# pysb Documentation

***Release 1.0.0***

**Jeremy Muhlich**

June 10, 2016

# Contents

PySB is a framework for building mathematical rule-based models of biochemical systems as Python programs. PySB abstracts the complex process of creating equations describing interactions among multiple proteins (or other biomolecules) into a simple and intuitive domain specific language embedded within Python. PySB accomplishes this by automatically generating sets of BNGL or Kappa rules and using the rules for simulation or analysis. PySB makes it straightforward to divide models into modules and to call libraries of reusable elements (macros) that encode standard biochemical actions. These features promote model transparency, reuse and accuracy. PySB interoperates with standard scientific Python libraries such as NumPy, SciPy and SymPy to enable model simulation and analysis.

Contents:

CHAPTER 1

# Installation

There are two different ways to install and use PySB:

1. **Download and run the virtual machine containing the complete PySB installation.** Users wishing to try out PySB, who are unfamiliar with the procedure for installing Python packages or who just want a simpler installation procedure, should choose this option.

   *OR*

2. **Install the necessary software dependencies natively on your computer.** Users who are comfortable with installing Python packages and compiling source code should choose this option.

## 1.1 Option 1: The PySB virtual machine

For easy installation, we provide a pre-configured virtual machine (VM) running the Ubuntu Linux operating system that comes with all necessary software installed. It also includes other useful software (e.g., Git, IPython, GraphViz, Kappa, OCaml), and has been designed to make getting up-to-date versions of PySB and other required packages easy. The VM will require 2GB of free hard drive space to install, plus an extra 500MB during the download and import process.

In addition to the PySB virtual machine file itself, you'll need virtualization software to run it, such as Oracle's free and open-source VirtualBox. The instructions given below are for VirtualBox, but other virtualization software such as VMWare Player (free) or Parallels can also be used. Here's the installation procedure:

1. Download VirtualBox and install it.
2. Download the PySB OVA (Open Virtualization Appliance) file. The file is approximately 500MB. Double-click the downloaded .ova file to open it in VirtualBox, if your web browser doesn't offer to do so.
3. VirtualBox will now display the Appliance Import Wizard. Click the "Import" button to continue. Note that the newly created VM will occupy about 2GB of hard drive space. Once the import is complete, you may delete the .ova file.
4. In the VirtualBox Manager window, double-click the "PySB demo" entry to launch the VM.

Now you may use the virtual machine to create and work with PySB models. All files created in the VM will be saved on a virtual disk image, and you may shut down the VM and re-launch it later from the VirtualBox Manager without losing your work. If you would like to share files between the VM and your desktop system, see the VirtualBox documentation for instructions.

## 1.2 Option 2: Installing the dependencies yourself

### 1.2.1 Required software

These are the minimum requirements needed to simulate a model and plot the results. Listed versions are the ones that are known to work well. Later versions should work but earlier versions may not. The major exception to this guideline is Python itself – see below.

- Python 2.7

  **PySB requires Python 2.7**! Earlier versions of Python (2.6 and lower) are not compatible, nor are later versions (3.x).

- SciPy 0.9
- NumPy 1.6
- SymPy 0.7
- matplotlib 1.1
- BioNetGen 2.2 (requires Perl – see below)
- Perl 5.8

  Any newer 5.x version is OK too. Mac and Linux users can use the version of Perl included with their operating system. Windows users should get Strawberry Perl from http://strawberryperl.com/.

### 1.2.2 Recommended software

- IPython: An alternate interactive Python shell, much improved over the standard one.
- Kappa: A rule-based modeling tool that can produce several useful model visualizations or perform an agent-based model simulation. PySB provides direct integration with some of these capabilities. Both the older *Kappa* (simplx/complx) and the newer *KaSim* packages are supported.

CHAPTER 2

# Tutorial

## 2.1 Introduction

This tutorial will walk you through the creation and simulation of a PySB model.

