Test Documentation

Release 2.0.0

Test

May 16, 2019

CONTENTS

1 Tutorials **3**
- 1.1 1. Brief Tour of E-Cell4 Simulations . . . 3
- 1.2 2. How to Build a Model . . . 8
- 1.3 3. How to Setup the Initial Condition . . . 15
- 1.4 4. How to Run a Simulation . . . 25
- 1.5 5. How to Log and Visualize Simulations . . . 30
- 1.6 6. How to Solve ODEs with Rate Law Functions . . . 36
- 1.7 7. Introduction of Rule-based Modeling . . . 44
- 1.8 8. More about 1. Brief Tour of E-Cell4 Simulations . . . 51
- 1.9 9. Spatial Gillespie Method . . . 55
- 1.10 10. Spatiocyte Simulations at Single-Molecule Resolution . . . 63

2 Examples **75**
- 2.1 Attractors . . . 75
- 2.2 Drosophila Circadian Clock . . . 77
- 2.3 Dual Phosphorylation Cycle . . . 78
- 2.4 Simple EGFR model . . . 80
- 2.5 A Simple Model of the Glycolysis of Human Erythrocytes . . . 86
- 2.6 Hodgkin-Huxley Model . . . 87
- 2.7 FitzHugh–Nagumo Model . . . 88
- 2.8 Lotka-Volterra 2D . . . 90
- 2.9 MinDE System with Mesoscopic Simulator . . . 92
- 2.10 MinDE System with Spatiocyte Simulator . . . 94
- 2.11 Simple Equilibrium . . . 96
- 2.12 Tyson1991 . . . 98

3 API **101**
- 3.1 E-Cell4 core API . . . 101
- 3.2 E-Cell4 gillespie API . . . 122
- 3.3 E-Cell4 ode API . . . 125
- 3.4 E-Cell4 meso API . . . 128
- 3.5 E-Cell4 spatiocyte API . . . 132
- 3.6 E-Cell4 bd API . . . 136
- 3.7 E-Cell4 egfrd API . . . 140

Python Module Index **145**

E-Cell System is a software platform for modeling, simulation and analysis of complex, heterogeneous and multi-scale systems like the cell.

E-Cell4 is a free and open-source software licensed under the GNU General Public License version 2. The source code is available on GitHub.

Please refer to https://github.com/ecell/ecell4 for information about **installation instructions**.

CHAPTER

ONE

TUTORIALS

1.1 1. Brief Tour of E-Cell4 Simulations

First of all, you have to load the E-Cell4 library:

[1]:
```
%matplotlib inline
from ecell4 import *
```

1.1.1 1.1. Quick Demo

There are three fundamental components consisting of E-Cell System version 4, which are `Model`, `World`, and `Simulator`. These components describe concepts in simulation.

- `Model` describes a problem to simulate as its name suggests.
- `World` describes a state, e.g. an initial state and a state at a time-point.
- `Simulator` describes a solver.

`Model` is independent from solvers. Every solver can share a single `Model` instance. Each solver alogrithm has a corresponding pair of `World` and `Simulator` (these pairs are capsulized into `Factory` class). `World` is not necessarily needed to be bound to `Model` and `Simulator`, but `Simulator` needs both `Model` and `World`.

Before running a simulation, you have to make a `Model`. E-Cell4 supports multiple ways to buld a `Model` (See **2. How to Build a Model** *local ipynb* readthedocs). Here, we explain the simplest way using the `with` statement with `reaction_rules`:

[2]:
```
with reaction_rules():
    A + B > C | 0.01  # equivalent to create_binding_reaction_rule
    C > A + B | 0.3   # equivalent to create_unbinding_reaction_rule

m1 = get_model()
print(m1)
```

```
<ecell4.core.NetworkModel object at 0x000002C0F32444B0>
```

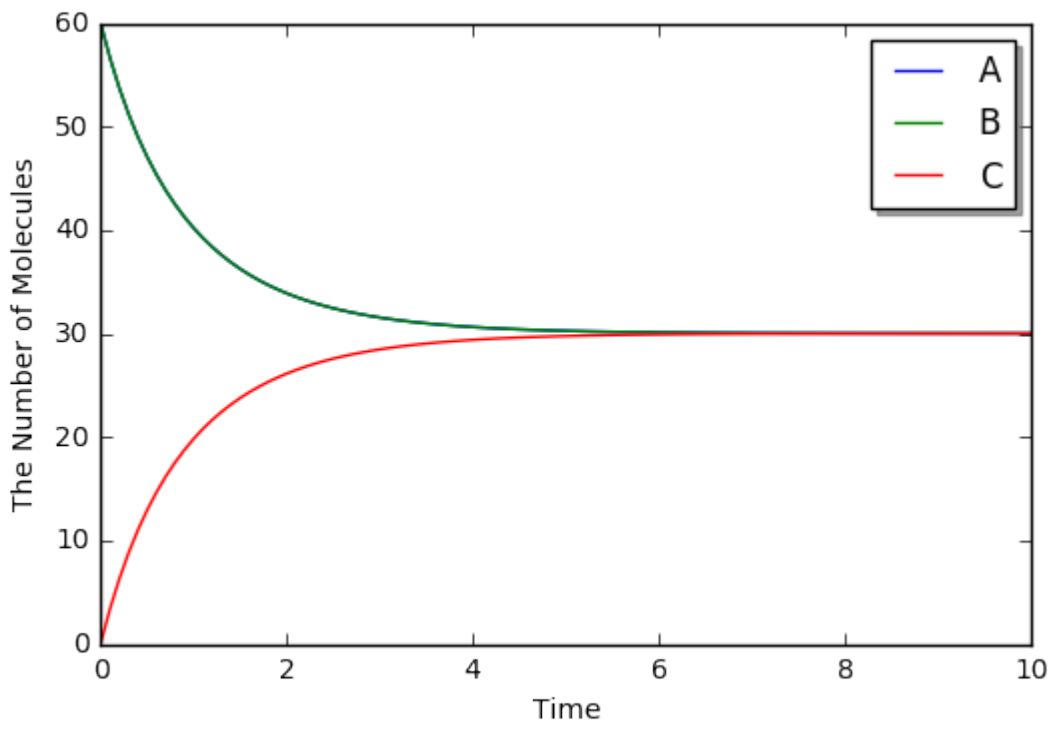
Please remember to write parentheses `()` after `reaction_rules`. Here, a `Model` with two `ReactionRules` named `m1` was built. Lines in the `with` block describe `ReactionRules`, a binding and unbinding reaction respectively. A kinetic rate for the mass action reaction is defined after a separator `|`, i.e. `0.01` or `0.3`. In the form of ordinary differential equations, this model can be described as:

$$[\mathrm{A}]' = [\mathrm{B}]' = -[\mathrm{C}] = -0.01[\mathrm{A}][\mathrm{B}] + 0.3[\mathrm{C}]$$

For more compact description, `A + B == C | (0.01, 0.3)` is also acceptable.

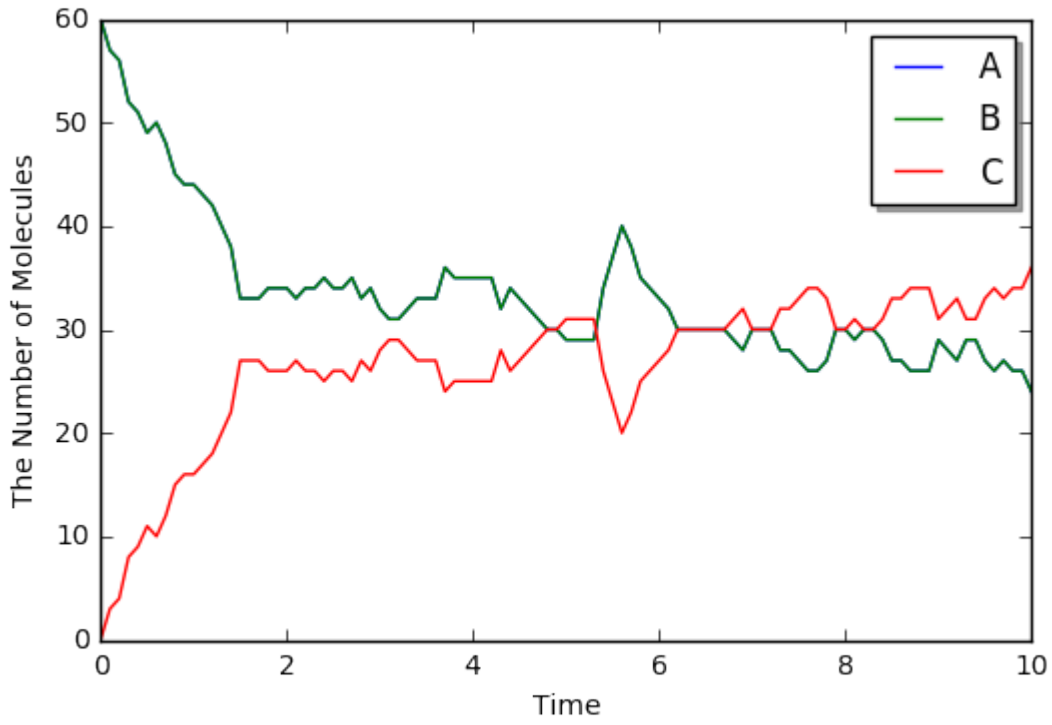
E-Cell4 has a simple interface for running simulation on a given model, called `run_simulation`. This enables for you to run simulations without instanciate `World` and `Simulator` by yourself. To solve this model, you have to give a volume, an initial value for each `Species` and duration of time:

[3]:
```
run_simulation(10.0, model=m1, y0={'A': 60, 'B': 60}, volume=1.0)
```



To switch simulation algorithm, you only need to give the type of solver (`ode` is used as a default) as follows:

[4]:
```
run_simulation(10.0, model=m1, y0={'A': 60, 'B': 60}, solver='gillespie')
```



1.1.2 1.2. Spatial Simulation and Visualization

E-Cell4 now supports multiple spatial simulation algorithms, `egfrd`, `spatiocyte` and `meso`. In addition to the model used in non-spatial solvers (`ode` and `gillespie`), these spatial solvers need extra information about each `Species`, i.e. a diffusion coefficient and radius.

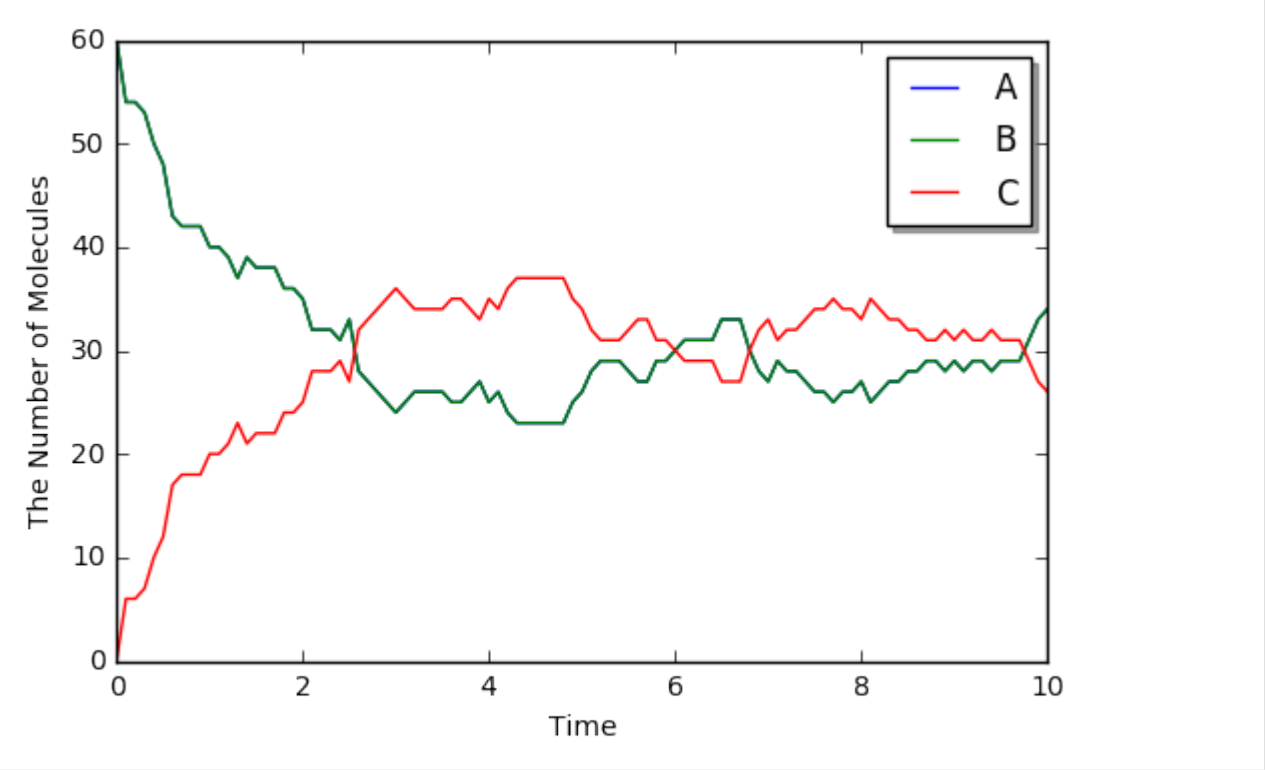
The `with` statement with `species_attributes` is available to describe these properties:

[5]:
```
with species_attributes():
    A | B | C | {'radius': '0.005', 'D': '1'}  # 'D' is for the diffusion coefficient

with reaction_rules():
    A + B == C | (0.01, 0.3)

m2 = get_model()
```

Even though the properties indicate a floating number, each attribute must be given as a string.

Now you can run a spatial simulation in the same way as above (in the case of `egfrd`, the time it takes to finish the simulation will be longer):

[6]:
```
run_simulation(10.0, model=m2, y0={'A': 60, 'B': 60}, solver='meso')
```



Structure (e.g. membrane, cytoplasm and nucleus) is only supported by `spatiocyte` and `meso` for now. For the simulation, `location` that each species belongs to must be specified in its attribute.

[7]:
```
with species_attributes():
    A | {'D': '1', 'location': 'S'}  # 'S' is a name of the structure

m3 = get_model()  # with no reactions
```

E-Cell4 supports primitive shapes as a structure like `Sphere`:

[8]:
```
sphere = Sphere(Real3(0.5, 0.5, 0.5), 0.48)  # a center position and radius
```

E-Cell4 provides various kinds of `Observers` which log the state during a simulation. In the following two observers are declared to record the position of the molecule. `FixedIntervalTrajectoryObserver` logs a trajectory of a molecule, and `FixedIntervalHDF5Observer` saves `World` to a HDF5 file at the given time interval:

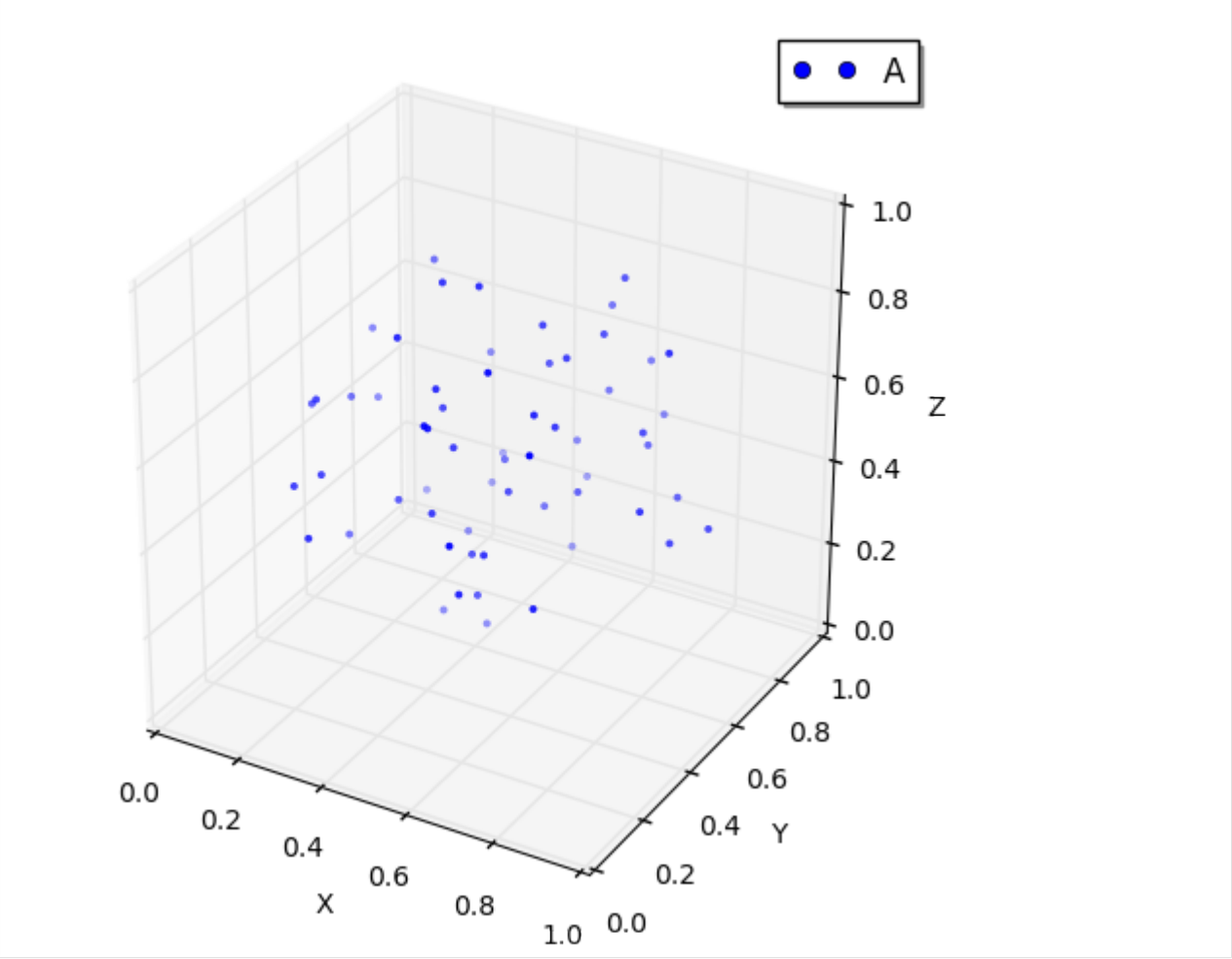
[9]:
```
obs1 = FixedIntervalTrajectoryObserver(1e-3)
obs2 = FixedIntervalHDF5Observer(0.1, 'test%02d.h5')
```

`run_simulation` can accept structures and observers as arguments `structure` and `observers` (see also `help(run_simulation)`):

[10]:
```
run_simulation(1.0, model=m3, y0={'A': 60}, structures={'S': sphere},
               solver='spatiocyte', observers=(obs1, obs2), return_type=None)
```

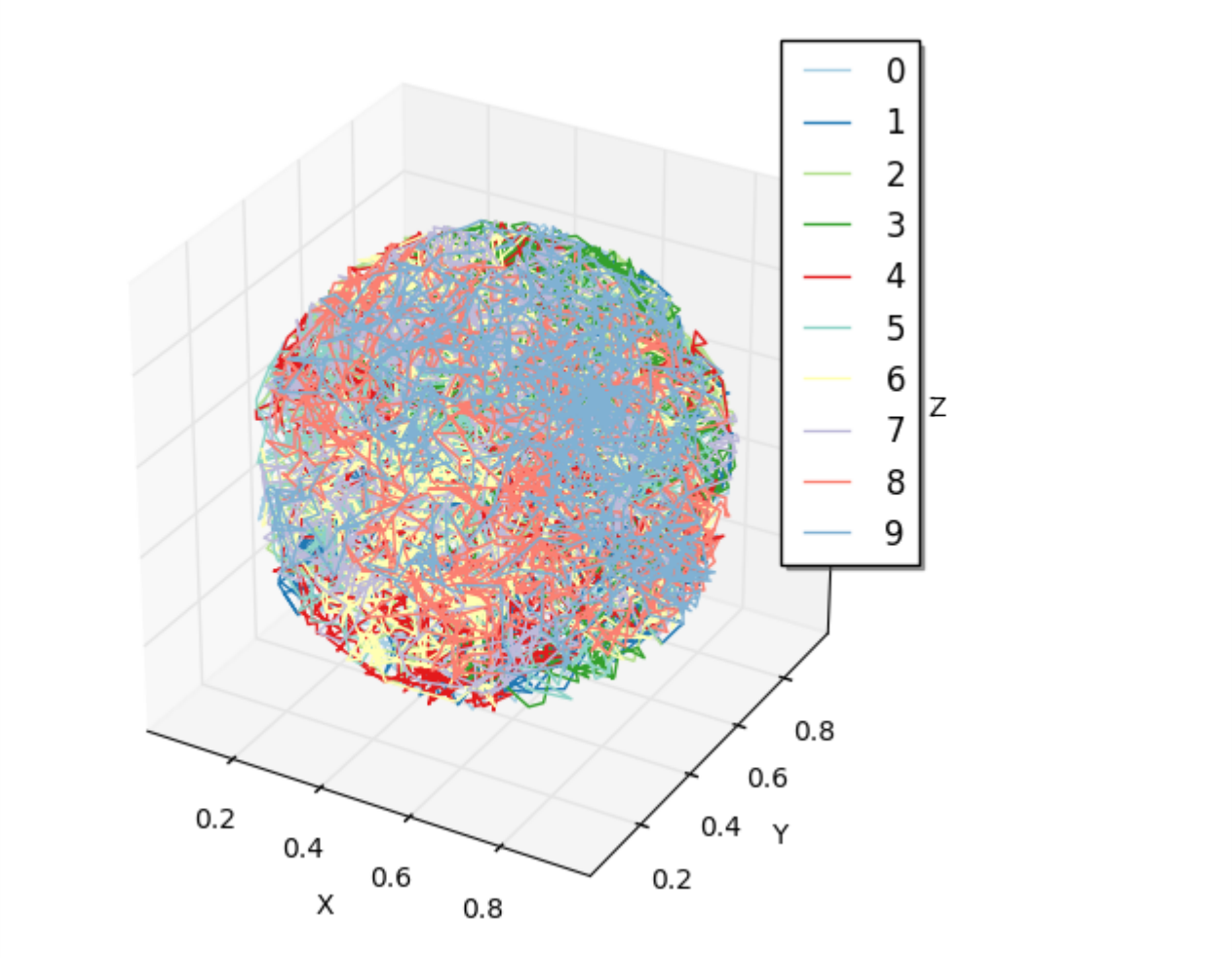
E-Cell4 has a function to visualize the world which is also capable of interactive visualization named `viz.plot_world`. `viz.plot_world` plots positions of molecules in 3D. In addition, by using `load_world`, you can easily restore the state of `World` from a HDF5 file:

[11]:
```
# viz.plot_world(load_world('test00.h5'), species_list=['A'])
viz.plot_world(load_world('test00.h5'), species_list=['A'], interactive=True)
```



Also, for `FixedIntervalTrajectoryObserver`,`viz.plot_trajectory` generates a plot of the trajectory. (Again, it is possible to generate interactive plots.):

[12]:
```
# viz.plot_trajectory(obs1)
viz.plot_trajectory(obs1, interactive=True)
```



For more details, see **5. How to Log and Visualize Simulations** local ipynb readthedocs.

1.2 2. How to Build a Model

`Model` is composed of a set of `Species` and `ReactionRules`.

- `Species` describes a molecule entitie (e.g. a type or state of a protein) in the model. `Species` also has its attributes like the size.
- `ReactionRule` describes the interactions between `Species` (e.g. binding and unbinding).

[1]:
```
%matplotlib inline
from ecell4 import *
```

1.2.1 2.1. Species

`Species` can be generated by giving its name:

[2]:
```
sp1 = Species("A")
print(sp1.serial())
```

```
A
```

There are some naming conventions for the name of `Species`. This naming convention requires careful use of special symbols (e.g. parenthesis `()`, dot `.`, underbar _), numbers and blank.

`Species` has a set of APIs for handling its attributes:

[3]:
```
sp1.set_attribute("radius", "0.005")
sp1.set_attribute("D", "1")
sp1.set_attribute("location", "cytoplasm")
print(sp1.get_attribute("radius"))
sp1.remove_attribute("radius")
print(sp1.has_attribute("radius"))
```

```
0.005
False
```

The arguments in `set_attribute` is the name of attribute and its value. Both of them have to be string. There is a shortcut to set the attributes above at once because `radius`, `D` (a diffusion coefficient) and `location` are frequently used.

[4]:
```
sp1 = Species("A", "0.005", "1", "cytoplasm")  # serial, radius, D, location
```

The equality between `Species` is just evaluated based on their `serial`:

[5]:
```
print(Species("A") == Species("B"), Species("A") == Species("A"))
```

```
False True
```

A Species consists of one or more UnitSpecies:

[6]:
```
sp1 = Species()
usp1 = UnitSpecies("C")
print(usp1.serial())
sp1.add_unit(usp1)
sp1.add_unit(UnitSpecies("A"))
sp1.add_unit(UnitSpecies("B"))
print(sp1.serial(), len(sp1.units()))
```

```
C
C.A.B 3
```

A `Species` can be reproduced from a serial. In a `serial`, all `UnitSpecies` are joined with the separator, dot `.`. The order of `UnitSpecies` affects the `Species` comparison.

[7]:
```
sp1 = Species("C.A.B")
print(sp1.serial())
print(Species("A.B.C") == Species("C.A.B"))
print(Species("A.B.C") == Species("A.B.C"))
```

```
C.A.B
False
True
```

`UnitSpecies` can have sites. Sites consists of a `name`, `state` and `bond`, and are sorted automatically in `UnitSpecies`. `name` must be unique in a `UnitSpecies`. All the value have to be string. Do not include parenthesis, dot and blank, and not start from numbers except for `bond`.

[8]:
```
usp1 = UnitSpecies("A")
usp1.add_site("us", "u", "")
usp1.add_site("ps", "p", "_")
usp1.add_site("bs", "", "_")
print(usp1.serial())
```

```
A(bs^_,ps=p^_,us=u)
```

`UnitSpecies` can be also reproduced from its serial. Please be careful with the order of sites where a site with a state must be placed after sites with no state specification:

[9]:
```
usp1 = UnitSpecies()
usp1.deserialize("A(bs^_, us=u, ps=p^_)")
print(usp1.serial())
```

```
A(bs^_,ps=p^_,us=u)
```

Of course, a site of `UnitSpecies` is available even in `Species`' serial.

[10]:
```
sp1 = Species("A(bs^1, ps=u).A(bs, ps=p^1)")
print(sp1.serial())
print(len(sp1.units()))
```

```
A(bs^1,ps=u).A(bs,ps=p^1)
2
```

The information (`UnitSpecies` and its `site`) is used for rule-based modeling. The way of rule-based modeling in E-Cell4 will be explained in **7. Introduction of Rule-based Modeling** *local ipynb* readthedocs.

1.2.2 2.2. ReactionRule

`ReactionRule` consists of `reactants`, `products` and `k`. `reactants` and `products` are a list of `Species`, and `k` is a kinetic rate given as a floating number.

[11]:
```
rr1 = ReactionRule()
rr1.add_reactant(Species("A"))
rr1.add_reactant(Species("B"))
rr1.add_product(Species("C"))
rr1.set_k(1.0)
```

Here is a binding reaction from `A` and `B` to `C`. In this reaction definition, you don't need to set attributes to `Species`. The above series of operations can be written in one line using `create_binding_reaction_rule(Species("A"), Species("B"), Species("C"), 1.0)`.

You can use `as_string` function to check `ReactionRule`:

[12]:
```
rr1 = create_binding_reaction_rule(Species("A"), Species("B"), Species("C"), 1.0)
print(rr1.as_string())
```

```
A+B>C|1
```

You can also provide components to the constructor:

[13]:
```
rr1 = ReactionRule([Species("A"), Species("B")], [Species("C")], 1.0)
print(rr1.as_string())
```

```
A+B>C|1
```

Basically `ReactionRule` represents a mass action reaction with the rate `k`. `ode` solver also supports rate laws thought it's under development yet. `ode.ODERatelaw` is explained in **6. How to Solve ODEs with Rate Law Functions** *local ipynb* readthedocs.

1.2.3 2.3. NetworkModel

You have learned how to create some `Model` components. Next let's put the components in a `Model`.

[14]:
```
sp1 = Species("A", "0.005", "1")
sp2 = Species("B", "0.005", "1")
sp3 = Species("C", "0.01", "0.5")
```

[15]:
```
rr1 = create_binding_reaction_rule(Species("A"), Species("B"), Species("C"), 0.01)
rr2 = create_unbinding_reaction_rule(Species("C"), Species("A"), Species("B"), 0.3)
```

You can put the `Species` and `ReactionRule` with `add_species_attribute` and `add_reaction_rule`.

[16]:
```
m1 = NetworkModel()
m1.add_species_attribute(sp1)
m1.add_species_attribute(sp2)
m1.add_species_attribute(sp3)
m1.add_reaction_rule(rr1)
m1.add_reaction_rule(rr2)
```

Now we have a simple model with the binding and unbinding reactions. You can use `species_attributes` and `reaction_rules` to check the `Model`.

[17]:
```
print([sp.serial() for sp in m1.species_attributes()])
print([rr.as_string() for rr in m1.reaction_rules()])
```

```
['A', 'B', 'C']
['A+B>C|0.01', 'C>A+B|0.3']
```

The `Species` attributes are required for the spatial `Model`, but not required for the nonspatial `Model` (i.e. `gillespie` or `ode`). The attribute pushed first has higher priority than one pushed later. You can also attribute a `Species` based on the attributes in a `Model`.

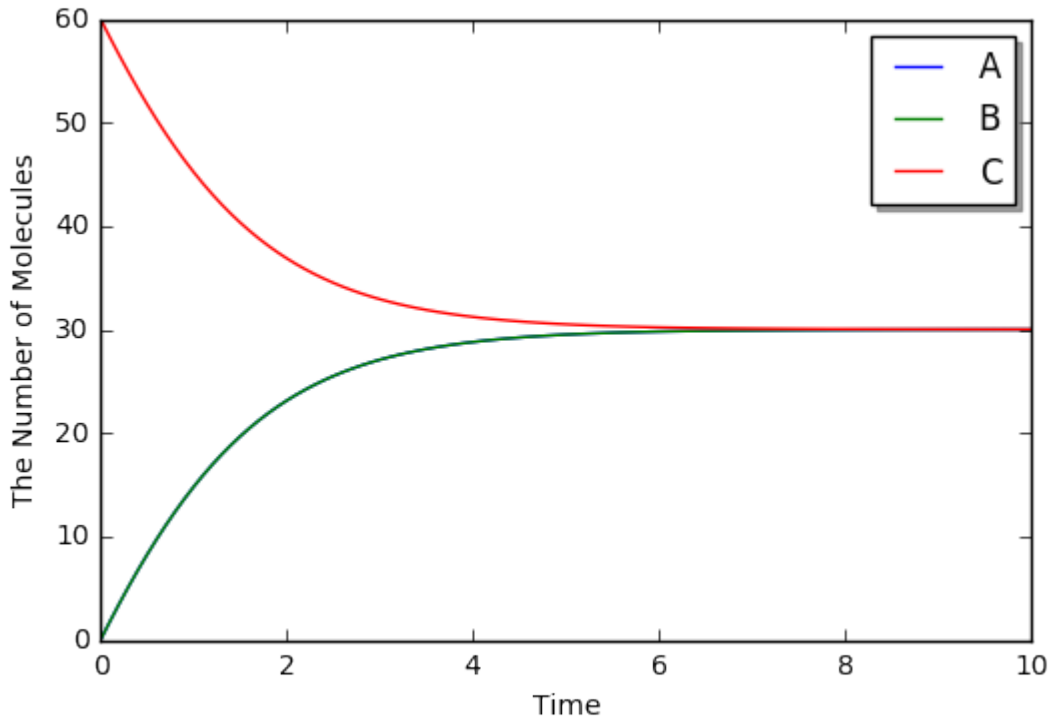
[18]:
```
sp1 = Species("A")
print(sp1.has_attribute("radius"))
sp2 = m1.apply_species_attributes(sp1)
print(sp2.has_attribute("radius"))
print(sp2.get_attribute("radius"))
```

```
False
True
0.005
```

For your information, all functions related to `Species`, `ReactionRule` and `NetworkModel` above are also available on C++ in the same way.

Finally, you can solve this model with `run_simulation` as explained in **1. Brief Tour of E-Cell4 Simulations** *local ipynb* readthedocs :

[19]:
```
run_simulation(10.0, model=m1, y0={'C': 60})
```



1.2.4 2.4. Python Utilities to Build a Model

As shown in **1. Brief Tour of E-Cell4 Simulations** *local ipynb* readthedocs, E-Cell4 also provides the easier way to build a model using `with` statement:

[20]:
```
with species_attributes():
    A | B | {'radius': '0.005', 'D': '1'}
    C | {'radius': '0.01', 'D': '0.5'}

with reaction_rules():
    A + B == C | (0.01, 0.3)

m1 = get_model()
```

For reversible reactions, <> is also available instead of == on Python 2, but deprecated on Python3. In the `with` statement, undeclared variables are automaticaly assumed to be a `Species`. Any Python variables, functions and statement are available even in the `with` block.

[21]:
```
from math import log

ka, kd, kf = 0.01, 0.3, 0.1
tau = 10.0

with reaction_rules():
    E0 + S == ES | (ka, kd)

    if tau > 0:
        ES > E1 + P | kf
        E1 > E0 | log(2) / tau
```

(continues on next page)

(continued from previous page)

```
    else:
        ES > E0 + P | kf

m1 = get_model()
del ka, kd, kf, tau
```

Meanwhile, once some variable is declared even outside the block, you can not use its name as a `Species` as follows:

[22]:
```
A = 10

try:
    with reaction_rules():
        A + B == C | (0.01, 0.3)
except Exception as e:
    print(repr(e))

del A
```

```
RuntimeError('invalid expression; "10" given',)
```

where `A + B == C` exactly means `10 + B == C`.

In the absence of left or right hand side of `ReactionRule` like synthesis or degradation, you may want to describe like:

```
with reaction_rules():
    A > | 1.0  # XXX: will raise SyntaxError
    > A | 1.0  # XXX: will raise SyntaxError
```

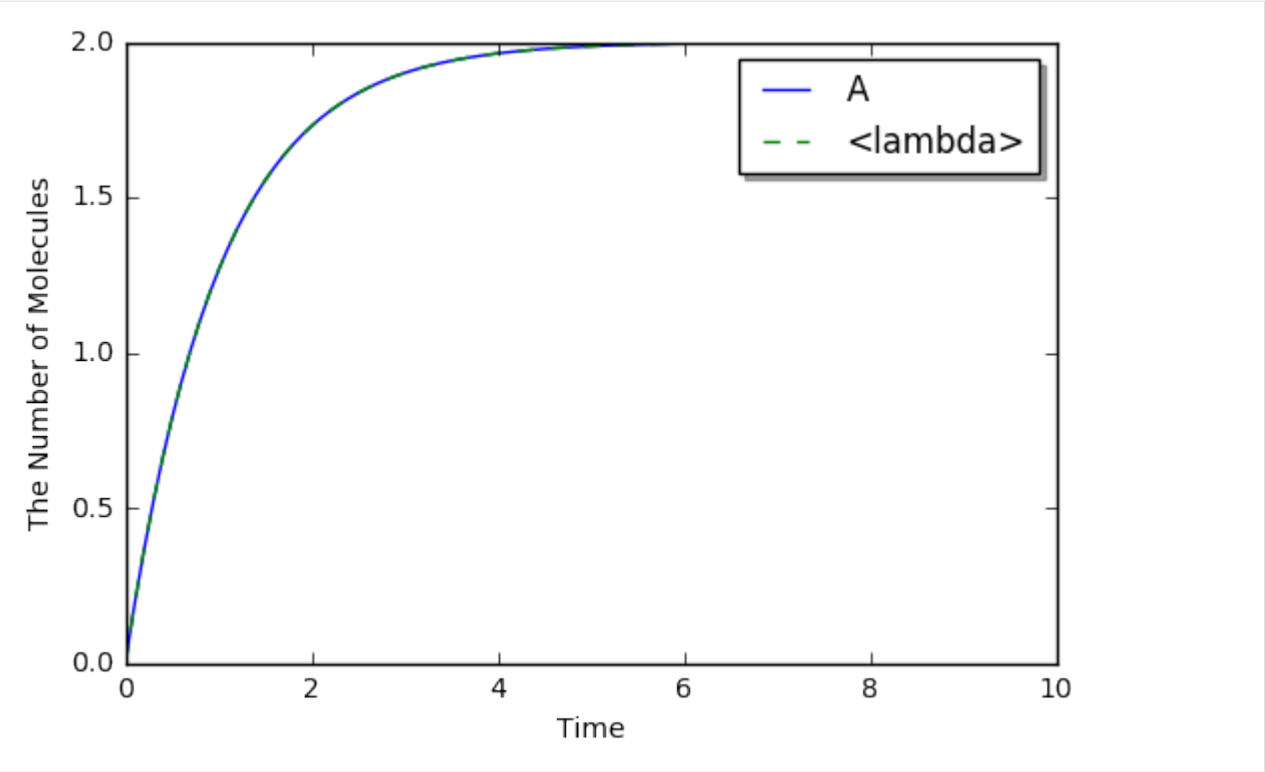
However, this is not accepted because of `SyntaxError` on Python. To describe this case, a special operator, tilde `~`, is available. `~` sets a stoichiometric coefficient of the following `Species` as zero, which means the `Species` is just ignored in the `ReactionRule`.

[23]:
```
with reaction_rules():
    ~A > A | 2.0  # equivalent to `create_synthesis_reaction_rule`
    A > ~A | 1.0  # equivalent to `create_degradation_reaction_rule`

m1 = get_model()
print([rr.as_string() for rr in m1.reaction_rules()])
```

```
['>A|2', 'A>|1']
```

The following `Species`' name is not necessarily needed to be the same as others. The model above describes $[A]' = 2 - [A]$:

[24]:
```
from math import exp
run_simulation(10.0, model=m1, opt_args=['-', lambda t: 2.0 * (1 - exp(-t)), '--'])
```



A chain of reactions can be described in one line. To split a line into two or more physical lines, wrap lines in a parenthesis:

[25]:
```
with reaction_rules():
    (E + S == ES | (0.5, 1.0)
         > E + P | 1.5)

m1 = get_model()
print([rr.as_string() for rr in m1.reaction_rules()])
```

```
['E+S>ES|0.5', 'ES>E+S|1', 'ES>E+P|1.5']
```

The method uses global variables in `ecell4.util.decorator` (e.g. `REACTION_RULES`) to cache objects created in the `with` statement:

[26]:
```
import ecell4.util.decorator

with reaction_rules():
    A + B == C | (0.01, 0.3)

print(ecell4.util.decorator.REACTION_RULES)  #XXX: Only for debugging
get_model()
print(ecell4.util.decorator.REACTION_RULES)  #XXX: Only for debugging
```

```
[<ecell4.core.ReactionRule object at 0x000001A123D02BA0>, <ecell4.core.ReactionRule␣
↪object at 0x000001A123D02CD8>]
[]
```

Python decorator functions are also available. Decorator functions improve the modularity of the `Model`.

[27]:
```
@species_attributes
def attrgen1(radius, D):
    A | B | {'radius': str(radius), 'D': str(D)}
    C | {'radius': str(radius * 2), 'D': str(D * 0.5)}

@reaction_rules
def rrgen1(kon, koff):
    A + B == C | (kon, koff)

attrs1 = attrgen1(0.005, 1)
rrs1 = rrgen1(0.01, 0.3)
print(attrs1)
print(rrs1)
```

```
[<ecell4.core.Species object at 0x000001A123D02CD8>, <ecell4.core.Species object at␣
↪0x000001A123D02BA0>, <ecell4.core.Species object at 0x000001A123D02DC8>]
[<ecell4.core.ReactionRule object at 0x000001A123D02D20>, <ecell4.core.ReactionRule␣
↪object at 0x000001A123D02DE0>]
```

In contrast to the `with` statement, do **not** add parentheses after the decorator here. The functions decorated by `reaction_rules` and `species_attributes` return a list of `ReactionRules` and `Species` respectively. The list can be registered to `Model` at once by using `add_reaction_rules` and `add_species_attributes`.

[28]:
```
m1 = NetworkModel()
m1.add_species_attributes(attrs1)
m1.add_reaction_rules(rrs1)
print(m1.num_reaction_rules())
```

```
2
```

This method is modular and reusable relative to the way using `with` statement.

1.3 3. How to Setup the Initial Condition

Here, we explain the basics of `World` classes. In E-Cell4, six types of World classes are supported now: `spatiocyte.SpatiocyteWorld`, `egfrd.EGFRDWorld`, `bd.BDWorld`, `meso.MesoscopicWorld`, `gillespie.GillespieWorld`, and `ode.ODEWorld`.

In the most of softwares, the initial condition is supposed to be a part of `Model`. But, in E-Cell4, the initial condition must be set up as `World` separately from `Model`. `World` stores an information about the state at a time point, such as a current time, the number of molecules, coordinate of molecules, structures, and random number generator. Meanwhile, `Model` contains the type of interactions between molecules and the common properties of molecules.

[1]:
```
import ecell4
```

1.3.1 3.1. Common APIs of World

Even though `World` describes the spatial representation specific to the corresponding algorithm, it has compatible APIs. In this section, we introduce the common interfaces of the six `World` classes.

[2]:
```
from ecell4.core import *
from ecell4.gillespie import GillespieWorld
from ecell4.ode import ODEWorld
from ecell4.spatiocyte import SpatiocyteWorld
```

(continues on next page)

(continued from previous page)

```
from ecell4.bd import BDWorld
from ecell4.meso import MesoscopicWorld
from ecell4.egfrd import EGFRDWorld
```

`World` classes accept different sets of arguments in the constructor, which determine the parameters specific to the algorithm. However, at least, all `World` classes can be instantiated only with their size, named `edge_lengths`. The type of `edge_lengths` is `Real3`, which represents a triplet of `Reals`. In E-Cell4, all 3-dimensional positions are treated as a `Real3`. See also 8. More about 1. Brief Tour of E-Cell4 Simulations.

[3]:
```
edge_lengths = Real3(1, 2, 3)
w1 = GillespieWorld(edge_lengths)
w2 = ODEWorld(edge_lengths)
w3 = SpatiocyteWorld(edge_lengths)
w4 = BDWorld(edge_lengths)
w5 = MesoscopicWorld(edge_lengths)
w6 = EGFRDWorld(edge_lengths)
```

`World` has getter methods for the size and volume.

[4]:
```
print(tuple(w1.edge_lengths()), w1.volume())
print(tuple(w2.edge_lengths()), w2.volume())
print(tuple(w3.edge_lengths()), w3.volume())
print(tuple(w4.edge_lengths()), w4.volume())
print(tuple(w5.edge_lengths()), w5.volume())
print(tuple(w6.edge_lengths()), w6.volume())
```

```
((1.0, 2.0, 3.0), 6.0)
((1.0, 2.0, 3.0), 6.0)
((1.0, 2.0, 3.0), 6.0)
((1.0, 2.0, 3.0), 6.0)
((1.0, 2.0, 3.0), 6.0)
((1.0, 2.0, 3.0), 6.0)
```

Next, let's add molecules into the World. Here, you must give `Species` attributed with **radius** and **D** to tell the shape of molecules. In the example below **0.0025** corresponds to **radius** and **1** to **D**. Positions of the molecules are randomly determined in the `World` if needed. **10** in **add_molecules** function represents the number of molecules to be added.

[5]:
```
sp1 = Species("A", "0.0025", "1")
w1.add_molecules(sp1, 10)
w2.add_molecules(sp1, 10)
w3.add_molecules(sp1, 10)
w4.add_molecules(sp1, 10)
w5.add_molecules(sp1, 10)
w6.add_molecules(sp1, 10)
```

After model is bound to world, you do not need to rewrite the **radius** and **D** once set in Species (unless you want to change it).

[6]:
```
m = NetworkModel()
m.add_species_attribute(Species("A", "0.0025", "1"))
m.add_species_attribute(Species("B", "0.0025", "1"))

w1.bind_to(m)
w2.bind_to(m)
w3.bind_to(m)
w4.bind_to(m)
```

(continues on next page)

(continued from previous page)

```
w5.bind_to(m)
w6.bind_to(m)
w1.add_molecules(Species("B"), 20)
w2.add_molecules(Species("B"), 20)
w3.add_molecules(Species("B"), 20)
w4.add_molecules(Species("B"), 20)
w5.add_molecules(Species("B"), 20)
w6.add_molecules(Species("B"), 20)
```

Similarly, `remove_molecules` and `num_molecules_exact` are also available.