## 2.2 First steps

Once you have installed PySB, run the following commands from a Python interpreter to check that the basic functionality is working. This will define a model that synthesizes a molecule "A" at the rate of 3 copies per second, simulates that model from t=0 to 60 seconds and displays the amount of A sampled at intervals of 10 seconds:

```
>>> from pysb import *
>>> from pysb.integrate import Solver
>>> Model()
<Model '<interactive>' (monomers: 0, rules: 0, parameters: 0, compartments: 0) at ...>
>>> Monomer('A')
Monomer('A')
>>> Parameter('k', 3.0)
Parameter('k', 3.0)
>>> Rule('synthesize_A', None >> A(), k)
Rule('synthesize_A', None >> A(), k)
>>> t = [0, 10, 20, 30, 40, 50, 60]
>>> solver = Solver(model, t)
>>> solver.run()
>>> print solver.y[:, 1]
[   0.    30.    60.    90.   120.   150.   180.]
```

## 2.3 Creating a model

The example above notwithstanding, PySB model definition is not meant to be performed in an interactive environment. The proper way to create a model is to write the code in a .py file which can then be loaded interactively or in other scripts for analysis and simulation. Here are the Python statements necessary to define the model from *First steps* above. Save this code in a file named `tutorial_a.py` (you can find a copy of this file and all other named scripts from the tutorial in `pysb/examples/`):

```
from pysb import *

Model()
```

```
Monomer('A')
Parameter('k', 3.0)
Rule('synthesize_A', None >> A(), k)
```

Note that we did not import `pysb.integrate`, define the `t` variable or create a `Solver` object. These are part of model usage, not definition, so they do not belong here.

You may also be wondering why there are no assignment statements to be found. This is because every PySB model component automatically assigns itself to a variable named identically to the component's name (`A`, `k` and `synthesize_A` above), or `model` in the case of the `Model` object itself. This is not standard Python behavior but it makes models much more readable. The *Component* section below explains a bit more about this feature, and technical readers can find even more in the *Self-export* section.

## 2.4 Using a model

Now that we have created a model file, we will see how to load it and do something with it. Here is `run_tutorial_a.py`, the code corresponding to the rest of the example from *First steps*.

```
from pysb.integrate import Solver
from tutorial_a import model

t = [0, 10, 20, 30, 40, 50, 60]
solver = Solver(model, t)
solver.run()
print solver.y[:, 1]
```

The one line that's been added relative to the original listing is `from tutorial_a import model`. Since PySB models are just Python code, we use the standard python `import` mechanism to load them. The variable `model` which holds the `Model` object is explicitly chosen for import. All other model components defined in `tutorial_a.py` are accessible through `model`, so there is little need to import them separately.

## 2.5 Model creation in depth

Every model file must begin with these two lines:

```
from pysb import *
Model()
```

The first line brings in all of the Python classes needed to define a model. The second line creates an instance of the `Model` class and implicitly assigns this object to the variable `model`. We won't have to refer to `model` within the model file itself, rather this is the symbol we will later import from *other* code in order to make use of the model.

The rest of the model file will be component declarations. There are several types of components, some required and others optional. The required types are `Monomer`, `Parameter` and `Rule` – we have already encountered these in `tutorial_a.py`. The optional ones are `Observable` and `Compartment`. Each of these component types is represented by a Python class which inherits from the base class `Component`. The following sections will explain what each of these component types does in a model and how to create them.

### 2.5.1 Component

The base `Component` class is never explicitly used in a model, but it defines two pieces of basic functionality that are common to all component types. The first is a `name` attribute, which is specified as the first argument to the constructor for all subclasses of `Component`. The second is the "self-export" functionality, which automatically assigns every component to a local variable named for its `name` attribute. Self-export helps streamline model definition, making it feel much more like a domain-specific language like BNGL or Kappa. A justification for the technically-minded for this somewhat unusual behavior may be found in the *Self-export* section near the end of the tutorial.

### 2.5.2 Monomer

Monomers are the indivisible elements that will make up the molecules and complexes whose behavior you intend to model. Typically they will represent a specific protein or other biomolecule such as "EGFR" or "ATP". Monomers have a *name* (like all components) as well as a list of *sites*. Sites are named locations on the monomer which can *bind* with a site on another monomer and/or take on a *state*. Binding merely represents aggregation, not necessarily a formal chemical bond. States can range from the biochemically specific (e.g. "phosphorylated/unphosphorylated" to the generic (e.g. "active/inactive"). The site list is technically optional (as seen in `tutorial_a.py`) but only the simplest toy models will be able to get by without them.