[7]:
```
w1.remove_molecules(Species("B"), 5)
w2.remove_molecules(Species("B"), 5)
w3.remove_molecules(Species("B"), 5)
w4.remove_molecules(Species("B"), 5)
w5.remove_molecules(Species("B"), 5)
w6.remove_molecules(Species("B"), 5)
```

[8]:
```
print(w1.num_molecules_exact(Species("A")), w2.num_molecules_exact(Species("A")), w3.
↪num_molecules_exact(Species("A")), w4.num_molecules_exact(Species("A")), w5.num_
↪molecules_exact(Species("A")), w6.num_molecules_exact(Species("A")))
print(w1.num_molecules_exact(Species("B")), w2.num_molecules_exact(Species("B")), w3.
↪num_molecules_exact(Species("B")), w4.num_molecules_exact(Species("B")), w5.num_
↪molecules_exact(Species("B")), w6.num_molecules_exact(Species("B")))
```

```
(10, 10, 10, 10, 10, 10)
(15, 15, 15, 15, 15, 15)
```

Unlike `num_molecules_exact`, `num_molecules` returns the numbers that match a given `Species` in rule-based fashion. When all `Species` in the `World` has no molecular interaction, `num_molecules` is same with `num_molecules_exact`.

[9]:
```
print(w1.num_molecules(Species("A")), w2.num_molecules(Species("A")), w3.num_
↪molecules(Species("A")), w4.num_molecules(Species("A")), w5.num_molecules(Species("A
↪")), w6.num_molecules(Species("A")))
print(w1.num_molecules(Species("B")), w2.num_molecules(Species("B")), w3.num_
↪molecules(Species("B")), w4.num_molecules(Species("B")), w5.num_molecules(Species("B
↪")), w6.num_molecules(Species("B")))
```

```
(10, 10, 10, 10, 10, 10)
(15, 15, 15, 15, 15, 15)
```

`World` holds its simulation time.

[10]:
```
print(w1.t(), w2.t(), w3.t(), w4.t(), w5.t(), w6.t())
w1.set_t(1.0)
w2.set_t(1.0)
w3.set_t(1.0)
w4.set_t(1.0)
w5.set_t(1.0)
w6.set_t(1.0)
print(w1.t(), w2.t(), w3.t(), w4.t(), w5.t(), w6.t())
```

```
(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
(1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

Finally, you can `save` and `load` the state of a `World` into/from a HDF5 file.

[11]:
```
w1.save("gillespie.h5")
w2.save("ode.h5")
w3.save("spatiocyte.h5")
w4.save("bd.h5")
w5.save("meso.h5")
w6.save("egfrd.h5")
del w1, w2, w3, w4, w5, w6
```

[12]:
```
w1 = GillespieWorld()
w2 = ODEWorld()
w3 = SpatiocyteWorld()
w4 = BDWorld()
w5 = MesoscopicWorld()
w6 = EGFRDWorld()
print(w1.t(), tuple(w1.edge_lengths()), w1.volume(), w1.num_molecules(Species("A")),␣
↪w1.num_molecules(Species("B")))
print(w2.t(), tuple(w2.edge_lengths()), w2.volume(), w2.num_molecules(Species("A")),␣
↪w2.num_molecules(Species("B")))
print(w3.t(), tuple(w3.edge_lengths()), w3.volume(), w3.num_molecules(Species("A")),␣
↪w3.num_molecules(Species("B")))
print(w4.t(), tuple(w4.edge_lengths()), w4.volume(), w4.num_molecules(Species("A")),␣
↪w4.num_molecules(Species("B")))
print(w5.t(), tuple(w5.edge_lengths()), w5.volume(), w5.num_molecules(Species("A")),␣
↪w5.num_molecules(Species("B")))
print(w6.t(), tuple(w6.edge_lengths()), w6.volume(), w6.num_molecules(Species("A")),␣
↪w6.num_molecules(Species("B")))
```

```
(0.0, (1.0, 1.0, 1.0), 1.0, 0, 0)
(0.0, (1.0, 1.0, 1.0), 1.0, 0, 0)
(0.0, (1.0, 1.0, 1.0), 1.0, 0, 0)
(0.0, (1.0, 1.0, 1.0), 1.0, 0, 0)
(0.0, (1.0, 1.0, 1.0), 1.0, 0, 0)
(0.0, (1.0, 1.0, 1.0), 1.0, 0, 0)
```

[13]:
```
w1.load("gillespie.h5")
w2.load("ode.h5")
w3.load("spatiocyte.h5")
w4.load("bd.h5")
w5.load("meso.h5")
w6.load("egfrd.h5")
print(w1.t(), tuple(w1.edge_lengths()), w1.volume(), w1.num_molecules(Species("A")),␣
↪w1.num_molecules(Species("B")))
print(w2.t(), tuple(w2.edge_lengths()), w2.volume(), w2.num_molecules(Species("A")),␣
↪w2.num_molecules(Species("B")))
print(w3.t(), tuple(w3.edge_lengths()), w3.volume(), w3.num_molecules(Species("A")),␣
↪w3.num_molecules(Species("B")))
print(w4.t(), tuple(w4.edge_lengths()), w4.volume(), w4.num_molecules(Species("A")),␣
↪w4.num_molecules(Species("B")))
print(w5.t(), tuple(w5.edge_lengths()), w5.volume(), w5.num_molecules(Species("A")),␣
↪w5.num_molecules(Species("B")))
print(w6.t(), tuple(w6.edge_lengths()), w6.volume(), w6.num_molecules(Species("A")),␣
↪w6.num_molecules(Species("B")))
del w1, w2, w3, w4, w5, w6
```

```
(1.0, (1.0, 2.0, 3.0), 6.0, 10, 15)
(1.0, (1.0, 2.0, 3.0), 6.0, 10, 15)
(1.0, (1.0, 2.0, 3.0), 6.0, 10, 15)
(1.0, (1.0, 2.0, 3.0), 6.0, 10, 15)
```

(continues on next page)

(continued from previous page)

```
(1.0, (1.0, 2.0, 3.0), 6.0, 10, 15)
(1.0, (1.0, 2.0, 3.0), 6.0, 10, 15)
```

All the `World` classes also accept a HDF5 file path as an unique argument of the constructor.

[14]:
```
print(GillespieWorld("gillespie.h5").t())
print(ODEWorld("ode.h5").t())
print(SpatiocyteWorld("spatiocyte.h5").t())
print(BDWorld("bd.h5").t())
print(MesoscopicWorld("meso.h5").t())
print(EGFRDWorld("egfrd.h5").t())
```

```
1.0
1.0
1.0
1.0
1.0
1.0
```

1.3.2 3.2. How to Get Positions of Molecules

`World` also has the common functions to access the coordinates of the molecules.

[15]:
```
w1 = GillespieWorld()
w2 = ODEWorld()
w3 = SpatiocyteWorld()
w4 = BDWorld()
w5 = MesoscopicWorld()
w6 = EGFRDWorld()
```

First, you can place a molecule at the certain position with `new_particle`.

[16]:
```
sp1 = Species("A", "0.0025", "1")
pos = Real3(0.5, 0.5, 0.5)
(pid1, p1), suc1 = w1.new_particle(sp1, pos)
(pid2, p2), suc2 = w2.new_particle(sp1, pos)
(pid3, p3), suc3 = w3.new_particle(sp1, pos)
(pid4, p4), suc4 = w4.new_particle(sp1, pos)
(pid5, p5), suc5 = w5.new_particle(sp1, pos)
(pid6, p6), suc6 = w6.new_particle(sp1, pos)
```

`new_particle` returns a particle created and whether it's succeeded or not. The resolution in representation of molecules differs. For example, `GillespieWorld` has almost no information about the coordinate of molecules. Thus, it simply ignores the given position, and just counts up the number of molecules here.

`ParticleID` is a pair of `Integers` named `lot` and `serial`.

[17]:
```
print(pid6.lot(), pid6.serial())
print(pid6 == ParticleID((0, 1)))
```

```
(0, 1L)
True
```

Particle simulators, i.e. `spatiocyte`, `bd` and `egfrd`, provide an interface to access a particle by its id. `has_particle` returns if a particles exists or not for the given `ParticleID`.

[18]:
```
# print(w1.has_particle(pid1))
# print(w2.has_particle(pid2))
print(w3.has_particle(pid3))  # => True
print(w4.has_particle(pid4))  # => True
# print(w5.has_particle(pid5))
print(w6.has_particle(pid6))  # => True
```

```
True
True
True
```

After checking the existency, you can get the partcle by `get_particle` as follows.

[19]:
```
# pid1, p1 = w1.get_particle(pid1)
# pid2, p2 = w2.get_particle(pid2)
pid3, p3 = w3.get_particle(pid3)
pid4, p4 = w4.get_particle(pid4)
# pid5, p5 = w5.get_particle(pid5)
pid6, p6 = w6.get_particle(pid6)
```

`Particle` consists of `species`, `position`, `radius` and `D`.

[20]:
```
# print(p1.species().serial(), tuple(p1.position()), p1.radius(), p1.D())
# print(p2.species().serial(), tuple(p2.position()), p2.radius(), p2.D())
print(p3.species().serial(), tuple(p3.position()), p3.radius(), p3.D())
print(p4.species().serial(), tuple(p4.position()), p4.radius(), p4.D())
# print(p5.species().serial(), tuple(p5.position()), p5.radius(), p5.D())
print(p6.species().serial(), tuple(p6.position()), p6.radius(), p6.D())
```

```
(u'A', (0.5062278801751902, 0.5080682368868706, 0.5), 0.0025, 1.0)
(u'A', (0.5, 0.5, 0.5), 0.0025, 1.0)
(u'A', (0.5, 0.5, 0.5), 0.0025, 1.0)
```

In the case of `spatiocyte`, a particle position is automatically round to the center of the voxel nearest to the given position.

You can even move the position of the particle. `update_particle` replace the particle specified with the given `ParticleID` with the given `Particle` and return `False`. If no corresponding particle is found, create new particle and return `True`. If you give a `Particle` with the different type of `Species`, the `Species` of the `Particle` will be also changed.

[21]:
```
newp = Particle(sp1, Real3(0.3, 0.3, 0.3), 0.0025, 1)
# print(w1.update_particle(pid1, newp))
# print(w2.update_particle(pid2, newp))
print(w3.update_particle(pid3, newp))
print(w4.update_particle(pid4, newp))
# print(w5.update_particle(pid5, newp))
print(w6.update_particle(pid6, newp))
```

```
False
False
False
```

`list_particles` and `list_particles_exact` return a list of pairs of `ParticleID` and `Particle` in the `World`. `World` automatically makes up for the gap with random numbers. For example, `GillespieWorld` returns a list of positions randomly distributed in the `World` size.

[22]:
```
print(w1.list_particles_exact(sp1))
# print(w2.list_particles_exact(sp1))  # ODEWorld has no member named list_particles
print(w3.list_particles_exact(sp1))
print(w4.list_particles_exact(sp1))
print(w5.list_particles_exact(sp1))
print(w6.list_particles_exact(sp1))
```

```
[(<ecell4.core.ParticleID object at 0x7fdfd28e8b58>, <ecell4.core.Particle object at␣
↪0x7fdfd28e8bd0>)]
[(<ecell4.core.ParticleID object at 0x7fdfd28e8b58>, <ecell4.core.Particle object at␣
↪0x7fdfd28e8af8>)]
[(<ecell4.core.ParticleID object at 0x7fdfd28e8b58>, <ecell4.core.Particle object at␣
↪0x7fdfd28e8a38>)]
[(<ecell4.core.ParticleID object at 0x7fdfd28e8b58>, <ecell4.core.Particle object at␣
↪0x7fdfd28e8bd0>)]
[(<ecell4.core.ParticleID object at 0x7fdfd28e8b58>, <ecell4.core.Particle object at␣
↪0x7fdfd28e8af8>)]
```

You can remove a specific particle with `remove_particle`.

[23]:
```
# w1.remove_particle(pid1)
# w2.remove_particle(pid2)
w3.remove_particle(pid3)
w4.remove_particle(pid4)
# w5.remove_particle(pid5)
w6.remove_particle(pid6)
# print(w1.has_particle(pid1))
# print(w2.has_particle(pid2))
print(w3.has_particle(pid3))  # => False
print(w4.has_particle(pid4))  # => False
# print(w5.has_particle(pid5))
print(w6.has_particle(pid6))  # => False
```

```
False
False
False
```

1.3.3 3.3. Lattice-based Coordinate

In addition to the common interface, each `World` can have their own interfaces. As an example, we explain methods to handle lattice-based coordinate here. `SpatiocyteWorld` is based on a space discretized to hexiagonal close packing lattices, `LatticeSpace`.

[24]:
```
w = SpatiocyteWorld(Real3(1, 2, 3), voxel_radius=0.01)
w.bind_to(m)
```

The size of a single lattice, called `Voxel`, can be obtained by `voxel_radius()`. `SpatiocyteWorld` has methods to get the numbers of rows, columns, and layers. These sizes are automatically calculated based on the given `edge_lengths` at the construction.

[25]:
```
print(w.voxel_radius())  # => 0.01
print(tuple(w.shape()))  # => (62, 152, 116)
# print(w.col_size(), w.row_size(), w.layer_size())  # => (62, 152, 116)
print(w.size())  # => 1093184 = 62 * 152 * 116
```

```
0.01
(62, 152, 116)
```

(continues on next page)

(continued from previous page)

```
(62, 152, 116)
1093184
```

A position in the lattice-based space is treated as an `Integer3`, column, row and layer, called a global coordinate. Thus, `SpatiocyteWorld` provides the function to convert the `Real3` into a lattice-based coordinate.

[26]:
```
# p1 = Real3(0.5, 0.5, 0.5)
# g1 = w.position2global(p1)
# p2 = w.global2position(g1)
# print(tuple(g1))  # => (31, 25, 29)
# print(tuple(p2))  # => (0.5062278801751902, 0.5080682368868706, 0.5)
```

```
(31, 25, 29)
(0.5062278801751902, 0.5080682368868706, 0.5)
```

In `SpatiocyteWorld`, the global coordinate is translated to a single integer. It is just called a coordinate. You can also treat the coordinate as in the same way with a global coordinate.

[27]:
```
# p1 = Real3(0.5, 0.5, 0.5)
# c1 = w.position2coordinate(p1)
# p2 = w.coordinate2position(c1)
# g1 = w.coord2global(c1)
# print(c1)  # => 278033
# print(tuple(p2))  # => (0.5062278801751902, 0.5080682368868706, 0.5)
# print(tuple(g1))  # => (31, 25, 29)
```

```
278033
(0.5062278801751902, 0.5080682368868706, 0.5)
(31, 25, 29)
```

With these coordinates, you can handle a `Voxel`, which represents a `Particle` object. Instead of `new_particle`, `new_voxel` provides the way to create a new `Voxel` with a coordinate.

[28]:
```
c1 = w.position2coordinate(Real3(0.5, 0.5, 0.5))
((pid, v), is_succeeded) = w.new_voxel(Species("A"), c1)
print(pid, v, is_succeeded)
```

```
(<ecell4.core.ParticleID object at 0x7fdfd28e8a38>, <ecell4.core.Voxel object at␣
↪0x7fdfd28e8c18>, True)
```

A `Voxel` consists of `species`, `coordinate`, `radius` and `D`.

[29]:
```
print(v.species().serial(), v.coordinate(), v.radius(), v.D())  # => (u'A', 278033, 0.
↪0025, 1.0)
```

```
(u'A', 278033, 0.0025, 1.0)
```

Of course, you can get a voxel and list voxels with `get_voxel` and `list_voxels_exact` similar to `get_particle` and `list_particles_exact`.

[30]:
```
print(w.num_voxels_exact(Species("A")))
print(w.list_voxels_exact(Species("A")))
print(w.get_voxel(pid))
```

```
1
[(<ecell4.core.ParticleID object at 0x7fdfd28e8ae0>, <ecell4.core.Voxel object at␣
↪0x7fdfd28e8c30>)]
(<ecell4.core.ParticleID object at 0x7fdfd28e8ae0>, <ecell4.core.Voxel object at␣
↪0x7fdfd28e8bd0>)
```

You can move and update the voxel with `update_voxel` corresponding to `update_particle`.

[31]:
```
c2 = w.position2coordinate(Real3(0.5, 0.5, 1.0))
w.update_voxel(pid, Voxel(v.species(), c2, v.radius(), v.D()))
pid, newv = w.get_voxel(pid)
print(c2)  # => 278058
print(newv.species().serial(), newv.coordinate(), newv.radius(), newv.D())  # => (u'A
↪', 278058, 0.0025, 1.0)
print(w.num_voxels_exact(Species("A")))  # => 1
```

```
278058
(u'A', 278058, 0.0025, 1.0)
1
```

Finally, `remove_voxel` remove a voxel as `remove_particle` does.

[32]:
```
print(w.has_voxel(pid))  # => True
w.remove_voxel(pid)
print(w.has_voxel(pid))  # => False
```

```
True
False
```

1.3.4 3.4 Structure

[33]:
```
w1 = GillespieWorld()
w2 = ODEWorld()
w3 = SpatiocyteWorld()
w4 = BDWorld()
w5 = MesoscopicWorld()
w6 = EGFRDWorld()
```

By using a `Shape` object, you can confine initial positions of molecules to a part of `World`. In the case below, 60 molecules are positioned inside the given `Sphere`. Diffusion of the molecules placed here is **NOT** restricted in the `Shape`. This `Shape` is only for the initialization.

[34]:
```
sp1 = Species("A", "0.0025", "1")
sphere = Sphere(Real3(0.5, 0.5, 0.5), 0.3)
w1.add_molecules(sp1, 60, sphere)
w2.add_molecules(sp1, 60, sphere)
w3.add_molecules(sp1, 60, sphere)
w4.add_molecules(sp1, 60, sphere)
w5.add_molecules(sp1, 60, sphere)
w6.add_molecules(sp1, 60, sphere)
```

A property of `Species`, `'location'`, is available to restrict diffusion of molecules. `'location'` is not fully supported yet, but only supported in `spatiocyte` and `meso`. `add_structure` defines a new structure given as a pair of `Species` and `Shape`.

[35]:
```
membrane = SphericalSurface(Real3(0.5, 0.5, 0.5), 0.4)  # This is equivalent to call␣
↪`Sphere(Real3(0.5, 0.5, 0.5), 0.4).surface()`
w3.add_structure(Species("M"), membrane)
w5.add_structure(Species("M"), membrane)
```

After defining a structure, you can bind molecules to the structure as follows:

[36]:
```
sp2 = Species("B", "0.0025", "0.1", "M")  # `'location'` is the fourth argument
w3.add_molecules(sp2, 60)
w5.add_molecules(sp2, 60)
```

The molecules bound to a `Species` named `B` diffuse on a structure named `M`, which has a shape of `SphericalSurface` (a hollow sphere). In `spatiocyte`, a structure is represented as a set of particles with `Species` `M` occupying a voxel. It means that molecules not belonging to the structure is not able to overlap the voxel and it causes a collision. On the other hand, in `meso`, a structure means a list of subvolumes. Thus, a structure doesn't avoid an incursion of other particles.

1.3.5 3.5. Random Number Generator

A random number generator is also a part of `World`. All `World` except `ODEWorld` store a random number generator, and updates it when the simulation needs a random value. On E-Cell4, only one class `GSLRandomNumberGenerator` is implemented as a random number generator.

[37]:
```
rng1 = GSLRandomNumberGenerator()
print([rng1.uniform_int(1, 6) for _ in range(20)])
```

```
[6, 1, 2, 6, 2, 3, 6, 5, 4, 5, 5, 4, 2, 5, 4, 2, 3, 3, 2, 2]
```

With no argument, the random number generator is always initialized with a seed, `0`.

[38]:
```
rng2 = GSLRandomNumberGenerator()
print([rng2.uniform_int(1, 6) for _ in range(20)])  # => same as above
```

```
[6, 1, 2, 6, 2, 3, 6, 5, 4, 5, 5, 4, 2, 5, 4, 2, 3, 3, 2, 2]
```

You can initialize the seed with an integer as follows:

[39]:
```
rng2 = GSLRandomNumberGenerator()
rng2.seed(15)
print([rng2.uniform_int(1, 6) for _ in range(20)])
```

```
[6, 5, 2, 4, 1, 1, 3, 5, 2, 6, 4, 1, 2, 5, 2, 5, 1, 2, 2, 6]
```

When you call the `seed` function with no input, the seed is drawn from the current time.

[40]:
```
rng2 = GSLRandomNumberGenerator()
rng2.seed()
print([rng2.uniform_int(1, 6) for _ in range(20)])
```

```
[6, 2, 1, 5, 6, 2, 6, 2, 6, 6, 5, 6, 2, 3, 5, 3, 6, 1, 4, 3]
```

`GSLRandomNumberGenerator` provides several ways to get a random number.

[41]:
```
print(rng1.uniform(0.0, 1.0))
print(rng1.uniform_int(0, 100))
print(rng1.gaussian(1.0))
```

```
0.0303352042101
33
0.893555545521
```

`World` accepts a random number generator at the construction. As a default, `GSLRandomNumberGenerator()` is used. Thus, when you don't give a generator, behavior of the simulation is always same (determinisitc).

[42]:
```
rng = GSLRandomNumberGenerator()
rng.seed()
w1 = GillespieWorld(Real3(1, 1, 1), rng=rng)
```

You can access the `GSLRandomNumberGenerator` in a `World` through `rng` function.

[43]:
```
print(w1.rng().uniform(0.0, 1.0))
```

```
0.966634188779
```

`rng()` returns a shared pointer to the `GSLRandomNumberGenerator`. Thus, in the example above, `rng` and `w1.rng()` point exactly the same thing.

1.4 4. How to Run a Simulation

In sections 2 and 3, we explained the way to build a model and to setup the intial state. Now, it is the time to run a simulation. Corresponding to `World` classes, six `Simulator` classes are there: `spatiocyte.SpatiocyteSimulator`, `egfrd.EGFRDSimulator`, `bd.BDSimulator`, `meso.MesoscopicSimulator`, `gillespie.GillespieSimulator`, and `ode.ODESimulator`. Each `Simulator` class only accepts the corresponding type of `World`, but all of them allow the same `Model`.

[1]:
```
import ecell4
```

1.4.1 4.1. How to Setup a Simulator

Except for the initialization (so-called constructor function) with arguments specific to the algorithm, all `Simulators` have the same APIs.

[2]:
```
from ecell4.core import *
from ecell4.gillespie import GillespieWorld, GillespieSimulator
from ecell4.ode import ODEWorld, ODESimulator
from ecell4.spatiocyte import SpatiocyteWorld, SpatiocyteSimulator
from ecell4.bd import BDWorld, BDSimulator
from ecell4.meso import MesoscopicWorld, MesoscopicSimulator
from ecell4.egfrd import EGFRDWorld, EGFRDSimulator
```

Before constructing a `Simulator`, parepare a `Model` and a `World` corresponding to the type of `Simulator`.

[3]:
```
from ecell4 import species_attributes, reaction_rules, get_model

with species_attributes():
    A | B | C | {'D': '1', 'radius': '0.005'}

with reaction_rules():
    A + B == C | (0.01, 0.3)

m = get_model()
```

[4]:
```
w1 = GillespieWorld()
w2 = ODEWorld()
w3 = SpatiocyteWorld()
w4 = BDWorld()
```

(continues on next page)

(continued from previous page)

```
w5 = MesoscopicWorld()
w6 = EGFRDWorld()
```

`Simulator` requires both `Model` and `World` in this order at the construction.

[5]:
```
sim1 = GillespieSimulator(m, w1)
sim2 = ODESimulator(m, w2)
sim3 = SpatiocyteSimulator(m, w3)
sim4 = BDSimulator(m, w4)
sim5 = MesoscopicSimulator(m, w5)
sim6 = EGFRDSimulator(m, w6)
```

If you bind the `Model` to the `World`, you need only the `World` to create a `Simulator`.

[6]:
```
w1.bind_to(m)
w2.bind_to(m)
w3.bind_to(m)
w4.bind_to(m)
w5.bind_to(m)
w6.bind_to(m)
```

[7]:
```
sim1 = GillespieSimulator(w1)
sim2 = ODESimulator(w2)
sim3 = SpatiocyteSimulator(w3)
sim4 = BDSimulator(w4)
sim5 = MesoscopicSimulator(w5)
sim6 = EGFRDSimulator(w6)
```

Of course, the `Model` and `World` bound to a `Simulator` can be drawn from `Simulator` in the way below:

[8]:
```
print(sim1.model(), sim1.world())
print(sim2.model(), sim2.world())
print(sim3.model(), sim3.world())
print(sim4.model(), sim4.world())
print(sim5.model(), sim5.world())
print(sim6.model(), sim6.world())
```

```
(<ecell4.core.Model object at 0x7f6b329d1af8>, <ecell4.gillespie.GillespieWorld␣
↪object at 0x7f6b329d1a08>)
(<ecell4.ode.ODENetworkModel object at 0x7f6b329d1af8>, <ecell4.ode.ODEWorld object␣
↪at 0x7f6b329d1ae0>)
(<ecell4.core.Model object at 0x7f6b329d1af8>, <ecell4.spatiocyte.SpatiocyteWorld␣
↪object at 0x7f6b329d1a08>)
(<ecell4.core.Model object at 0x7f6b329d1af8>, <ecell4.bd.BDWorld object at␣
↪0x7f6b329d1ae0>)
(<ecell4.core.Model object at 0x7f6b329d1af8>, <ecell4.meso.MesoscopicWorld object at␣
↪0x7f6b329d1ac8>)
(<ecell4.core.Model object at 0x7f6b329d1af8>, <ecell4.egfrd.EGFRDWorld object at␣
↪0x7f6b329d1ae0>)
```

After updating the `World` by yourself, you must initialize the internal state of a `Simulator` before running simulation.

[9]:
```
w1.add_molecules(Species('C'), 60)
w2.add_molecules(Species('C'), 60)
w3.add_molecules(Species('C'), 60)
```

(continues on next page)

(continued from previous page)

```
w4.add_molecules(Species('C'), 60)
w5.add_molecules(Species('C'), 60)
w6.add_molecules(Species('C'), 60)
```

[10]:
```
sim1.initialize()
sim2.initialize()
sim3.initialize()
sim4.initialize()
sim5.initialize()
sim6.initialize()
```

For algorithms with a fixed step interval, the `Simulator` also requires `dt`.

[11]:
```
sim2.set_dt(1e-6)  # ODESimulator. This is optional
sim4.set_dt(1e-6)  # BDSimulator
```

1.4.2 4.2. Running Simulation

For running simulation, `Simulator` provides two APIs, `step` and `run`.

`step()` advances a simulation for the time that the `Simulator` expects, `next_time()`.

[12]:
```
print(sim1.t(), sim1.next_time(), sim1.dt())
print(sim2.t(), sim2.next_time(), sim2.dt())  # => (0.0, 1e-6, 1e-6)
print(sim3.t(), sim3.next_time(), sim3.dt())
print(sim4.t(), sim4.next_time(), sim4.dt())  # => (0.0, 1e-6, 1e-6)
print(sim5.t(), sim5.next_time(), sim5.dt())
print(sim6.t(), sim6.next_time(), sim6.dt())  # => (0.0, 0.0, 0.0)
```

```
(0.0, 0.032171880483933143, 0.032171880483933143)
(0.0, 1e-06, 1e-06)
(0.0, 1.6666666666666667e-05, 1.6666666666666667e-05)
(0.0, 1e-06, 1e-06)
(0.0, 0.029999537310460653, 0.029999537310460653)
(0.0, 0.0, 0.0)
```

[13]:
```
sim1.step()
sim2.step()
sim3.step()
sim4.step()
sim5.step()
sim6.step()
```

[14]:
```
print(sim1.t())
print(sim2.t())  # => 1e-6
print(sim3.t())
print(sim4.t())  # => 1e-6
print(sim5.t())
print(sim6.t())  # => 0.0
```

```
0.0321718804839
1e-06
1.66666666667e-05
1e-06
0.0299995373105
0.0
```

`last_reactions()` returns a list of pairs of `ReactionRule` and `ReactionInfo` which occurs at the last step. Each algorithm have its own implementation of `ReactionInfo`. See `help(module.ReactionInfo)` for details.

[15]:
```
print(sim1.last_reactions())
# print(sim2.last_reactions())
print(sim3.last_reactions())
print(sim4.last_reactions())
print(sim5.last_reactions())
print(sim6.last_reactions())
```

```
[(<ecell4.core.ReactionRule object at 0x7f6b329d1ae0>, <ecell4.gillespie.ReactionInfo␣
↪object at 0x7f6b329d1af8>)]
[]
[]
[(<ecell4.core.ReactionRule object at 0x7f6b329d1ae0>, <ecell4.meso.ReactionInfo␣
↪object at 0x7f6b329d1af8>)]
[]
```

`step(upto)` advances a simulation for `next_time` if `next_time` is less than `upto`, or for `upto` otherwise. `step(upto)` returns whether the time does **NOT** reach the limit, `upto`.

[16]:
```
print(sim1.step(1.0), sim1.t())
print(sim2.step(1.0), sim2.t())
print(sim3.step(1.0), sim3.t())
print(sim4.step(1.0), sim4.t())
print(sim5.step(1.0), sim5.t())
print(sim6.step(1.0), sim6.t())
```

```
(True, 0.11785771174833615)
(True, 2e-06)
(True, 3.3333333333333335e-05)
(True, 2e-06)
(True, 0.0709074540101994)
(True, 0.0)
```

To run a simulation just until the time, `upto`, call `step(upto)` while it returns `True`.

[17]:
```
while sim1.step(1.0): pass
while sim2.step(0.001): pass
while sim3.step(0.001): pass
while sim4.step(0.001): pass
while sim5.step(1.0): pass
while sim6.step(0.001): pass
```

[18]:
```
print(sim1.t())  # => 1.0
print(sim2.t())  # => 0.001
print(sim3.t())  # => 0.001
print(sim4.t())  # => 0.001
print(sim5.t())  # => 1.0
print(sim6.t())  # => 0.001
```

```
1.0
0.001
0.001
0.001
1.0
0.001
```

This is just what `run` does. `run(tau)` advances a simulation upto `t()+tau`.

[19]:
```
sim1.run(1.0)
sim2.run(0.001)
sim3.run(0.001)
sim4.run(0.001)
sim5.run(1.0)
sim6.run(0.001)
```

[20]:
```
print(sim1.t())  # => 2.0
print(sim2.t())  # => 0.002
print(sim3.t())  # => 0.002
print(sim4.t())  # => 0.002
print(sim5.t())  # => 2.0
print(sim6.t())  # => 0.02
```

```
2.0
0.002
0.002
0.002
2.0
0.002
```

`num_steps` returns the number of steps during the simulation.

[21]:
```
print(sim1.num_steps())
print(sim2.num_steps())
print(sim3.num_steps())
print(sim4.num_steps())
print(sim5.num_steps())
print(sim6.num_steps())
```

```
37
2001
120
2001
34
581
```

1.4.3 4.3. Capsulizing Algorithm into a Factory Class

Owing to the portability of a `Model` and consistent APIs of `Worlds` and `Simulators`, it is very easy to write a script common in algorithms. However, when switching the algorithm, still we have to rewrite the name of classes in the code, one by one.

To avoid the trouble, E-Cell4 also provides a `Factory` class for each algorithm. `Factory` encapsulates `World` and `Simulator` with their arguments needed for the construction. By using `Factory` class, your script could be portable and robust agaist changes in the algorithm.

[22]:
```
from ecell4.gillespie import GillespieFactory
from ecell4.ode import ODEFactory
from ecell4.spatiocyte import SpatiocyteFactory
from ecell4.bd import BDFactory
from ecell4.meso import MesoscopicFactory
from ecell4.egfrd import EGFRDFactory
```

`Factory` just provides two functions, `create_world` and `create_simulator`.

[23]:
```
def singlerun(f, m):
    w = f.create_world(Real3(1, 1, 1))
    w.bind_to(m)
    w.add_molecules(Species('C'), 60)

    sim = f.create_simulator(w)
    sim.run(0.01)
    print(sim.t(), w.num_molecules(Species('C')))
```

`singlerun` above is free from the algorithm. Thus, by just switching `Factory`, you can easily compare the results.

[24]:
```
singlerun(GillespieFactory(), m)
singlerun(ODEFactory(), m)
singlerun(SpatiocyteFactory(), m)
singlerun(BDFactory(bd_dt_factor=1), m)
singlerun(MesoscopicFactory(), m)
singlerun(EGFRDFactory(), m)
```

```
(0.01, 60)
(0.01, 59)
(0.01, 60)
(0.01, 60)
(0.01, 60)
(0.01, 60)
```

When you need to provide several parameters to initialize `World` or `Simulator`, `run_simulation` also accepts `Factory` instead of `solver`.

[25]:
```
from ecell4.util import run_simulation
print(run_simulation(0.01, model=m, y0={'C': 60}, return_type='array',␣
↪solver=GillespieFactory())[-1])
print(run_simulation(0.01, model=m, y0={'C': 60}, return_type='array',␣
↪solver=ODEFactory())[-1])
print(run_simulation(0.01, model=m, y0={'C': 60}, return_type='array',␣
↪solver=SpatiocyteFactory())[-1])
print(run_simulation(0.01, model=m, y0={'C': 60}, return_type='array',␣
↪solver=BDFactory(bd_dt_factor=1))[-1])
print(run_simulation(0.01, model=m, y0={'C': 60}, return_type='array',␣
↪solver=MesoscopicFactory())[-1])
print(run_simulation(0.01, model=m, y0={'C': 60}, return_type='array',␣
↪solver=EGFRDFactory())[-1])
```

```
[0.01, 0.0, 0.0, 60.0]
[0.01, 0.17972919304001073, 0.17972919304001067, 59.82027080696036]
[0.01, 0.0, 0.0, 60.0]
[0.01, 0.0, 0.0, 60.0]
[0.01, 0.0, 0.0, 60.0]
[0.01, 0.0, 0.0, 60.0]
```

1.5 5. How to Log and Visualize Simulations

Here we explain how to take a log of simulation results and how to visualize it.

[1]:
```
%matplotlib inline
import math
from ecell4 import *
```

1.5.1 5.1. Logging Simulations with Observers

E-Cell4 provides special classes for logging, named `Observer`. `Observer` class is given when you call the `run` function of `Simulator`.

[2]:
```
def create_simulator(f=gillespie.GillespieFactory()):
    m = NetworkModel()
    A, B, C = Species('A', '0.005', '1'), Species('B', '0.005', '1'), Species('C', '0.
↪005', '1')
    m.add_reaction_rule(create_binding_reaction_rule(A, B, C, 0.01))
    m.add_reaction_rule(create_unbinding_reaction_rule(C, A, B, 0.3))
    w = f.create_world()
    w.bind_to(m)
    w.add_molecules(C, 60)
    sim = f.create_simulator(w)
    sim.initialize()
    return sim
```

One of most popular `Observer` is `FixedIntervalNumberObserver`, which logs the number of molecules with the given time interval. `FixedIntervalNumberObserver` requires an interval and a list of serials of `Species` for logging.

[3]:
```
obs1 = FixedIntervalNumberObserver(0.1, ['A', 'B', 'C'])
sim = create_simulator()
sim.run(1.0, obs1)
```

`data` function of `FixedIntervalNumberObserver` returns the data logged.

[4]:
```
print(obs1.data())
```

```
[[0.0, 0.0, 0.0, 60.0], [0.1, 0.0, 0.0, 60.0], [0.2, 0.0, 0.0, 60.0], [0.
↪30000000000000004, 5.0, 5.0, 55.0], [0.4, 8.0, 8.0, 52.0], [0.5, 8.0, 8.0, 52.0],␣
↪[0.6000000000000001, 8.0, 8.0, 52.0], [0.7000000000000001, 11.0, 11.0, 49.0], [0.8,␣
↪14.0, 14.0, 46.0], [0.9, 14.0, 14.0, 46.0], [1.0, 15.0, 15.0, 45.0]]
```

`targets()` returns a list of `Species`, which you specified as an argument of the constructor.

[5]:
```
print([sp.serial() for sp in obs1.targets()])
```

```
['A', 'B', 'C']
```

`NumberObserver` logs the number of molecules after every steps when a reaction occurs. This observer is useful to log all reactions, but not available for `ode`.

[6]:
```
obs1 = NumberObserver(['A', 'B', 'C'])
sim = create_simulator()
sim.run(1.0, obs1)
print(obs1.data())
```

```
[[0.0, 0.0, 0.0, 60.0], [0.143979748309557, 1.0, 1.0, 59.0], [0.263959301731782, 2.0,␣
↪2.0, 58.0], [0.28549985484415447, 3.0, 3.0, 57.0], [0.2920290844608921, 4.0, 4.0,␣
↪56.0], [0.3151450201904092, 5.0, 5.0, 55.0], [0.32962099109027987, 6.0, 6.0, 54.0],␣
↪[0.33943151256600035, 7.0, 7.0, 53.0], [0.38459684239249475, 8.0, 8.0, 52.0], [0.
↪4402392101074317, 9.0, 9.0, 51.0], [0.4488367232148584, 10.0, 10.0, 50.0], [0.
↪4724492407095052, 11.0, 11.0, 49.0], [0.48969169545303415, 12.0, 12.0, 48.0], [0.
↪5926648902430811, 13.0, 13.0, 47.0], [0.7112014766673606, 14.0, 14.0, 46.0], [0.
↪7268968273742534, 15.0, 15.0, 45.0], [0.7889733307696482, 16.0, 16.0, 44.0], [0.
↪7938779425321273, 17.0, 17.0, 43.0], [0.9703902424683863, 18.0, 18.0, 42.0], [0.
↪9712948910756202, 17.0, 17.0, 43.0], [1.0, 17.0, 17.0, 43.0]]
```

`TimingNumberObserver` allows you to give the times for logging as an argument of its constructor.

[7]:
```
obs1 = TimingNumberObserver([0.0, 0.1, 0.2, 0.5, 1.0], ['A', 'B', 'C'])
sim = create_simulator()
sim.run(1.0, obs1)
print(obs1.data())
```

```
[[0.0, 0.0, 0.0, 60.0], [0.1, 0.0, 0.0, 60.0], [0.2, 0.0, 0.0, 60.0], [0.5, 10.0, 10.
↪0, 50.0], [1.0, 16.0, 16.0, 44.0]]
```

`run` function accepts multile `Observers` at once.

[8]:
```
obs1 = NumberObserver(['C'])
obs2 = FixedIntervalNumberObserver(0.1, ['A', 'B'])
sim = create_simulator()
sim.run(1.0, [obs1, obs2])
print(obs1.data())
print(obs2.data())
```

```
[[0.0, 60.0], [0.22087040257109863, 59.0], [0.22720792132387785, 58.0], [0.
↪24968763863882104, 57.0], [0.2637930785790673, 56.0], [0.27337221983130383, 55.0],␣
↪[0.35394818219048224, 54.0], [0.36231206722433756, 53.0], [0.38536272493723256, 52.
↪0], [0.6152314487155562, 51.0], [0.6759207676775343, 50.0], [0.6807518102635762, 49.
↪0], [0.7009006683808036, 48.0], [0.7485557651303876, 47.0], [0.7530285146153741, 46.
↪0], [0.9214853607237395, 45.0], [1.0, 45.0]]
[[0.0, 0.0, 0.0], [0.1, 0.0, 0.0], [0.2, 0.0, 0.0], [0.30000000000000004, 5.0, 5.0],␣
↪[0.4, 8.0, 8.0], [0.5, 8.0, 8.0], [0.6000000000000001, 8.0, 8.0], [0.
↪7000000000000001, 11.0, 11.0], [0.8, 14.0, 14.0], [0.9, 14.0, 14.0], [1.0, 15.0, 15.
↪0]]
```

`FixedIntervalHDF5Observedr` logs the whole data in a `World` to an output file with the fixed interval. Its second argument is a prefix for output filenames. `filename()` returns the name of a file scheduled to be saved next. At most one format string like `%02d` is allowed to use a step count in the file name. When you do not use the format string, it overwrites the latest data to the file.

[9]:
```
obs1 = FixedIntervalHDF5Observer(0.2, 'test%02d.h5')
print(obs1.filename())
sim = create_simulator()
sim.run(1.0, obs1) # Now you have steped 5 (1.0/0.2) times
print(obs1.filename())
```

```
test00.h5
test06.h5
```

[10]:
```
w = load_world('test05.h5')
print(w.t(), w.num_molecules(Species('C')))
```

```
1.0 44
```

The usage of `FixedIntervalCSVObserver` is almost same with that of `FixedIntervalHDF5Observer`. It saves positions (x, y, z) of particles with the radius (r) and serial number of `Species` (sid) to a CSV file.

[11]:
```
obs1 = FixedIntervalCSVObserver(0.2, "test%02d.csv")
print(obs1.filename())
sim = create_simulator()
sim.run(1.0, obs1)
print(obs1.filename())
```

```
test00.csv
test06.csv
```

Here is the first 10 lines in the output CSV file.

[12]:
```
print(''.join(open("test05.csv").readlines()[: 10]))
```

```
x,y,z,r,sid
0.17508278810419142,0.12197625380940735,0.53627523058094084,0,0
0.24078186159022152,0.62765785818919539,0.7254820850212127,0,0
0.40445487247779965,0.16720660566352308,0.92957371729426086,0,0
0.24083335651084781,0.59580284473486245,0.51183571317233145,0,0
0.90290508698672056,0.4137285016477108,0.39729581261053681,0,0
0.51044316007755697,0.26319809840060771,0.40705726365558803,0,0
0.84177179471589625,0.085374288959428668,0.64473599148914218,0,0
0.54177056765183806,0.098819603677839041,0.17240164172835648,0,0
0.28669063560664654,0.71829434810206294,0.91102162329480052,0,0
```

For particle simulations, E-Cell4 also provides `Observer` to trace a trajectory of a molecule, named `FixedIntervalTrajectoryObserver`. When no `ParticleID` is specified, it logs all the trajectories. Once some `ParticleID` is lost for the reaction during a simulation, it just stop to trace the particle any more.

[13]:
```
sim = create_simulator(spatiocyte.SpatiocyteFactory(0.005))
obs1 = FixedIntervalTrajectoryObserver(0.01)
sim.run(0.1, obs1)
```

[14]:
```
print([tuple(pos) for pos in obs1.data()[0]])
```