The *`Monomer constructor`* takes a name, followed by a list of site names, and finally a dict specifying the allowable states for the sites. Sites used only for binding may be omitted from the dict.

Here we will define a monomer representing the protein **Raf**, for use in a model of the MAPK signaling cascade. We choose to give our Raf monomer two sites: **s** represents the serine residue on which it is phosphorylated by Ras to activate its own kinase activity, and **k** represents the catalytic kinase domain with which it can subsequently phosphorylate MEK. Site **s** can take on two states: 'u' for unphosphorylated and 'p' for phosphorylated:

```
Monomer('Raf', ['s', 'k'], {'s': ['u', 'p']})
```

Now let's provide a definition for **MEK**, the substrate of Raf. MEK has two serine residues at positions 218 and 222 in the amino acid sequence which are both phosphorylated by Raf. We can't call them both **s** as site names must be unique within a monomer, so we've used the residue numbers in the sites' names to distinguish them: **s218** and **s222**. MEK has a kinase domain of its own for which we've again used **k**:

```
Monomer('MEK', ['s218', 's222', 'k'], {'s218': ['u', 'p'], 's222': ['u', 'p']})
```

Adding these two monomer definitions to a new model file `tutorial_b.py` yields the following:

```
from pysb import *

Model()
Monomer('Raf', ['s', 'k'], {'s': ['u', 'p']})
Monomer('MEK', ['s218', 's222', 'k'], {'s218': ['u', 'p'], 's222': ['u', 'p']})
```

We can import this model in an interactive Python session and explore its monomers:

```
>>> from tutorial_b import model
>>> model.monomers
ComponentSet([
 Monomer('Raf', ['s', 'k'], {'s': ['u', 'p']}),
 Monomer('MEK', ['s218', 's222', 'k'], {'s218': ['u', 'p'], 's222': ['u', 'p']}),
 ])
>>> [m.name for m in model.monomers]
['Raf', 'MEK']
>>> model.monomers[0]
Monomer('Raf', ['s', 'k'], {'s': ['u', 'p']})
>>> model.monomers.keys()
['Raf', 'MEK']
>>> model.monomers['MEK']
Monomer('MEK', ['s218', 's222', 'k'], {'s218': ['u', 'p'], 's222': ['u', 'p']})
>>> model.monomers['MEK'].sites
['s218', 's222', 'k']
```

The Model class has a container for each component type, for example monomers holds the monomers. These component objects are the very same ones you defined in your model script – they were implicitly added to the model's monomers container by the self-export system. This container is a ComponentSet, a special PySB class which acts like a list, a dict and a set rolled into one, although it can only hold Component objects and can only be appended to (never deleted from). Its list personality allows you to iterate over the components or index an individual component by integer position, with the ordering of the values corresponding to the order in which the components were defined in the model. Its dict personality allows you to index an individual component with its string name and use the standard keys and items methods. The set personality allows set operations with ordering retained. For binary set operators, the left-hand operand's ordering takes precedence.

We can also access the fields of a Monomer object such as name and sites. See the PySB core (pysb.core) section of the module reference for documentation on the fields and methods of all the component classes.

## 2.5.3 Parameter

Parameters are constant numerical values that represent biological constants. A parameter can be used as a reaction rate constant, compartment volume or initial (boundary) condition for a molecular species. Other than *name*, the only other attribute of a parameter is its numerical *value*.

The *Parameter constructor* takes the name and value as its two arguments. The value is optional and defaults to 0.