```
[(0.46540305112880387, 0.739008344562721, 0.48), (0.2122891110412088, 0.
↪9266471820493494, 0.665), (0.49806291436591293, 0.8689121551303868, 0.685), (0.
↪7838367176906171, 0.8313843876330611, 0.64), (0.9961258287318259, 0.
↪7967433714816836, 0.5), (0.7920016834998943, 0.9122134253196088, 0.63), (0.
↪7348469228349536, 0.7534421012924616, 0.515), (0.9716309313039941, 0.
↪851591647054698, 0.525), (1.1349302474895393, 0.9122134253196088, 0.63), (0.
↪9961258287318259, 0.9439676901250381, 0.725), (0.718516991216399, 0.
↪8833459118601275, 0.93)]
```

Generally, `World` assumes a periodic boundary for each plane. To avoid the big jump of a particle at the edge due to the boundary condition, `FixedIntervalTrajectoryObserver` tries to keep the shift of positions. Thus, the positions stored in the `Observer` are not necessarily limited in the cuboid given for the `World`. To track the diffusion over the boundary condition accurately, the step interval for logging must be small enough. Of course, you can disable this option. See `help(FixedIntervalTrajectoryObserver)`.

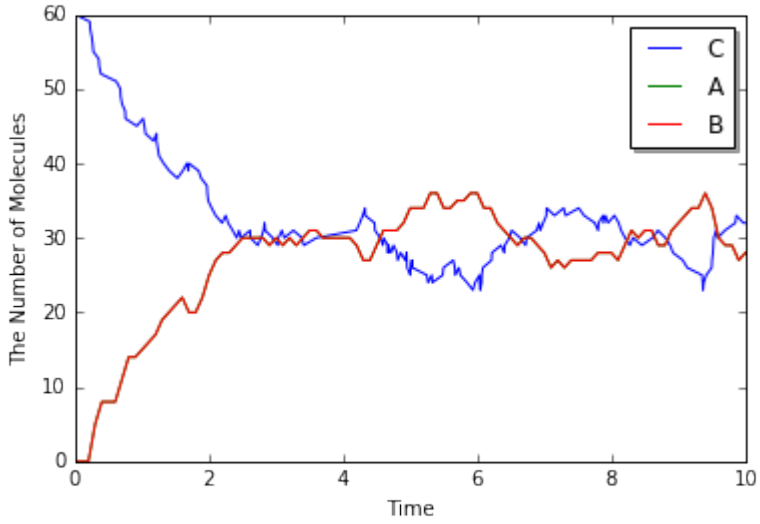
1.5.2 5.2. Visualization of Data Logged

In this section, we explain the visualization tools for data logged by `Observer`.

Firstly, for time course data, `viz.plot_number_observer` plots the data provided by `NumberObserver`, `FixedIntervalNumberObserver` and `TimingNumberObserver`. For the detailed usage of `viz.plot_number_observer`, see `help(viz.plot_number_observer)`.

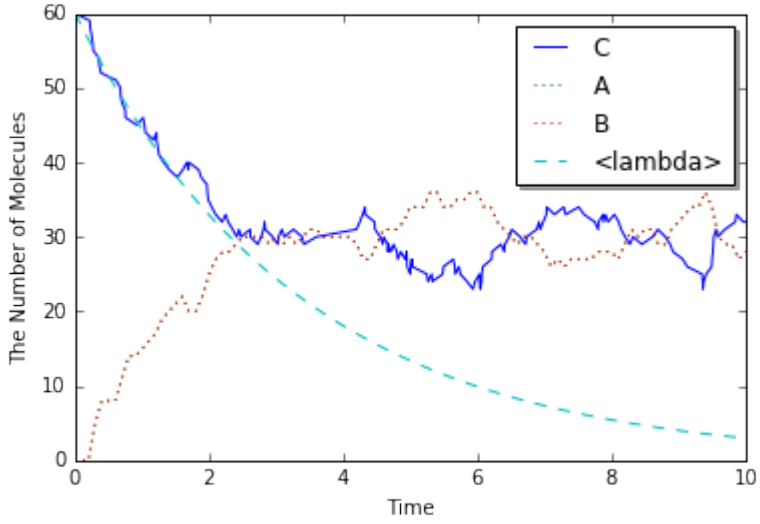
[15]:
```
obs1 = NumberObserver(['C'])
obs2 = FixedIntervalNumberObserver(0.1, ['A', 'B'])
sim = create_simulator()
sim.run(10.0, [obs1, obs2])
```

[16]:
```
viz.plot_number_observer(obs1, obs2)
```



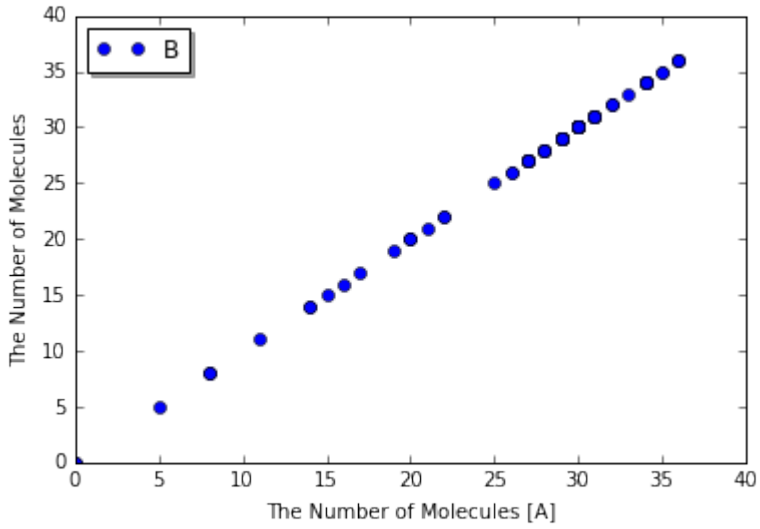
You can set the style for plotting, and even add an arbitrary function to plot.

[17]:
```
viz.plot_number_observer(obs1, '-', obs2, ':', lambda t: 60 * math.exp(-0.3 * t), '--
↪')
```



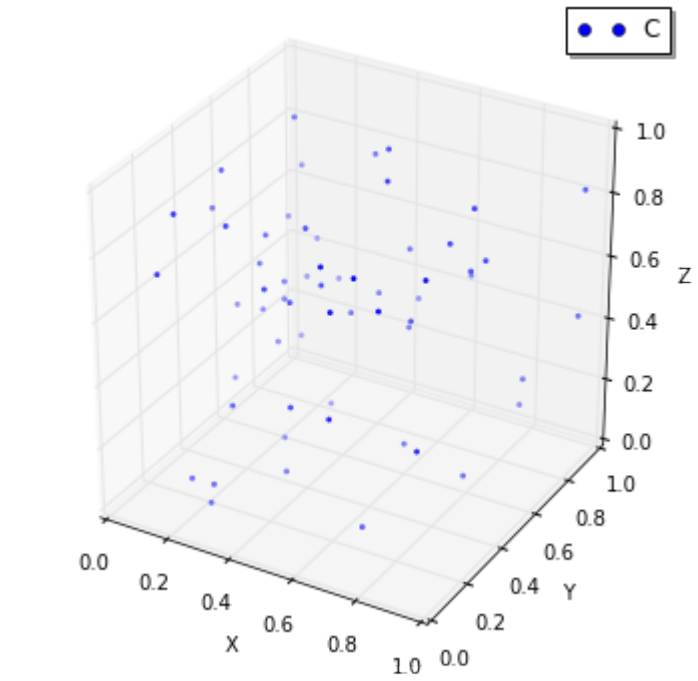
Plotting in the phase plane is also available by specifing the x-axis and y-axis.

[18]:
```
viz.plot_number_observer(obs2, 'o', x='A', y='B')
```



For spatial simulations, to visualize the state of `World`, `viz.plot_world` is available. This function plots the points of particles in three-dimensional volume in the interactive way. You can save the image by clicking a right button on the drawing region.

[19]:
```
sim = create_simulator(spatiocyte.SpatiocyteFactory(0.005))
# viz.plot_world(sim.world())
viz.plot_world(sim.world(), interactive=False)
```



You can also make a movie from a series of HDF5 files, given as a `FixedIntervalHDF5Observer`. NOTE: `viz.plot_movie` requires an extra library, `ffmpeg`, when `interactive=False`.

[20]:
```
sim = create_simulator(spatiocyte.SpatiocyteFactory(0.005))
obs1 = FixedIntervalHDF5Observer(0.02, 'test%02d.h5')
```

(continues on next page)

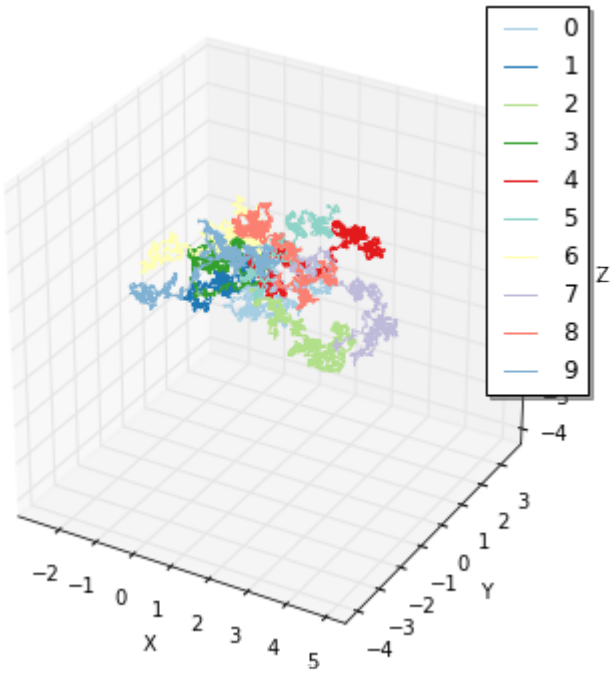
(continued from previous page)

```
sim.run(1.0, obs1)
viz.plot_movie(obs1)
```

```
<IPython.core.display.HTML object>
```

Finally, corresponding to `FixedIntervalTrajectoryObserver`, `viz.plot_trajectory` provides a visualization of particle trajectories.

[21]:
```
sim = create_simulator(spatiocyte.SpatiocyteFactory(0.005))
obs1 = FixedIntervalTrajectoryObserver(1e-3)
sim.run(1, obs1)
# viz.plot_trajectory(obs1)
viz.plot_trajectory(obs1, interactive=False)
```



1.6 6. How to Solve ODEs with Rate Law Functions

Although the `ode` solver accepts `NetworkModel`, `ode.ODESimulator` owns its model class `ode.ODENetworkModel` and `ode.ODEReactionRule` for the extension. The interface of these classes are almost same with `Model` classes and `ReactionRule`. Here, we explain the usage specific to `ode` especially about `ode.ODERatelaw`.

[1]:
```
%matplotlib inline
from ecell4 import *
```

However, the rate law support in `ode` is under development. Some functions might be deprecated in the future. Currently, to enable rate laws, the option `ecell4.util.decorator.ENABLE_RATELAW` must be activated as follows:

[2]:
```
util.decorator.ENABLE_RATELAW = True
```

1.6.1 6.1. ode.ODEReactionRule

`ode.ODEReactionRule` has almost same members with `ReactionRule`.

[3]:
```
rr1 = ReactionRule()
rr1.add_reactant(Species("A"))
rr1.add_reactant(Species("B"))
rr1.add_product(Species("C"))
rr1.set_k(1.0)
print(len(rr1.reactants()))  # => 2
print(len(rr1.products()))  # => 1
print(rr1.k())  # => 1.0
print(rr1.as_string())  # => A+B>C|1
```

```
2
1
1.0
A+B>C|1
```

[4]:
```
rr2 = ode.ODEReactionRule()
rr2.add_reactant(Species("A"))
rr2.add_reactant(Species("B"))
rr2.add_product(Species("C"))
rr2.set_k(1.0)
print(len(rr2.reactants()))  # => 2
print(len(rr2.products()))  # => 1
print(rr2.k())  # => 1.0
print(rr2.as_string())  # => A+B>C|1
```

```
2
1
1.0
A+B>C|1
```

In addition to the common members, `ode.ODEReactionRule` can store stoichiometric coefficients for each `Species`:

[5]:
```
rr2 = ode.ODEReactionRule()
rr2.add_reactant(Species("A"), 1.0)
rr2.add_reactant(Species("B"), 2.0)
rr2.add_product(Species("C"), 2.5)
rr2.set_k(1.0)
print(rr2.as_string())
```

```
A+2*B>2.5*C|1
```

You can also access to the coefficients as follows:

[6]:
```
print(rr2.reactants_coefficients())  # => [1.0, 2.0]
print(rr2.products_coefficients())  # => [2.5]
```

```
[1.0, 2.0]
[2.5]
```

1.6.2 6.2. ode.ODERatelaw

`ode.ODEReactionRule` can be bound to a `ode.ODERatelaw`. `ode.ODERatelaw` provides a function to calculate a derivative (flux or velocity) based on the given values of `Species`. `ode.ODERatelawMassAction`

is a default class bound to `ode.ODEReactionRule`.

[7]:
```
rr1 = ode.ODEReactionRule()
rr1.add_reactant(Species("A"))
rr1.add_reactant(Species("B"))
rr1.add_product(Species("C"))
rl1 = ode.ODERatelawMassAction(2.0)
rr1.set_ratelaw(rl1)  # equivalent to rr1.set_k(2.0)
print(rr1.as_string())
```

```
A+B>C|2
```

`ode.ODERatelawCallback` enables the user-defined function for calculating a flux.

[8]:
```
def mass_action(reactants, products, volume, t, rr):
    veloc = 2.0 * volume
    for value in reactants:
        veloc *= value / volume
    return veloc

rl2 = ode.ODERatelawCallback(mass_action)
rr1.set_ratelaw(rl2)
print(rr1.as_string())
```

```
A+B>C|mass_action
```

The function bound must accept five arguments and return a floating number as a velocity. The first and second list contain a value for each reactants and products respectively. When you need to access the stoichiometric coefficients, use `rr` (`ode.ODEReactionRule`) in the arguments.

A `lambda` function is available too.

[9]:
```
rl2 = ode.ODERatelawCallback(lambda r, p, v, t, rr: 2.0 * r[0] * r[1])
rr1.set_k(0)
rr1.set_ratelaw(rl2)
print(rr1.as_string())
```

```
A+B>C|<lambda>
```

1.6.3 6.3. ode.ODENetworkModel

`ode.ODENetworkModel` accepts both `ReactionRule` and `ode.ODEReactionRule`. `ReactionRule` is implicitly converted and stored as a `ode.ODEReactionRule`.

[10]:
```
m1 = ode.ODENetworkModel()
rr1 = create_unbinding_reaction_rule(Species("C"), Species("A"), Species("B"), 3.0)
m1.add_reaction_rule(rr1)
rr2 = ode.ODEReactionRule(create_binding_reaction_rule(Species("A"), Species("B"),␣
↪Species("C"), 0.0))
rr2.set_ratelaw(ode.ODERatelawCallback(lambda r, p, v, t, rr: 0.1 * r[0] * r[1]))
m1.add_reaction_rule(rr2)
```

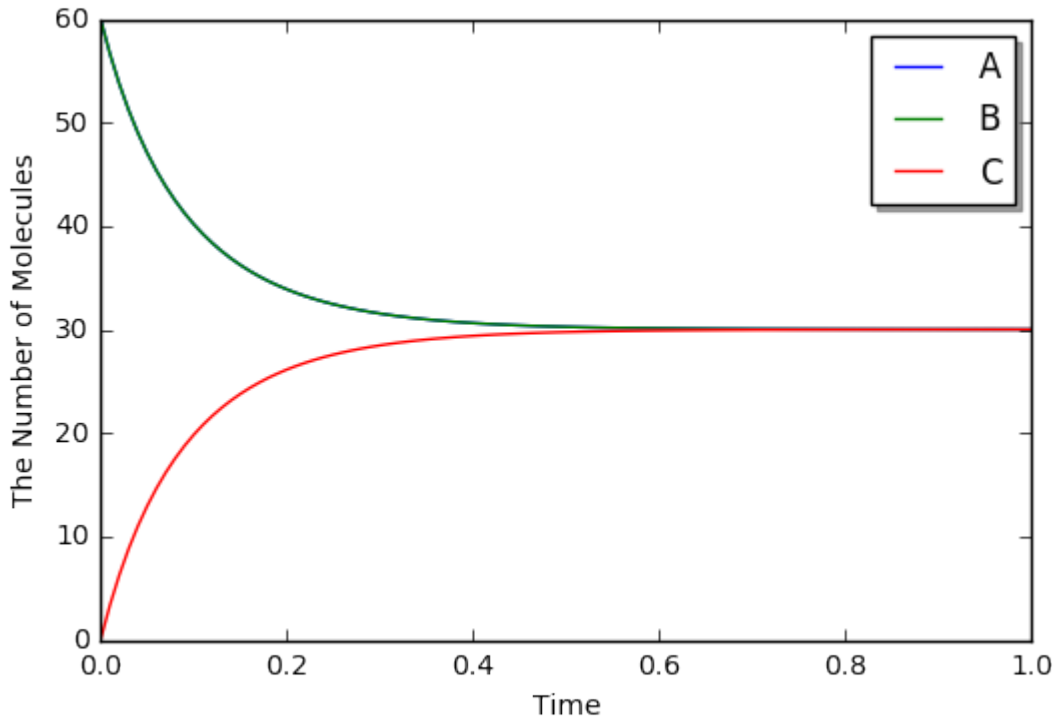
You can access to the list of `ode.ODEReactionRules` in `ode.ODENetworkModel` via its member `reaction_rules()`.

[11]:
```
print([rr.as_string() for rr in m1.reaction_rules()])
```

```
['C>A+B|3', 'A+B>C|<lambda>']
```

Finally, you can run simulations in the same way with other solvers as follows:

[12]:
```
run_simulation(1.0, model=m1, y0={'A': 60, 'B': 60})
```



Modeling with Python decorators is also available by specifying a function instead of a rate (floating number). When a floating number is set, it is assumed to be a kinetic rate of a mass action reaction, but not a constant velocity.

[13]:
```
with reaction_rules():
    A + B == C | (lambda r, *args: 0.1 * reduce(mul, r), 3.0)

m1 = get_model()
```

For the simplicity, you can directory defining the equation with `Species` names as follows:

[14]:
```
with reaction_rules():
    A + B == C | (0.1 * A * B, 3.0)

m1 = get_model()
```

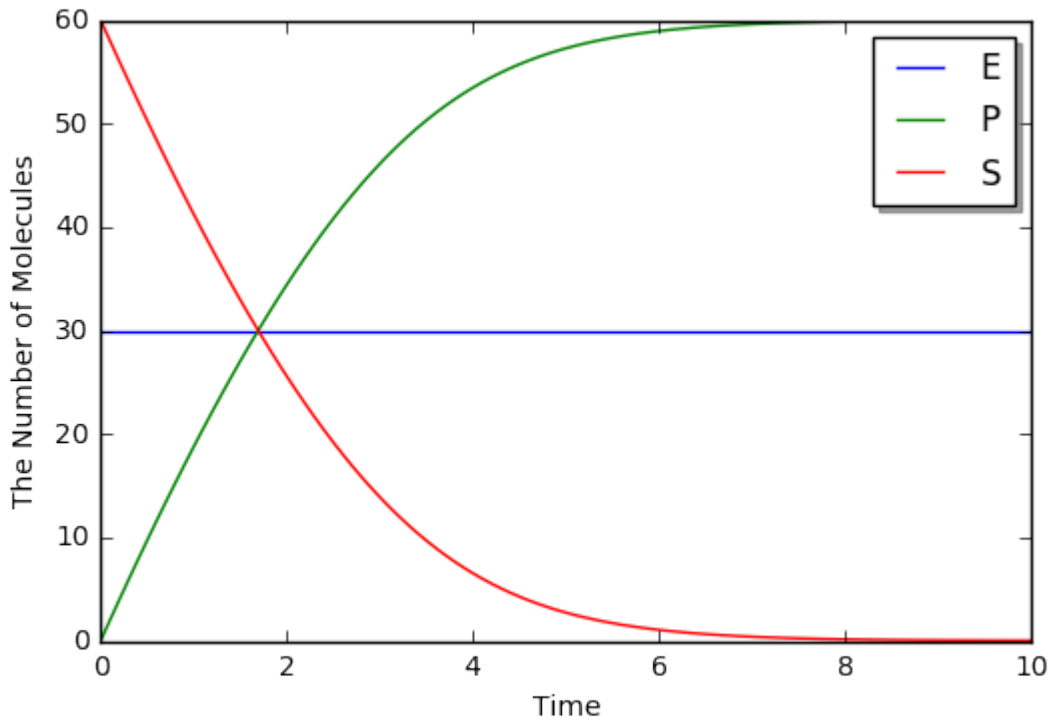
When you call a `Species` (in the rate law) which is not listed as a reactant or product, it is automatically added to the list as an enzyme.

[15]:
```
with reaction_rules():
    S > P | 1.0 * E * S / (30.0 + S)

m1 = get_model()
print(m1.reaction_rules()[0].as_string())
```

```
S+E>P+E|((1.0*E*S)/(30.0+S))
```

where `E` in the equation is appended to both reacant and product lists.

[16]:
```
run_simulation(10.0, model=m1, y0={'S': 60, 'E': 30})
```



Please be careful about typo in `Species`' name. When you make a typo, it is unintentionally recognized as a new enzyme:

[17]:
```
with reaction_rules():
    A13P2G > A23P2G | 1500 * A13B2G  # typo: A13P2G -> A13B2G

m1 = get_model()
print(m1.reaction_rules()[0].as_string())
```

```
A13P2G+A13B2G>A23P2G+A13B2G|(1500*A13B2G)
```

When you want to disable the automatic declaration of enzymes, inactivate `util.decorator.ENABLE_IMPLICIT_DECLARATION`. If its value is `False`, the above case will raise an error:

[20]:
```
util.decorator.ENABLE_IMPLICIT_DECLARATION = False

try:
    with reaction_rules():
        A13P2G > A23P2G | 1500 * A13B2G
except RuntimeError as e:
    print(repr(e))

util.decorator.ENABLE_IMPLICIT_DECLARATION = True
```

```
RuntimeError('unknown variable [A13B2G] was used.',)
```

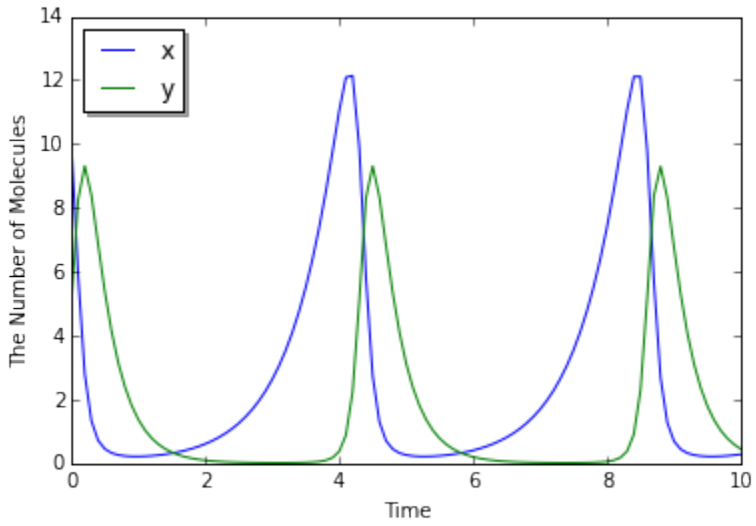
Although E-Cell4 is specialized for a simulation of biochemical reaction network, by using a synthetic reaction rule,

ordinary differential equations can be translated intuitively. For example, the Lotka-Volterra equations:

$$\frac{dx}{dt} = Ax - Bxy$$
$$\frac{dy}{dt} = -Cx + Dxy$$

where $A = 1.5, B = 1, C = 3, D = 1, x(0) = 10, y(0) = 5$, are solved as follows:

[19]:
```
with reaction_rules():
    A, B, C, D = 1.5, 1, 3, 1

    ~x > x | A * x - B * x * y
    ~y > y | -C * y + D * x * y

run_simulation(10, model=get_model(), y0={'x': 10, 'y': 5})
```



1.6.4 6.4. References in a Rate Law

Here, we exlain the details in the rate law definition.

First, when you use simpler definitions of a rate law with `Species`, only a limited number of mathematical functions (e.g. `exp`, `log`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, and `pi`) are available there even if you declare the function outside the block.

[20]:
```
try:
    from math import erf

    with reaction_rules():
        S > P | erf(S / 30.0)
except TypeError as e:
    print(repr(e))
```

```
TypeError('a float is required',)
```

This is because `erf` is tried to be evaluated agaist `S / 30.0` first, but it is not a floating number. In contrast, the following case is acceptable:

[21]:
```
from math import erf

with reaction_rules():
    S > P | erf(2.0) * S

m1 = get_model()
print(m1.reaction_rules()[0].as_string())
```

```
S>P|(0.995322265019*S)
```

where only the result of `erf(2.0)`, `0.995322265019`, is passed to the rate law. Thus, the rate law above has no reference to the `erf` function. Similarly, a value of variables declared outside is acceptable, but not as a reference.

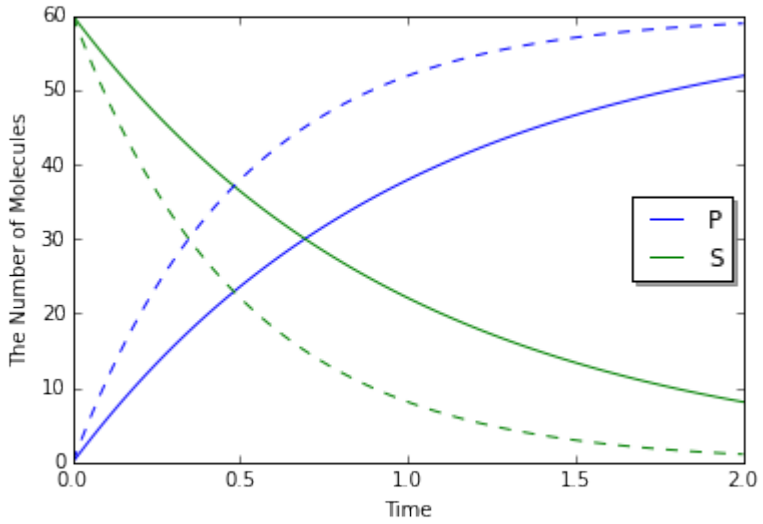
[22]:
```
kcat, Km = 1.0, 30.0

with reaction_rules():
    S > P | kcat * E * S / (Km + S)

m1 = get_model()
print(m1.reaction_rules()[0].as_string())
kcat = 2.0
print(m1.reaction_rules()[0].as_string())
```

```
S+E>P+E|((1.0*E*S)/(30.0+S))
S+E>P+E|((1.0*E*S)/(30.0+S))
```

Even if you change the value of a variable, it does **not** affect the rate law.

On the other hand, when you use your own function to define a rate law, it can hold a reference to variables outside.

[23]:
```
k1 = 1.0

with reaction_rules():
    S > P | (lambda r, *args: k1 * r[0])  # referring k1

m1 = get_model()

obs1 = run_simulation(2, model=m1, y0={"S": 60}, return_type='observer')
k1 = 2.0
obs2 = run_simulation(2, model=m1, y0={"S": 60}, return_type='observer')

viz.plot_number_observer(obs1, '-', obs2, '--')
```



However, in this case, it is better to make a new model for each set of parameters.

[24]:
```
def create_model(k):
    with reaction_rules():
        S > P | k

    return get_model()

obs1 = run_simulation(2, model=create_model(k=1.0), y0={"S": 60}, return_type=
↪'observer')
obs2 = run_simulation(2, model=create_model(k=2.0), y0={"S": 60}, return_type=
↪'observer')
# viz.plot_number_observer(obs1, '-', obs2, '--')
```

1.6.5 6.5. More about ode

In `ode.ODEWorld`, a value for each `Species` is a floating number. However, for the compatibility, the common member `num_molecules` and `add_molecules` regard the value as an integer.

[25]:
```
w = ode.ODEWorld()
w.add_molecules(Species("A"), 2.5)
print(w.num_molecules(Species("A")))
```

```
2
```

To set/get a real number, use `set_value` and `get_value`:

[26]:
```
w.set_value(Species("B"), 2.5)
print(w.get_value(Species("A")))
print(w.get_value(Species("B")))
```

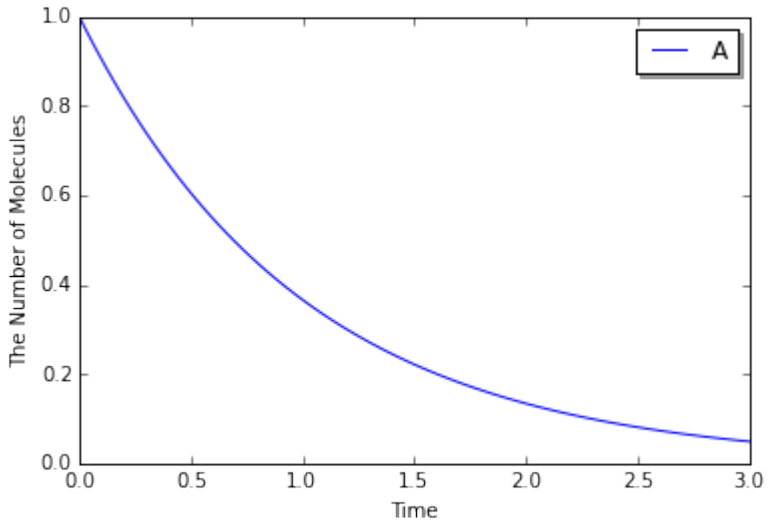
```
2.0
2.5
```

As a default, `ode.ODESimulator` employs the Rosenblock method, called `ROSENBROCK4_CONTROLLER`, to solve ODEs. In addition to that, two solvers, `EULER` and `RUNGE_KUTTA_CASH_KARP54`, are available. `ROSENBROCK4_CONTROLLER` and `RUNGE_KUTTA_CASH_KARP54` adaptively change the step size during time evolution due to error controll, but `EULER` does not.

[27]:
```
with reaction_rules():
    A > ~A | 1.0

m1 = get_model()

w1 = ode.ODEWorld()
w1.set_value(Species("A"), 1.0)
sim1 = ode.ODESimulator(m1, w1, ode.EULER)
sim1.set_dt(0.01) # This is only effective for EULER
sim1.run(3.0, obs1)
```

`ode.ODEFactory` also accepts a solver type and a default step interval.

[28]:
```
run_simulation(3.0, model=m1, y0={"A": 1.0}, solver=('ode', ode.EULER, 0.01))
```



See also examples listed below:

- Glycolysis of Human Erythrocytes
- Drosophila Circadian Clock
- Attractors

1.7 7. Introduction of Rule-based Modeling

E-Cell4 provides the rule-based modeling environment.

[1]:
```
%matplotlib inline
from ecell4 import *
```

1.7.1 7.1. Species.count

First, `Species` provides a function `count`. `Species.count` counts the number of matches between `Species`.

[2]:
```
sp1 = Species("A(b^1).B(b^1)")
sp2 = Species("A(b^1).A(b^1)")
pttrn1 = Species("A")
print(pttrn1.count(sp1))   # => 1
print(pttrn1.count(sp2))   # => 2
```

```
1
2
```

In the above case, `Species.count` just returns the number of `UnitSpecies` named `A` in `Species` regardless of its sites. To specify the occupancy of a bond:

[3]:
```
pttrn1 = Species("A(b)")
pttrn2 = Species("A(b^_)")
print(pttrn1.count(sp1))   # => 0
print(pttrn2.count(sp1))   # => 1
```

```
0
1
```

where `A(b)` suggests that bond `b` is empty, and `A(b^_)` does that bond `b` is occupied. Underscore _ means wildcard here. Similarly, you can also specify the state of sites.

[4]:
```
sp1 = Species("A(b=u)")
pttrn1 = Species("A(b)")
pttrn2 = Species("A(b=u)")
print(pttrn1.count(sp1))   # => 1
print(pttrn2.count(sp1))   # => 1
```

```
1
1
```

`A(b)` says nothing about the state, but `A(b=u)` specifies both state and bond. `A(b=u)` means that `UnitSpecies` named `A` has a site named `b` which state is `u` and the bond is empty. Wildcard _ is acceptable even in a state and name.

[5]:
```
sp1 = Species("A(b=u^1).B(b=p^1)")
pttrn1 = Species("A(b=_^_)")  # This is equivalent to `A(b^_)` here
pttrn2 = Species("_(b^_)")
print(pttrn1.count(sp1))   # => 1
print(pttrn2.count(sp1))   # => 2
```

```
1
2
```

Wildcard _ matches anything, and the pattern matched is not consistent between wildcards even in the `Species`. On the other hand, named wildcards, `_1`, `_2` and so on, confer the consistency within the match.

[6]:
```
sp1 = Species("A(b^1).B(b^1)")
pttrn1 = Species("_._")
pttrn2 = Species("_1._1")
print(pttrn1.count(sp1))   # => 2
print(pttrn2.count(sp1))   # => 0
```

```
2
0
```

where the first pattern matches in two ways (`A.B` and `B.A`), but the second matches nothing. `Species.count` always distinguishes the order of `UnitSpecies` even in the symmetric case. Thus, `_1._1` does **not** mean the number of dimers.

[7]:
```
sp1 = Species("A(b^1).A(b^1)")
pttrn1 = Species("_1._1")
print(pttrn1.count(sp1))  # => 2
```

```
2
```

1.7.2 7.2. ReactionRule.count and generate

`ReactionRule` also has a function to count matches agaist the given list of reactants.

[8]:
```
rr1 = create_unimolecular_reaction_rule(Species("A(p=u)"), Species("A(p=p)"), 1.0)
sp1 = Species("A(b^1,p=u).B(b^1)")
print(rr1.count([sp1]))  # => 1
```

```
1
```

`ReactionRule.generate` returns a list of `ReactionRules` generated based on the matches.

[9]:
```
print([rr.as_string() for rr in rr1.generate([sp1])])
```

```
[u'A(b^1,p=u).B(b^1)>A(b^1,p=p).B(b^1)|1']
```

`ReactionRule.generate` matters the order of `Species` in the given list:

[10]:
```
rr1 = create_binding_reaction_rule(Species("A(b)"), Species("B(b)"), Species("A(b^1).
↪B(b^1)"), 1.0)
sp1 = Species("A(b)")
sp2 = Species("B(b)")
print([rr.as_string() for rr in rr1.generate([sp1, sp2])])
print([rr.as_string() for rr in rr1.generate([sp2, sp1])])
```

```
[u'A(b)+B(b)>A(b^1).B(b^1)|1']
[]
```

On the current implementation, `ReactionRule.generate` does **not** always return a list of unique `ReactionRules`.

[11]:
```
sp1 = Species("A(b,c^1).A(b,c^1)")
sp2 = Species("B(b,c^1).B(b,c^1)")
print(rr1.count([sp1, sp2]))  # => 4
print([rr.as_string() for rr in rr1.generate([sp1, sp2])])
```

```
4
[u'A(b,c^1).A(b,c^1)+B(b,c^1).B(b,c^1)>A(b^1,c^2).A(b,c^2).B(b^1,c^3).B(b,c^3)|1',␣
↪u'A(b,c^1).A(b,c^1)+B(b,c^1).B(b,c^1)>A(b^1,c^2).A(b,c^2).B(b,c^3).B(b^1,c^3)|1',␣
↪u'A(b,c^1).A(b,c^1)+B(b,c^1).B(b,c^1)>A(b,c^1).A(b^2,c^1).B(b^2,c^3).B(b,c^3)|1',␣
↪u'A(b,c^1).A(b,c^1)+B(b,c^1).B(b,c^1)>A(b,c^1).A(b^2,c^1).B(b,c^3).B(b^2,c^3)|1']
```

`ReactionRules` listed above look different, but all the products suggest the same.

[12]:
```
print(set([unique_serial(rr.products()[0]) for rr in rr1.generate([sp1, sp2])]))
```

```
set([u'A(b,c^1).A(b^2,c^1).B(b^2,c^3).B(b,c^3)'])
```

This is because these `ReactionRules` are generated based on the diffent matches though they produces the same `Species`. For details, See the section below.

Wildcard is also available in `ReactionRule`.

[13]:
```
rr1 = create_unimolecular_reaction_rule(Species("A(p=u^_)"), Species("A(p=p^_)"), 1.0)
print([rr.as_string() for rr in rr1.generate([Species("A(p=u^1).B(p^1)")])])
```

```
[u'A(p=u^1).B(p^1)>A(p=p^1).B(p^1)|1']
```

Of course, wildcard is accepted as a name of `UnitSpecies`.

[14]:
```
rr1 = create_unimolecular_reaction_rule(Species("_(p=u)"), Species("_(p=p)"), 1.0)
print([rr.as_string() for rr in rr1.generate([Species("A(p=u)")])])
print([rr.as_string() for rr in rr1.generate([Species("B(b^1,p=u).C(b^1,p=u)")])])
```

```
[u'A(p=u)>A(p=p)|1']
[u'B(b^1,p=u).C(b^1,p=u)>B(b^1,p=p).C(b^1,p=u)|1', u'B(b^1,p=u).C(b^1,p=u)>B(b^1,p=u).
↪C(b^1,p=p)|1']
```

Named wildcards in a state is useful to specify the correspondence between sites.

[15]:
```
rr1 = create_unbinding_reaction_rule(Species("AB(a=_1, b=_2)"), Species("B(b=_2)"),␣
↪Species("A(a=_1)"), 1.0)
print([rr.as_string() for rr in rr1.generate([Species("AB(a=x, b=y)")])])
print([rr.as_string() for rr in rr1.generate([Species("AB(a=y, b=x)")])])
```

```
[u'AB(a=x,b=y)>B(b=y)+A(a=x)|1']
[u'AB(a=y,b=x)>B(b=x)+A(a=y)|1']
```

Nameless wildcard _ does not care about equality between matches. Products are generated in order.

[16]:
```
rr1 = create_binding_reaction_rule(Species("_(b)"), Species("_(b)"), Species("_(b^1)._
↪(b^1)"), 1.0)
print(rr1.as_string())
print([rr.as_string() for rr in rr1.generate([Species("A(b)"), Species("A(b)")])])
print([rr.as_string() for rr in rr1.generate([Species("A(b)"), Species("B(b)")])])
```

```
_(b)+_(b)>_(b^1)._(b^1)|1
[u'A(b)+A(b)>A(b^1).A(b^1)|1']
[u'A(b)+B(b)>A(b^1).B(b^1)|1']
```

For its symmetry, the former case above is sometimes preffered to have a half of the original kinetic rate. This is because the number of combinations of molecules in the former is $n(n-1)/2$ even though that in the later is n^2, where both numbers of A and B molecules are n. This is true for `gillespie` and `ode`. However, in `egfrd` and `spatiocyte`, a kinetic rate is intrinsic one, and not affected by the number of combinations. Thus, in E-Cell4, no modification in the rate is done even for the case. See Homodimerization and Annihilation for the difference between algorithms.

In constrast to nameless wildcard, named wildcard keeps its consistency, and always suggests the same value in the `ReactionRule`.

[17]:
```
rr1 = create_binding_reaction_rule(Species("_1(b)"), Species("_1(b)"), Species("_1(b^
↪1)._1(b^1)"), 1.0)
print(rr1.as_string())
print([rr.as_string() for rr in rr1.generate([Species("A(b)"), Species("A(b)")])])
print([rr.as_string() for rr in rr1.generate([Species("A(b)"), Species("B(b)")])])  #␣
↪=> []
```

```
_1(b)+_1(b)>_1(b^1)._1(b^1)|1
[u'A(b)+A(b)>A(b^1).A(b^1)|1']
[]
```

Named wildcard is consistent even between `UnitSpecies`' and `site`'s names, technically.

[18]:
```
rr1 = create_binding_reaction_rule(Species("A(b=_1)"), Species("_1(b)"), Species(
↪"A(b=_1^1)._1(b^1)"), 1.0)
print(rr1.as_string())
print([rr.as_string() for rr in rr1.generate([Species("A(b=B)"), Species("A(b)")])])
↪# => []
print([rr.as_string() for rr in rr1.generate([Species("A(b=B)"), Species("B(b)")])])
```

```
A(b=_1)+_1(b)>A(b=_1^1)._1(b^1)|1
[]
[u'A(b=B)+B(b)>A(b=B^1).B(b^1)|1']
```

1.7.3 7.3. NetfreeModel

`NetfreeModel` is a `Model` class for the rule-based modeling. The interface of `NetfreeModel` is almost same with `NetworkModel`, but takes into account rules and matches.

[19]:
```
rr1 = create_binding_reaction_rule(Species("A(r)"), Species("A(l)"), Species("A(r^1).
↪A(l^1)"), 1.0)

m1 = NetfreeModel()
m1.add_reaction_rule(rr1)
print(m1.num_reaction_rules())

m2 = NetworkModel()
m2.add_reaction_rule(rr1)
print(m2.num_reaction_rules())
```

```
1
1
```

Python notation explained in 2. How to Build a Model is available too. To get a model as `NetfreeModel`, set `is_netfree` `True` in `get_model`:

[20]:
```
with reaction_rules():
    A(r) + A(l) > A(r^1).A(l^1) | 1.0

m1 = get_model(is_netfree=True)
print(repr(m1))
```

```
<ecell4.core.NetfreeModel object at 0x7fbb44dc7708>
```

`Model.query_reaction_rules` returns a list of `ReactionRules` agaist the given reactants. `NetworkModel` just returns `ReactionRules` based on the equality of `Species`.

[21]:
```
print(len(m2.query_reaction_rules(Species("A(r)"), Species("A(l)"))))  # => 1
print(len(m2.query_reaction_rules(Species("A(l,r)"), Species("A(l,r)"))))  # => 0
```

```
1
0
```

On the other hand, `NetfreeModel` genarates the list by applying the stored `ReactionRules` in the rule-based way.

[22]:
```
print(len(m1.query_reaction_rules(Species("A(r)"), Species("A(l)"))))  # => 1
print(len(m1.query_reaction_rules(Species("A(l,r)"), Species("A(l,r)"))))  # => 1
```

```
1
1
```

`NetfreeModel` does not cache generated objects. Thus, `NetfreeModel.query_reaction_rules` is slow, but needs less memory.

[23]:
```
print(m1.query_reaction_rules(Species("A(l,r)"), Species("A(l,r)"))[0].as_string())
print(m1.query_reaction_rules(Species("A(l,r^1).A(l^1,r)"), Species("A(l,r)"))[0].as_
↪string())
print(m1.query_reaction_rules(Species("A(l,r^1).A(l^1,r)"), Species("A(l,r^1).A(l^1,r)
↪"))[0].as_string())
```

```
A(l,r)+A(l,r)>A(l,r^1).A(l^1,r)|2
A(l,r^1).A(l^1,r)+A(l,r)>A(l,r^1).A(l^1,r^2).A(l^2,r)|2
A(l,r^1).A(l^1,r)+A(l,r^1).A(l^1,r)>A(l,r^1).A(l^1,r^2).A(l^2,r^3).A(l^3,r)|2
```

`NetfreeModel.expand` expands `NetfreeModel` to `NetworkModel` by iteratively applying `ReactionRules` agaist the given seeds.

[24]:
```
with reaction_rules():
    _(b) + _(b) == _(b^1)._(b^1) | (1.0, 1.0)

m3 = get_model(True)
print(m3.num_reaction_rules())

m4 = m3.expand([Species("A(b)"), Species("B(b)")])
print(m4.num_reaction_rules())
print([rr.as_string() for rr in m4.reaction_rules()])
```

```
2
6
[u'A(b)+A(b)>A(b^1).A(b^1)|1', u'A(b)+B(b)>A(b^1).B(b^1)|1', u'B(b)+B(b)>B(b^1).
↪B(b^1)|1', u'A(b^1).A(b^1)>A(b)+A(b)|1', u'A(b^1).B(b^1)>A(b)+B(b)|1', u'B(b^1).
↪B(b^1)>B(b)+B(b)|1']
```

To avoid the infinite iteration for expansion, you can limit the maximum number of iterations and of `UnitSpecies` in a `Species`.

[25]:
```
m2 = m1.expand([Species("A(l, r)")], 100, {Species("A"): 4})
print(m2.num_reaction_rules())  # => 4
print([rr.as_string() for rr in m2.reaction_rules()])
```

```
4
[u'A(l,r)+A(l,r)>A(l,r^1).A(l^1,r)|2', u'A(l,r^1).A(l^1,r)+A(l,r^1).A(l^1,r)>A(l,r^1).
↪A(l^1,r^2).A(l^2,r^3).A(l^3,r)|2', u'A(l,r)+A(l,r^1).A(l^1,r)>A(l,r^1).A(l^1,r^2).
↪A(l^2,r)|2', u'A(l,r)+A(l,r^1).A(l^1,r^2).A(l^2,r)>A(l,r^1).A(l^1,r^2).A(l^2,r^3).
↪A(l^3,r)|2']
```

1.7.4 7.4. Differences between Species, ReactionRule and NetfreeModel

The functions explained above is similar, but there are some differences in the criteria.

[26]:
```
sp1 = Species("A(b^1).A(b^1)")
sp2 = Species("A(b)")
rr1 = create_unbinding_reaction_rule(sp1, sp2, sp2, 1.0)
print(sp1.count(sp1))
print([rr.as_string() for rr in rr1.generate([sp1])])
```

```
2
[u'A(b^1).A(b^1)>A(b)+A(b)|1']
```

Though `Species.count` suggests two different ways for matching (left-right and right-left), `ReactionRule.generate` returns only one `ReactionRule` because the order doesn't affect the product.

[27]:
```
sp1 = Species("A(b^1).B(b^1)")
rr1 = create_unbinding_reaction_rule(
    sp1, Species("A(b)"), Species("B(b)"), 1.0)
sp2 = Species("A(b^1,c^2).A(b^3,c^2).B(b^1).B(b^3)")
print(sp1.count(sp2))
print([rr.as_string() for rr in rr1.generate([sp2])])
```

```
2
[u'A(b^1,c^2).A(b^3,c^2).B(b^1).B(b^3)>A(b,c^1).A(b^2,c^1).B(b^2)+B(b)|1', u'A(b^1,
↪c^2).A(b^3,c^2).B(b^1).B(b^3)>A(b^1,c^2).A(b,c^2).B(b^1)+B(b)|1']
```

In this case, `ReactionRule.generate` works similarly with `Species.count`. However, `Netfree.query_reaction_rules` returns only one `ReationRule` with higher kinetic rate:

[28]:
```
m1 = NetfreeModel()
m1.add_reaction_rule(rr1)
print([rr.as_string() for rr in m1.query_reaction_rules(sp2)])
```

```
[u'A(b^1,c^2).A(b^3,c^2).B(b^1).B(b^3)>A(b,c^1).A(b^2,c^1).B(b^2)+B(b)|2']
```

`NetfreeModel.query_reaction_rules` checks if each `ReactionRule` generated is the same with others, and summalizes it if possible.

As explaned above, `ReactionRule.generate` matters the order of `Species`, but `Netfree.query_reaction_rules` does not.

[29]:
```
sp1 = Species("A(b)")
sp2 = Species("B(b)")
rr1 = create_binding_reaction_rule(sp1, sp2, Species("A(b^1).B(b^1)"), 1.0)
m1 = NetfreeModel()
m1.add_reaction_rule(rr1)
print([rr.as_string() for rr in rr1.generate([sp1, sp2])])
print([rr.as_string() for rr in m1.query_reaction_rules(sp1, sp2)])
print([rr.as_string() for rr in rr1.generate([sp2, sp1])])  # => []
print([rr.as_string() for rr in m1.query_reaction_rules(sp2, sp1)])
```

```
[u'A(b)+B(b)>A(b^1).B(b^1)|1']
[u'A(b)+B(b)>A(b^1).B(b^1)|1']
[]
[u'B(b)+A(b)>A(b^1).B(b^1)|1']
```

Named wildcards must be consistent in the context while nameless wildcards are not necessarily consistent.

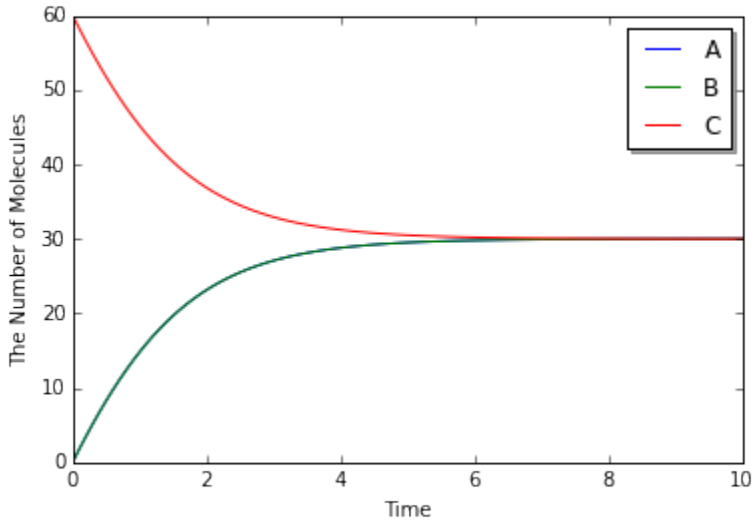
[30]:
```
sp1 = Species("_(b)")
sp2 = Species("_1(b)")
sp3 = Species("A(b)")
sp4 = Species("B(b)")
rr1 = create_binding_reaction_rule(sp1, sp1, Species("_(b^1)._(b^1)"), 1)
rr2 = create_binding_reaction_rule(sp2, sp2, Species("_1(b^1)._1(b^1)"), 1)
print(sp1.count(sp2))  # => 1
print(sp1.count(sp3))  # => 1
print(sp2.count(sp2))  # => 1
print(sp2.count(sp3))  # => 1
print([rr.as_string() for rr in rr1.generate([sp3, sp3])])
print([rr.as_string() for rr in rr1.generate([sp3, sp4])])
print([rr.as_string() for rr in rr2.generate([sp3, sp3])])
print([rr.as_string() for rr in rr2.generate([sp3, sp4])])  # => []
```

```
1
1
1
1
[u'A(b)+A(b)>A(b^1).A(b^1)|1']
[u'A(b)+B(b)>A(b^1).B(b^1)|1']
[u'A(b)+A(b)>A(b^1).A(b^1)|1']
[]
```

1.8 8. More about 1. Brief Tour of E-Cell4 Simulations

Once you read through 1. Brief Tour of E-Cell4 Simulations, it is NOT difficult to use `World` and `Simulator`. `volume` and `{'C': 60}` is equivalent of the `World` and solver is the `Simulator` below.

[1]:
```
%matplotlib inline
from ecell4 import *

with reaction_rules():
    A + B == C | (0.01, 0.3)

y = run_simulation(10.0, {'C': 60}, volume=1.0)
```



Here we give you a breakdown for `run_simulation`. `run_simulation` use ODE simulator by default, so we create `ODEWorld` step by step.

1.8.1 8.1. Creating ODEWorld

You can create `World` like this.

[2]:
```
w = ode.ODEWorld(Real3(1, 1, 1))
```

`Real3` is a coordinate vector. In this example, the first argument for `ODEWorld` constructor is a cube. Note that you can NOT use volume for `ode.ODEWorld` argument, like `run_simulation` argument.

Now you created a cube box for simulation, next let's throw molecules into the cube.

[3]:
```
w = ode.ODEWorld(Real3(1, 1, 1))
w.add_molecules(Species('C'), 60)
print(w.t(), w.num_molecules(Species('C')))  # must return (0.0, 60)
```

```
(0.0, 60)
```

Use `add_molecules` to add molecules, `remove_molecules` to remove molecules, `num_molecules` to know the number of molecules. First argument for each method is the `Species` you want to know. You can get current time by `t` method. However the number of molecules in ODE solver is real number, in these `_molecules` functions work only for integer number. When you handle real numbers in ODE, use `set_value` and `get_value`.

1.8.2 8.2. How to Use Real3

Before the detail of `Simulator`, we explaing more about `Real3`.

[4]:
```
pos = Real3(1, 2, 3)
print(pos)  # must print like <ecell4.core.Real3 object at 0x7f44e118b9c0>
print(tuple(pos))  # must print (1.0, 2.0, 3.0)
```

```
<ecell4.core.Real3 object at 0x7f05b0e6f630>
(1.0, 2.0, 3.0)
```

You can not print the contents in `Real3` object directly. You need to convert `Real3` to Python tuple or list once.

[5]:
```
pos1 = Real3(1, 1, 1)
x, y, z = pos[0], pos[1], pos[2]
pos2 = pos1 + pos1
pos3 = pos1 * 3
pos4 = pos1 / 5
print(length(pos1))  # must print 1.73205080757
print(dot_product(pos1, pos3))  # must print 9.0
```

```
1.73205080757
9.0
```

You can use basic function like `dot_product`. Of course, you can convert `Real3` to numpy array too.

[6]:
```
import numpy
a = numpy.asarray(tuple(Real3(1, 2, 3)))
print(a)  # must print [ 1.  2.  3.]
```

```
[ 1.  2.  3.]
```

`Integer3` represents a triplet of integers.

[7]:
```
g = Integer3(1, 2, 3)
print(tuple(g))
```

```
(1, 2, 3)
```

Of course, you can also apply simple arithmetics to `Integer3`.

[8]:
```
print(tuple(Integer3(1, 2, 3) + Integer3(4, 5, 6)))  # => (5, 7, 9)
print(tuple(Integer3(4, 5, 6) - Integer3(1, 2, 3)))  # => (3, 3, 3)
print(tuple(Integer3(1, 2, 3) * 2))  # => (2, 4, 6)
print(dot_product(Integer3(1, 2, 3), Integer3(4, 5, 6)))  # => 32
print(length(Integer3(1, 2, 3)))  # => 3.74165738677
```

```
(5, 7, 9)
(3, 3, 3)
(2, 4, 6)
32
3.74165738677
```

1.8.3 8.3. Creating and Running ODESimulator

You can create a `Simulator` with `Model` and `World` like

[9]:
```
with reaction_rules():
    A + B > C | 0.01  # equivalent to create_binding_reaction_rule
    C > A + B | 0.3   # equivalent to create_unbinding_reaction_rule

m = get_model()

sim = ode.ODESimulator(m, w)
sim.run(10.0)
```

then call `run` method, the simulation will run. In this example the simulation runs for 10seconds.

You can check the state of the `World` like this.

[10]:
```
print(w.t(), w.num_molecules(Species('C')))  # must return (10.0, 30)
```

```
(10.0, 30)
```

You can see that the number of the `Species C` decreases from 60 to 30.

`World` describes the state of a timepoint, so you can NOT see the transition of the simulation with the `World`. To obtain the time-series result, use `Observer`.

[11]:
```
w = ode.ODEWorld(Real3(1, 1, 1))
w.add_molecules(Species('C'), 60)
sim = ode.ODESimulator(m, w)

obs = FixedIntervalNumberObserver(0.1, ('A', 'C'))
sim.run(10.0, obs)
print(obs.data())  # must return [[0.0, 0.0, 60.0], ..., [10.0, 29.994446899691276,␣
↪30.005553100308752]]
```

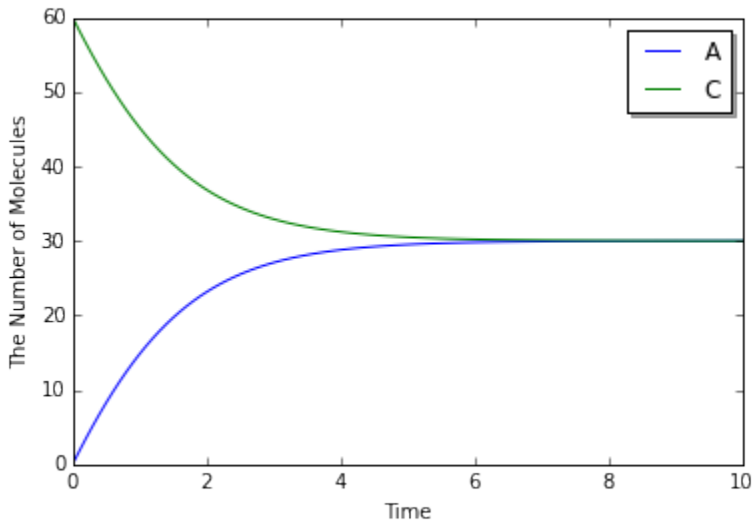
```
[[0.0, 0.0, 60.0], [0.1, 1.7722206098711988, 58.227779390128795], [0.2, 3.
↪4860124889661757, 56.51398751103382], [0.30000000000000004, 5.1376332715496495, 54.
↪862366728450354], [0.4, 6.724090809612153, 53.27590919038786], [0.5, 8.
↪243129756755453, 51.75687024324456], [0.6000000000000001, 9.69320376680592, 50.
↪3067962331941], [0.7000000000000001, 11.073435590968808, 48.92656440903121], [0.8,␣
↪12.383567691608423, 47.616432308391595], [0.9, 13.62390591657045, 46.
↪37609408342957], [1.0, 14.795258681171735, 45.204741318828293], [1.1, 15.
↪898873899780316, 44.10112610021971], [1.2000000000000002, 16.936375633755194, 43.
↪06362436624483], [1.3, 17.90970211303103, 42.090297886969], [1.4000000000000001, 18.
↪821046466966358, 41.17895353303366], [1.5, 19.67280118133671, 40.32719881866331],␣
↪[1.6, 20.46750699887015, 39.53249300112988], [1.7000000000000002, 21.
↪207806714875233, 38.792193285124796], [1.8, 21.896404094818397, 38.10359590518163],␣
↪[1.9000000000000001, 22.536027939969333, 37.463972060030684], [2.0, 23.
↪129401186565122, 36.8705988134349], [2.1, 23.679214801318086, 36.320785198681946],␣
↪[2.2, 24.188106157761755, 35.81189384223828], [2.3000000000000003, 24.
↪658641522840725, 35.34135847715931], [2.4000000000000004, 25.093302252968826, 34.
↪9066977470312], [2.5, 25.49447428958285, 34.505525710417181, [2.6, 25.
↪86440054777012, 34.13555945222991], [2.7, 26.205375806607968, 33.794624193392071,␣
↪[2.8000000000000003, 26.51934373464207, 33.48065626535796], [2.9000000000000004, 26.
↪808295707975571, 33.191704292024461, [3.0, 27.07407111744664, 32.92592888255339], [3.
↪1, 27.318398885233666, 32.681601114766361, [3.2, 27.542899949227504, 32.
↪457100050772524], [3.3000000000000003, 27.749090501388586, 32.25090949861144], [3.
↪4000000000000004, 27.938385796523626, 32.0616142034764], [3.5, 28.112104375214862,␣
↪31.88789562478517], [3.6, 28.271472567998387, 31.728527432001645], [3.7, 28.
```

(continues on next page)

(continued from previous page)

There are several types of `Observers` for E-Cell4. `FixedIntervalNumberObserver` is the simplest `Observer` to obtain the time-series result. As its name suggests, this `Observer` records the number of molecules for each time-step. The 1st argument is the time-step, the 2nd argument is the molecule types. You can check the result with `data` method, but there is a shortcut for this.

[12]:
```
viz.plot_number_observer(obs)
```



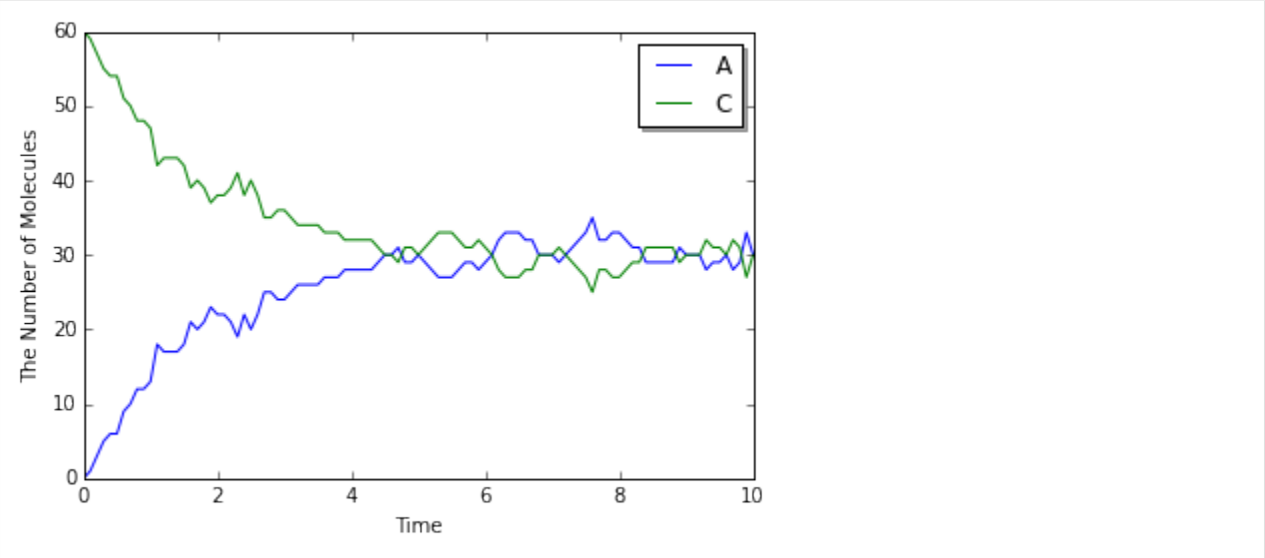
This plots the time-series result easily.

We explained the internal of `run_simulation` function. When you change the `World` after creating the `Simulator`, you need to indicate it to `Simulator`. So do NOT forget to call `sim.initialize()` after that.

1.8.4 8.4. Switching the Solver

It is NOT difficult to switch the solver to stochastic method, as we showed `run_simulation`.

[13]:
```
from ecell4 import *

with reaction_rules():
    A + B == C | (0.01, 0.3)

m = get_model()

# ode.ODEWorld -> gillespie.GillespieWorld
w = gillespie.GillespieWorld(Real3(1, 1, 1))
w.add_molecules(Species('C'), 60)

# ode.ODESimulator -> gillespie.GillespieSimulator
sim = gillespie.GillespieSimulator(m, w)
obs = FixedIntervalNumberObserver(0.1, ('A', 'C'))
sim.run(10.0, obs)

viz.plot_number_observer(obs)
```



`World` and `Simulator` never change the `Model` itself, so you can switch several `Simulators` for 1 `Model`.

1.9 9. Spatial Gillespie Method

1.9.1 9.1. Spaces in E-Cell4

What the space in E-Cell4 looks like?

[1]:
```
from ecell4 import *

w1 = ode.ODEWorld(Real3(1, 1, 1))
w2 = gillespie.GillespieWorld(Real3(1, 1, 1))
```

We created a cube size, `1`, on a side for `ODEWorld` and `GillespieWorld`. In this case the volume only matters, that is

[2]:
```
w3 = ode.ODEWorld(Real3(2, 0.5, 1))  # is almost equivalent to 'w1'
w4 = gillespie.GillespieWorld(Real3(2, 2, 0.25))  # is almost equivalent to 'w2'
```

This returns the same results. Because the volume is same as `1`.

This seems reasonable in homogeneous system, but the cell is NOT homogeneous. So we need to consider a space for molecular localization.

You can use several types of space and simulation methods in E-Cell4. We show an example with spatial Gillespie method here.

1.9.2 9.2. Spatial Gillespie Method

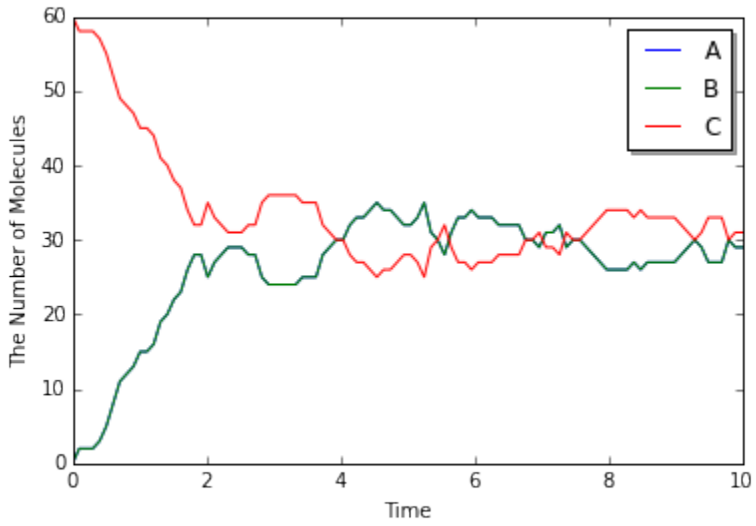
In E-Cell4, the Spatial Gillespie method is included in `meso` module. Let's start with `run_simulation` like `ode`.

[3]:
```
%matplotlib inline
import numpy
from ecell4 import *
```

(continues on next page)

(continued from previous page)

```
with reaction_rules():
    A + B == C | (0.01, 0.3)

y = run_simulation(numpy.linspace(0, 10, 100), {'C': 60}, solver='meso')
```



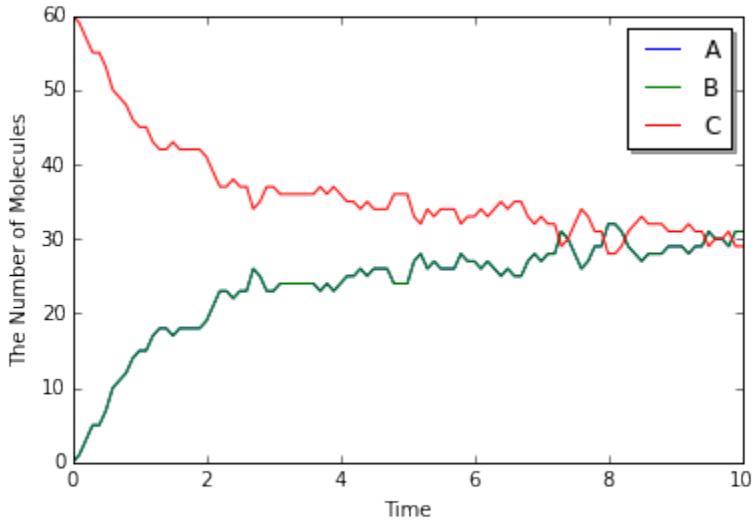
At the steady state, the number of `C` is given as follows:

$$\frac{dC}{dt} = 0.01 \cdot \frac{A}{V} \cdot \frac{B}{V} - 0.3 \cdot \frac{C}{V} = 0$$
$$0.01\,(60 - C)^2 = 0.3C \times V$$
$$C = 30.$$

You will obtain almost the same result with `ode` or `gillespie` (may take longer time than `ode` or `gillespie`). This is not surprising because `meso` module is almost same with Gillespie unless you give additional spatial parameter.

Next we will decompose `run_simulation`.

[4]:
```
from ecell4 import *

with reaction_rules():
    A + B == C | (0.01, 0.3)

m = get_model()

w = meso.MesoscopicWorld(Real3(1, 1, 1), Integer3(1, 1, 1))  # XXX: Point2
w.bind_to(m)  # XXX: Point1
w.add_molecules(Species('C'), 60)

sim = meso.MesoscopicSimulator(w)  # XXX: Point1
obs = FixedIntervalNumberObserver(0.1, ('A', 'B', 'C'))
sim.run(10, obs)

viz.plot_number_observer(obs)
```

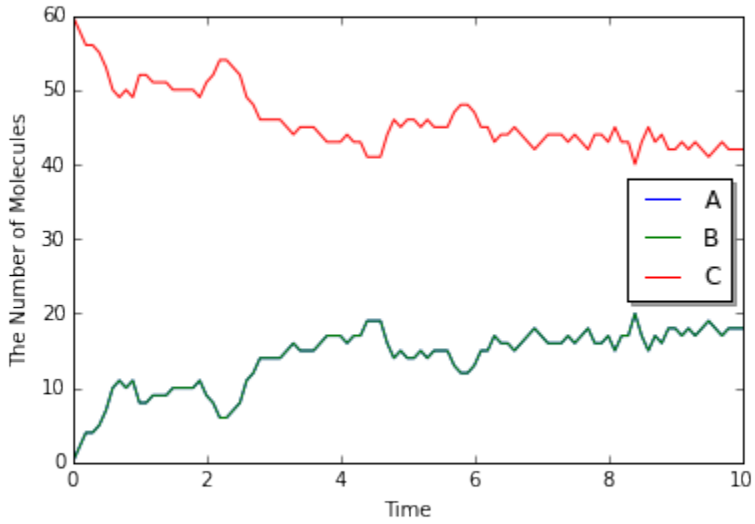


This is nothing out of the ordinary one except for `MesoscopicWorld` and `MesoscopicSimulator`, but you can see some new elements.

First in `w.bind_to(m)` we asscociated a `Model` to the `World`. In the basic exercises before, we did NOT do this. In spatial methods, `Species` attributes are necessary. Do not forget to call this. After that, only the `World` is required to create a `MesoscopicSimulator`.

Next, the important difference is the second argument for MesoscopicWorld, i.e. `Integer3(1, 1, 1)`. `ODEWorld` and `GillespieWorld` do NOT have this second argument. Before we explain this, let's change this argument and run the simulation again.

[5]:
```
from ecell4 import *

with reaction_rules():
    A + B == C | (0.01, 0.3)

m = get_model()

w = meso.MesoscopicWorld(Real3(1, 1, 1), Integer3(4, 4, 4))  # XXX: Point2
w.bind_to(m)  # XXX: Point1
w.add_molecules(Species('C'), 60)

sim = meso.MesoscopicSimulator(w)  # XXX: Point1
obs = FixedIntervalNumberObserver(0.1, ('A', 'B', 'C'))
sim.run(10, obs)

viz.plot_number_observer(obs)
```



You must have the different plot. If you increase value in the `Integer3`, you will have more different one.

Actually this second argument means the number of spatical partitions. `meso` is almost same with `gillespie`, but `meso` divides the space into cuboids (we call these cuboids subvolumes) and each subvolume has different molecular concentration by contrast `gillespie` has only one uniform closed space. So in the preceding example, we divided `1` cube with sides `1` into `64` (4x4x4) cubes with sides `0.25`. We threw 60 `C` molecules into the `World`. Thus, each `subvolume` has 1 species at most.

1.9.3 9.3. Defining Molecular Diffusion Coefficient

Where the difference is coming from? This is because we do NOT consider molecular diffusion coefficient, although we got a space with `meso`. To setup diffusion coefficient, use `Species` attribute `'D'` in the way described before (2. How to Build a Model). As shown in 1. Brief Tour of E-Cell4 Simulations, we use E-Cell4 special notation here.

[6]:
```
with species_attributes():
    A | {'D': '1'}
    B | {'D': '1'}
    C | {'D': '1'}

    # A | B | C | {'D': '1'}  # means the same as above

get_model()
```

[6]:
```
<ecell4.core.NetworkModel at 0x7ff0add53c48>
```

You can setup diffusion coefficient with `with species_attributes():` statement. Here we set all the diffusion coefficient as `1`. Let's simulate this model again. Now you must have the almost same result with `gillespie` even with large `Integer3` value (the simulation will takes much longer than `gillespie`).

How did the molecular diffusion work for the problem? Think about free diffusion (the diffusion coefficient of a Species is D) in 3D space. The unit of diffusion coefficient is the square of length divided by time like $\mu\mathrm{m}^2/s$ or $\mathrm{nm}^2/\mu s$.

It is known that the average of the square of point distance from time 0 to t is equal to $6Dt$. Conversely the average of the time scale in a space with length scale l is about $l^2/6D$.

In the above case, the size of each subvolume is 0.25 and the diffusion coefficient is 1. Thus the time scale is about 0.01 sec. If the molecules of the `Species` `A` and `B` are in the same subvolume, it takes about 1.5 sec to react, so in most cases the diffusion is faster than the reaction and the molecules move to other subvolume even dissociated in the same subvolume. The smaller l, the smaller subvolume's volume l^3, so the reaction rate after dissociation is faster, and the time of the diffusion and the transition between the subvolume gets smaller too.

1.9.4 9.4. Molecular localization

We have used `add_molecules` function to add molecules to `World` in the same manner as `ode` or `gillespie`. Meanwhile in `MesoscopicWorld`, you can put in molecules according to the spatial presentation.

[7]:
```
from ecell4 import *

w = meso.MesoscopicWorld(Real3(1, 1, 1), Integer3(3, 3, 3))
w.add_molecules(Species('A'), 120)
w.add_molecules(Species('B'), 120, Integer3(1, 1, 1))
```

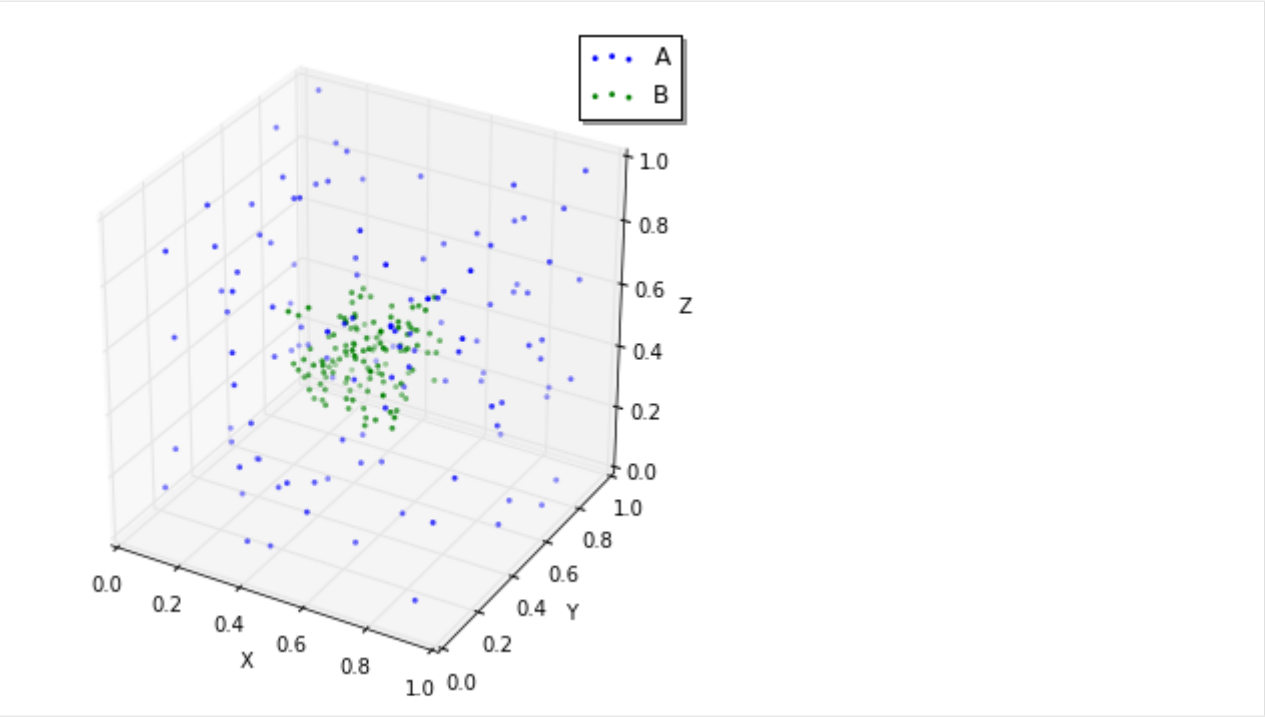
In `MesoscopicWorld`, you can set the subvolume and the molecule locations by giving the third argument `Integer3` to `add_molecules`. In the above example, the molecule type `A` spreads all over the space, but the molecule type `B` only locates in a subvolume at the center of the volume. To check this, use `num_molecules` function with a coordinate.

[8]:
```
print(w.num_molecules(Species('B')))  # must print 120
print(w.num_molecules(Species('B'), Integer3(0, 0, 0)))  # must print 0
print(w.num_molecules(Species('B'), Integer3(1, 1, 1)))  # must print 120
```

```
120
0
120
```

Furthermore, if you have IPython Notebook environment, you can visualize the molecular localization with `ecell4.viz` module.

[9]:
```
# viz.plot_world(w, radius=0.01)
viz.plot_world(w, interactive=False)
```



`viz.plot_world` function visualize the location of the molecules in IPython Notebook cell by giving the `World`. You can set the molecule size with `radius`. Now you can set the molecular localization to the `World`, next let's simulate this. In the above example, we set the diffusion coefficient 1 and the World side 1, so 10 seconds is enough to stir this. After the simulation, check the result with calling `viz.plot_world` again.

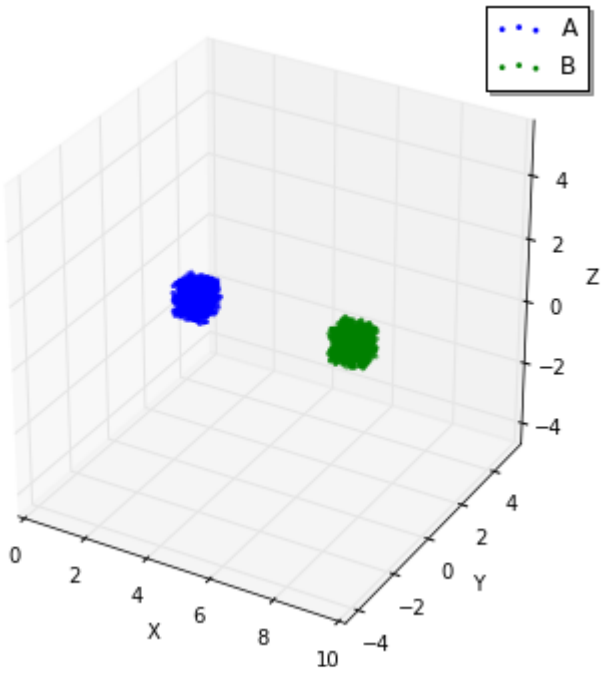
1.9.5 9.5. Molecular initial location and the reaction

This is an extreme example to check how the molecular localization affects the reaction.

[10]:
```
%matplotlib inline
from ecell4 import *

with species_attributes():
    A | B | C | {'D': '1'}

with reaction_rules():
    A + B > C | 0.01

m = get_model()
w = meso.MesoscopicWorld(Real3(10, 1, 1), Integer3(10, 1, 1))
w.bind_to(m)
```

This model consists only of a simple binding reaction. The `World` is a long x axis cuboid, and molecules are located off-center.

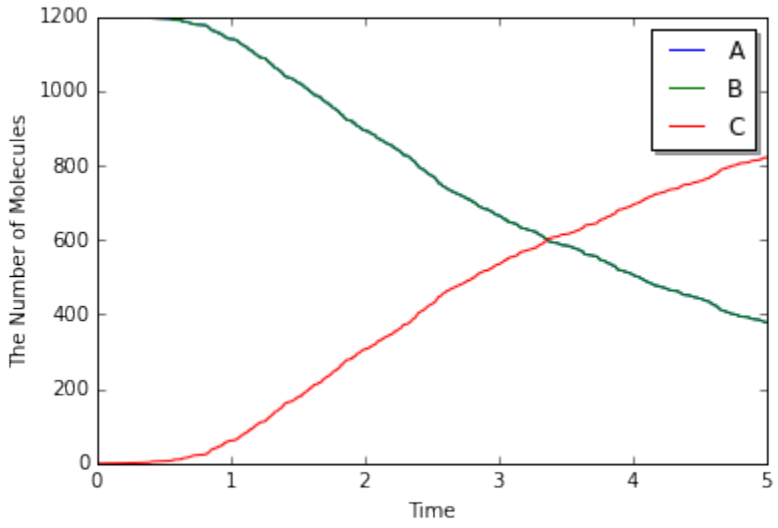
[11]:
```
w.add_molecules(Species('A'), 1200, Integer3(2, 0, 0))
w.add_molecules(Species('B'), 1200, Integer3(7, 0, 0))
# viz.plot_world(w, radius=0.025)
viz.plot_world(w, interactive=False)
```



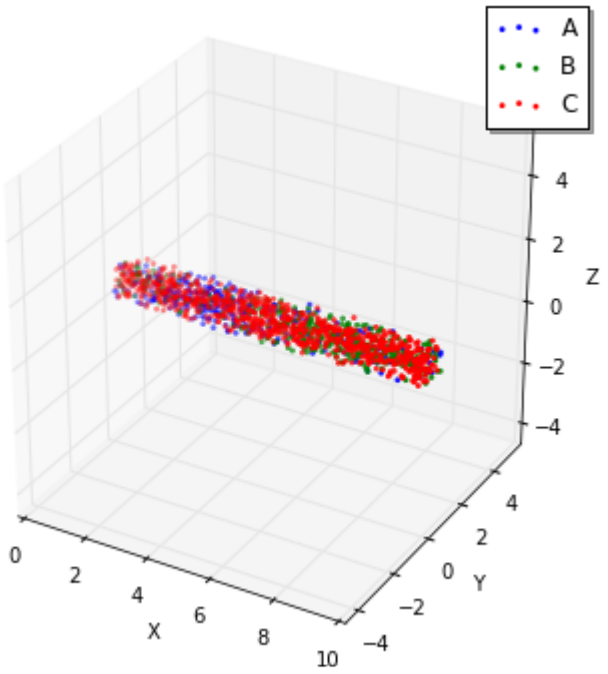
On a different note, there is a reason not to set `Integer3(0, 0, 0)` or `Integer3(9, 0, 0)`. In E-Cell4, basically we adopt periodic boundary condition for everything. So the forementioned two subvolumes are actually adjoining.

After realizing the location expected, simulate it with `MesoscopicSimulator`.

[12]:
```
sim = meso.MesoscopicSimulator(w)
obs1 = NumberObserver(('A', 'B', 'C'))  # XXX: saves the numbers after every steps
sim.run(5, obs1)
viz.plot_number_observer(obs1)
```

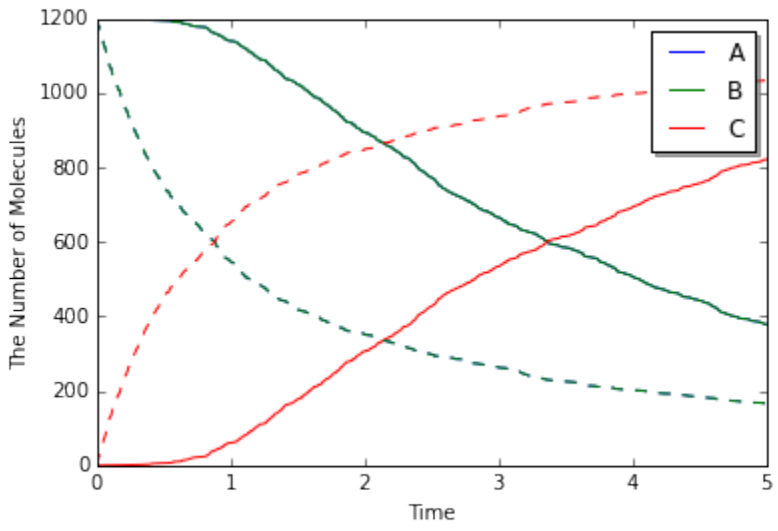


[13]:
```
# viz.plot_world(w, radius=0.025)
viz.plot_world(w, interactive=False)
```



To check the effect of initial coordinates, we recommend that you locate the molecules homogeneously with `meso` or simulate with `gillespie`.

[14]:
```
w = meso.MesoscopicWorld(Real3(10, 1, 1), Integer3(10, 1, 1))
w.bind_to(m)
w.add_molecules(Species('A'), 1200)
w.add_molecules(Species('B'), 1200)

sim = meso.MesoscopicSimulator(w)
obs2 = NumberObserver(('A', 'B', 'C'))  # XXX: saves the numbers after every steps
sim.run(5, obs2)
viz.plot_number_observer(obs1, "-", obs2, "--")
```



The solid line is biased case, and the dash line is non-biased. The biased reaction is obviously slow. And you may notice that the shape of time-series is also different between the solid and dash lines. This is because it takes some time for the molecule A and B to collide due to the initial separation. Actually it takes $4^2/2(D_{\mathrm{A}} + D_{\mathrm{B}}) = 4$ seconds to move the initial distance between A and B (about 4).

1.10 10. Spatiocyte Simulations at Single-Molecule Resolution

We showed an example of E-Cell4 spatial representation.
Next let's simulate the models with more detailed spatial representation called "single molecule resolution".

[1]:
```
%matplotlib inline
from ecell4 import *
```

1.10.1 10.1. Spatiocyte Lattice-based Method

In spatical Gillespie method, we divided the Space into smaller Space, then we diffuse the molecules in the subvolumes. However, we treated the molecules in each subvolume just as the number of the molecules, and the location of the molecules are NOT determinated.

In other words, the spatical resolution of spatical Gillespie method is equal to the side of a subvolume l. To improve this resolution, we need to make the size of l small. But in this method the l must be larger than the (at least) 10 times the diameter of molecule R.

How can we improve the spatical resolution to the size of the molecule? The answer is the simulation with single-molecule resolution. This method simulate the molecule not with the number of the molecules, but with the spatical reaction diffusion of each molecule.

E-Cell4 has multiple single-molecule resolution method, here we explain about Spatiocyte lattice-based method. Spatiocyte treats each molecule as hard spheres, and diffuses the molecules on hexagonal close-packed lattice.

Spatiocyte has an ID for each molecule and the position of the molecule with single-molecule resolution. For the higher spatial resolution, Spatiocyte has 100 times smaller time-step than spatical Gillespie, because the time scale of diffusion increases with the square of the distance.

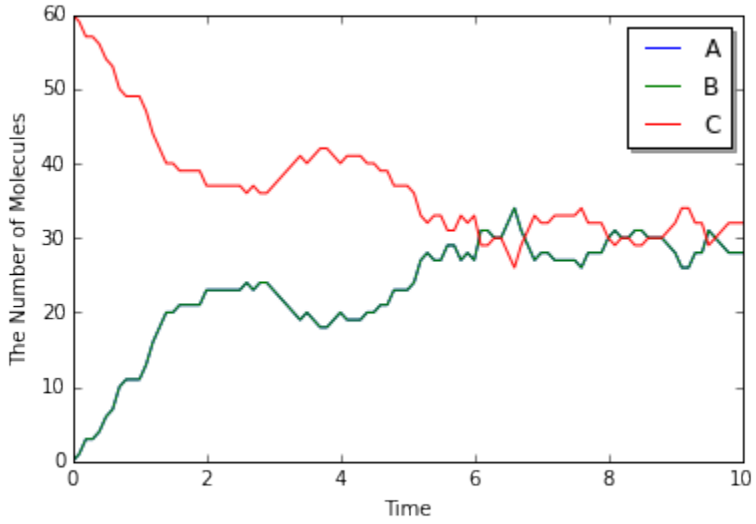
Next, let's try the Spatiocyte method.

[2]:
```
with species_attributes():
    A | B | C | {'D': '1'}

with reaction_rules():
    A + B == C | (0.01, 0.3)

m = get_model()

w = spatiocyte.SpatiocyteWorld(Real3(1, 1, 1), 0.005)  # The second argument is
↪'voxel_radius'.
w.bind_to(m)
w.add_molecules(Species('C'), 60)

sim = spatiocyte.SpatiocyteSimulator(w)
obs = FixedIntervalNumberObserver(0.1, ('A', 'B', 'C'))
sim.run(10, obs)
```

There is a distinct difference in the second argument for `SpatiocyteWorld`. This is called `voxel radius`. Spatiocyte defines the locations of the molecules with dividing the space with molecule size, and call the minimum unit for this space as `Voxel`.

In most cases the size of the molecule would be good for `voxel radius`. In this example, we set 5 nm as the radius of the molecule in the space with the side 1 μm . It takes more time to simulate, but the result is same with ODE or Gillespie.

[3]:
```
viz.plot_number_observer(obs)
```



1.10.2 10.2. The Diffusion Movement of Single Molecule

Next let's simulate single molecule diffusion to check the resolution.

[4]:
```
with species_attributes():
    A | {'D': '1'}

m = get_model()

w = spatiocyte.SpatiocyteWorld(Real3(1, 1, 1), 0.005)
w.bind_to(m)

(pid, p), suc = w.new_particle(Species('A'), Real3(0.5, 0.5, 0.5))
```

`new_particle` method tries to place a particle to a coordinate in `SpatiocyteWorld`. It returns the particle's `ParticleID` (`pid`), the `Particle` object (`p`), and the boolean value (`suc`). If `new_particle` succeeds in putting the particle,`suc` will be True. If a particle is already placed in the coordinate, you can NOT place a particle over it and `suc` will be `False` and fail.

`p` contains the particle position, species type, radius, and diffusion coefficient. You can inspect the `p` with the particle's ID, `pid`.

Let's check `p` first.

[5]:
```
pid, p = w.get_particle(pid)
print(p.species().serial())  # must print: A
print(p.radius(), p.D())  # must print: (0.005, 1.0)
print(tuple(p.position()))  # must print: (0.49806291436591293, 0.49652123150307814,
↪0.5)
```

(continues on next page)

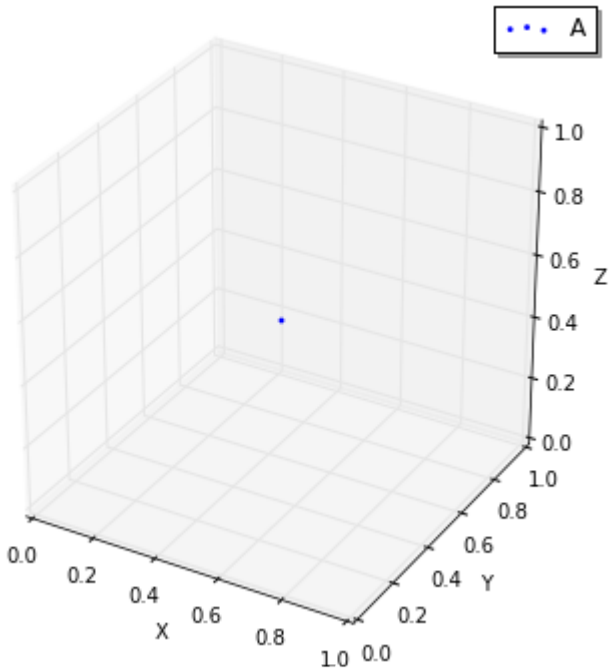
(continued from previous page)

```
A
(0.005, 1.0)
(0.49806291436591293, 0.49652123150307814, 0.5)
```

`get_particle` method receives a particle ID and returns the ID and particle (of cource the ID are same with the given one). You can inspect the coordinate of the particle as `Real3` with `position()` method. It is hard to directly read the coordinate, here we printed it after converting to tuple. As you can see the tuple coodinate is slightly different from the original position given as a `Real3`. This is because Spatiocyte can place the molecule only on the lattice. `SpatiocyteWorld` places the molecule a center position of the nearest lattice for the argument `Real3`.

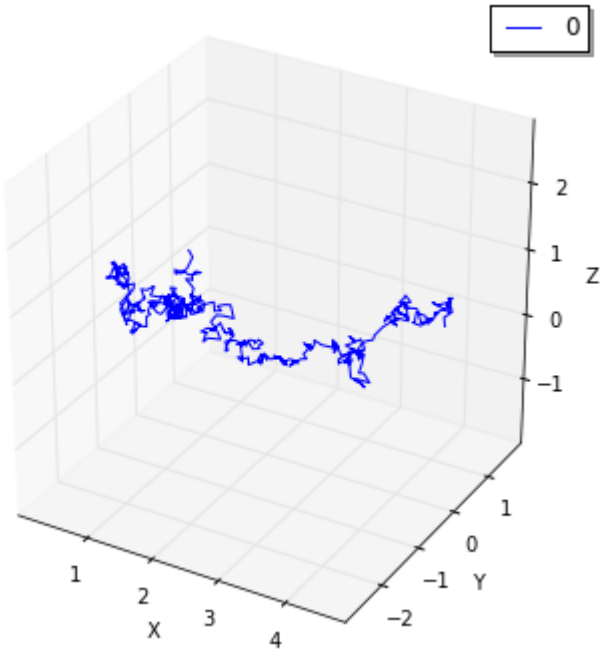
You can visualize the coordinate of the molecule with `viz.plot_world` method, and check the molecule in the center of the World.

[6]:
```
viz.plot_world(w, interactive=False)
# viz.plot_world(w)
```



And you can use `FixedIntervalTrajectoryObserver` to track the trajectory of molecular diffusion process.

[7]:
```
sim = spatiocyte.SpatiocyteSimulator(w)
obs = FixedIntervalTrajectoryObserver(0.002, (pid,))
sim.run(1, obs)
viz.plot_trajectory(obs, interactive=False)
# viz.plot_trajectory(obs)
```



Here we visualized the trajectory with `viz.plot_trajectory` method, you can also obtain it as `Real3` list with `data()` method.

[8]:
```
print(len(obs.data()))  # => 1
print(len(obs.data()[0]))  # => 501
```

```
1
501
```

`data()` method returns nested list. First index means the index of the particle. Second index means the index of the `Real3`. In this case we threw just one particle, so the first result is `1`, and next `501` means time-series coordinate of the only one particle (initial coordinate and the coordinates in 1/0.002 = 500 time points).

Also you can obtain the particles in bulk with `list_particles_exact` method and a `Species`.

[9]:
```
w.add_molecules(Species('A'), 5)

particles = w.list_particles_exact(Species('A'))
for pid, p in particles:
    print(p.species().serial(), tuple(p.position()))
```

```
(u'A', (0.7103520254071217, 0.8256108849411649, 0.88))
(u'A', (0.5062278801751902, 0.034641016151377546, 0.77))
(u'A', (0.4082482904638631, 0.7188010851410841, 0.985))
(u'A', (0.30210373494325865, 0.30599564267050167, 0.68))
(u'A', (0.28577380332470415, 0.26269437248127975, 1.0050000000000001))
(u'A', (0.563382640840131, 0.0202072594216369, 0.545))
```

Please remember `list_particles_exact` method, this method can be used for other World as well as `add_molecules` method.

On a different note, in Spatiocyte proper method to inspect the single molecule is `list_voxels_exact`, and the coordinate is described with index of voxel (not `Real3`).

1.10.3 10.3 The Diffusion Coefficient and the Second-order Reaction

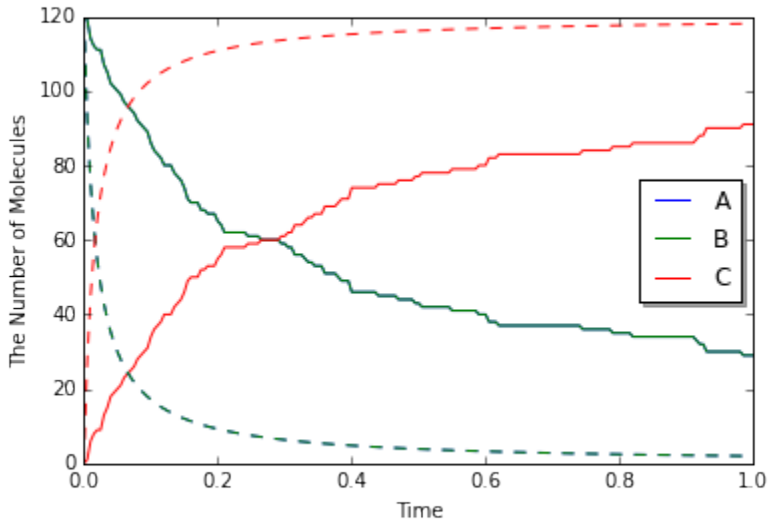
The models we have addressed contains a second-order reaction. Let's look at the relationship between this second-order reaction and the diffusion coefficient in Spatiocyte.

[10]:
```
with species_attributes():
    A | B | C | {'D': '1'}

with reaction_rules():
    A + B > C | 1.0

m = get_model()
```

[11]:
```
w = spatiocyte.SpatiocyteWorld(Real3(2, 1, 1), 0.005)
w.bind_to(m)
w.add_molecules(Species('A'), 120)
w.add_molecules(Species('B'), 120)

obs = FixedIntervalNumberObserver(0.005, ('A', 'B', 'C'))
sim = spatiocyte.SpatiocyteSimulator(w)
sim.run(1.0, obs)
```

[12]:
```
odew = ode.ODEWorld(Real3(2, 1, 1))
# odew.bind_to(m)
odew.add_molecules(Species('A'), 120)
odew.add_molecules(Species('B'), 120)

odeobs = FixedIntervalNumberObserver(0.005, ('A', 'B', 'C'))
odesim = ode.ODESimulator(m, odew)
odesim.run(1.0, odeobs)
```

[13]:
```
viz.plot_number_observer(obs, "-", odeobs, "--")
```



Although we used faster kinetic constant than before, the result is same. But by contrast with ODE simulation, you can find the difference between them (solid line is `spatiocyte`, dash line is `ode`). Is this fault of Spatiocyte? (No) Actually Spatiocyte reaction rate couldn't be faster, while ODE reaction rate can be faster infinitely.

This is caused by the difference between the definition of reaction rate constant in ODE solver and single molecule

simulation method. The former is called "macroscopic" or "effective" reaction rate constant, the latter is called "microscopic" or "intrinsic" reaction rate constant.

The "macroscopic" rate represents the reaction rate in mixed molecular state, meanwhile "microscopic" rate represents the reactivity in molecule collision. So in "microscopic" perspective, the first thing molecules need to react is collision. In Spatiocyte, however, you make this "microscopic" rate faster, you can NOT make the actual reaction rate faster than collision rate. This is called "diffusion-limited" condition. This is similar to what the molecules coordinated disproportionately need time to react.

It is known that there is a relationship between this macroscopic rate constant k_{on} and microscopic rate constant k_a in 3D space.

$\frac{1}{k_{\mathrm{on}}} = \frac{1}{k_a} + \frac{1}{4\pi R D_{\mathrm{tot}}}$,

where R is the sum of two molecule's radius in collision, D_{tot} is the sum of diffusion coefficients.

In the case of the above Jupyter Notebook cell, $k_D = 4\pi R D_{\mathrm{tot}}$ is almost 0.25 and "microscopic" rate constant is 1.0. So the "macroscopic" rate constant is almost 0.2. (However unless you specify the configuration for Spatiocyte, the second order reaction rate must be slower than $3\sqrt{2}RD$, and the dissociation constant k_D is also $3\sqrt{2}RD$.) The single molecule simulation method can separate molecular "diffusion" and "reaction" in accurate manner contrary to ODE or Gillespie method supposed well mixed system (that is diffusion coefficient is infinite). However if the microscopic rate constant k_D is small enough, the macroscopic rate constant is almost equal to microscopic one (reaction late-limit).

1.10.4 10.4. The Structure in the Spatiocyte Method

Next we explain a way to create a structure like cell membrane. Although The structure feature in E-Cell4 is still in development, Spatiocyte supports the structure on some level. Let's look a sphere structure as an example.

To restrict the molecular diffusion inside of the sphere, first we create it.

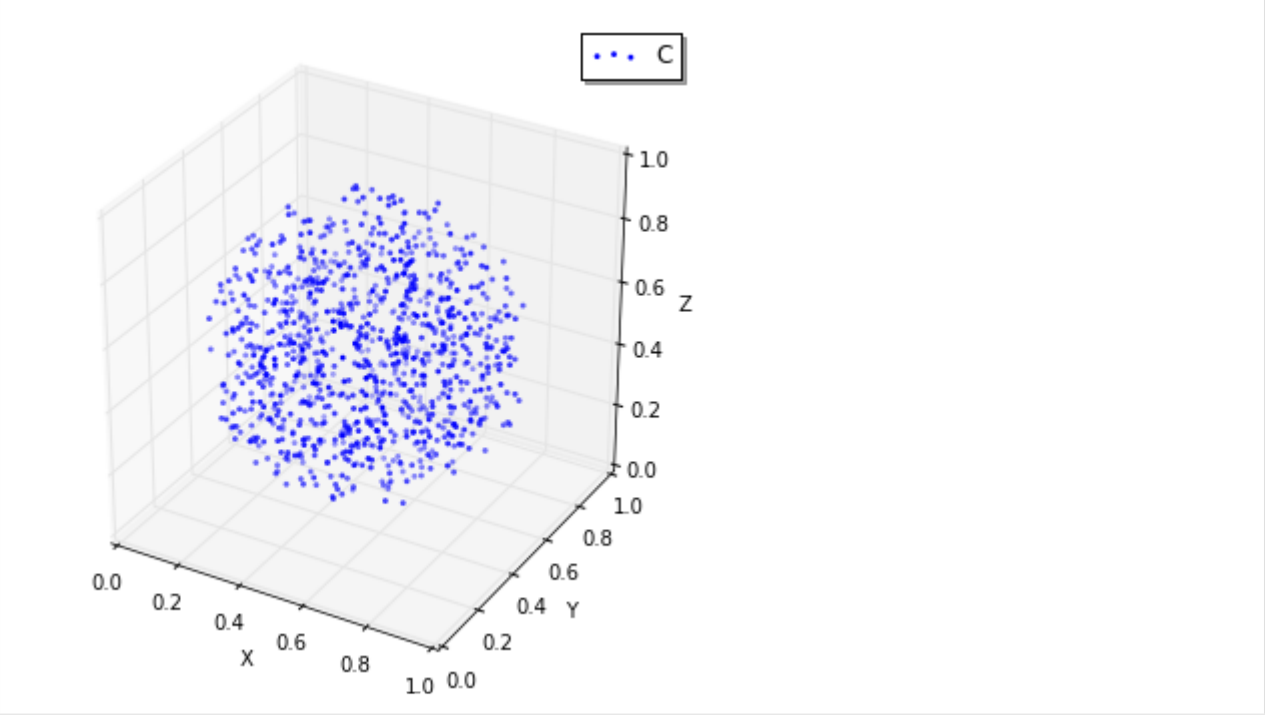
[14]:
```
with species_attributes():
    A | {'D': '1', 'location': 'C'}

m = get_model()
```

[15]:
```
w = spatiocyte.SpatiocyteWorld(Real3(1, 1, 1), 0.005)
w.bind_to(m)
sph = Sphere(Real3(0.5, 0.5, 0.5), 0.45)
print(w.add_structure(Species('C'), sph))  # will print 539805
```

```
539805
```

Visualize the state of the `World`.

[16]:
```
viz.plot_world(w, interactive=False)
# viz.plot_world(w)
```



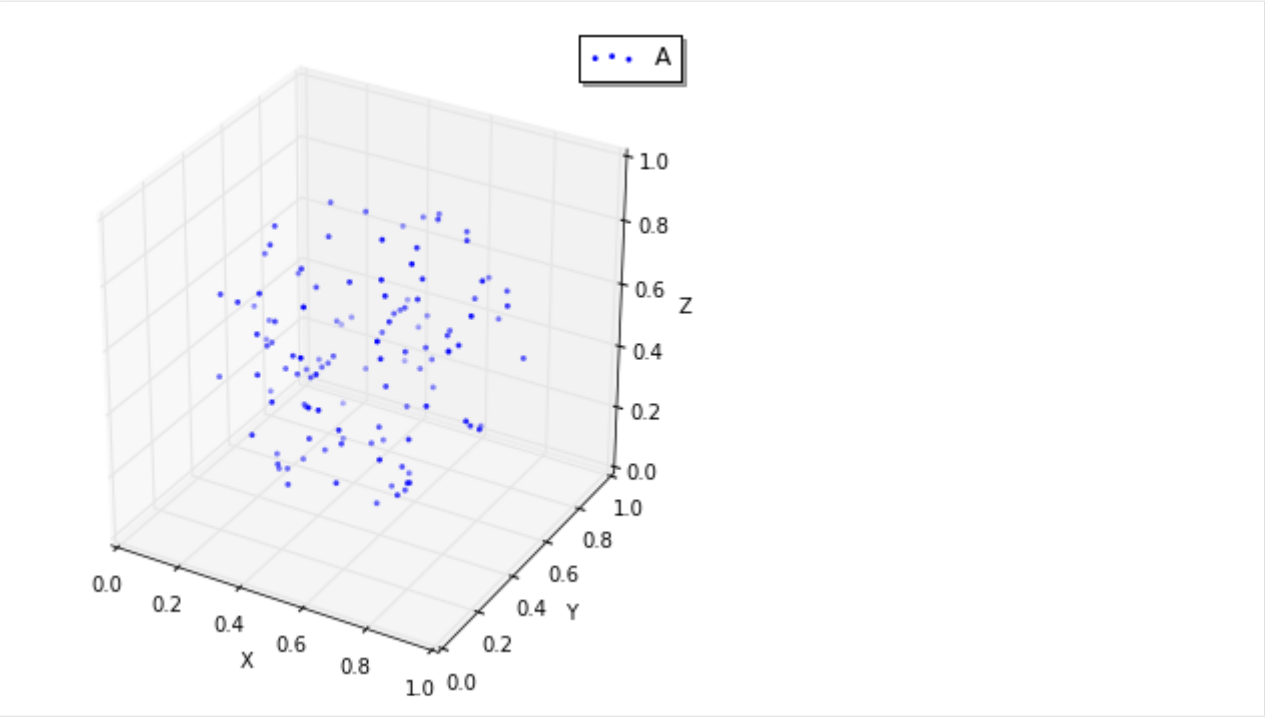
The `Sphere` class first argument is the center of the sphere, and second argument is the radius. Then we created a `Species` named `C` and added it inside the `Sphere`. The structure in the Spatiocyte method is described by filling the space with the `Voxel`. In the example above, the `Voxel`s in the sphere are occupied with `Species` named `C`.

You can see those distribution with `viz.plot_world` as above. (However, the number of the species is too large to visualize all. So we plot only a part of it, but actually the sphere is fully occupied with the `Species`.)

Next we create Species moving inside this sphere. To that end we give `location` attribute to the `Species`. After that, you just throw-in molecules to the `World` with `add_molecules` function.

[17]:
```
w.add_molecules(Species('A'), 120)
```

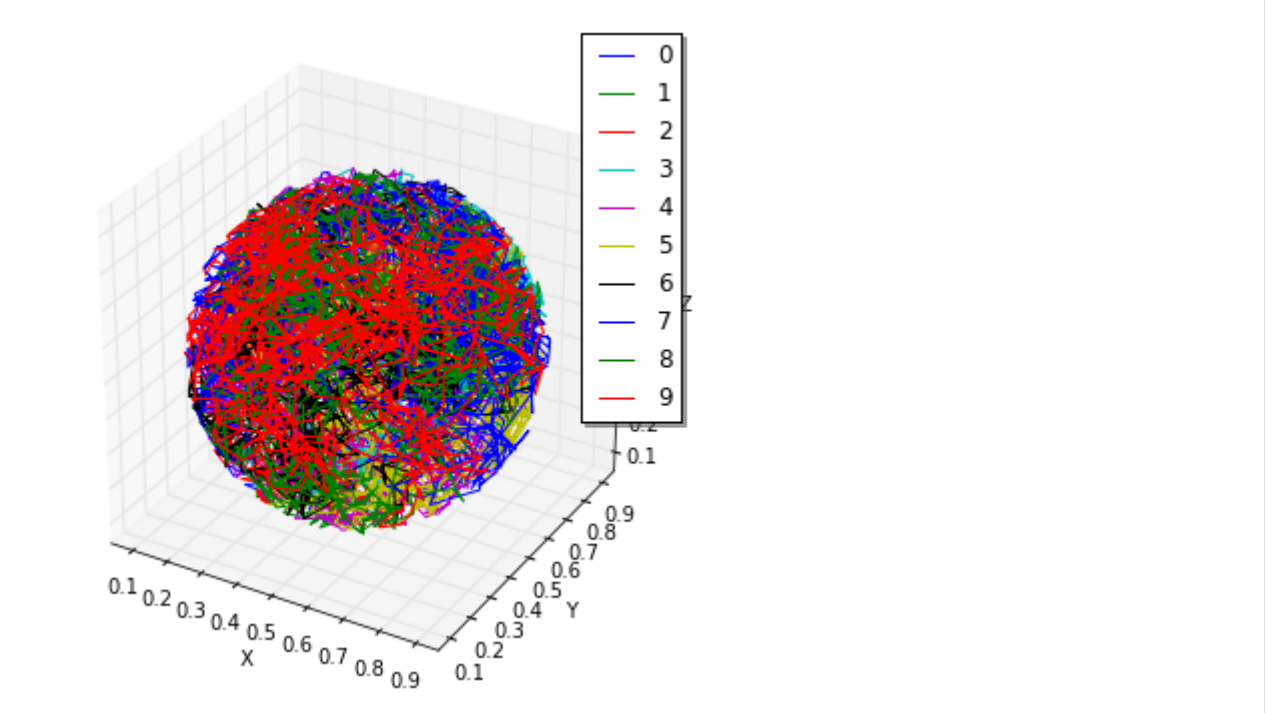
[18]:
```
viz.plot_world(w, species_list=('A',), interactive=False)  # visualize A-molecules␣
↪only
# viz.plot_world(w, species_list=('A',))  # visualize A-molecules only
```



Now we restricted the trajectories of `Species A` on the structure of Species `C`, and `add_molecules` works like that. As a note, you need to create the structure before `add_molecule`.

We can use `FixedIntervalTrajectoryObserver` to check the restriction of the diffusion area.

[19]:
```
pid_list = [pid for pid, p in w.list_particles(Species('A'))[: 10]]
obs = FixedIntervalTrajectoryObserver(1e-3, pid_list)
sim = spatiocyte.SpatiocyteSimulator(w)
sim.run(1, obs)
viz.plot_trajectory(obs, interactive=False)
# viz.plot_trajectory(obs)
```



`pid_list` is a list of the first 10 `ParticleID`s of `A` molecules. The trajectories are colored by this 10 species. Certainly the trajectories are restricted in the sphere.

1.10.5 10.5 The structure and the reaction

At the end, we explain about molecular translocation among the structures.

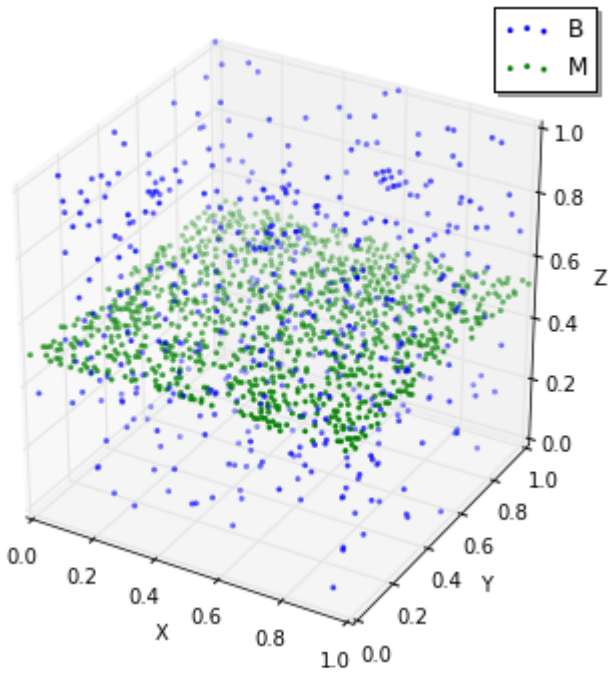
A `Species` without `location` attribute is not an member of any structures. In the example above, if you do NOT write `location` attribute with Species `A`, `A` is placed outside of the sphere.

Next let's create a planar surface structure. To create a surface, we need to use three Real3, those are original point (`origin`) and two axis vector (`unit0`, `unit1`): `ps = PlanarSurface(origin, unit0, unit1)`.

Suppose Species `A` on the surface, `ps`, and a normal Species `B`.

[20]:
```
with species_attributes():
    A | {'D': '0.1', 'location': 'M'}
    B | {'D': '1'}

m  = get_model()

w = spatiocyte.SpatiocyteWorld(Real3(1, 1, 1))
w.bind_to(m)

origin = Real3(0, 0, 0.5)
unit0 = Real3(1, 0, 0)
unit1 = Real3(0, 1, 0)
w.add_structure(
    Species('M'), PlanarSurface(origin, unit0, unit1))  # Create a structure first

w.add_molecules(Species('B'), 480)  # Throw-in B-molecules
```

[21]:
```
viz.plot_world(w, species_list=('B', 'M'), interactive=False)
# viz.plot_world(w, species_list=('B', 'M'))
```



It might be hard to see them, but actually the `Species B` are placed only not on a surface. Then how can we make absorbed this `Species B` to a surface `M` and synthesize a `Species A`?

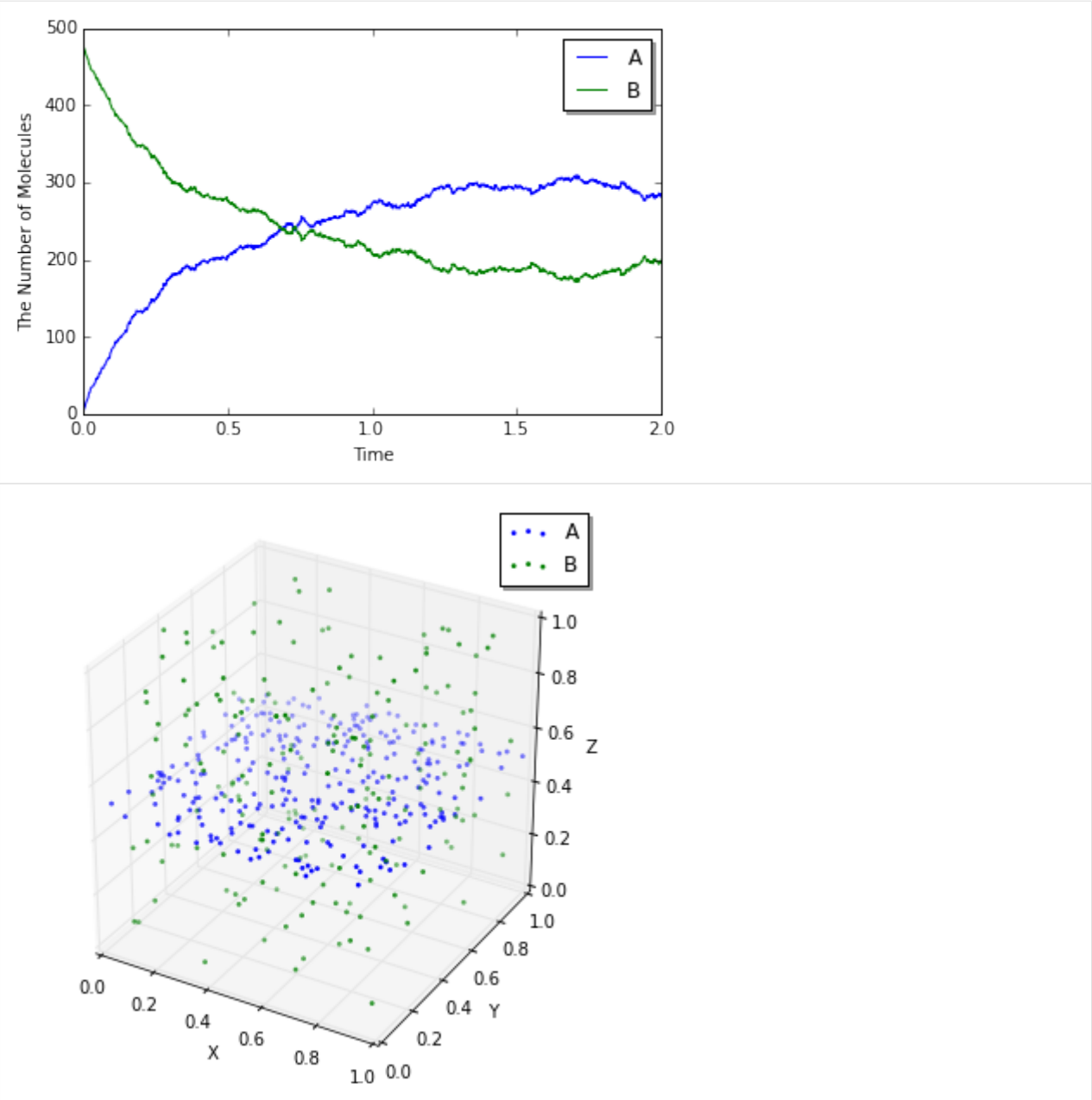
[22]:
```
with species_attributes():
    A | {'D': '0.1', 'location': 'M'}
    B | {'D': '1'}

with reaction_rules():
    B + M == A | (1.0, 1.5)

m = get_model()
```

This means that a `Species B` becomes `A` when `B` collides with a structure `M`. On the other hand, a species `A` dissociates from the structure, and becomes `M` and `B` on as the reverse reaction direction.

Now you can simulate this model with a structure.

[23]:
```
w.bind_to(m)

sim = spatiocyte.SpatiocyteSimulator(w)
obs = NumberObserver(('A', 'B'))
sim.run(2, obs)
```

[24]:
```
viz.plot_number_observer(obs)
viz.plot_world(w, species_list=('A', 'B'), interactive=False)
# viz.plot_world(w, species_list=('A', 'B'))
```



In the dissociation from a structure, you can skip to write the structure. Thus, `A > B` just means the same as `A > B + M` in the above. But in the binding, you can **NOT**. Because it is impossible to create `A` from `B` with no `M` around there. By contrast the species `A` wherever on the sphere `M` can create the species `B`. The first order reaction occurs in either presence or absence of the structure. But, in the case of the binding, the second order reaction turns into the first order reaction and the meaning of rate constant also changes if you ignore `M` in the left-hand side.

CHAPTER

TWO

EXAMPLES

2.1 Attractors

[1]:
```
%matplotlib inline
import numpy
from ecell4 import *
util.decorator.ENABLE_RATELAW = True
```

2.1.1 Rössler attractor

[2]:
```
a, b, c = 0.2, 0.2, 5.7

with reaction_rules():
    ~x > x | (-y - z)
    ~y > y | (x + a * y)
    ~z > z | (b + z * (x - c))
```

[3]:
```
run_simulation(numpy.linspace(0, 200, 4001), y0={'x': 1.0}, return_type='nyaplot',
               opt_args={'x': 'x', 'y': ('y', 'z'), 'to_png': True})
```

```
<IPython.core.display.HTML object>
```

2.1.2 Modified Chua chaotic attractor

[4]:
```
alpha, beta = 10.82, 14.286
a, b, d = 1.3, 0.1, 0.2

with reaction_rules():
    h = -b * sin(numpy.pi * x / (2 * a) + d)
    ~x > x | (alpha * (y - h))
    ~y > y | (x - y + z)
    ~z > z | (-beta * y)
```

[5]:
```
run_simulation(numpy.linspace(0, 250, 5001),
               y0={'x': 0, 'y': 0.49899, 'z': 0.2}, return_type='nyaplot',
               opt_args={'x': 'x', 'y': 'y', 'to_png': True})
```

```
<IPython.core.display.HTML object>
```

2.1.3 Lorenz system

[6]:
```
p, r, b = 10, 28, 8.0 / 3

with reaction_rules():
    ~x > x | (-p * x + p * y)
    ~y > y | (-x * z + r * x - y)
    ~z > z | (x * y - b * z)
```

[7]:
```
run_simulation(numpy.linspace(0, 25, 2501),
               y0={'x': 10, 'y': 1, 'z': 1}, return_type='nyaplot',
               opt_args={'x': 'x', 'y': ('y', 'z'), 'to_png': True})
```

```
<IPython.core.display.HTML object>
```

2.1.4 Tamari attractor

[8]:
```
a = 1.013
b = -0.021
c = 0.019
d = 0.96
e = 0
f = 0.01
g = 1
u = 0.05
i = 0.05

with reaction_rules():
    ~x > x | ((x - a * y) * cos(z) - b * y * sin(z))
    ~y > y | ((x + c * y) * sin(z) + d * y * cos(z))
    ~z > z | (e + f * z + g * a * atan((1 - u) / (1 - i) * x * y))
```

[9]:
```
run_simulation(numpy.linspace(0, 800, 8001),
               y0={'x': 0.9, 'y': 1, 'z': 1}, return_type='nyaplot',
               opt_args={'x': 'x', 'y': ('y', 'z'), 'to_png': True})
```

```
<IPython.core.display.HTML object>
```

2.1.5 Moore-Spiegel attractor

[10]:
```
T, R = 6, 20
with reaction_rules():
    ~x > x | y
    ~y > y | z
    ~z > z | (-z - (T - R + R * x * x) * y - T * x)
```

[11]:
```
run_simulation(numpy.linspace(0, 100, 5001),
               y0={'x': 1, 'y': 0, 'z': 0}, return_type='nyaplot',
               opt_args={'x': 'x', 'y': 'y', 'to_png': True})
```

```
<IPython.core.display.HTML object>
```

2.2 Drosophila Circadian Clock

This is a model of the oscillating Drosophila period protein(PER). This model is based on the model introduced in the following publication.

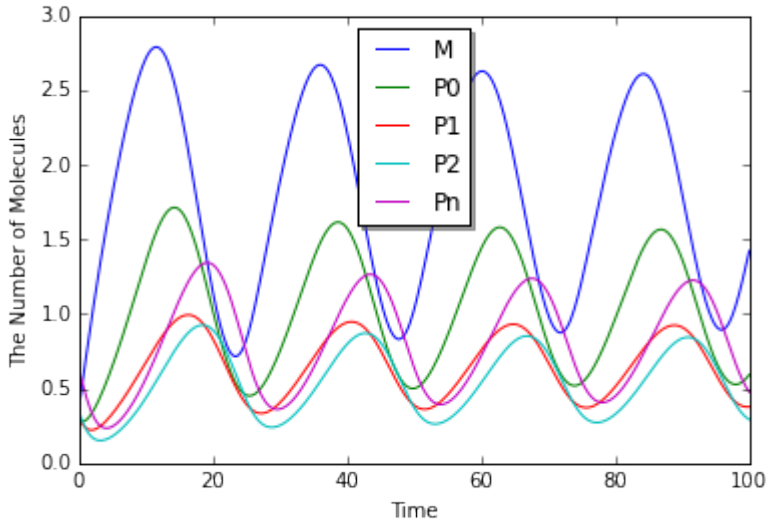
- A. Goldbeter, "A model for circadian oscillations in the Drosophila period protein(PER)", Proc R Soc Lond B Biol Sci, Vol.261:319-324, Sep 1995.

[1]:
```
%matplotlib inline
import numpy
from ecell4 import *
util.decorator.ENABLE_RATELAW = True
```

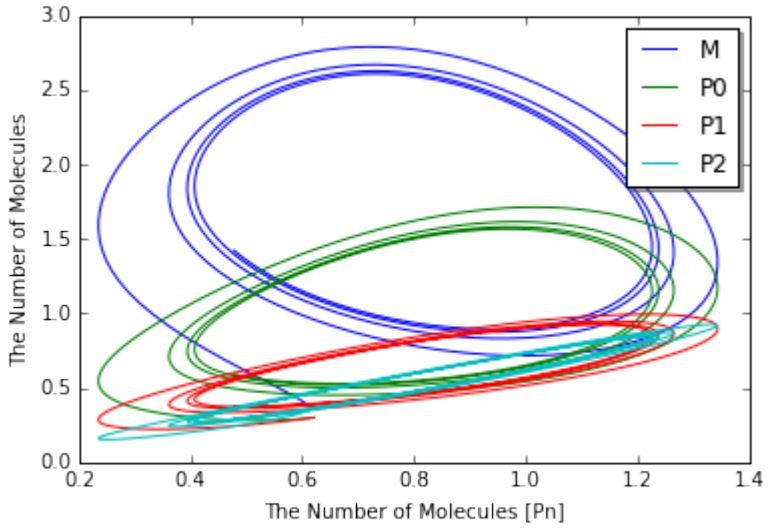
[2]:
```
with reaction_rules():
    ~M > M | 0.76 / (1 + Pn ** 3)
    M > ~M | 0.65 * M / (0.5 + M)
    ~P0 > P0 | 0.38 * M
    P0 == P1 | (3.2 * P0 / (2 + P0), 1.58 * P1 / (2 + P1))
    P1 == P2 | (5 * P1 / (2 + P1), 2.5 * P2 / (2 + P2))
    P2 == Pn | (1.9, 1.3)
    P2 > ~P2 | 0.95 * P2 / (0.2 + P2)
```

[3]:
```
y0 = {"M": 3.61328202E-01, "Pn": 6.21367E-01, "P0": 3.01106835E-01, "P1": 3.01106835E-
↪01, "P2": 3.61328202E-01}
obs = run_simulation(numpy.linspace(0, 100, 400), y0, return_type='observer')
```

[4]:
```
# viz.plot_number_observer(obs, interactive=True)
viz.plot_number_observer(obs)
```



[5]:
```
# viz.plot_number_observer(obs, x="Pn", y=("M", "P0", "P1", "P2"), interactive=True)
viz.plot_number_observer(obs, x="Pn", y=("M", "P0", "P1", "P2"))
```



2.3 Dual Phosphorylation Cycle

[1]:
```
from ecell4.core import *
from ecell4.util import *
```

[2]:
```
@species_attributes
def attrgen(radius, D):
    K | {"radius": radius, "D": D}
    Kp | {"radius": radius, "D": D}
    Kpp | {"radius": radius, "D": D}
    KK | {"radius": radius, "D": D}
    PP | {"radius": radius, "D": D}
    K_KK | {"radius": radius, "D": D}
    Kp_KK | {"radius": radius, "D": D}
    Kpp_PP | {"radius": radius, "D": D}
    Kp_PP | {"radius": radius, "D": D}

@reaction_rules
def rulegen(kon1, koff1, kcat1, kon2, koff2, kcat2):
    (K + KK == K_KK | (kon1, koff1)
        > Kp + KK | kcat1
        == Kp_KK | (kon2, koff2)
        > Kpp + KK | kcat2)

    (Kpp + PP == Kpp_PP | (kon1, koff1)
        > Kp + PP | kcat1
        == Kp_PP | (kon2, koff2)
        > K + PP | kcat2)
```

[3]:
```
m = NetworkModel()
```

[4]:
```
for i, sp in enumerate(attrgen("0.0025", "1")):
    print(i, sp.serial(), sp.get_attribute("radius"), sp.get_attribute("D"))
    m.add_species_attribute(sp)
```

```
(0, u'K', u'0.0025', u'1')
(1, u'Kp', u'0.0025', u'1')
(2, u'Kpp', u'0.0025', u'1')
(3, u'KK', u'0.0025', u'1')
(4, u'PP', u'0.0025', u'1')
(5, u'K_KK', u'0.0025', u'1')
(6, u'Kp_KK', u'0.0025', u'1')
(7, u'Kpp_PP', u'0.0025', u'1')
(8, u'Kp_PP', u'0.0025', u'1')
```

[5]:
```
ka1, kd1, kcat1 = 0.044834550867869 13, 1.35, 1.5
ka2, kd2, kcat2 = 0.09299017957780264, 1.73, 15.0

for i, rr in enumerate(rulegen(ka1, kd2, kcat1, ka2, kd2, kcat2)):
    reactants, products, k = rr.reactants(), rr.products(), rr.k()
    print(i, rr.as_string())
    m.add_reaction_rule(rr)
```

```
(0, u'K+KK>K_KK|0.0448346')
(1, u'K_KK>K+KK|1.73')
(2, u'K_KK>Kp+KK|1.5')
(3, u'Kp+KK>Kp_KK|0.0929902')
(4, u'Kp_KK>Kp+KK|1.73')
(5, u'Kp_KK>Kpp+KK|15')
(6, u'Kpp+PP>Kpp_PP|0.0448346')
(7, u'Kpp_PP>Kpp+PP|1.73')
(8, u'Kpp_PP>Kp+PP|1.5')
(9, u'Kp+PP>Kp_PP|0.0929902')
(10, u'Kp_PP>Kp+PP|1.73')
(11, u'Kp_PP>K+PP|15')
```

[6]:
```
from ecell4.gillespie import GillespieWorld as world_type, GillespieSimulator as␣
↪simulator_type
# from ecell4.ode import ODEWorld as world_type, ODESimulator as simulator_type

w = world_type(Real3(1, 1, 1))
# w.bind_to(m)
w.add_molecules(Species("K"), 120)
w.add_molecules(Species("KK"), 30)
w.add_molecules(Species("PP"), 30)
sim = simulator_type(m, w)
```

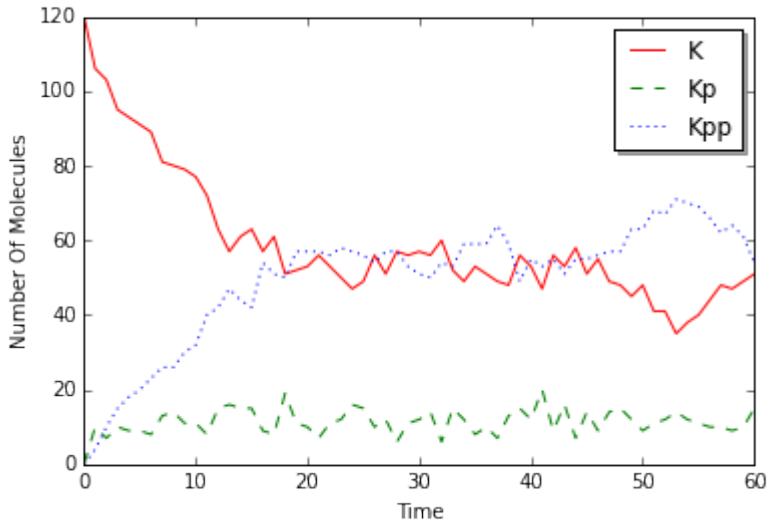
[7]:
```
obs = FixedIntervalNumberObserver(1.0, ["K", "K_KK", "Kp", "Kp_KK", "Kp_PP", "Kpp",
↪"Kpp_PP"])
sim.run(60, [obs])
```

[8]:
```
%matplotlib inline
import matplotlib.pylab as plt
from numpy import array

data = array(obs.data()).T
plt.plot(data[0], data[1] + data[2], "r-", label="K")
plt.plot(data[0], data[3] + data[4] + data[5], "g--", label="Kp")
plt.plot(data[0], data[6] + data[7], "b:", label="Kpp")
plt.xlabel("Time")
plt.ylabel("Number Of Molecules")
```

(continues on next page)

(continued from previous page)

```
plt.xlim(data[0][0], data[0][-1])
plt.legend(loc="best", shadow=True)
plt.show()
```



2.4 Simple EGFR model

- http://bionetgen.org/index.php/Simple_EGFR_model
- M.L. Blinov, J.R. Faeder, B. Goldstein, W.S. Hlavacek, "A network model of early events in epidermal growth factor receptor signaling that accounts for combinatorial complexity.", Biosystems, 83(2-3), 136-151, 2006.

[1]:
```
%matplotlib inline
from ecell4 import *
```

[2]:
```
NA = 6.02e23  # Avogadro's number (molecules/mol)
f = 1  # Fraction of the cell to simulate
Vo = f * 1.0e-10  # Extracellular volume=1/cell_density (L)
V = f * 3.0e-12  # Cytoplasmic volume (L)

EGF_init = 20 * 1e-9 * NA * Vo  # Initial amount of ligand (20 nM) converted to␣
↪copies per cell

# Initial amounts of cellular components (copies per cell)
EGFR_init = f * 1.8e5
Grb2_init = f * 1.5e5
Sos1_init = f * 6.2e4

# Rate constants
# Divide by NA*V to convert bimolecular rate constants
# from /M/sec to /(molecule/cell)/sec
kp1 = 9.0e7 / (NA * Vo)  # ligand-monomer binding
km1 = 0.06  # ligand-monomer dissociation
kp2 = 1.0e7 / (NA * V)  # aggregation of bound monomers
km2 = 0.1  # dissociation of bound monomers
kp3 = 0.5  # dimer transphosphorylation
```

(continues on next page)

(continued from previous page)

```
km3 = 4.505  # dimer dephosphorylation
kp4 = 1.5e6 / (NA * V)  # binding of Grb2 to receptor
km4 = 0.05  # dissociation of Grb2 from receptor
kp5 = 1.0e7 / (NA * V)  # binding of Grb2 to Sos1
km5 = 0.06  # dissociation of Grb2 from Sos1
deg = 0.01  # degradation of receptor dimers
```

[3]:
```
with reaction_rules():
  # R1: Ligand-receptor binding
  EGFR(L, CR1) + EGF(R) == EGFR(L^1, CR1).EGF(R^1) | (kp1, km1)

  # R2: Receptor-aggregation
  EGFR(L^_, CR1) + EGFR(L^_, CR1) == EGFR(L^_,CR1^1).EGFR(L^_,CR1^1) | (kp2, km2)

  # R3: Transphosphorylation of EGFR by RTK
  EGFR(CR1^_, Y1068=U) > EGFR(CR1^_, Y1068=P) | kp3

  # R4: Dephosphorylation
  EGFR(Y1068=P) > EGFR(Y1068=U) | km3

  # R5: Grb2 binding to pY1068
  EGFR(Y1068=P) + Grb2(SH2) == EGFR(Y1068=P^1).Grb2(SH2^1) | (kp4, km4)

  # R6: Grb2 binding to Sos1
  Grb2(SH3) + Sos1(PxxP) == Grb2(SH3^1).Sos1(PxxP^1) | (kp5, km5)

  # R7: Receptor dimer internalization/degradation
  (EGF(R^1).EGF(R^2).EGFR(L^1,CR1^3).EGFR(L^2,CR1^3) > ~EmptySet | deg
       | _policy(ReactionRule.STRICT | ReactionRule.DESTROY))

m = get_model(is_netfree=True, effective=True)
```

[4]:
```
y0 = {"EGF(R)": EGF_init, "EGFR(L, CR1, Y1068=U)": EGFR_init, "Grb2(SH2, SH3)": Grb2_
↪init, "Sos1(PxxP)": Sos1_init}
```

[5]:
```
newm = m.expand([Species(serial) for serial in y0.keys()])
```

[6]:
```
print("{} species and {} reactions were generated.".format(len(newm.list_species()),␣
↪len(newm.reaction_rules())))

for i, sp in enumerate(newm.list_species()):
    print("{}: {}".format(i + 1, sp.serial()))

for i, rr in enumerate(newm.reaction_rules()):
    print("{}: {}".format(i + 1, rr.as_string()))
```

```
22 species and 86 reactions were generated.
1: EGF(R)
2: EGF(R^1).EGFR(CR1,L^1,Y1068=P)
3: EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3)
4: EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3^3).Sos1(PxxP^3)
5: EGF(R^1).EGFR(CR1,L^1,Y1068=U)
6: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P).EGF(R^3)
7: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,SH3)
8: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3^5).Sos1(PxxP^5)
```

(continues on next page)

(continued from previous page)

```
9: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)
10: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3).Grb2(SH2^3,SH3)
11: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3).Grb2(SH2^3,SH3^6).Sos1(PxxP^6)
12: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3^6).Sos1(PxxP^6).Grb2(SH2^3,SH3^7).Sos1(PxxP^7)
13: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).Grb2(SH2^3,
↪SH3)
14: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).Grb2(SH2^3,
↪SH3^5).Sos1(PxxP^5)
15: EGF(R^1).EGFR(CR1^2,L^1,Y1068=U).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)
16: EGFR(CR1,L,Y1068=P)
17: EGFR(CR1,L,Y1068=P^1).Grb2(SH2^1,SH3)
18: EGFR(CR1,L,Y1068=P^1).Grb2(SH2^1,SH3^2).Sos1(PxxP^2)
19: EGFR(CR1,L,Y1068=U)
20: Grb2(SH2,SH3)
21: Grb2(SH2,SH3^1).Sos1(PxxP^1)
22: Sos1(PxxP)
1: EGF(R)+EGFR(CR1,L,Y1068=U)>EGF(R^1).EGFR(CR1,L^1,Y1068=U)|1.49502e-06
2: Grb2(SH2,SH3)+Sos1(PxxP)>Grb2(SH2,SH3^1).Sos1(PxxP^1)|5.5371e-06
3: EGF(R^1).EGFR(CR1,L^1,Y1068=U)>EGF(R)+EGFR(CR1,L,Y1068=U)|0.06
4: EGF(R^1).EGFR(CR1,L^1,Y1068=U)+EGF(R^1).EGFR(CR1,L^1,Y1068=U)>EGF(R^1).EGFR(CR1^2,
↪L^1,Y1068=U).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)|2.76855e-06
5: Grb2(SH2,SH3^1).Sos1(PxxP^1)>Grb2(SH2,SH3)+Sos1(PxxP)|0.06
6: EGF(R^1).EGFR(CR1^2,L^1,Y1068=U).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)>EGF(R^1).
↪EGFR(CR1,L^1,Y1068=U)+EGF(R^1).EGFR(CR1,L^1,Y1068=U)|0.1
7: EGF(R^1).EGFR(CR1^2,L^1,Y1068=U).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)>EGF(R^1).
↪EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)|1
8: EGF(R^1).EGFR(CR1^2,L^1,Y1068=U).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)>|0.01
9: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)>EGF(R^1).
↪EGFR(CR1,L^1,Y1068=P)+EGF(R^1).EGFR(CR1,L^1,Y1068=U)|0.1
10: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)>EGF(R^1).
↪EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P).EGF(R^3)|0.5
11: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)>EGF(R^1).
↪EGFR(CR1^2,L^1,Y1068=U).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)|4.505
12: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)+Grb2(SH2,SH3^1).
↪Sos1(PxxP^1)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).
↪Grb2(SH2^3,SH3^5).Sos1(PxxP^5)|8.30565e-07
13: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)+Grb2(SH2,
↪SH3)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).Grb2(SH2^3,
↪SH3)|8.30565e-07
14: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)>|0.01
15: EGF(R^1).EGFR(CR1,L^1,Y1068=P)>EGF(R)+EGFR(CR1,L,Y1068=P)|0.06
16: EGF(R^1).EGFR(CR1,L^1,Y1068=P)+EGF(R^1).EGFR(CR1,L^1,Y1068=P)>EGF(R^1).EGFR(CR1^2,
↪L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P).EGF(R^3)|2.76855e-06
17: EGF(R^1).EGFR(CR1,L^1,Y1068=P)+EGF(R^1).EGFR(CR1,L^1,Y1068=U)>EGF(R^1).EGFR(CR1^2,
↪L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)|5.5371e-06
18: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P).EGF(R^3)>EGF(R^1).
↪EGFR(CR1,L^1,Y1068=P)+EGF(R^1).EGFR(CR1,L^1,Y1068=P)|0.1
19: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).Grb2(SH2^3,
↪SH3^5).Sos1(PxxP^5)>EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3^3).
↪Sos1(PxxP^3)+EGF(R^1).EGFR(CR1,L^1,Y1068=U)|0.1
20: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).Grb2(SH2^3,
↪SH3)>EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3)+EGF(R^1).EGFR(CR1,L^1,
↪Y1068=U)|0.1
21: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).Grb2(SH2^3,
↪SH3^5).Sos1(PxxP^5)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=U).
↪EGF(R^3).Grb2(SH2^4,SH3^5).Sos1(PxxP^5)|0.5
```

(continues on next page)

(continued from previous page)

```
22: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).Grb2(SH2^3,
↪SH3)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3)|0.5
23: EGF(R^1).EGFR(CR1,L^1,Y1068=P)>EGF(R^1).EGFR(CR1,L^1,Y1068=U)|4.505
24: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P).EGF(R^3)>EGF(R^1).
↪EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)|9.01
25: EGF(R^1).EGFR(CR1,L^1,Y1068=P)+Grb2(SH2,SH3^1).Sos1(PxxP^1)>EGF(R^1).EGFR(CR1,L^1,
↪Y1068=P^2).Grb2(SH2^2,SH3^3).Sos1(PxxP^3)|8.30565e-07
26: EGF(R^1).EGFR(CR1,L^1,Y1068=P)+Grb2(SH2,SH3)>EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).
↪Grb2(SH2^2,SH3)|8.30565e-07
27: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P).EGF(R^3)+Grb2(SH2,SH3^1).
↪Sos1(PxxP^1)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).
↪Grb2(SH2^4,SH3^5).Sos1(PxxP^5)|1.66113e-06
28: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P).EGF(R^3)+Grb2(SH2,
↪SH3)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3)|1.66113e-06
29: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).Grb2(SH2^3,
↪SH3^5).Sos1(PxxP^5)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=U).
↪EGF(R^3)+Grb2(SH2,SH3^1).Sos1(PxxP^1)|0.05
30: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).Grb2(SH2^3,
↪SH3)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=U).EGF(R^3)+Grb2(SH2,
↪SH3)|0.05
31: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).Grb2(SH2^3,
↪SH3)+Sos1(PxxP)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).
↪Grb2(SH2^3,SH3^5).Sos1(PxxP^5)|5.5371e-06
32: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).Grb2(SH2^3,
↪SH3^5).Sos1(PxxP^5)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).
↪EGF(R^4).Grb2(SH2^3,SH3)+Sos1(PxxP)|0.06
33: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P).EGF(R^3)>|0.01
34: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).Grb2(SH2^3,
↪SH3^5).Sos1(PxxP^5)>|0.01
35: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).Grb2(SH2^3,
↪SH3)>|0.01
36: EGF(R)+EGFR(CR1,L,Y1068=P)>EGF(R^1).EGFR(CR1,L^1,Y1068=P)|1.49502e-06
37: EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3^3).Sos1(PxxP^3)>EGF(R)+EGFR(CR1,L,
↪Y1068=P^1).Grb2(SH2^1,SH3^2).Sos1(PxxP^2)|0.06
38: EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3)>EGF(R)+EGFR(CR1,L,Y1068=P^1).
↪Grb2(SH2^1,SH3)|0.06
39: EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3^3).Sos1(PxxP^3)+EGF(R^1).EGFR(CR1,
↪L^1,Y1068=P^2).Grb2(SH2^2,SH3^3).Sos1(PxxP^3)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).
↪EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,SH3^6).Sos1(PxxP^6).Grb2(SH2^3,SH3^7).
↪Sos1(PxxP^7)|2.76855e-06
40: EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3)+EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).
↪Grb2(SH2^2,SH3^3).Sos1(PxxP^3)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,
↪Y1068=P^5).EGF(R^4).Grb2(SH2^5,SH3).Grb2(SH2^3,SH3^6).Sos1(PxxP^6)|5.5371e-06
41: EGF(R^1).EGFR(CR1,L^1,Y1068=P)+EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3^3).
↪Sos1(PxxP^3)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).
↪Grb2(SH2^4,SH3^5).Sos1(PxxP^5)|5.5371e-06
42: EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3^3).Sos1(PxxP^3)+EGF(R^1).EGFR(CR1,
↪L^1,Y1068=U)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).
↪Grb2(SH2^3,SH3^5).Sos1(PxxP^5)|5.5371e-06
43: EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3)+EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).
↪Grb2(SH2^2,SH3)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).
↪EGF(R^4).Grb2(SH2^5,SH3).Grb2(SH2^3,SH3)|2.76855e-06
44: EGF(R^1).EGFR(CR1,L^1,Y1068=P)+EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,
↪SH3)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3)|5.5371e-06
```

(continues on next page)

(continued from previous page)

```
45: EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3)+EGF(R^1).EGFR(CR1,L^1,
↪Y1068=U)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).
↪Grb2(SH2^3,SH3)|5.5371e-06
46: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3^5).Sos1(PxxP^5)>EGF(R^1).EGFR(CR1,L^1,Y1068=P)+EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).
↪Grb2(SH2^2,SH3^3).Sos1(PxxP^3)|0.1
47: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3)>EGF(R^1).EGFR(CR1,L^1,Y1068=P)+EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,
↪SH3)|0.1
48: EGFR(CR1,L,Y1068=P)>EGFR(CR1,L,Y1068=U)|4.505
49: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3^5).Sos1(PxxP^5)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).
↪EGF(R^4).Grb2(SH2^3,SH3^5).Sos1(PxxP^5)|4.505
50: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=U).EGF(R^4).Grb2(SH2^3,
↪SH3)|4.505
51: EGFR(CR1,L,Y1068=P)+Grb2(SH2,SH3^1).Sos1(PxxP^1)>EGFR(CR1,L,Y1068=P^1).Grb2(SH2^1,
↪SH3^2).Sos1(PxxP^2)|8.30565e-07
52: EGFR(CR1,L,Y1068=P)+Grb2(SH2,SH3)>EGFR(CR1,L,Y1068=P^1).Grb2(SH2^1,SH3)|8.
↪30565e-07
53: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3^5).Sos1(PxxP^5)+Grb2(SH2,SH3^1).Sos1(PxxP^1)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).
↪EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,SH3^6).Sos1(PxxP^6).Grb2(SH2^3,SH3^7).
↪Sos1(PxxP^7)|8.30565e-07
54: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3^5).Sos1(PxxP^5)+Grb2(SH2,SH3)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,
↪Y1068=P^5).EGF(R^4).Grb2(SH2^5,SH3).Grb2(SH2^3,SH3^6).Sos1(PxxP^6)|8.30565e-07
55: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3)+Grb2(SH2,SH3^1).Sos1(PxxP^1)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,
↪Y1068=P^5).EGF(R^4).Grb2(SH2^5,SH3).Grb2(SH2^3,SH3^6).Sos1(PxxP^6)|8.30565e-07
56: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3)+Grb2(SH2,SH3)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).
↪EGF(R^4).Grb2(SH2^5,SH3).Grb2(SH2^3,SH3)|8.30565e-07
57: EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3^3).Sos1(PxxP^3)>EGF(R^1).EGFR(CR1,
↪L^1,Y1068=P)+Grb2(SH2,SH3^1).Sos1(PxxP^1)|0.05
58: EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3)>EGF(R^1).EGFR(CR1,L^1,
↪Y1068=P)+Grb2(SH2,SH3)|0.05
59: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3^5).Sos1(PxxP^5)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P).
↪EGF(R^3)+Grb2(SH2,SH3^1).Sos1(PxxP^1)|0.05
60: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P).EGF(R^3)+Grb2(SH2,
↪SH3)|0.05
61: EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3)+Sos1(PxxP)>EGF(R^1).EGFR(CR1,L^1,
↪Y1068=P^2).Grb2(SH2^2,SH3^3).Sos1(PxxP^3)|5.5371e-06
62: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3)+Sos1(PxxP)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).
↪Grb2(SH2^4,SH3^5).Sos1(PxxP^5)|5.5371e-06
63: EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3^3).Sos1(PxxP^3)>EGF(R^1).EGFR(CR1,
↪L^1,Y1068=P^2).Grb2(SH2^2,SH3)+Sos1(PxxP)|0.06
64: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3^5).Sos1(PxxP^5)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).
↪EGF(R^3).Grb2(SH2^4,SH3)+Sos1(PxxP)|0.06
65: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3^5).Sos1(PxxP^5)>|0.01
66: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,
↪SH3)>|0.01
```

(continues on next page)

(continued from previous page)

```
67: EGF(R)+EGFR(CR1,L,Y1068=P^1).Grb2(SH2^1,SH3^2).Sos1(PxxP^2)>EGF(R^1).EGFR(CR1,L^1,
↪Y1068=P^2).Grb2(SH2^2,SH3^3).Sos1(PxxP^3)|1.49502e-06
68: EGF(R)+EGFR(CR1,L,Y1068=P^1).Grb2(SH2^1,SH3)>EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).
↪Grb2(SH2^2,SH3)|1.49502e-06
69: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3^6).Sos1(PxxP^6).Grb2(SH2^3,SH3^7).Sos1(PxxP^7)>EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).
↪Grb2(SH2^2,SH3^3).Sos1(PxxP^3)+EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3^3).
↪Sos1(PxxP^3)|0.1
70: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3).Grb2(SH2^3,SH3^6).Sos1(PxxP^6)>EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,
↪SH3)+EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3^3).Sos1(PxxP^3)|0.1
71: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3).Grb2(SH2^3,SH3)>EGF(R^1).EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3)+EGF(R^1).
↪EGFR(CR1,L^1,Y1068=P^2).Grb2(SH2^2,SH3)|0.1
72: EGFR(CR1,L,Y1068=P^1).Grb2(SH2^1,SH3^2).Sos1(PxxP^2)>EGFR(CR1,L,Y1068=P)+Grb2(SH2,
↪SH3^1).Sos1(PxxP^1)|0.05
73: EGFR(CR1,L,Y1068=P^1).Grb2(SH2^1,SH3)>EGFR(CR1,L,Y1068=P)+Grb2(SH2,SH3)|0.05
74: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3^6).Sos1(PxxP^6).Grb2(SH2^3,SH3^7).Sos1(PxxP^7)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).
↪EGFR(CR1^2,L^3,Y1068=P^4).EGF(R^3).Grb2(SH2^4,SH3^5).Sos1(PxxP^5)+Grb2(SH2,SH3^1).
↪Sos1(PxxP^1)|0.1
75: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3).Grb2(SH2^3,SH3^6).Sos1(PxxP^6)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,
↪Y1068=P^4).EGF(R^3).Grb2(SH2^4,SH3)+Grb2(SH2,SH3^1).Sos1(PxxP^1)|0.05
76: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3).Grb2(SH2^3,SH3^6).Sos1(PxxP^6)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,
↪Y1068=P^4).EGF(R^3).Grb2(SH2^4,SH3^5).Sos1(PxxP^5)+Grb2(SH2,SH3)|0.05
77: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3).Grb2(SH2^3,SH3)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P).EGFR(CR1^2,L^3,Y1068=P^4).
↪EGF(R^3).Grb2(SH2^4,SH3)+Grb2(SH2,SH3)|0.1
78: EGFR(CR1,L,Y1068=P^1).Grb2(SH2^1,SH3)+Sos1(PxxP)>EGFR(CR1,L,Y1068=P^1).Grb2(SH2^1,
↪SH3^2).Sos1(PxxP^2)|5.5371e-06
79: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3).Grb2(SH2^3,SH3^6).Sos1(PxxP^6)+Sos1(PxxP)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).
↪EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,SH3^6).Sos1(PxxP^6).Grb2(SH2^3,SH3^7).
↪Sos1(PxxP^7)|5.5371e-06
80: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3).Grb2(SH2^3,SH3)+Sos1(PxxP)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,
↪Y1068=P^5).EGF(R^4).Grb2(SH2^5,SH3).Grb2(SH2^3,SH3^6).Sos1(PxxP^6)|1.10742e-05
81: EGFR(CR1,L,Y1068=P^1).Grb2(SH2^1,SH3^2).Sos1(PxxP^2)>EGFR(CR1,L,Y1068=P^1).
↪Grb2(SH2^1,SH3)+Sos1(PxxP)|0.06
82: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3^6).Sos1(PxxP^6).Grb2(SH2^3,SH3^7).Sos1(PxxP^7)>EGF(R^1).EGFR(CR1^2,L^1,
↪Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,SH3).Grb2(SH2^3,SH3^6).
↪Sos1(PxxP^6)+Sos1(PxxP)|0.12
83: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3).Grb2(SH2^3,SH3^6).Sos1(PxxP^6)>EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,
↪L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,SH3).Grb2(SH2^3,SH3)+Sos1(PxxP)|0.06
84: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3^6).Sos1(PxxP^6).Grb2(SH2^3,SH3^7).Sos1(PxxP^7)>|0.01
85: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3).Grb2(SH2^3,SH3^6).Sos1(PxxP^6)>|0.01
86: EGF(R^1).EGFR(CR1^2,L^1,Y1068=P^3).EGFR(CR1^2,L^4,Y1068=P^5).EGF(R^4).Grb2(SH2^5,
↪SH3).Grb2(SH2^3,SH3)>|0.01
```

[7]:
```
species_list = ["EGFR",
```

(continues on next page)

(continued from previous page)

```
                "EGF(R)",
                "EGFR(CR1^_)",
                "EGFR(Y1068=P^_0)",
                "Grb2(SH2, SH3^1).Sos1(PxxP^1)",
                "EGFR(Y1068^1).Grb2(SH2^1, SH3^2).Sos1(PxxP^2)"]
run_simulation(120, model=newm, y0=y0, species_list=species_list,
               opt_kwargs={'interactive': True, 'to_png': True})
```

```
<IPython.core.display.HTML object>
```

2.5 A Simple Model of the Glycolysis of Human Erythrocytes

This is a model for the glycolysis of human erythrocytes which takes into account ATP-synthesis and -consumption. This model is based on the model introduced in the following publication.

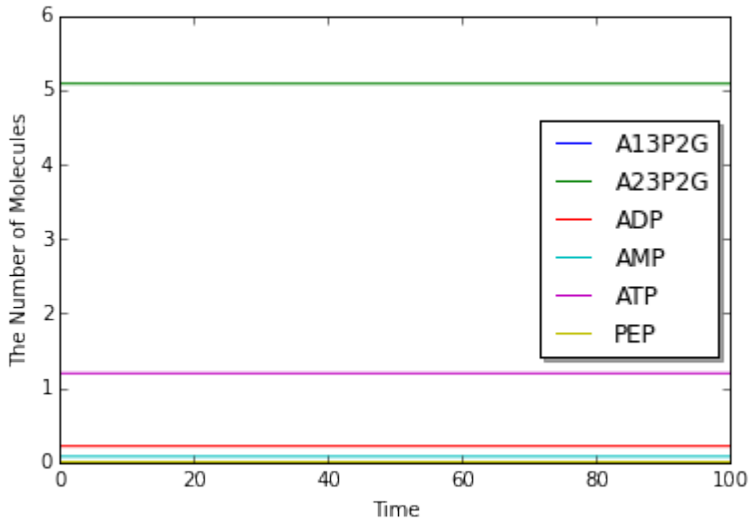
- Rapoport, T.A. and Heinrich, R. (1975) "Mathematical analysis of multienzyme systems. I. Modelling of the glycolysis of human erythrocytes.", Biosystems., 7, 1, 120-129.
- Heinrich, R. and Rapoport, T.A. (1975) "Mathematical analysis of multienzyme systems. II. Steady state and transient control.", Biosystems., 7, 1, 130-136.

[1]:
```
%matplotlib inline
from ecell4 import *
util.decorator.ENABLE_RATELAW = True
```

[2]:
```
with reaction_rules():
    2 * ATP > 2 * A13P2G + 2 * ADP | (3.2 * ATP / (1.0 + (ATP / 1.0) ** 4.0))
    A13P2G > A23P2G | 1500
    A23P2G > PEP | 0.15
    A13P2G + ADP > PEP + ATP | 1.57e+4
    PEP + ADP > ATP | 559
    AMP + ATP > 2 * ADP | (1.0 * (AMP * ATP - 2.0 * ADP * ADP))
    ATP > ADP | 1.46
```

[3]:
```
y0 = {"A13P2G": 0.0005082, "A23P2G": 5.0834, "PEP": 0.020502,
      "AMP": 0.080139, "ADP": 0.2190, "ATP": 1.196867}
```

[4]:
```
run_simulation(100, y0=y0)
```



2.6 Hodgkin-Huxley Model

- A.L. Hodgkin, A.F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve", J. Physiol., 117, 500-544, 1952.

[1]:
```
%matplotlib inline
import numpy as np
from ecell4 import *
```

[2]:
```
Q10 = 3.0
GNa = 120.0 # mS/cm^2
GK = 36.0 # mS/cm^2
gL = 0.3 # mS/cm^2
EL = -64.387 # mV
ENa = 40.0 # mV
EK = -87.0 # mV
Cm = 1.0 # uF/cm^2

T = 6.3 # degrees C
Iext = 10.0 # nA

with reaction_rules():
    Q = Q10 ** ((T - 6.3) / 10)

    alpha_m = -0.1 * (Vm + 50) / (exp(-(Vm + 50) / 10) - 1)
    beta_m = 4 * exp(-(Vm + 75) / 18)
    ~m > m | Q * (alpha_m * (1 - m) - beta_m * m)

    alpha_h = 0.07 * exp(-(Vm + 75) / 20)
    beta_h = 1.0 / (exp(-(Vm + 45) / 10) + 1)
    ~h > h | Q * (alpha_h * (1 - h) - beta_h * h)

    alpha_n = -0.01 * (Vm + 65) / (exp(-(Vm + 65) / 10) - 1)
    beta_n = 0.125 * exp(-(Vm + 75) / 80)
    ~n > n | Q * (alpha_n * (1 - n) - beta_n * n)
```

(continues on next page)

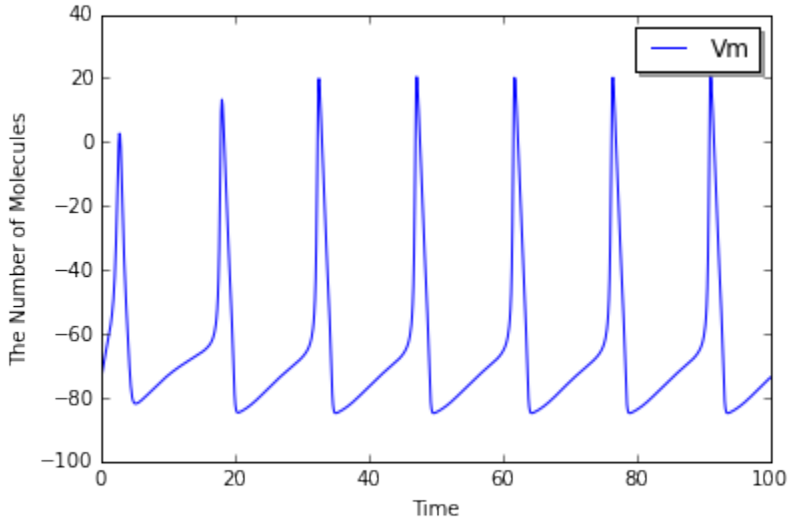
(continued from previous page)

```
    gNa = (m ** 3) * h * GNa
    INa = gNa * (Vm - ENa)
    gK = (n ** 4) * GK
    IK = gK * (Vm - EK)
    IL = gL * (Vm - EL)
    ~Vm > Vm | (Iext - (IL + INa + IK)) / Cm

hhm = get_model()
```

[3]:
```
for rr in hhm.reaction_rules():
    print(rr.as_string())
```

```
Vm>m+Vm|(1.0*((((-0.1*(Vm+50))/(exp((-(Vm+50)/10))-1))*(1-m))-(4*exp((-(Vm+75)/
↪18))*m)))
Vm>h+Vm|(1.0*((0.07*exp((-(Vm+75)/20))*(1-h))-((1.0/(exp((-(Vm+45)/10))+1))*h)))
Vm>n+Vm|(1.0*((((-0.01*(Vm+65))/(exp((-(Vm+65)/10))-1))*(1-n))-(0.125*exp((-(Vm+75)/
↪80))*n)))
m+h+n>Vm+m+h+n|((10.0-((0.3*(Vm-64.387))+((m**3)*h*120.0*(Vm-40.0))+((n**4)*36.
↪0*(Vm-87.0))))/1.0)
```

[4]:
```
run_simulation(np.linspace(0, 100, 1001), model=hhm, y0={'Vm': -75}, species_list=['Vm
↪'])
```



2.7 FitzHugh–Nagumo Model

- R. FitzHugh, “Mathematical models of threshold phenomena in the nerve membrane.”, Bull. Math. Biophysics, 17:257—278, 1955.

[5]:
```
a = 0.7
b = 0.8
c = 12.5
Iext = 0.5
```

(continues on next page)

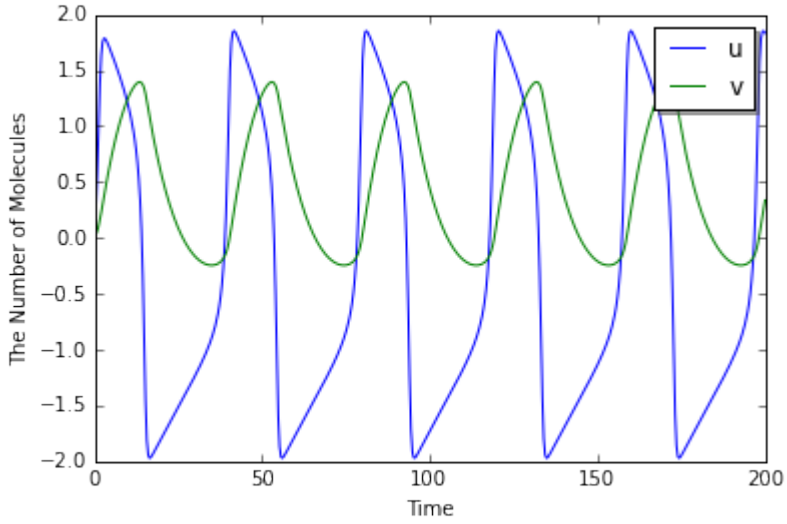
(continued from previous page)

```
with reaction_rules():
    ~u > u | -v + u - (u ** 3) / 3 + Iext
    ~v > v | (u - b * v + a) / c

fnm = get_model()
```

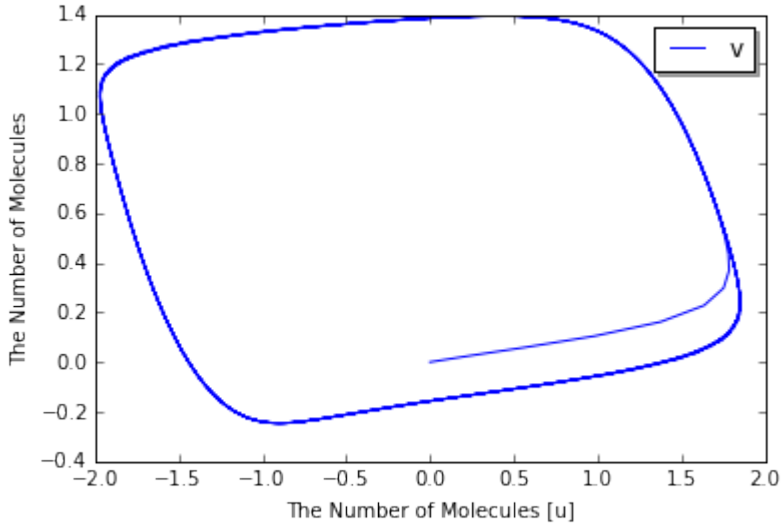
[6]:
```
for rr in fnm.reaction_rules():
    print(rr.as_string())
```

```
v>u+v|(((-v+u)-((u**3)/3))+0.5)
u>v+u|(((u-(0.8*v))+0.7)/12.5)
```

[7]:
```
run_simulation(np.linspace(0, 200, 501), model=fnm)
```



[8]:
```
run_simulation(np.linspace(0, 200, 501), model=fnm, # return_type='nyaplot',
               opt_kwargs={'x': 'u', 'y': ['v']})
```



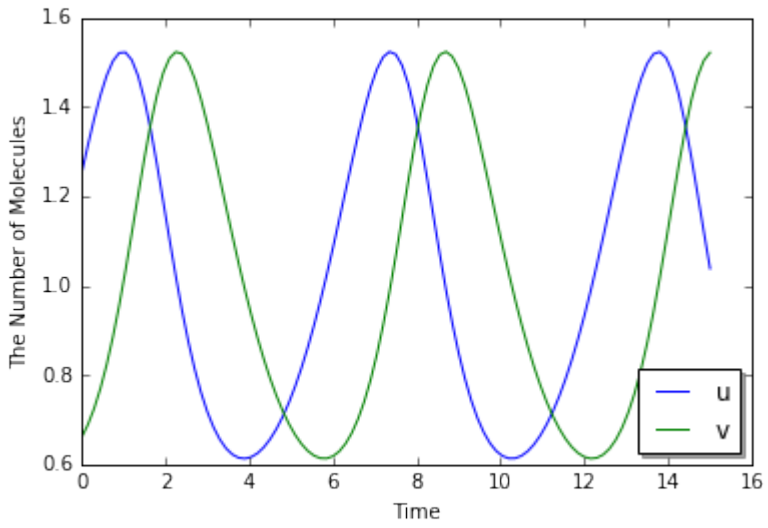
2.8 Lotka-Volterra 2D

2.8.1 The Original Model in Ordinary Differential Equations

[1]:
```
%matplotlib inline
from ecell4 import *
```

[2]:
```
alpha = 1

with reaction_rules():
    ~u > u | u * (1 - v)
    ~v > v | alpha * v * (u - 1)

m = get_model()
```

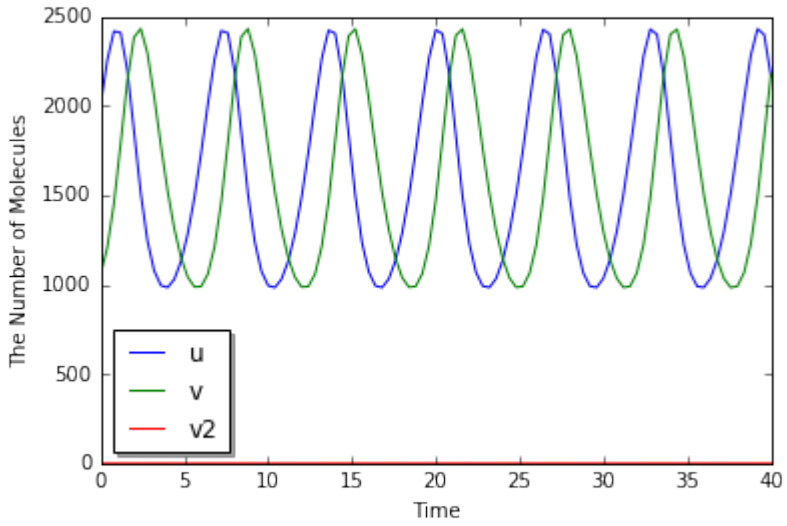
[3]:
```
run_simulation(15, {'u': 1.25, 'v': 0.66}, model=m)
```



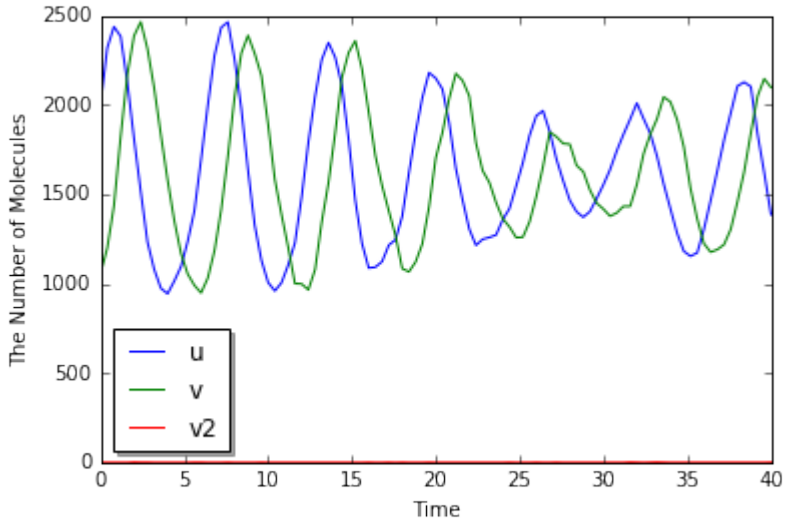
2.8.2 The Modified Model Decomposed into Elementary Reactions

[4]:
```
alpha = 1

with species_attributes():
    u | {'D': '0.1'}
    v | {'D': '0.1'}

with reaction_rules():
    u > u + u | 1.0
    u + v > v | 1.0

    u + v > u + v2 | alpha
    v2 > v + v | alpha * 10000.0
    v > ~v | alpha

m = get_model()
```

[5]:
```
run_simulation(40, {'u': 1.25 * 1600, 'v': 0.66 * 1600}, volume=1600, model=m)
```



[6]:
```
run_simulation(40, {'u': 1.25 * 1600, 'v': 0.66 * 1600}, volume=1600, model=m, solver=
↪'gillespie')
```



2.8.3 A Lotka-Volterra-like Model in 2D

[7]:
```
rng = GSLRandomNumberGenerator()
rng.seed(0)
```

[8]:
```
w = meso.MesoscopicWorld(Real3(40, 40, 1), Integer3(160, 160, 1), rng)
w.bind_to(m)
```

[9]:
```
V = w.volume()
print(V)
```

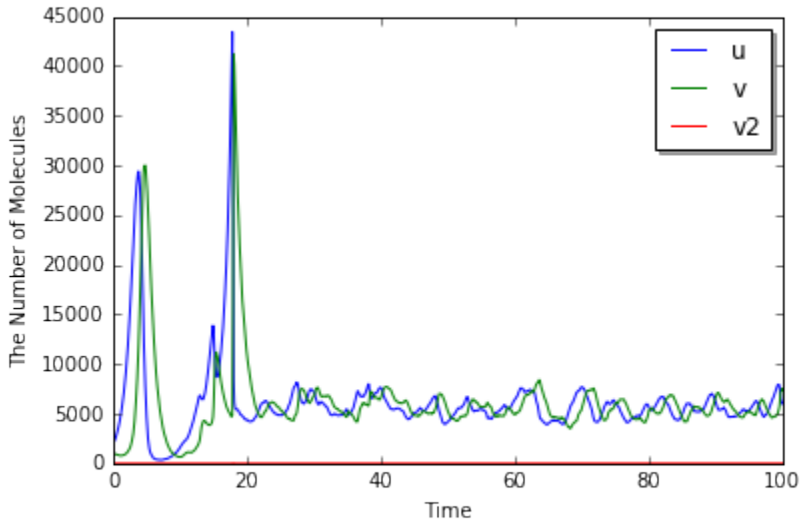
```
1600.0
```

[10]:
```
w.add_molecules(Species("u"), int(1.25 * V))
w.add_molecules(Species("v"), int(0.66 * V))
```

[11]:
```
sim = meso.MesoscopicSimulator(w)
obs1 = FixedIntervalNumberObserver(0.1, ('u', 'v', 'v2'))
obs2 = FixedIntervalHDF5Observer(2, "test%03d.h5")
```

[12]:
```
sim.run(100, (obs1, obs2))
```

[13]:
```
viz.plot_number_observer(obs1)
```



[14]:
```
viz.plot_world(w, radius=0.2)
```

```
<IPython.core.display.HTML object>
```

[15]:
```
viz.plot_movie_with_attractive_mpl(
    obs2, linewidth=0, noaxis=True, figsize=6, whratio=1.4,
    angle=(-90, 90, 6), bitrate='10M')
```

```
<IPython.core.display.HTML object>
```

[16]:
```
# from zipfile import ZipFile
# with ZipFile('test.zip', 'w') as myzip:
#     for i in range(obs2.num_steps()):
#         myzip.write('test{:03d}.h5'.format(i))
```

2.9 MinDE System with Mesoscopic Simulator

Fange D, Elf J (2006) Noise-Induced Min Phenotypes in E. coli. PLoS Comput Biol 2(6): e80. doi:10.1371/journal.pcbi.0020080

[1]:
```
%matplotlib inline
from ecell4 import *
```

Declaring `Species` and `ReactionRules`:

[2]:
```
with species_attributes():
    D | DE | {"D": "0.01", "location": "M"}
    D_ADP | D_ATP | E | {"D": "2.5", "location": "C"}

with reaction_rules():
    D_ATP + M > D | 0.0125
    D_ATP + D > D + D | 9e+6 * (1e+15 / N_A)
    D + E > DE | 5.58e+7 * (1e+15 / N_A)
    DE > D_ADP + E | 0.7
    D_ADP > D_ATP | 0.5

m = get_model()
```

Make a `World`. The second argument, `0.05`, means its subvolume length:

[3]:
```
w = meso.MesoscopicWorld(Real3(4.6, 1.1, 1.1), 0.05)
w.bind_to(m)
```

Make a structures. `Species` `C` is for cytoplasm, and `M` is for membrane:

[4]:
```
rod = Rod(3.5, 0.55, w.edge_lengths() * 0.5)
w.add_structure(Species("C"), rod)
w.add_structure(Species("M"), rod.surface())
```

Throw-in molecules:

[5]:
```
w.add_molecules(Species("D_ATP"), 2001)
w.add_molecules(Species("D_ADP"), 2001)
w.add_molecules(Species("E"), 1040)
```

Run a simulation for 120 seconds. Two `Observers` below are for logging. `obs1` logs only the number of molecules, and `obs2` does a whole state of the `World`.

[6]:
```
sim = meso.MesoscopicSimulator(w)
obs1 = FixedIntervalNumberObserver(0.1, [sp.serial() for sp in m.list_species()])
obs2 = FixedIntervalHDF5Observer(1.0, 'minde%03d.h5')
```

[7]:
```
from ecell4.util.progressbar import progressbar
```

[8]:
```
duration = 120
progressbar(sim, timeout=1).run(duration, (obs1, obs2))
```

```
  [##############################]    100.0%  Elapsed:  00:28:11 ETA:  00:00:00
```

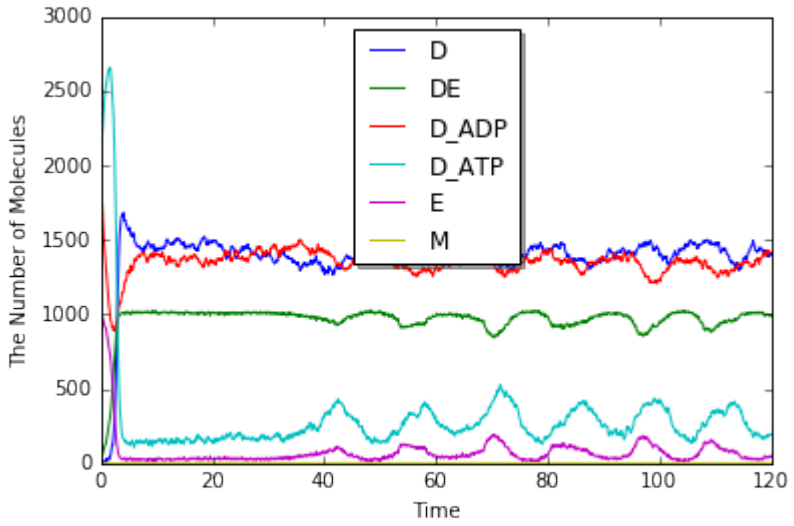
Visualize the final state of the `World`:

[9]:
```
viz.plot_world(w, radius=0.01, species_list=('D', 'DE'))
```

```
<IPython.core.display.HTML object>
```

Plot a time course of the number of molecules:

[10]:
```
viz.plot_number_observer(obs1)
```



[11]:
```
viz.plot_movie_with_matplotlib(obs2, species_list=('D', 'DE'))
```

```
<IPython.core.display.HTML object>
```

[ ]:

2.10 MinDE System with Spatiocyte Simulator

[1]:
```
%matplotlib inline
from ecell4 import *
```

Declaring Species and ReactionRules:

[2]:
```
with species_attributes():
    cytoplasm | {'radius': '1e-8', 'D': '0'}
    MinDatp | MinDadp | {'radius': '1e-8', 'D': '16e-12', 'location': 'cytoplasm'}
    MinEE_C | {'radius': '1e-8', 'D': '10e-12', 'location': 'cytoplasm'}
    membrane | {'radius': '1e-8', 'D': '0', 'location': 'cytoplasm'}
    MinD | MinEE_M | MinDEE | MinDEED | {'radius': '1e-8', 'D': '0.02e-12', 'location
↪': 'membrane'}

with reaction_rules():
    membrane + MinDatp > MinD | 2.2e-8
    MinD + MinDatp > MinD + MinD | 3e-20
    MinD + MinEE_C > MinDEE | 5e-19
    MinDEE > MinEE_M + MinDadp | 1
    MinDadp > MinDatp | 5
    MinDEE + MinD > MinDEED | 5e-15
    MinDEED > MinDEE + MinDadp | 1
    MinEE_M > MinEE_C | 0.83

m = get_model()
```

Make a World.

[3]:
```
f = spatiocyte.SpatiocyteFactory(1e-8)
w = f.create_world(Real3(4.6e-6, 1.1e-6, 1.1e-6))
w.bind_to(m)
```

Make a Structures.

[4]:
```
rod = Rod(3.5e-6, 0.51e-6, w.edge_lengths() * 0.5)
w.add_structure(Species('cytoplasm'), rod)
w.add_structure(Species('membrane'), rod.surface())
```

[4]:
```
47500
```

Throw-in molecules.

[5]:
```
w.add_molecules(Species('MinDadp'), 1300)
w.add_molecules(Species('MinDEE'), 700)
```

Run a simulation for 240 seconds.

[6]:
```
sim = f.create_simulator(m, w)
```

[7]:
```
# from functools import reduce
# alpha = reduce(lambda x, y: min(x, sim.calculate_alpha(y)), m.reaction_rules())
# sim.set_alpha(alpha)
```

[8]:
```
from ecell4.util.progressbar import progressbar
```

[9]:
```
obs1 = FixedIntervalNumberObserver(0.1, ('MinDatp', 'MinDadp', 'MinEE_C', 'MinD',
↪'MinEE_M', 'MinDEE', 'MinDEED'))
```

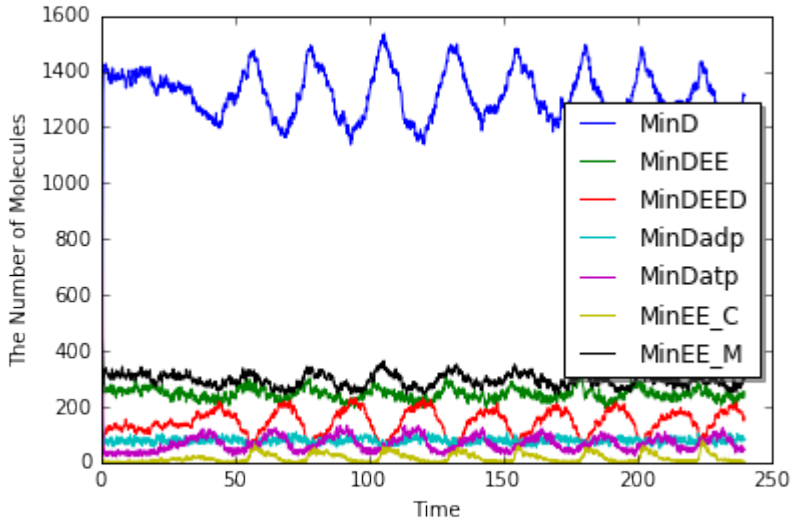
[10]:
```
obs2 = FixedIntervalHDF5Observer(1.0, "minde%03d.h5")
```

[11]:
```
duration = 240
```

[12]:
```
progressbar(sim, timeout=1).run(duration, (obs1, obs2))
```

```
  [##############################]    100.0%  Elapsed:  01:04:40 ETA:  00:00:00
```

[13]:
```
viz.plot_number_observer(obs1)
```



[14]:
```
viz.plot_movie_with_matplotlib([spatiocyte.SpatiocyteWorld("minde%03d.h5" % i) for i␣
↪in range(obs2.num_steps())], species_list=('MinD', 'MinEE_M', 'MinDEE', 'MinDEED'))
```

```
<IPython.core.display.HTML object>
```

[15]:
```
viz.plot_world(spatiocyte.SpatiocyteWorld("minde240.h5"), species_list=('MinD',
↪'MinEE_M', 'MinDEE', 'MinDEED'))
```

```
<IPython.core.display.HTML object>
```

[ ]:
```
```

2.11 Simple Equilibrium

This is a simple equilibrium model as an example. Here, we explain how to model and run a simulation without using decorators (`species_attributes` and `reaction_rules`) and `run_simulation` method.

[1]:
```
%matplotlib inline
from ecell4 import *
```

Choose one module from a list of methods supported on E-Cell4.

[2]:
```
# f = gillespie.GillespieFactory
# f = ode.ODEFactory()
# f = spatiocyte.SpatiocyteFactory()
# f = bd.BDFactory()
# f = meso.MesoscopicFactory()
f = egfrd.EGFRDFactory()
```

Set up parameters:

[3]:
```
L, N, kd, U, D, radius = 1.0, 60, 0.1, 0.5, "1", "0.01"
volume = L * L * L
ka = kd * volume * (1 - U) / (U * U * N)
```

(continues on next page)

(continued from previous page)

```
sp1, sp2, sp3 = Species("A", radius, D), Species("B", radius, D), Species("A_B",␣
↪radius, D)
rr1, rr2 = create_binding_reaction_rule(sp1, sp2, sp3, ka), create_unbinding_reaction_
↪rule(sp3, sp1, sp2, kd)
```

Create a model:

[4]:
```
m = NetworkModel()
m.add_species_attribute(sp1)
m.add_species_attribute(sp2)
m.add_species_attribute(sp3)
m.add_reaction_rule(rr1)
m.add_reaction_rule(rr2)
```

Create a world and simulator:

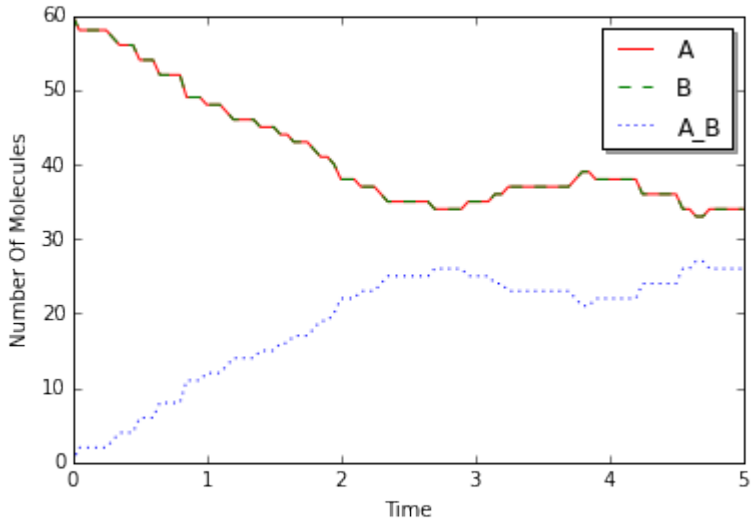
[5]:
```
w = f.create_world(Real3(L, L, L))
w.bind_to(m)
w.add_molecules(Species("A"), N)
w.add_molecules(Species("B"), N)

sim = f.create_simulator(w)
sim.set_dt(1e-3) #XXX: This is too large to get the accurate result with BDSimulator.
```

Run a simulation:

[6]:
```
next_time, dt = 0.0, 0.05
data = [(w.t(), w.num_molecules(sp1), w.num_molecules(sp2), w.num_molecules(sp3))]
for i in range(100):
    next_time += dt
    while (sim.step(next_time)): pass
    data.append((w.t(), w.num_molecules(sp1), w.num_molecules(sp2), w.num_
↪molecules(sp3)))
```

Plot with Matplotlib:

[7]:
```
import matplotlib.pylab as plt
from numpy import array

data = array(data)
plt.plot(data.T[0], data.T[1], "r-", label=sp1.serial())
plt.plot(data.T[0], data.T[2], "g--", label=sp2.serial())
plt.plot(data.T[0], data.T[3], "b:", label=sp3.serial())
plt.xlabel("Time")
plt.ylabel("Number Of Molecules")
plt.xlim(data.T[0][0], data.T[0][-1])
plt.legend(loc="best", shadow=True)
plt.show()
```



See also Reversible and Reversible (Diffusion-limited) in the Tests section for more detailed comparisons between methods.

2.12 Tyson1991

This model is described in the article:

- J.J. Tyson, "Modeling the cell division cycle: cdc2 and cyclin interactions.", Proc. Natl. Acad. Sci. U.S.A., 88(16), 7328-32, 1991.

Abstract: The proteins cdc2 and cyclin form a heterodimer (maturation promoting factor) that controls the major events of the cell cycle. A mathematical model for the interactions of cdc2 and cyclin is constructed. Simulation and analysis of the model show that the control system can operate in three modes: as a steady state with high maturation promoting factor activity, as a spontaneous oscillator, or as an excitable switch. We associate the steady state with metaphase arrest in unfertilized eggs, the spontaneous oscillations with rapid division cycles in early embryos, and the excitable switch with growth-controlled division cycles typical of nonembryonic cells.

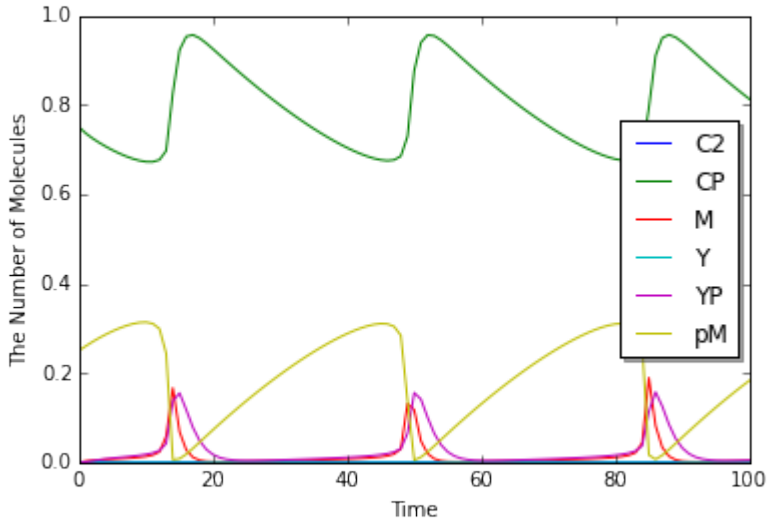
[1]:
```
%matplotlib inline
from ecell4 import *
```

[2]:
```
with reaction_rules():
    YT = Y + YP + M + pM
    CT = C2 + CP + M + pM

    ~Y > Y | 0.015 / CT
    Y > ~Y | 0.0 * Y
    CP + Y > pM | 200.0 * CP * Y / CT
    pM > M | pM * (0.018 + 180 * ((M / CT) ** 2))
    M > pM | 0.0 * M
    M > C2 + YP | 1.0 * M
    YP > ~YP | 0.6 * YP
    C2 > CP | 1000000.0 * C2
    CP > C2 | 1000.0 * CP

m = get_model()
```

[3]:
```
for rr in m.reaction_rules():
    print(rr.as_string())
```

```
C2+CP+M+pM>Y+C2+CP+M+pM|(0.015/(C2+CP+M+pM))
Y>|(0.0*Y)
CP+Y+C2+M>pM+C2+M|((200.0*CP*Y)/(C2+CP+M+pM))
pM+C2+CP>M+C2+CP|(pM*(0.018+(180*((M/(C2+CP+M+pM))**2))))
M>pM|(0.0*M)
M>C2+YP|(1.0*M)
YP>|(0.6*YP)
C2>CP|(1000000.0*C2)
CP>C2|(1000.0*CP)
```

[4]:
```
run_simulation(100.0, model=m, y0={'CP': 0.75, 'pM': 0.25})
```



CHAPTER

THREE

API

3.1 E-Cell4 core API

A submodule of ecell4_base

class ecell4_base.core.**AABB**
Bases: *ecell4_base.core.Shape*

Methods

dimension()
distance()
is_inside()
lower()
surface()
upper()

distance()

lower()

surface()

upper()

class ecell4_base.core.**AffineTransformation**
Bases: *ecell4_base.core.Shape*

Methods

dimension()
first()
is_inside()
rescale()
root()
second()
shift()
surface()
third()

Continued on next page

Table 2 – continued from previous page

translate()
xroll()
yroll()
zroll()

first()

rescale()

root()

second()

shift()

surface()

third()

translate()

xroll()

yroll()

zroll()

class ecell4_base.core.**CSVObserver**

Bases: *ecell4_base.core.Observer*

Methods

filename()
log()
next_time()
num_steps()
reset()
set_formatter()
set_header()

filename()

log()

set_formatter()

set_header()

class ecell4_base.core.**Complement**

Bases: *ecell4_base.core.Shape*

Methods

another()
dimension()
is_inside()

Continued on next page

Table 4 – continued from previous page

one()
surface()

another()

one()

surface()

class ecell4_base.core.**Cylinder**
Bases: *ecell4_base.core.Shape*

Methods

axis()
center()
dimension()
distance()
half_height()
is_inside()
surface()

axis()

center()

distance()

half_height()

surface()

class ecell4_base.core.**CylindricalSurface**
Bases: *ecell4_base.core.Shape*

Methods

axis()
center()
dimension()
distance()
half_height()
inside()
is_inside()
radius()

axis()

center()

distance()

half_height()

inside()

radius()

class ecell4_base.core.**FixedIntervalCSVObserver**
Bases: *ecell4_base.core.Observer*

Methods

filename()
log()
next_time()
num_steps()
reset()
set_formatter()
set_header()

filename()

log()

set_formatter()

set_header()

class ecell4_base.core.**FixedIntervalHDF5Observer**
Bases: *ecell4_base.core.Observer*

Methods

filename()	Overloaded function.
next_time()	
num_steps()	
prefix()	
reset()	

filename()
Overloaded function.

1. filename(self: ecell4_base.core.FixedIntervalHDF5Observer) -> str
2. filename(self: ecell4_base.core.FixedIntervalHDF5Observer, arg0: int) -> str

prefix()

class ecell4_base.core.**FixedIntervalNumberObserver**
Bases: *ecell4_base.core.Observer*

Methods

data()
next_time()
num_steps()
reset()

Continued on next page

Table 9 – continued from previous page

save()
targets()

data()

save()

targets()

class ecell4_base.core.**FixedIntervalPythonHooker**

Bases: *ecell4_base.core.Observer*

Methods

next_time()
num_steps()
reset()

class ecell4_base.core.**FixedIntervalTrackingObserver**

Bases: *ecell4_base.core.Observer*

Methods

data()
next_time()
num_steps()
num_tracers()
reset()
t()

data()

num_tracers()

t()

class ecell4_base.core.**FixedIntervalTrajectoryObserver**

Bases: *ecell4_base.core.Observer*

Methods

data()
next_time()
num_steps()
num_tracers()
reset()
t()

data()

num_tracers()

t()

class ecell4_base.core.**GSLRandomNumberGenerator**
Bases: *ecell4_base.core.RandomNumberGenerator*

Methods

binomial()	
gaussian()	
load()	
save()	
seed()	Overloaded function.
uniform()	Overloaded function.
uniform_int()	

class ecell4_base.core.**Integer3**
Bases: pybind11_builtins.pybind11_object

Attributes

col

layer

row

col

layer

row

class ecell4_base.core.**MeshSurface**
Bases: *ecell4_base.core.Shape*

Methods

dimension()	
edge_lengths()	
filename()	
is_inside()	

edge_lengths()

filename()

class ecell4_base.core.**Model**
Bases: pybind11_builtins.pybind11_object

Methods

add_reaction_rule()	
add_reaction_rules()	
add_species_attribute()	
add_species_attributes()	
apply_species_attributes()	
expand()	Overloaded function.
has_reaction_rule()	
has_species_attribute()	
list_species()	
num_reaction_rules()	
query_reaction_rules()	Overloaded function.
reaction_rules()	
remove_reaction_rule()	
remove_species_attribute()	
species_attributes()	
update_species_attribute()	

add_reaction_rule()

add_reaction_rules()

add_species_attribute()

add_species_attributes()

apply_species_attributes()

expand()
Overloaded function.

1. expand(self: ecell4_base.core.Model, arg0: List[ecell4_base.core.Species], arg1: int, arg2: Dict[ecell4_base.core.Species, int]) -> ecell4_base.core.Model
2. expand(self: ecell4_base.core.Model, arg0: List[ecell4_base.core.Species], arg1: int) -> ecell4_base.core.Model
3. expand(self: ecell4_base.core.Model, arg0: List[ecell4_base.core.Species]) -> ecell4_base.core.Model

has_reaction_rule()

has_species_attribute()

list_species()

num_reaction_rules()

query_reaction_rules()
Overloaded function.

1. query_reaction_rules(self: ecell4_base.core.Model, arg0: ecell4_base.core.Species) -> List[ecell4_base.core.ReactionRule]
2. query_reaction_rules(self: ecell4_base.core.Model, arg0: ecell4_base.core.Species, arg1: ecell4_base.core.Species) -> List[ecell4_base.core.ReactionRule]

reaction_rules()

remove_reaction_rule()

remove_species_attribute()

species_attributes()

update_species_attribute()

class ecell4_base.core.**NetfreeModel**

Bases: *ecell4_base.core.Model*

Methods

add_reaction_rule()	
add_reaction_rules()	
add_species_attribute()	
add_species_attributes()	
apply_species_attributes()	
effective()	
expand()	Overloaded function.
has_reaction_rule()	
has_species_attribute()	
list_species()	
num_reaction_rules()	
query_reaction_rules()	Overloaded function.
reaction_rules()	
remove_reaction_rule()	
remove_species_attribute()	
set_effective()	
species_attributes()	
update_species_attribute()	

effective()

set_effective()

class ecell4_base.core.**NetworkModel**

Bases: *ecell4_base.core.Model*

Methods

add_reaction_rule()	
add_reaction_rules()	
add_species_attribute()	
add_species_attributes()	
apply_species_attributes()	
expand()	Overloaded function.
has_reaction_rule()	
has_species_attribute()	
list_species()	
num_reaction_rules()	
query_reaction_rules()	Overloaded function.
reaction_rules()	
remove_reaction_rule()	
remove_species_attribute()	

Continued on next page

Table 17 – continued from previous page

species_attributes()
update_species_attribute()

class ecell4_base.core.**NumberObserver**
Bases: *ecell4_base.core.Observer*

Methods

data()
next_time()
num_steps()
reset()
save()
targets()

data()

save()

targets()

class ecell4_base.core.**Observer**
Bases: pybind11_builtins.pybind11_object

Methods

next_time()
num_steps()
reset()

next_time()

num_steps()

reset()

class ecell4_base.core.**Particle**
Bases: pybind11_builtins.pybind11_object

Methods

D()
position()
radius()
species()

D()

position()

radius()

species()

class ecell4_base.core.**ParticleID**
Bases: pybind11_builtins.pybind11_object

Methods

lot()
serial()

lot()

serial()

class ecell4_base.core.**ParticleVoxel**
Bases: pybind11_builtins.pybind11_object

Methods

D()
coordinate()
loc()
radius()
species()

D()

coordinate()

loc()

radius()

species()

class ecell4_base.core.**PlanarSurface**
Bases: *ecell4_base.core.Shape*

Methods

dimension()
e0()
e1()
is_inside()
normal()
origin()

e0()

e1()

normal()

origin()

ecell4_base.core.**Quantity**
alias of *ecell4_base.core.Quantity_Real*

class ecell4_base.core.**Quantity_Integer**
Bases: pybind11_builtins.pybind11_object

Attributes

magnitude

units

magnitude

units

class ecell4_base.core.**Quantity_Real**
Bases: pybind11_builtins.pybind11_object

Attributes

magnitude

units

magnitude

units

class ecell4_base.core.**RandomNumberGenerator**
Bases: pybind11_builtins.pybind11_object

Methods

binomial()	
gaussian()	
load()	
save()	
seed()	Overloaded function.
uniform()	Overloaded function.
uniform_int()	

binomial()

gaussian()

load()

save()

seed()
Overloaded function.

1. seed(self: ecell4_base.core.RandomNumberGenerator) -> None
2. seed(self: ecell4_base.core.RandomNumberGenerator, arg0: int) -> None

uniform()
Overloaded function.

1. uniform(self: ecell4_base.core.RandomNumberGenerator, arg0: float, arg1: float) -> float
2. uniform(self: ecell4_base.core.RandomNumberGenerator, arg0: float, arg1: float) -> float

uniform_int()

class ecell4_base.core.**ReactionRule**

Bases: pybind11_builtins.pybind11_object

Methods

add_product()	
add_reactant()	
as_string()	
count()	
generate()	
get_descriptor()	
get_k()	
has_descriptor()	
k()	
policy()	
products()	
reactants()	
reset_descriptor()	
set_descriptor()	
set_k()	Overloaded function.
set_policy()	

add_product()

add_reactant()

as_string()

count()

generate()

get_descriptor()

get_k()

has_descriptor()

k()

policy()

products()

reactants()

reset_descriptor()

set_descriptor()

set_k()

Overloaded function.

1. set_k(self: ecell4_base.core.ReactionRule, arg0: float) -> None

2. set_k(self: ecell4_base.core.ReactionRule, arg0: ecell4_base.core.Quantity_Real) -> None

set_policy()

class ecell4_base.core.**ReactionRuleDescriptor**
Bases: pybind11_builtins.pybind11_object

Methods

product_coefficients()
propensity()
reactant_coefficients()
set_product_coefficient()
set_product_coefficients()
set_reactant_coefficient()
set_reactant_coefficients()

product_coefficients()

propensity()

reactant_coefficients()

set_product_coefficient()

set_product_coefficients()

set_reactant_coefficient()

set_reactant_coefficients()

class ecell4_base.core.**ReactionRuleDescriptorMassAction**
Bases: *ecell4_base.core.ReactionRuleDescriptor*

Methods

get_k()	
k()	
product_coefficients()	
propensity()	
reactant_coefficients()	
set_k()	Overloaded function.
set_product_coefficient()	
set_product_coefficients()	
set_reactant_coefficient()	
set_reactant_coefficients()	

get_k()

k()

set_k()
Overloaded function.

1. set_k(self: ecell4_base.core.ReactionRuleDescriptorMassAction, arg0: float) -> None
2. set_k(self: ecell4_base.core.ReactionRuleDescriptorMassAction, arg0: ecell4_base.core.Quantity_Real) -> None

class ecell4_base.core.**ReactionRuleDescriptorPyfunc**

Bases: *ecell4_base.core.ReactionRuleDescriptor*

Methods

as_string()
get()
product_coefficients()
propensity()
reactant_coefficients()
set_name()
set_product_coefficient()
set_product_coefficients()
set_reactant_coefficient()
set_reactant_coefficients()

as_string()

get()

set_name()

class ecell4_base.core.**ReactionRulePolicy**

Bases: pybind11_builtins.pybind11_object

Members:

STRICT

IMPLICIT

DESTROY

Attributes

name (self: handle) -> str

DESTROY = ReactionRulePolicy.DESTROY

IMPLICIT = ReactionRulePolicy.IMPLICIT

STRICT = ReactionRulePolicy.STRICT

name

(self: handle) -> str

class ecell4_base.core.**Real3**

Bases: pybind11_builtins.pybind11_object

class ecell4_base.core.**Rod**

Bases: *ecell4_base.core.Shape*

Methods

dimension()
distance()
is_inside()
length()

Continued on next page

Table 29 – continued from previous page

origin()
radius()
shift()
surface()

distance()

length()

origin()

radius()

shift()

surface()

class ecell4_base.core.**RodSurface**
Bases: *ecell4_base.core.Shape*

Methods

dimension()
distance()
is_inside()
length()
origin()
radius()
shift()

distance()

length()

origin()

radius()

shift()

class ecell4_base.core.**Shape**
Bases: pybind11_builtins.pybind11_object

Methods

dimension()
is_inside()

dimension()

is_inside()

class ecell4_base.core.**Simulator**
Bases: pybind11_builtins.pybind11_object

Methods

check_reaction()	
dt()	
initialize()	
next_time()	
num_steps()	
set_dt()	
step()	Overloaded function.
t()	

check_reaction()

dt()

initialize()

next_time()

num_steps()

set_dt()

step()
Overloaded function.

1. step(self: ecell4_base.core.Simulator) -> None
2. step(self: ecell4_base.core.Simulator, arg0: float) -> bool

t()

class ecell4_base.core.**Species**
Bases: pybind11_builtins.pybind11_object

Methods

D()	
add_unit()	
count()	
dimension()	
get_attribute()	
has_attribute()	
list_attributes()	
location()	
radius()	
remove_attribute()	
serial()	
set_attribute()	Overloaded function.
units()	

D()

add_unit()

count()

dimension()

get_attribute()

has_attribute()

list_attributes()

location()

radius()

remove_attribute()

serial()

set_attribute()
Overloaded function.

1. set_attribute(self: ecell4_base.core.Species, arg0: str, arg1: str) -> None
2. set_attribute(self: ecell4_base.core.Species, arg0: str, arg1: str) -> None
3. set_attribute(self: ecell4_base.core.Species, arg0: str, arg1: float) -> None
4. set_attribute(self: ecell4_base.core.Species, arg0: str, arg1: int) -> None
5. set_attribute(self: ecell4_base.core.Species, arg0: str, arg1: ecell4_base.core.Quantity_Real) -> None
6. set_attribute(self: ecell4_base.core.Species, arg0: str, arg1: ecell4_base.core.Quantity_Integer) -> None

units()

class ecell4_base.core.**Sphere**
Bases: *ecell4_base.core.Shape*

Methods

center()
dimension()
distance()
is_inside()
radius()
surface()

center()

distance()

radius()

surface()

class ecell4_base.core.**SphericalSurface**
Bases: *ecell4_base.core.Shape*

Methods

center()
dimension()
distance()
inside()
is_inside()
radius()

center()

distance()

inside()

radius()

class ecell4_base.core.**Surface**
Bases: *ecell4_base.core.Shape*

Methods

dimension()
is_inside()
root()

root()

class ecell4_base.core.**TimeoutObserver**
Bases: *ecell4_base.core.Observer*

Methods

accumulation()
duration()
interval()
next_time()
num_steps()
reset()

accumulation()

duration()

interval()

class ecell4_base.core.**TimingNumberObserver**
Bases: *ecell4_base.core.Observer*

Methods

data()

Continued on next page

Table 38 – continued from previous page

next_time()
num_steps()
reset()
save()
targets()

data()

save()

targets()

class ecell4_base.core.**TimingTrajectoryObserver**
Bases: *ecell4_base.core.Observer*

Methods

data()
next_time()
num_steps()
num_tracers()
reset()
t()

data()

num_tracers()

t()

class ecell4_base.core.**Union**
Bases: *ecell4_base.core.Shape*

Methods

another()
dimension()
is_inside()
one()
surface()

another()

one()

surface()

class ecell4_base.core.**UnitSpecies**
Bases: pybind11_builtins.pybind11_object

Methods

add_site()	
deserialize()	
name()	
serial()	

add_site()

deserialize()

name()

serial()

class ecell4_base.core.**WorldInterface**

Bases: pybind11_builtins.pybind11_object

Methods

edge_lengths()	
get_particle()	
get_value()	
get_value_exact()	
has_particle()	
has_species()	
list_particles()	Overloaded function.
list_particles_exact()	
list_species()	
load()	
num_molecules()	
num_molecules_exact()	
num_particles()	Overloaded function.
num_particles_exact()	
save()	
set_t()	
t()	
volume()	

edge_lengths()

get_particle()

get_value()

get_value_exact()

has_particle()

has_species()

list_particles()

Overloaded function.

1. list_particles(self: ecell4_base.core.WorldInterface) -> List[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle]]

2. list_particles(self: ecell4_base.core.WorldInterface, arg0: ecell4_base.core.Species) -> List[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle]]

list_particles_exact()

list_species()

load()

num_molecules()

num_molecules_exact()

num_particles()
Overloaded function.

1. num_particles(self: ecell4_base.core.WorldInterface) -> int

2. num_particles(self: ecell4_base.core.WorldInterface, arg0: ecell4_base.core.Species) -> int

num_particles_exact()

save()

set_t()

t()

volume()

ecell4_base.core.**cbrt**()

ecell4_base.core.**count_species_matches**()

ecell4_base.core.**create_binding_reaction_rule**()

ecell4_base.core.**create_degradation_reaction_rule**()

ecell4_base.core.**create_synthesis_reaction_rule**()

ecell4_base.core.**create_unbinding_reaction_rule**()

ecell4_base.core.**create_unimolecular_reaction_rule**()

ecell4_base.core.**create_x_plane**()

ecell4_base.core.**create_y_plane**()

ecell4_base.core.**create_z_plane**()

ecell4_base.core.**cross_product**()

ecell4_base.core.**dot_product**()
Overloaded function.

1. dot_product(arg0: ecell4_base.core.Real3, arg1: ecell4_base.core.Real3) -> float

2. dot_product(arg0: ecell4_base.core.Integer3, arg1: ecell4_base.core.Integer3) -> int

ecell4_base.core.**format_species**()

ecell4_base.core.**integer3_abs**()

ecell4_base.core.**integer3_add**()

ecell4_base.core.**integer3_dot_product**()

ecell4_base.core.**integer3_length**()

ecell4_base.core.**integer3_length_sq**()

ecell4_base.core.**integer3_multiply**()

ecell4_base.core.**integer3_subtract**()

ecell4_base.core.**length**()
Overloaded function.

1. length(arg0: ecell4_base.core.Real3) -> float
2. length(arg0: ecell4_base.core.Integer3) -> float

ecell4_base.core.**length_sq**()
Overloaded function.

1. length_sq(arg0: ecell4_base.core.Real3) -> float
2. length_sq(arg0: ecell4_base.core.Integer3) -> int

ecell4_base.core.**load_version_information**()

ecell4_base.core.**ones**()

ecell4_base.core.**real3_abs**()

ecell4_base.core.**real3_add**()

ecell4_base.core.**real3_divide**()

ecell4_base.core.**real3_dot_product**()

ecell4_base.core.**real3_length**()

ecell4_base.core.**real3_length_sq**()

ecell4_base.core.**real3_multiply**()

ecell4_base.core.**real3_subtract**()

ecell4_base.core.**unitx**()

ecell4_base.core.**unity**()

ecell4_base.core.**unitz**()

3.2 E-Cell4 gillespie API

A submodule of ecell4_base

ecell4_base.gillespie.**Factory**
alias of *ecell4_base.gillespie.GillespieFactory*

class ecell4_base.gillespie.**GillespieFactory**
Bases: pybind11_builtins.pybind11_object

Methods

rng()	
simulator()	Overloaded function.
world()	Overloaded function.

rng()

simulator()
Overloaded function.

1. simulator(self: ecell4_base.gillespie.GillespieFactory, world: ecell4::gillespie::GillespieWorld) -> ecell4::gillespie::GillespieSimulator
2. simulator(self: ecell4_base.gillespie.GillespieFactory, world: ecell4::gillespie::GillespieWorld, model: ecell4_base.core.Model) -> ecell4::gillespie::GillespieSimulator

world()
Overloaded function.

1. world(self: ecell4_base.gillespie.GillespieFactory, edge_lengths: ecell4_base.core.Real3 = <ecell4_base.core.Real3 object at 0x7fb3e1a9c998>) -> ecell4::gillespie::GillespieWorld
2. world(self: ecell4_base.gillespie.GillespieFactory, volume: float) -> ecell4::gillespie::GillespieWorld
3. world(self: ecell4_base.gillespie.GillespieFactory, filename: str) -> ecell4::gillespie::GillespieWorld
4. world(self: ecell4_base.gillespie.GillespieFactory, model: ecell4_base.core.Model) -> ecell4::gillespie::GillespieWorld

class ecell4_base.gillespie.**GillespieSimulator**
Bases: *ecell4_base.core.Simulator*

Methods

check_reaction()	
dt()	
initialize()	
last_reactions()	
model()	
next_time()	
num_steps()	
run()	Overloaded function.
set_dt()	
set_t()	
step()	Overloaded function.
t()	
world()	

last_reactions()

model()

run()
Overloaded function.

1. run(self: ecell4_base.gillespie.GillespieSimulator, duration: float, is_dirty: bool = True) -> None
2. run(self: ecell4_base.gillespie.GillespieSimulator, duration: float, observer: ecell4_base.core.Observer, is_dirty: bool = True) -> None
3. run(self: ecell4_base.gillespie.GillespieSimulator, duration: float, observers: List[ecell4_base.core.Observer], is_dirty: bool = True) -> None

set_t()

world()

class ecell4_base.gillespie.**GillespieWorld**
Bases: *ecell4_base.core.WorldInterface*

Methods

add_molecules()	Overloaded function.
bind_to()	
edge_lengths()	
get_particle()	
get_value()	
get_value_exact()	
has_particle()	
has_species()	
list_particles()	Overloaded function.
list_particles_exact()	
list_species()	
load()	
new_particle()	Overloaded function.
num_molecules()	
num_molecules_exact()	
num_particles()	Overloaded function.
num_particles_exact()	
remove_molecules()	
rng()	
save()	
set_t()	
set_value()	
t()	
volume()	

add_molecules()
Overloaded function.

1. add_molecules(self: ecell4_base.gillespie.GillespieWorld, arg0: ecell4_base.core.Species, arg1: int) -> None
2. add_molecules(self: ecell4_base.gillespie.GillespieWorld, arg0: ecell4_base.core.Species, arg1: int, arg2: ecell4_base.core.Shape) -> None

bind_to()

new_particle()
Overloaded function.

1. new_particle(self: ecell4_base.gillespie.GillespieWorld, arg0: ecell4_base.core.Particle) -> Tuple[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle], bool]
2. new_particle(self: ecell4_base.gillespie.GillespieWorld, arg0: ecell4_base.core.Species, arg1: ecell4_base.core.Real3) -> Tuple[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle], bool]

remove_molecules()

rng()

set_value()

class ecell4_base.gillespie.**ReactionInfo**
Bases: pybind11_builtins.pybind11_object

Methods

products()
reactants()
t()

products()

reactants()

t()

ecell4_base.gillespie.**Simulator**
alias of *ecell4_base.gillespie.GillespieSimulator*

ecell4_base.gillespie.**World**
alias of *ecell4_base.gillespie.GillespieWorld*

3.3 E-Cell4 ode API

A submodule of ecell4_base

ecell4_base.ode.**Factory**
alias of *ecell4_base.ode.ODEFactory*

class ecell4_base.ode.**ODEFactory**
Bases: pybind11_builtins.pybind11_object

Methods

rng()	
simulator()	Overloaded function.
world()	Overloaded function.

rng()

simulator()
Overloaded function.

1. simulator(self: ecell4_base.ode.ODEFactory, world: ecell4::ode::ODEWorld) -> ecell4::ode::ODESimulator

2. simulator(self: ecell4_base.ode.ODEFactory, world: ecell4::ode::ODEWorld, model: ecell4_base.core.Model) -> ecell4::ode::ODESimulator

world()
Overloaded function.

1. world(self: ecell4_base.ode.ODEFactory, edge_lengths: ecell4_base.core.Real3 = <ecell4_base.core.Real3 object at 0x7fb3e1ab9298>) -> ecell4::ode::ODEWorld

2. world(self: ecell4_base.ode.ODEFactory, volume: float) -> ecell4::ode::ODEWorld

3. world(self: ecell4_base.ode.ODEFactory, filename: str) -> ecell4::ode::ODEWorld
4. world(self: ecell4_base.ode.ODEFactory, model: ecell4_base.core.Model) -> ecell4::ode::ODEWorld

class ecell4_base.ode.**ODESimulator**
Bases: *ecell4_base.core.Simulator*

Methods

absolute_tolerance()	
check_reaction()	
dt()	
initialize()	
model()	
next_time()	
num_steps()	
relative_tolerance()	
run()	Overloaded function.
set_absolute_tolerance()	
set_dt()	
set_relative_tolerance()	
set_t()	
step()	Overloaded function.
t()	
world()	

absolute_tolerance()

model()

relative_tolerance()

run()
Overloaded function.

1. run(self: ecell4_base.ode.ODESimulator, duration: float, is_dirty: bool = True) -> None
2. run(self: ecell4_base.ode.ODESimulator, duration: float, observer: ecell4_base.core.Observer, is_dirty: bool = True) -> None
3. run(self: ecell4_base.ode.ODESimulator, duration: float, observers: List[ecell4_base.core.Observer], is_dirty: bool = True) -> None

set_absolute_tolerance()

set_relative_tolerance()

set_t()

world()

class ecell4_base.ode.**ODESolverType**
Bases: pybind11_builtins.pybind11_object

Members:

RUNGE_KUTTA_CASH_KARP54

ROSENBROCK4_CONTROLLER

EULER

Attributes

name (self: handle) -> str

EULER = ODESolverType.EULER

ROSENBROCK4_CONTROLLER = ODESolverType.ROSENBROCK4_CONTROLLER

RUNGE_KUTTA_CASH_KARP54 = ODESolverType.RUNGE_KUTTA_CASH_KARP54

name
(self: handle) -> str

class ecell4_base.ode.**ODEWorld**
Bases: *ecell4_base.core.WorldInterface*

Methods

add_molecules()	Overloaded function.
bind_to()	
edge_lengths()	
evaluate()	
get_particle()	
get_value()	
get_value_exact()	
has_particle()	
has_species()	
list_particles()	Overloaded function.
list_particles_exact()	
list_species()	
load()	
new_particle()	Overloaded function.
num_molecules()	
num_molecules_exact()	
num_particles()	Overloaded function.
num_particles_exact()	
release_species()	
remove_molecules()	
reserve_species()	
save()	
set_t()	
set_value()	
set_volume()	
t()	
volume()	

add_molecules()
Overloaded function.

1. add_molecules(self: ecell4_base.ode.ODEWorld, arg0: ecell4_base.core.Species, arg1: float) -> None

2. add_molecules(self: ecell4_base.ode.ODEWorld, arg0: ecell4_base.core.Species, arg1: int, arg2: ecell4_base.core.Shape) -> None

bind_to()

evaluate()

new_particle()
Overloaded function.

1. new_particle(self: ecell4_base.ode.ODEWorld, arg0: ecell4_base.core.Particle) -> Tuple[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle], bool]

2. new_particle(self: ecell4_base.ode.ODEWorld, arg0: ecell4_base.core.Species, arg1: ecell4_base.core.Real3) -> Tuple[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle], bool]

release_species()

remove_molecules()

reserve_species()

set_value()

set_volume()

ecell4_base.ode.**Simulator**
alias of *ecell4_base.ode.ODESimulator*

ecell4_base.ode.**World**
alias of *ecell4_base.ode.ODEWorld*

3.4 E-Cell4 meso API

A submodule of ecell4_base

ecell4_base.meso.**Factory**
alias of *ecell4_base.meso.MesoscopicFactory*

class ecell4_base.meso.**MesoscopicFactory**
Bases: pybind11_builtins.pybind11_object

Methods

rng()	
simulator()	Overloaded function.
world()	Overloaded function.

rng()

simulator()
Overloaded function.

1. simulator(self: ecell4_base.meso.MesoscopicFactory, world: ecell4::meso::MesoscopicWorld) -> ecell4::meso::MesoscopicSimulator

2. simulator(self: ecell4_base.meso.MesoscopicFactory, world: ecell4::meso::MesoscopicWorld, model: ecell4_base.core.Model) -> ecell4::meso::MesoscopicSimulator

world()
Overloaded function.

1. world(self: ecell4_base.meso.MesoscopicFactory, edge_lengths: ecell4_base.core.Real3 = <ecell4_base.core.Real3 object at 0x7fb3e1a9c810>) -> ecell4::meso::MesoscopicWorld
2. world(self: ecell4_base.meso.MesoscopicFactory, volume: float) -> ecell4::meso::MesoscopicWorld
3. world(self: ecell4_base.meso.MesoscopicFactory, filename: str) -> ecell4::meso::MesoscopicWorld
4. world(self: ecell4_base.meso.MesoscopicFactory, model: ecell4_base.core.Model) -> ecell4::meso::MesoscopicWorld

class ecell4_base.meso.**MesoscopicSimulator**

Bases: *ecell4_base.core.Simulator*

Methods

check_reaction()	
dt()	
initialize()	
last_reactions()	
model()	
next_time()	
num_steps()	
run()	Overloaded function.
set_dt()	
set_t()	
step()	Overloaded function.
t()	
world()	

last_reactions()

model()

run()

Overloaded function.

1. run(self: ecell4_base.meso.MesoscopicSimulator, duration: float, is_dirty: bool = True) -> None
2. run(self: ecell4_base.meso.MesoscopicSimulator, duration: float, observer: ecell4_base.core.Observer, is_dirty: bool = True) -> None
3. run(self: ecell4_base.meso.MesoscopicSimulator, duration: float, observers: List[ecell4_base.core.Observer], is_dirty: bool = True) -> None

set_t()

world()

class ecell4_base.meso.**MesoscopicWorld**

Bases: *ecell4_base.core.WorldInterface*

Methods

add_molecules()	Overloaded function.
add_structure()	

Continued on next page

Table 52 – continued from previous page

bind_to()	
check_structure()	
coord2global()	
edge_lengths()	
get_occupancy()	Overloaded function.
get_particle()	
get_value()	
get_value_exact()	
get_volume()	
global2coord()	
has_particle()	
has_species()	
has_structure()	
list_coordinates()	
list_coordinates_exact()	
list_particles()	Overloaded function.
list_particles_exact()	
list_species()	
load()	
matrix_sizes()	
new_particle()	Overloaded function.
num_molecules()	Overloaded function.
num_molecules_exact()	
num_particles()	Overloaded function.
num_particles_exact()	
num_subvolumes()	Overloaded function.
on_structure()	
position2coordinate()	
position2global()	
remove_molecules()	Overloaded function.
rng()	
save()	
set_t()	
set_value()	
subvolume()	
subvolume_edge_lengths()	
t()	
volume()	

add_molecules()
Overloaded function.

1. add_molecules(self: ecell4_base.meso.MesoscopicWorld, arg0: ecell4_base.core.Species, arg1: int) -> None
2. add_molecules(self: ecell4_base.meso.MesoscopicWorld, arg0: ecell4_base.core.Species, arg1: int, arg2: int) -> None
3. add_molecules(self: ecell4_base.meso.MesoscopicWorld, arg0: ecell4_base.core.Species, arg1: int, arg2: ecell4_base.core.Integer3) -> None
4. add_molecules(self: ecell4_base.meso.MesoscopicWorld, arg0: ecell4_base.core.Species, arg1: int, arg2: ecell4_base.core.Shape) -> None

add_structure()

bind_to()

check_structure()

coord2global()

get_occupancy()
Overloaded function.

1. get_occupancy(self: ecell4_base.meso.MesoscopicWorld, arg0: ecell4_base.core.Species, arg1: int) -> float

2. get_occupancy(self: ecell4_base.meso.MesoscopicWorld, arg0: ecell4_base.core.Species, arg1: ecell4_base.core.Integer3) -> float

get_volume()

global2coord()

has_structure()

list_coordinates()

list_coordinates_exact()

matrix_sizes()

new_particle()
Overloaded function.

1. new_particle(self: ecell4_base.meso.MesoscopicWorld, arg0: ecell4_base.core.Particle) -> Tuple[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle], bool]

2. new_particle(self: ecell4_base.meso.MesoscopicWorld, arg0: ecell4_base.core.Species, arg1: ecell4_base.core.Real3) -> Tuple[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle], bool]

num_molecules()
Overloaded function.

1. num_molecules(self: ecell4_base.meso.MesoscopicWorld, arg0: ecell4_base.core.Species) -> int

2. num_molecules(self: ecell4_base.meso.MesoscopicWorld, arg0: ecell4_base.core.Species, arg1: int) -> int

3. num_molecules(self: ecell4_base.meso.MesoscopicWorld, arg0: ecell4_base.core.Species, arg1: ecell4_base.core.Integer3) -> int

num_subvolumes()
Overloaded function.

1. num_subvolumes(self: ecell4_base.meso.MesoscopicWorld) -> int

2. num_subvolumes(self: ecell4_base.meso.MesoscopicWorld, arg0: ecell4_base.core.Species) -> int

on_structure()

position2coordinate()

position2global()

remove_molecules()
Overloaded function.

1. remove_molecules(self: ecell4_base.meso.MesoscopicWorld, arg0: ecell4_base.core.Species, arg1: int) -> None

2. remove_molecules(self: ecell4_base.meso.MesoscopicWorld, arg0: ecell4_base.core.Species, arg1: int, arg2: ecell4_base.core.Integer3) -> None
3. remove_molecules(self: ecell4_base.meso.MesoscopicWorld, arg0: ecell4_base.core.Species, arg1: int, arg2: int) -> None

rng()

set_value()

subvolume()

subvolume_edge_lengths()

class ecell4_base.meso.**ReactionInfo**
Bases: pybind11_builtins.pybind11_object

Methods

coordinate()
products()
reactants()
t()

coordinate()

products()

reactants()

t()

ecell4_base.meso.**Simulator**
alias of *ecell4_base.meso.MesoscopicSimulator*

ecell4_base.meso.**World**
alias of *ecell4_base.meso.MesoscopicWorld*

3.5 E-Cell4 spatiocyte API

A submodule of ecell4_base

ecell4_base.spatiocyte.**Factory**
alias of *ecell4_base.spatiocyte.SpatiocyteFactory*

class ecell4_base.spatiocyte.**ReactionInfo**
Bases: pybind11_builtins.pybind11_object

Methods

products()
reactants()
t()

products()

reactants()

t()

class ecell4_base.spatiocyte.**ReactionInfoItem**
Bases: pybind11_builtins.pybind11_object

Attributes

pid

species

voxel

pid

species

voxel

ecell4_base.spatiocyte.**Simulator**
alias of *ecell4_base.spatiocyte.SpatiocyteSimulator*

class ecell4_base.spatiocyte.**SpatiocyteFactory**
Bases: pybind11_builtins.pybind11_object

Methods

rng()	
simulator()	Overloaded function.
world()	Overloaded function.

rng()

simulator()
Overloaded function.

1. simulator(self: ecell4_base.spatiocyte.SpatiocyteFactory, world: ecell4::spatiocyte::SpatiocyteWorld) -> ecell4::spatiocyte::SpatiocyteSimulator
2. simulator(self: ecell4_base.spatiocyte.SpatiocyteFactory, world: ecell4::spatiocyte::SpatiocyteWorld, model: ecell4_base.core.Model) -> ecell4::spatiocyte::SpatiocyteSimulator

world()
Overloaded function.

1. world(self: ecell4_base.spatiocyte.SpatiocyteFactory, edge_lengths: ecell4_base.core.Real3 = <ecell4_base.core.Real3 object at 0x7fb3e1ab9b90>) -> ecell4::spatiocyte::SpatiocyteWorld
2. world(self: ecell4_base.spatiocyte.SpatiocyteFactory, volume: float) -> ecell4::spatiocyte::SpatiocyteWorld
3. world(self: ecell4_base.spatiocyte.SpatiocyteFactory, filename: str) -> ecell4::spatiocyte::SpatiocyteWorld
4. world(self: ecell4_base.spatiocyte.SpatiocyteFactory, model: ecell4_base.core.Model) -> ecell4::spatiocyte::SpatiocyteWorld

class ecell4_base.spatiocyte.**SpatiocyteSimulator**
Bases: *ecell4_base.core.Simulator*

Methods

check_reaction()	
dt()	
initialize()	
last_reactions()	
model()	
next_time()	
num_steps()	
run()	Overloaded function.
set_dt()	
set_t()	
step()	Overloaded function.
t()	
world()	

last_reactions()

model()

run()
Overloaded function.

1. run(self: ecell4_base.spatiocyte.SpatiocyteSimulator, duration: float, is_dirty: bool = True) -> None
2. run(self: ecell4_base.spatiocyte.SpatiocyteSimulator, duration: float, observer: ecell4_base.core.Observer, is_dirty: bool = True) -> None
3. run(self: ecell4_base.spatiocyte.SpatiocyteSimulator, duration: float, observers: List[ecell4_base.core.Observer], is_dirty: bool = True) -> None

set_t()

world()

class ecell4_base.spatiocyte.**SpatiocyteWorld**
Bases: *ecell4_base.core.WorldInterface*

Methods

add_molecules()	Overloaded function.
add_structure()	
bind_to()	
calculate_hcp_lengths()	
calculate_shape()	
calculate_volume()	
calculate_voxel_volume()	
edge_lengths()	
get_particle()	
get_value()	
get_value_exact()	
get_volume()	
get_voxel()	
get_voxel_at()	

Continued on next page

Table 57 – continued from previous page

get_voxel_near_by()	
has_particle()	
has_species()	
list_non_structure_particles()	
list_particles()	Overloaded function.
list_particles_exact()	
list_species()	
list_structure_particles()	
load()	
new_particle()	Overloaded function.
new_voxel()	
new_voxel_structure()	
num_molecules()	
num_molecules_exact()	
num_particles()	Overloaded function.
num_particles_exact()	
remove_molecules()	
remove_particle()	
rng()	
save()	
set_t()	
set_value()	
shape()	
size()	
t()	
update_particle()	
volume()	
voxel_radius()	
voxel_volume()	

add_molecules()

Overloaded function.

1. add_molecules(self: ecell4_base.spatiocyte.SpatiocyteWorld, arg0: ecell4_base.core.Species, arg1: int) -> bool
2. add_molecules(self: ecell4_base.spatiocyte.SpatiocyteWorld, arg0: ecell4_base.core.Species, arg1: int, arg2: ecell4_base.core.Shape) -> bool

add_structure()

bind_to()

calculate_hcp_lengths()

calculate_shape()

calculate_volume()

calculate_voxel_volume()

get_volume()

get_voxel()

get_voxel_at()

get_voxel_near_by()

list_non_structure_particles()

list_structure_particles()

new_particle()
Overloaded function.

1. new_particle(self: ecell4_base.spatiocyte.SpatiocyteWorld, arg0: ecell4_base.core.Particle) -> Optional[ecell4_base.core.ParticleID]

2. new_particle(self: ecell4_base.spatiocyte.SpatiocyteWorld, arg0: ecell4_base.core.Species, arg1: ecell4_base.core.Real3) -> Optional[ecell4_base.core.ParticleID]

new_voxel()

new_voxel_structure()

remove_molecules()

remove_particle()

rng()

set_value()

shape()

size()

update_particle()

voxel_radius()

voxel_volume()

class ecell4_base.spatiocyte.**Voxel**
Bases: pybind11_builtins.pybind11_object

Methods

list_neighbors()
position()

list_neighbors()

position()

ecell4_base.spatiocyte.**World**
alias of *ecell4_base.spatiocyte.SpatiocyteWorld*

ecell4_base.spatiocyte.**create_spatiocyte_world_cell_list_impl**()

ecell4_base.spatiocyte.**create_spatiocyte_world_square_offlattice_impl**()

ecell4_base.spatiocyte.**create_spatiocyte_world_vector_impl**()

3.6 E-Cell4 bd API

A submodule of ecell4_base

class ecell4_base.bd.**BDFactory**
Bases: pybind11_builtins.pybind11_object

Methods

rng()	
simulator()	Overloaded function.
world()	Overloaded function.

rng()

simulator()
Overloaded function.

1. simulator(self: ecell4_base.bd.BDFactory, world: ecell4::bd::BDWorld) -> ecell4::bd::BDSimulator
2. simulator(self: ecell4_base.bd.BDFactory, world: ecell4::bd::BDWorld, model: ecell4_base.core.Model) -> ecell4::bd::BDSimulator

world()
Overloaded function.

1. world(self: ecell4_base.bd.BDFactory, edge_lengths: ecell4_base.core.Real3 = <ecell4_base.core.Real3 object at 0x7fb3e1abd570>) -> ecell4::bd::BDWorld
2. world(self: ecell4_base.bd.BDFactory, volume: float) -> ecell4::bd::BDWorld
3. world(self: ecell4_base.bd.BDFactory, filename: str) -> ecell4::bd::BDWorld
4. world(self: ecell4_base.bd.BDFactory, model: ecell4_base.core.Model) -> ecell4::bd::BDWorld

class ecell4_base.bd.**BDSimulator**
Bases: *ecell4_base.core.Simulator*

Methods

check_reaction()	
dt()	
initialize()	
last_reactions()	
model()	
next_time()	
num_steps()	
run()	Overloaded function.
set_dt()	
set_t()	
step()	Overloaded function.
t()	
world()	

last_reactions()

model()

run()
Overloaded function.

1. run(self: ecell4_base.bd.BDSimulator, duration: float, is_dirty: bool = True) -> None
2. run(self: ecell4_base.bd.BDSimulator, duration: float, observer: ecell4_base.core.Observer, is_dirty: bool = True) -> None
3. run(self: ecell4_base.bd.BDSimulator, duration: float, observers: List[ecell4_base.core.Observer], is_dirty: bool = True) -> None

set_t()

world()

class ecell4_base.bd.**BDWorld**

Bases: *ecell4_base.core.WorldInterface*

Methods

add_molecules()	Overloaded function.
apply_boundary()	
bind_to()	
distance()	
distance_sq()	
edge_lengths()	
get_particle()	
get_value()	
get_value_exact()	
has_particle()	
has_species()	
list_particles()	Overloaded function.
list_particles_exact()	
list_particles_within_radius()	Overloaded function.
list_species()	
load()	
new_particle()	Overloaded function.
num_molecules()	
num_molecules_exact()	
num_particles()	Overloaded function.
num_particles_exact()	
periodic_transpose()	
remove_molecules()	
remove_particle()	
rng()	
save()	
set_t()	
t()	
update_particle()	
volume()	

add_molecules()

Overloaded function.

1. add_molecules(self: ecell4_base.bd.BDWorld, arg0: ecell4_base.core.Species, arg1: int) -> None
2. add_molecules(self: ecell4_base.bd.BDWorld, arg0: ecell4_base.core.Species, arg1: int, arg2: ecell4_base.core.Shape) -> None

apply_boundary()

bind_to()

distance()

distance_sq()

list_particles_within_radius()
Overloaded function.

1. list_particles_within_radius(self: ecell4_base.bd.BDWorld, arg0: ecell4_base.core.Real3, arg1: float) -> List[Tuple[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle], float]]
2. list_particles_within_radius(self: ecell4_base.bd.BDWorld, arg0: ecell4_base.core.Real3, arg1: float, arg2: ecell4_base.core.ParticleID) -> List[Tuple[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle], float]]
3. list_particles_within_radius(self: ecell4_base.bd.BDWorld, arg0: ecell4_base.core.Real3, arg1: float, arg2: ecell4_base.core.ParticleID, arg3: ecell4_base.core.ParticleID) -> List[Tuple[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle], float]]

new_particle()
Overloaded function.

1. new_particle(self: ecell4_base.bd.BDWorld, arg0: ecell4_base.core.Particle) -> Tuple[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle], bool]
2. new_particle(self: ecell4_base.bd.BDWorld, arg0: ecell4_base.core.Species, arg1: ecell4_base.core.Real3) -> Tuple[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle], bool]

periodic_transpose()

remove_molecules()

remove_particle()

rng()

update_particle()

ecell4_base.bd.**Factory**
alias of *ecell4_base.bd.BDFactory*

class ecell4_base.bd.**ReactionInfo**
Bases: pybind11_builtins.pybind11_object

Methods

products()
reactants()
t()

products()

reactants()

t()

ecell4_base.bd.**Simulator**
alias of *ecell4_base.bd.BDSimulator*

ecell4_base.bd.**World**
alias of *ecell4_base.bd.BDWorld*

3.7 E-Cell4 egfrd API

A submodule of ecell4_base

class ecell4_base.egfrd.**BDFactory**
Bases: pybind11_builtins.pybind11_object

Methods

rng()	
simulator()	Overloaded function.
world()	Overloaded function.

rng()

simulator()
Overloaded function.

1. simulator(self: ecell4_base.egfrd.BDFactory, world: World<CyclicWorldTraits<double> >) -> ecell4_base.egfrd.BDSimulator
2. simulator(self: ecell4_base.egfrd.BDFactory, world: World<CyclicWorldTraits<double> >, model: ecell4_base.core.Model) -> ecell4_base.egfrd.BDSimulator

world()
Overloaded function.

1. world(self: ecell4_base.egfrd.BDFactory, edge_lengths: ecell4_base.core.Real3 = <ecell4_base.core.Real3 object at 0x7fb3e1ab19d0>) -> World<CyclicWorldTraits<double> >
2. world(self: ecell4_base.egfrd.BDFactory, volume: float) -> World<CyclicWorldTraits<double> >
3. world(self: ecell4_base.egfrd.BDFactory, filename: str) -> World<CyclicWorldTraits<double> >
4. world(self: ecell4_base.egfrd.BDFactory, model: ecell4_base.core.Model) -> World<CyclicWorldTraits<double> >

class ecell4_base.egfrd.**BDSimulator**
Bases: *ecell4_base.core.Simulator*

Methods

add_potential()	Overloaded function.
check_reaction()	
dt()	
dt_factor()	
initialize()	
last_reactions()	
model()	

Continued on next page

Table 64 – continued from previous page

next_time()	
num_steps()	
run()	Overloaded function.
set_dt()	
set_t()	
step()	Overloaded function.
t()	
world()	

add_potential()
Overloaded function.

1. add_potential(self: ecell4_base.egfrd.BDSimulator, arg0: ecell4_base.core.Species, arg1: float) -> None
2. add_potential(self: ecell4_base.egfrd.BDSimulator, arg0: ecell4_base.core.Species, arg1: ecell4_base.core.Shape) -> None
3. add_potential(self: ecell4_base.egfrd.BDSimulator, arg0: ecell4_base.core.Species, arg1: ecell4_base.core.Shape, arg2: float) -> None

dt_factor()

last_reactions()

model()

run()
Overloaded function.

1. run(self: ecell4_base.egfrd.BDSimulator, duration: float, is_dirty: bool = True) -> None
2. run(self: ecell4_base.egfrd.BDSimulator, duration: float, observer: ecell4_base.core.Observer, is_dirty: bool = True) -> None
3. run(self: ecell4_base.egfrd.BDSimulator, duration: float, observers: List[ecell4_base.core.Observer], is_dirty: bool = True) -> None

set_t()

world()

class ecell4_base.egfrd.**EGFRDFactory**
Bases: pybind11_builtins.pybind11_object

Methods

rng()	
simulator()	Overloaded function.
world()	Overloaded function.

rng()

simulator()
Overloaded function.

1. simulator(self: ecell4_base.egfrd.EGFRDFactory, world: World<CyclicWorldTraits<double> >) -> EGFRDSimulator<EGFRDSimulatorTraitsBase<World<CyclicWorldTraits<double> > > >

2. simulator(self: ecell4_base.egfrd.EGFRDFactory, world: World<CyclicWorldTraits<double> >, model: ecell4_base.core.Model) -> EGFRDSimulator<EGFRDSimulatorTraitsBase<World<CyclicWorldTraits<double> > > >

world()
Overloaded function.

1. world(self: ecell4_base.egfrd.EGFRDFactory, edge_lengths: ecell4_base.core.Real3 = <ecell4_base.core.Real3 object at 0x7fb3e1ab1768>) -> World<CyclicWorldTraits<double> >
2. world(self: ecell4_base.egfrd.EGFRDFactory, volume: float) -> World<CyclicWorldTraits<double> >
3. world(self: ecell4_base.egfrd.EGFRDFactory, filename: str) -> World<CyclicWorldTraits<double> >
4. world(self: ecell4_base.egfrd.EGFRDFactory, model: ecell4_base.core.Model) -> World<CyclicWorldTraits<double> >

class ecell4_base.egfrd.**EGFRDSimulator**
Bases: *ecell4_base.core.Simulator*

Methods

check_reaction()	
dt()	
initialize()	
last_reactions()	
model()	
next_time()	
num_steps()	
run()	Overloaded function.
set_dt()	
set_paranoiac()	
set_t()	
step()	Overloaded function.
t()	
world()	

last_reactions()

model()

run()
Overloaded function.

1. run(self: ecell4_base.egfrd.EGFRDSimulator, duration: float, is_dirty: bool = True) -> None
2. run(self: ecell4_base.egfrd.EGFRDSimulator, duration: float, observer: ecell4_base.core.Observer, is_dirty: bool = True) -> None
3. run(self: ecell4_base.egfrd.EGFRDSimulator, duration: float, observers: List[ecell4_base.core.Observer], is_dirty: bool = True) -> None

set_paranoiac()

set_t()

world()

class ecell4_base.egfrd.**EGFRDWorld**
Bases: *ecell4_base.core.WorldInterface*

Methods

add_molecules()	Overloaded function.
apply_boundary()	
bind_to()	
distance()	
edge_lengths()	
get_particle()	
get_value()	
get_value_exact()	
has_particle()	
has_species()	
list_particles()	Overloaded function.
list_particles_exact()	
list_particles_within_radius()	Overloaded function.
list_species()	
load()	
matrix_sizes()	
new_particle()	Overloaded function.
num_molecules()	
num_molecules_exact()	
num_particles()	Overloaded function.
num_particles_exact()	
remove_molecules()	
remove_particle()	
rng()	
save()	
set_t()	
set_value()	
t()	
update_particle()	
volume()	

add_molecules()
Overloaded function.

1. add_molecules(self: ecell4_base.egfrd.EGFRDWorld, arg0: ecell4_base.core.Species, arg1: int) -> None
2. add_molecules(self: ecell4_base.egfrd.EGFRDWorld, arg0: ecell4_base.core.Species, arg1: int, arg2: ecell4_base.core.Shape) -> None

apply_boundary()

bind_to()

distance()

list_particles_within_radius()
Overloaded function.

1. list_particles_within_radius(self: ecell4_base.egfrd.EGFRDWorld, arg0: ecell4_base.core.Real3, arg1: float) -> List[Tuple[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle], float]]
2. list_particles_within_radius(self: ecell4_base.egfrd.EGFRDWorld, arg0: ecell4_base.core.Real3, arg1: float, arg2: ecell4_base.core.ParticleID) -> List[Tuple[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle], float]]
3. list_particles_within_radius(self: ecell4_base.egfrd.EGFRDWorld, arg0: ecell4_base.core.Real3, arg1: float, arg2: ecell4_base.core.ParticleID, arg3: ecell4_base.core.ParticleID) -> List[Tuple[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle], float]]

matrix_sizes()

new_particle()
Overloaded function.

1. new_particle(self: ecell4_base.egfrd.EGFRDWorld, arg0: ecell4_base.core.Particle) -> Tuple[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle], bool]
2. new_particle(self: ecell4_base.egfrd.EGFRDWorld, arg0: ecell4_base.core.Species, arg1: ecell4_base.core.Real3) -> Tuple[Tuple[ecell4_base.core.ParticleID, ecell4_base.core.Particle], bool]

remove_molecules()

remove_particle()

rng()

set_value()

update_particle()

ecell4_base.egfrd.**Factory**
alias of *ecell4_base.egfrd.EGFRDFactory*

class ecell4_base.egfrd.**ReactionInfo**
Bases: pybind11_builtins.pybind11_object

Methods

products()
reactants()
t()

products()

reactants()

t()

ecell4_base.egfrd.**Simulator**
alias of *ecell4_base.egfrd.EGFRDSimulator*

ecell4_base.egfrd.**World**
alias of *ecell4_base.egfrd.EGFRDWorld*

PYTHON MODULE INDEX

e

ecell4_base.bd, 136
ecell4_base.core, 101
ecell4_base.egfrd, 140
ecell4_base.gillespie, 122
ecell4_base.meso, 128
ecell4_base.ode, 125
ecell4_base.spatiocyte, 132

INDEX

A

AABB (*class in ecell4_base.core*), 101
absolute_tolerance() (*ecell4_base.ode.ODESimulator method*), 126
accumulation() (*ecell4_base.core.TimeoutObserver method*), 118
add_molecules() (*ecell4_base.bd.BDWorld method*), 138
add_molecules() (*ecell4_base.egfrd.EGFRDWorld method*), 143
add_molecules() (*ecell4_base.gillespie.GillespieWorld method*), 124
add_molecules() (*ecell4_base.meso.MesoscopicWorld method*), 130
add_molecules() (*ecell4_base.ode.ODEWorld method*), 127
add_molecules() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 135
add_potential() (*ecell4_base.egfrd.BDSimulator method*), 141
add_product() (*ecell4_base.core.ReactionRule method*), 112
add_reactant() (*ecell4_base.core.ReactionRule method*), 112
add_reaction_rule() (*ecell4_base.core.Model method*), 107
add_reaction_rules() (*ecell4_base.core.Model method*), 107
add_site() (*ecell4_base.core.UnitSpecies method*), 120
add_species_attribute() (*ecell4_base.core.Model method*), 107
add_species_attributes() (*ecell4_base.core.Model method*), 107
add_structure() (*ecell4_base.meso.MesoscopicWorld method*), 130
add_structure() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 135
add_unit() (*ecell4_base.core.Species method*), 116
AffineTransformation (*class in ecell4_base.core*), 101
another() (*ecell4_base.core.Complement method*), 103
another() (*ecell4_base.core.Union method*), 119
apply_boundary() (*ecell4_base.bd.BDWorld method*), 138
apply_boundary() (*ecell4_base.egfrd.EGFRDWorld method*), 143
apply_species_attributes() (*ecell4_base.core.Model method*), 107
as_string() (*ecell4_base.core.ReactionRule method*), 112
as_string() (*ecell4_base.core.ReactionRuleDescriptorPyfunc method*), 114
axis() (*ecell4_base.core.Cylinder method*), 103
axis() (*ecell4_base.core.CylindricalSurface method*), 103

B

BDFactory (*class in ecell4_base.bd*), 136
BDFactory (*class in ecell4_base.egfrd*), 140
BDSimulator (*class in ecell4_base.bd*), 137
BDSimulator (*class in ecell4_base.egfrd*), 140
BDWorld (*class in ecell4_base.bd*), 138
bind_to() (*ecell4_base.bd.BDWorld method*), 139
bind_to() (*ecell4_base.egfrd.EGFRDWorld method*), 143
bind_to() (*ecell4_base.gillespie.GillespieWorld method*), 124
bind_to() (*ecell4_base.meso.MesoscopicWorld method*), 131
bind_to() (*ecell4_base.ode.ODEWorld method*), 127
bind_to() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 135
binomial() (*ecell4_base.core.RandomNumberGenerator method*), 111

C

calculate_hcp_lengths() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 135
calculate_shape() (*ecell4_base.spatiocyte.SpatiocyteWorld*

method), 135
calculate_volume() (ecell4_base.spatiocyte.SpatiocyteWorld method), 135
calculate_voxel_volume() (ecell4_base.spatiocyte.SpatiocyteWorld method), 135
cbrt() (in module ecell4_base.core), 121
center() (ecell4_base.core.Cylinder method), 103
center() (ecell4_base.core.CylindricalSurface method), 103
center() (ecell4_base.core.Sphere method), 117
center() (ecell4_base.core.SphericalSurface method), 118
check_reaction() (ecell4_base.core.Simulator method), 116
check_structure() (ecell4_base.meso.MesoscopicWorld method), 131
col (ecell4_base.core.Integer3 attribute), 106
Complement (class in ecell4_base.core), 102
coord2global() (ecell4_base.meso.MesoscopicWorld method), 131
coordinate() (ecell4_base.core.ParticleVoxel method), 110
coordinate() (ecell4_base.meso.ReactionInfo method), 132
count() (ecell4_base.core.ReactionRule method), 112
count() (ecell4_base.core.Species method), 116
count_species_matches() (in module ecell4_base.core), 121
create_binding_reaction_rule() (in module ecell4_base.core), 121
create_degradation_reaction_rule() (in module ecell4_base.core), 121
create_spatiocyte_world_cell_list_impl() (in module ecell4_base.spatiocyte), 136
create_spatiocyte_world_square_offlattice_impl() (in module ecell4_base.spatiocyte), 136
create_spatiocyte_world_vector_impl() (in module ecell4_base.spatiocyte), 136
create_synthesis_reaction_rule() (in module ecell4_base.core), 121
create_unbinding_reaction_rule() (in module ecell4_base.core), 121
create_unimolecular_reaction_rule() (in module ecell4_base.core), 121
create_x_plane() (in module ecell4_base.core), 121
create_y_plane() (in module ecell4_base.core), 121
create_z_plane() (in module ecell4_base.core), 121
cross_product() (in module ecell4_base.core), 121
CSVObserver (class in ecell4_base.core), 102
Cylinder (class in ecell4_base.core), 103
CylindricalSurface (class in ecell4_base.core), 103

D

D() (ecell4_base.core.Particle method), 109
D() (ecell4_base.core.ParticleVoxel method), 110
D() (ecell4_base.core.Species method), 116
data() (ecell4_base.core.FixedIntervalNumberObserver method), 105
data() (ecell4_base.core.FixedIntervalTrackingObserver method), 105
data() (ecell4_base.core.FixedIntervalTrajectoryObserver method), 105
data() (ecell4_base.core.NumberObserver method), 109
data() (ecell4_base.core.TimingNumberObserver method), 119
data() (ecell4_base.core.TimingTrajectoryObserver method), 119
deserialize() (ecell4_base.core.UnitSpecies method), 120
DESTROY (ecell4_base.core.ReactionRulePolicy attribute), 114
dimension() (ecell4_base.core.Shape method), 115
dimension() (ecell4_base.core.Species method), 116
distance() (ecell4_base.bd.BDWorld method), 139
distance() (ecell4_base.core.AABB method), 101
distance() (ecell4_base.core.Cylinder method), 103
distance() (ecell4_base.core.CylindricalSurface method), 103
distance() (ecell4_base.core.Rod method), 115
distance() (ecell4_base.core.RodSurface method), 115
distance() (ecell4_base.core.Sphere method), 117
distance() (ecell4_base.core.SphericalSurface method), 118
distance() (ecell4_base.egfrd.EGFRDWorld method), 143
distance_sq() (ecell4_base.bd.BDWorld method), 139
dot_product() (in module ecell4_base.core), 121
dt() (ecell4_base.core.Simulator method), 116
dt_factor() (ecell4_base.egfrd.BDSimulator method), 141
duration() (ecell4_base.core.TimeoutObserver method), 118

E

e0() (ecell4_base.core.PlanarSurface method), 110
e1() (ecell4_base.core.PlanarSurface method), 110
ecell4_base.bd (module), 136
ecell4_base.core (module), 101

ecell4_base.egfrd (*module*), 140
ecell4_base.gillespie (*module*), 122
ecell4_base.meso (*module*), 128
ecell4_base.ode (*module*), 125
ecell4_base.spatiocyte (*module*), 132
edge_lengths() (*ecell4_base.core.MeshSurface method*), 106
edge_lengths() (*ecell4_base.core.WorldInterface method*), 120
effective() (*ecell4_base.core.NetfreeModel method*), 108
EGFRDFactory (*class in ecell4_base.egfrd*), 141
EGFRDSimulator (*class in ecell4_base.egfrd*), 142
EGFRDWorld (*class in ecell4_base.egfrd*), 142
EULER (*ecell4_base.ode.ODESolverType attribute*), 127
evaluate() (*ecell4_base.ode.ODEWorld method*), 128
expand() (*ecell4_base.core.Model method*), 107

F

Factory (*in module ecell4_base.bd*), 139
Factory (*in module ecell4_base.egfrd*), 144
Factory (*in module ecell4_base.gillespie*), 122
Factory (*in module ecell4_base.meso*), 128
Factory (*in module ecell4_base.ode*), 125
Factory (*in module ecell4_base.spatiocyte*), 132
filename() (*ecell4_base.core.CSVObserver method*), 102
filename() (*ecell4_base.core.FixedIntervalCSVObserver method*), 104
filename() (*ecell4_base.core.FixedIntervalHDF5Observer method*), 104
filename() (*ecell4_base.core.MeshSurface method*), 106
first() (*ecell4_base.core.AffineTransformation method*), 102
FixedIntervalCSVObserver (*class in ecell4_base.core*), 104
FixedIntervalHDF5Observer (*class in ecell4_base.core*), 104
FixedIntervalNumberObserver (*class in ecell4_base.core*), 104
FixedIntervalPythonHooker (*class in ecell4_base.core*), 105
FixedIntervalTrackingObserver (*class in ecell4_base.core*), 105
FixedIntervalTrajectoryObserver (*class in ecell4_base.core*), 105
format_species() (*in module ecell4_base.core*), 121

G

gaussian() (*ecell4_base.core.RandomNumberGenerator method*), 111
generate() (*ecell4_base.core.ReactionRule method*), 112
get() (*ecell4_base.core.ReactionRuleDescriptorPyfunc method*), 114
get_attribute() (*ecell4_base.core.Species method*), 117
get_descriptor() (*ecell4_base.core.ReactionRule method*), 112
get_k() (*ecell4_base.core.ReactionRule method*), 112
get_k() (*ecell4_base.core.ReactionRuleDescriptorMassAction method*), 113
get_occupancy() (*ecell4_base.meso.MesoscopicWorld method*), 131
get_particle() (*ecell4_base.core.WorldInterface method*), 120
get_value() (*ecell4_base.core.WorldInterface method*), 120
get_value_exact() (*ecell4_base.core.WorldInterface method*), 120
get_volume() (*ecell4_base.meso.MesoscopicWorld method*), 131
get_volume() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 135
get_voxel() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 135
get_voxel_at() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 135
get_voxel_near_by() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 135
GillespieFactory (*class in ecell4_base.gillespie*), 122
GillespieSimulator (*class in ecell4_base.gillespie*), 123
GillespieWorld (*class in ecell4_base.gillespie*), 123
global2coord() (*ecell4_base.meso.MesoscopicWorld method*), 131
GSLRandomNumberGenerator (*class in ecell4_base.core*), 106

H

half_height() (*ecell4_base.core.Cylinder method*), 103
half_height() (*ecell4_base.core.CylindricalSurface method*), 103
has_attribute() (*ecell4_base.core.Species method*), 117
has_descriptor() (*ecell4_base.core.ReactionRule method*), 112
has_particle() (*ecell4_base.core.WorldInterface method*), 120
has_reaction_rule() (*ecell4_base.core.Model method*), 107

has_species() (*ecell4_base.core.WorldInterface method*), 120
has_species_attribute() (*ecell4_base.core.Model method*), 107
has_structure() (*ecell4_base.meso.MesoscopicWorld method*), 131

I

IMPLICIT (*ecell4_base.core.ReactionRulePolicy attribute*), 114
initialize() (*ecell4_base.core.Simulator method*), 116
inside() (*ecell4_base.core.CylindricalSurface method*), 103
inside() (*ecell4_base.core.SphericalSurface method*), 118
Integer3 (*class in ecell4_base.core*), 106
integer3_abs() (*in module ecell4_base.core*), 121
integer3_add() (*in module ecell4_base.core*), 121
integer3_dot_product() (*in module ecell4_base.core*), 121
integer3_length() (*in module ecell4_base.core*), 121
integer3_length_sq() (*in module ecell4_base.core*), 121
integer3_multiply() (*in module ecell4_base.core*), 121
integer3_subtract() (*in module ecell4_base.core*), 122
interval() (*ecell4_base.core.TimeoutObserver method*), 118
is_inside() (*ecell4_base.core.Shape method*), 115

K

k() (*ecell4_base.core.ReactionRule method*), 112
k() (*ecell4_base.core.ReactionRuleDescriptorMassAction method*), 113

L

last_reactions() (*ecell4_base.bd.BDSimulator method*), 137
last_reactions() (*ecell4_base.egfrd.BDSimulator method*), 141
last_reactions() (*ecell4_base.egfrd.EGFRDSimulator method*), 142
last_reactions() (*ecell4_base.gillespie.GillespieSimulator method*), 123
last_reactions() (*ecell4_base.meso.MesoscopicSimulator method*), 129
last_reactions() (*ecell4_base.spatiocyte.SpatiocyteSimulator method*), 134
layer (*ecell4_base.core.Integer3 attribute*), 106
length() (*ecell4_base.core.Rod method*), 115
length() (*ecell4_base.core.RodSurface method*), 115
length() (*in module ecell4_base.core*), 122
length_sq() (*in module ecell4_base.core*), 122
list_attributes() (*ecell4_base.core.Species method*), 117
list_coordinates() (*ecell4_base.meso.MesoscopicWorld method*), 131
list_coordinates_exact() (*ecell4_base.meso.MesoscopicWorld method*), 131
list_neighbors() (*ecell4_base.spatiocyte.Voxel method*), 136
list_non_structure_particles() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 135
list_particles() (*ecell4_base.core.WorldInterface method*), 120
list_particles_exact() (*ecell4_base.core.WorldInterface method*), 121
list_particles_within_radius() (*ecell4_base.bd.BDWorld method*), 139
list_particles_within_radius() (*ecell4_base.egfrd.EGFRDWorld method*), 143
list_species() (*ecell4_base.core.Model method*), 107
list_species() (*ecell4_base.core.WorldInterface method*), 121
list_structure_particles() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 136
load() (*ecell4_base.core.RandomNumberGenerator method*), 111
load() (*ecell4_base.core.WorldInterface method*), 121
load_version_information() (*in module ecell4_base.core*), 122
loc() (*ecell4_base.core.ParticleVoxel method*), 110
location() (*ecell4_base.core.Species method*), 117
log() (*ecell4_base.core.CSVObserver method*), 102
log() (*ecell4_base.core.FixedIntervalCSVObserver method*), 104
lot() (*ecell4_base.core.ParticleID method*), 110
lower() (*ecell4_base.core.AABB method*), 101

M

magnitude (*ecell4_base.core.Quantity_Integer attribute*), 111
magnitude (*ecell4_base.core.Quantity_Real attribute*), 111
matrix_sizes() (*ecell4_base.egfrd.EGFRDWorld method*), 144
matrix_sizes() (*ecell4_base.meso.MesoscopicWorld method*), 131

MeshSurface (*class in ecell4_base.core*), 106
MesoscopicFactory (*class in ecell4_base.meso*), 128
MesoscopicSimulator (*class in ecell4_base.meso*), 129
MesoscopicWorld (*class in ecell4_base.meso*), 129
Model (*class in ecell4_base.core*), 106
model() (*ecell4_base.bd.BDSimulator method*), 137
model() (*ecell4_base.egfrd.BDSimulator method*), 141
model() (*ecell4_base.egfrd.EGFRDSimulator method*), 142
model() (*ecell4_base.gillespie.GillespieSimulator method*), 123
model() (*ecell4_base.meso.MesoscopicSimulator method*), 129
model() (*ecell4_base.ode.ODESimulator method*), 126
model() (*ecell4_base.spatiocyte.SpatiocyteSimulator method*), 134

N

name (*ecell4_base.core.ReactionRulePolicy attribute*), 114
name (*ecell4_base.ode.ODESolverType attribute*), 127
name() (*ecell4_base.core.UnitSpecies method*), 120
NetfreeModel (*class in ecell4_base.core*), 108
NetworkModel (*class in ecell4_base.core*), 108
new_particle() (*ecell4_base.bd.BDWorld method*), 139
new_particle() (*ecell4_base.egfrd.EGFRDWorld method*), 144
new_particle() (*ecell4_base.gillespie.GillespieWorld method*), 124
new_particle() (*ecell4_base.meso.MesoscopicWorld method*), 131
new_particle() (*ecell4_base.ode.ODEWorld method*), 128
new_particle() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 136
new_voxel() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 136
new_voxel_structure() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 136
next_time() (*ecell4_base.core.Observer method*), 109
next_time() (*ecell4_base.core.Simulator method*), 116
normal() (*ecell4_base.core.PlanarSurface method*), 110
num_molecules() (*ecell4_base.core.WorldInterface method*), 121
num_molecules() (*ecell4_base.meso.MesoscopicWorld method*), 131
num_molecules_exact() (*ecell4_base.core.WorldInterface method*), 121
num_particles() (*ecell4_base.core.WorldInterface method*), 121
num_particles_exact() (*ecell4_base.core.WorldInterface method*), 121
num_reaction_rules() (*ecell4_base.core.Model method*), 107
num_steps() (*ecell4_base.core.Observer method*), 109
num_steps() (*ecell4_base.core.Simulator method*), 116
num_subvolumes() (*ecell4_base.meso.MesoscopicWorld method*), 131
num_tracers() (*ecell4_base.core.FixedIntervalTrackingObserver method*), 105
num_tracers() (*ecell4_base.core.FixedIntervalTrajectoryObserver method*), 105
num_tracers() (*ecell4_base.core.TimingTrajectoryObserver method*), 119
NumberObserver (*class in ecell4_base.core*), 109

O

Observer (*class in ecell4_base.core*), 109
ODEFactory (*class in ecell4_base.ode*), 125
ODESimulator (*class in ecell4_base.ode*), 126
ODESolverType (*class in ecell4_base.ode*), 126
ODEWorld (*class in ecell4_base.ode*), 127
on_structure() (*ecell4_base.meso.MesoscopicWorld method*), 131
one() (*ecell4_base.core.Complement method*), 103
one() (*ecell4_base.core.Union method*), 119
ones() (*in module ecell4_base.core*), 122
origin() (*ecell4_base.core.PlanarSurface method*), 110
origin() (*ecell4_base.core.Rod method*), 115
origin() (*ecell4_base.core.RodSurface method*), 115

P

Particle (*class in ecell4_base.core*), 109
ParticleID (*class in ecell4_base.core*), 110
ParticleVoxel (*class in ecell4_base.core*), 110
periodic_transpose() (*ecell4_base.bd.BDWorld method*), 139
pid (*ecell4_base.spatiocyte.ReactionInfoItem attribute*), 133
PlanarSurface (*class in ecell4_base.core*), 110
policy() (*ecell4_base.core.ReactionRule method*), 112
position() (*ecell4_base.core.Particle method*), 109
position() (*ecell4_base.spatiocyte.Voxel method*), 136

position2coordinate() (*ecell4_base.meso.MesoscopicWorld method*), 131
position2global() (*ecell4_base.meso.MesoscopicWorld method*), 131
prefix() (*ecell4_base.core.FixedIntervalHDF5Observer method*), 104
product_coefficients() (*ecell4_base.core.ReactionRuleDescriptor method*), 113
products() (*ecell4_base.bd.ReactionInfo method*), 139
products() (*ecell4_base.core.ReactionRule method*), 112
products() (*ecell4_base.egfrd.ReactionInfo method*), 144
products() (*ecell4_base.gillespie.ReactionInfo method*), 125
products() (*ecell4_base.meso.ReactionInfo method*), 132
products() (*ecell4_base.spatiocyte.ReactionInfo method*), 132
propensity() (*ecell4_base.core.ReactionRuleDescriptor method*), 113

Q

Quantity (*in module ecell4_base.core*), 110
Quantity_Integer (*class in ecell4_base.core*), 111
Quantity_Real (*class in ecell4_base.core*), 111
query_reaction_rules() (*ecell4_base.core.Model method*), 107

R

radius() (*ecell4_base.core.CylindricalSurface method*), 103
radius() (*ecell4_base.core.Particle method*), 109
radius() (*ecell4_base.core.ParticleVoxel method*), 110
radius() (*ecell4_base.core.Rod method*), 115
radius() (*ecell4_base.core.RodSurface method*), 115
radius() (*ecell4_base.core.Species method*), 117
radius() (*ecell4_base.core.Sphere method*), 117
radius() (*ecell4_base.core.SphericalSurface method*), 118
RandomNumberGenerator (*class in ecell4_base.core*), 111
reactant_coefficients() (*ecell4_base.core.ReactionRuleDescriptor method*), 113
reactants() (*ecell4_base.bd.ReactionInfo method*), 139
reactants() (*ecell4_base.core.ReactionRule method*), 112
reactants() (*ecell4_base.egfrd.ReactionInfo method*), 144
reactants() (*ecell4_base.gillespie.ReactionInfo method*), 125
reactants() (*ecell4_base.meso.ReactionInfo method*), 132
reactants() (*ecell4_base.spatiocyte.ReactionInfo method*), 132
reaction_rules() (*ecell4_base.core.Model method*), 107
ReactionInfo (*class in ecell4_base.bd*), 139
ReactionInfo (*class in ecell4_base.egfrd*), 144
ReactionInfo (*class in ecell4_base.gillespie*), 124
ReactionInfo (*class in ecell4_base.meso*), 132
ReactionInfo (*class in ecell4_base.spatiocyte*), 132
ReactionInfoItem (*class in ecell4_base.spatiocyte*), 133
ReactionRule (*class in ecell4_base.core*), 112
ReactionRuleDescriptor (*class in ecell4_base.core*), 112
ReactionRuleDescriptorMassAction (*class in ecell4_base.core*), 113
ReactionRuleDescriptorPyfunc (*class in ecell4_base.core*), 113
ReactionRulePolicy (*class in ecell4_base.core*), 114
Real3 (*class in ecell4_base.core*), 114
real3_abs() (*in module ecell4_base.core*), 122
real3_add() (*in module ecell4_base.core*), 122
real3_divide() (*in module ecell4_base.core*), 122
real3_dot_product() (*in module ecell4_base.core*), 122
real3_length() (*in module ecell4_base.core*), 122
real3_length_sq() (*in module ecell4_base.core*), 122
real3_multiply() (*in module ecell4_base.core*), 122
real3_subtract() (*in module ecell4_base.core*), 122
relative_tolerance() (*ecell4_base.ode.ODESimulator method*), 126
release_species() (*ecell4_base.ode.ODEWorld method*), 128
remove_attribute() (*ecell4_base.core.Species method*), 117
remove_molecules() (*ecell4_base.bd.BDWorld method*), 139
remove_molecules() (*ecell4_base.egfrd.EGFRDWorld method*), 144
remove_molecules() (*ecell4_base.gillespie.GillespieWorld method*), 124

remove_molecules() (*ecell4_base.meso.MesoscopicWorld method*), 131
remove_molecules() (*ecell4_base.ode.ODEWorld method*), 128
remove_molecules() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 136
remove_particle() (*ecell4_base.bd.BDWorld method*), 139
remove_particle() (*ecell4_base.egfrd.EGFRDWorld method*), 144
remove_particle() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 136
remove_reaction_rule() (*ecell4_base.core.Model method*), 107
remove_species_attribute() (*ecell4_base.core.Model method*), 107
rescale() (*ecell4_base.core.AffineTransformation method*), 102
reserve_species() (*ecell4_base.ode.ODEWorld method*), 128
reset() (*ecell4_base.core.Observer method*), 109
reset_descriptor() (*ecell4_base.core.ReactionRule method*), 112
rng() (*ecell4_base.bd.BDFactory method*), 137
rng() (*ecell4_base.bd.BDWorld method*), 139
rng() (*ecell4_base.egfrd.BDFactory method*), 140
rng() (*ecell4_base.egfrd.EGFRDFactory method*), 141
rng() (*ecell4_base.egfrd.EGFRDWorld method*), 144
rng() (*ecell4_base.gillespie.GillespieFactory method*), 122
rng() (*ecell4_base.gillespie.GillespieWorld method*), 124
rng() (*ecell4_base.meso.MesoscopicFactory method*), 128
rng() (*ecell4_base.meso.MesoscopicWorld method*), 132
rng() (*ecell4_base.ode.ODEFactory method*), 125
rng() (*ecell4_base.spatiocyte.SpatiocyteFactory method*), 133
rng() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 136
Rod (*class in ecell4_base.core*), 114
RodSurface (*class in ecell4_base.core*), 115
root() (*ecell4_base.core.AffineTransformation method*), 102
root() (*ecell4_base.core.Surface method*), 118
ROSENBROCK4_CONTROLLER (*ecell4_base.ode.ODESolverType attribute*), 127
row (*ecell4_base.core.Integer3 attribute*), 106
run() (*ecell4_base.bd.BDSimulator method*), 137
run() (*ecell4_base.egfrd.BDSimulator method*), 141
run() (*ecell4_base.egfrd.EGFRDSimulator method*), 142
run() (*ecell4_base.gillespie.GillespieSimulator method*), 123
run() (*ecell4_base.meso.MesoscopicSimulator method*), 129
run() (*ecell4_base.ode.ODESimulator method*), 126
run() (*ecell4_base.spatiocyte.SpatiocyteSimulator method*), 134
RUNGE_KUTTA_CASH_KARP54 (*ecell4_base.ode.ODESolverType attribute*), 127

S

save() (*ecell4_base.core.FixedIntervalNumberObserver method*), 105
save() (*ecell4_base.core.NumberObserver method*), 109
save() (*ecell4_base.core.RandomNumberGenerator method*), 111
save() (*ecell4_base.core.TimingNumberObserver method*), 119
save() (*ecell4_base.core.WorldInterface method*), 121
second() (*ecell4_base.core.AffineTransformation method*), 102
seed() (*ecell4_base.core.RandomNumberGenerator method*), 111
serial() (*ecell4_base.core.ParticleID method*), 110
serial() (*ecell4_base.core.Species method*), 117
serial() (*ecell4_base.core.UnitSpecies method*), 120
set_absolute_tolerance() (*ecell4_base.ode.ODESimulator method*), 126
set_attribute() (*ecell4_base.core.Species method*), 117
set_descriptor() (*ecell4_base.core.ReactionRule method*), 112
set_dt() (*ecell4_base.core.Simulator method*), 116
set_effective() (*ecell4_base.core.NetfreeModel method*), 108
set_formatter() (*ecell4_base.core.CSVObserver method*), 102
set_formatter() (*ecell4_base.core.FixedIntervalCSVObserver method*), 104
set_header() (*ecell4_base.core.CSVObserver method*), 102
set_header() (*ecell4_base.core.FixedIntervalCSVObserver method*), 104
set_k() (*ecell4_base.core.ReactionRule method*), 112
set_k() (*ecell4_base.core.ReactionRuleDescriptorMassAction method*), 113

set_name() (ecell4_base.core.ReactionRuleDescriptorPyfunc method), 114
set_paranoiac() (ecell4_base.egfrd.EGFRDSimulator method), 142
set_policy() (ecell4_base.core.ReactionRule method), 112
set_product_coefficient() (ecell4_base.core.ReactionRuleDescriptor method), 113
set_product_coefficients() (ecell4_base.core.ReactionRuleDescriptor method), 113
set_reactant_coefficient() (ecell4_base.core.ReactionRuleDescriptor method), 113
set_reactant_coefficients() (ecell4_base.core.ReactionRuleDescriptor method), 113
set_relative_tolerance() (ecell4_base.ode.ODESimulator method), 126
set_t() (ecell4_base.bd.BDSimulator method), 138
set_t() (ecell4_base.core.WorldInterface method), 121
set_t() (ecell4_base.egfrd.BDSimulator method), 141
set_t() (ecell4_base.egfrd.EGFRDSimulator method), 142
set_t() (ecell4_base.gillespie.GillespieSimulator method), 123
set_t() (ecell4_base.meso.MesoscopicSimulator method), 129
set_t() (ecell4_base.ode.ODESimulator method), 126
set_t() (ecell4_base.spatiocyte.SpatiocyteSimulator method), 134
set_value() (ecell4_base.egfrd.EGFRDWorld method), 144
set_value() (ecell4_base.gillespie.GillespieWorld method), 124
set_value() (ecell4_base.meso.MesoscopicWorld method), 132
set_value() (ecell4_base.ode.ODEWorld method), 128
set_value() (ecell4_base.spatiocyte.SpatiocyteWorld method), 136
set_volume() (ecell4_base.ode.ODEWorld method), 128
Shape (class in ecell4_base.core), 115
shape() (ecell4_base.spatiocyte.SpatiocyteWorld method), 136
shift() (ecell4_base.core.AffineTransformation method), 102
shift() (ecell4_base.core.Rod method), 115
shift() (ecell4_base.core.RodSurface method), 115
Simulator (class in ecell4_base.core), 115
Simulator (in module ecell4_base.bd), 139
Simulator (in module ecell4_base.egfrd), 144
Simulator (in module ecell4_base.gillespie), 125
Simulator (in module ecell4_base.meso), 132
Simulator (in module ecell4_base.ode), 128
Simulator (in module ecell4_base.spatiocyte), 133
simulator() (ecell4_base.bd.BDFactory method), 137
simulator() (ecell4_base.egfrd.BDFactory method), 140
simulator() (ecell4_base.egfrd.EGFRDFactory method), 141
simulator() (ecell4_base.gillespie.GillespieFactory method), 122
simulator() (ecell4_base.meso.MesoscopicFactory method), 128
simulator() (ecell4_base.ode.ODEFactory method), 125
simulator() (ecell4_base.spatiocyte.SpatiocyteFactory method), 133
size() (ecell4_base.spatiocyte.SpatiocyteWorld method), 136
SpatiocyteFactory (class in ecell4_base.spatiocyte), 133
SpatiocyteSimulator (class in ecell4_base.spatiocyte), 133
SpatiocyteWorld (class in ecell4_base.spatiocyte), 134
Species (class in ecell4_base.core), 116
species (ecell4_base.spatiocyte.ReactionInfoItem attribute), 133
species() (ecell4_base.core.Particle method), 109
species() (ecell4_base.core.ParticleVoxel method), 110
species_attributes() (ecell4_base.core.Model method), 107
Sphere (class in ecell4_base.core), 117
SphericalSurface (class in ecell4_base.core), 117
step() (ecell4_base.core.Simulator method), 116
STRICT (ecell4_base.core.ReactionRulePolicy attribute), 114
subvolume() (ecell4_base.meso.MesoscopicWorld method), 132
subvolume_edge_lengths() (ecell4_base.meso.MesoscopicWorld method), 132
Surface (class in ecell4_base.core), 118
surface() (ecell4_base.core.AABB method), 101
surface() (ecell4_base.core.AffineTransformation method), 102
surface() (ecell4_base.core.Complement method), 103
surface() (ecell4_base.core.Cylinder method), 103
surface() (ecell4_base.core.Rod method), 115

surface() (*ecell4_base.core.Sphere method*), 117
surface() (*ecell4_base.core.Union method*), 119

T

t() (*ecell4_base.bd.ReactionInfo method*), 139
t() (*ecell4_base.core.FixedIntervalTrackingObserver method*), 105
t() (*ecell4_base.core.FixedIntervalTrajectoryObserver method*), 106
t() (*ecell4_base.core.Simulator method*), 116
t() (*ecell4_base.core.TimingTrajectoryObserver method*), 119
t() (*ecell4_base.core.WorldInterface method*), 121
t() (*ecell4_base.egfrd.ReactionInfo method*), 144
t() (*ecell4_base.gillespie.ReactionInfo method*), 125
t() (*ecell4_base.meso.ReactionInfo method*), 132
t() (*ecell4_base.spatiocyte.ReactionInfo method*), 133
targets() (*ecell4_base.core.FixedIntervalNumberObserver method*), 105
targets() (*ecell4_base.core.NumberObserver method*), 109
targets() (*ecell4_base.core.TimingNumberObserver method*), 119
third() (*ecell4_base.core.AffineTransformation method*), 102
TimeoutObserver (*class in ecell4_base.core*), 118
TimingNumberObserver (*class in ecell4_base.core*), 118
TimingTrajectoryObserver (*class in ecell4_base.core*), 119
translate() (*ecell4_base.core.AffineTransformation method*), 102

U

uniform() (*ecell4_base.core.RandomNumberGenerator method*), 111
uniform_int() (*ecell4_base.core.RandomNumberGenerator method*), 111
Union (*class in ecell4_base.core*), 119
units (*ecell4_base.core.Quantity_Integer attribute*), 111
units (*ecell4_base.core.Quantity_Real attribute*), 111
units() (*ecell4_base.core.Species method*), 117
UnitSpecies (*class in ecell4_base.core*), 119
unitx() (*in module ecell4_base.core*), 122
unity() (*in module ecell4_base.core*), 122
unitz() (*in module ecell4_base.core*), 122
update_particle() (*ecell4_base.bd.BDWorld method*), 139
update_particle() (*ecell4_base.egfrd.EGFRDWorld method*), 144
update_particle() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 136
update_species_attribute() (*ecell4_base.core.Model method*), 108
upper() (*ecell4_base.core.AABB method*), 101

V

volume() (*ecell4_base.core.WorldInterface method*), 121
Voxel (*class in ecell4_base.spatiocyte*), 136
voxel (*ecell4_base.spatiocyte.ReactionInfoItem attribute*), 133
voxel_radius() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 136
voxel_volume() (*ecell4_base.spatiocyte.SpatiocyteWorld method*), 136

W

World (*in module ecell4_base.bd*), 139
World (*in module ecell4_base.egfrd*), 144
World (*in module ecell4_base.gillespie*), 125
World (*in module ecell4_base.meso*), 132
World (*in module ecell4_base.ode*), 128
World (*in module ecell4_base.spatiocyte*), 136
world() (*ecell4_base.bd.BDFactory method*), 137
world() (*ecell4_base.bd.BDSimulator method*), 138
world() (*ecell4_base.egfrd.BDFactory method*), 140
world() (*ecell4_base.egfrd.BDSimulator method*), 141
world() (*ecell4_base.egfrd.EGFRDFactory method*), 142
world() (*ecell4_base.egfrd.EGFRDSimulator method*), 142
world() (*ecell4_base.gillespie.GillespieFactory method*), 123
world() (*ecell4_base.gillespie.GillespieSimulator method*), 123
world() (*ecell4_base.meso.MesoscopicFactory method*), 128
world() (*ecell4_base.meso.MesoscopicSimulator method*), 129
world() (*ecell4_base.ode.ODEFactory method*), 125
world() (*ecell4_base.ode.ODESimulator method*), 126
world() (*ecell4_base.spatiocyte.SpatiocyteFactory method*), 133
world() (*ecell4_base.spatiocyte.SpatiocyteSimulator method*), 134
WorldInterface (*class in ecell4_base.core*), 120

X

xroll() (*ecell4_base.core.AffineTransformation method*), 102

Y

yroll() (*ecell4_base.core.AffineTransformation method*), 102

Z

zroll() (*ecell4_base.core.AffineTransformation method*), 102