Here we will define three parameters: a forward reaction rate for the binding of Raf and MEK and initial conditions for those two proteins:

```
Parameter('kf', 1e-5)
Parameter('Raf_0', 7e4)
Parameter('MEK_0', 3e6)
```

Add these parameter definitions to our tutorial_b model file to create tutorial_c.py:

```
from pysb import *

Model()
Monomer('Raf', ['s', 'k'], {'s': ['u', 'p']})
Monomer('MEK', ['s218', 's222', 'k'], {'s218': ['u', 'p'], 's222': ['u', 'p']})
Parameter('kf', 1e-5)
Parameter('Raf_0', 7e4)
Parameter('MEK_0', 3e6)
```

Then explore the parameters container:

```
>>> from tutorial_c import model
>>> model.parameters
ComponentSet([
 Parameter('kf', 1e-05),
 Parameter('Raf_0', 70000.0),
 Parameter('MEK_0', 3000000.0),
 ])
>>> model.parameters['Raf_0'].value
70000.0
```

Parameters as defined are unitless, so you'll need to maintain unit consistency on your own. Best practice is to use number of molecules for species concentrations (i.e. initial conditions) and S.I. units for everything else: unimolecular rate constants in $s^{-1}$, bimolecular rate constants in $\#molecules^{-1} \times s^{-1}$, compartment volumes in $L$, etc.

In the following sections we will see how parameters are used to build other model components.

### 2.5.4 Rules

Rules define chemical reactions between molecules and complexes. A rule consists of a *name*, a pattern describing which molecular species should act as the *reactants*, another pattern describing how reactants should be transformed into *products*, and parameters denoting the *rate constants*.

The `Rule constructor` takes a name, a `RuleExpression` containing the reactant and product patterns (more on that below) and one or two `Parameter` objects for the rate constants. It also takes several optional boolean flags as kwargs which alter the behavior of the rule in certain ways.

Rules, as described in this section, comprise the basic elements of procedural instructions that encode biochemical interactions. In its simplest form a rule is a chemical reaction that can be made general to a range of monomer states or very specific to only one kind of monomer in one kind of state. We follow the style for writing rules as described in BioNetGen but the style proposed by Kappa is quite similar with only some differences related to the implementation details (e.g. mass-action vs. stochastic simulations, compartments or no compartments, etc). We will write two rules to represent the interaction between the reactants and the products in a two-step manner.

The general pattern for a rule consists of the statement *Rule* and in parenthesis a series of statements separated by commas, namely the rule name (string), the rule interactions, and the rule parameters. The rule interactions make use of the following operators:

```
*+* operator to represent complexation
*<>* operator to represent backward/forward reaction
*>>* operator to represent forward-only reaction
*%* operator to represent a binding interaction between two species
```

To illustrate the use of the operators and the rule syntax we write the complex formation reaction with labels illustrating the parts of the rule:

```
Rule('C8_Bid_bind', C8(b=None) + Bid(b=None, S='u') <> C8(b=1) % Bid(b=1, S='u'), *[kf, kr])
          |              |     |           |         |     |    |     |             |
          |              |     |           |         |     |    |     |            parameter list
          |              |     |           |         |     |    |     |
          |              |     |           |         |     |    |    bound species
          |              |     |           |         |     |    |
          |              |     |           |         |     |   binding operator
          |              |     |           |         |     |
          |              |     |           |         |    bound species
          |              |     |           |         |
          |              |     |           |        forward/backward operator
          |              |     |           |
          |              |     |          unbound species
          |              |     |
          |              |    complexation / addition operator
          |              |
          |             unbound species
         rule name
```

The *rule name* can be any string and should be enclosed in single (') or double (") quotation marks. The species are *instances* of the mononmers in a specific state. In this case we are requiring that *C8* and *Bid* are both unbound, as we would not want any binding to occur with species that are previously bound. The *complexation* or *addition* operator tells the program that the two species are being added, that is, undergoing a transition, to form a new species as specified on the right side of the rule. The forward/backward operator states that the reaction is reversible. Finally the *binding* operator indicates that there is a bond formed between two or more species. This is indicated by the matching integer (in this case *1*) in the bonding site of both species along with the *binding* operator. If a non-reversible rule is desired, then the *forward-only* operator can be relplaced for the *forward/backward* operator.

In order to actually change the state of the Bid protein we must now edit the monomer so that have an acutal state site as follows:

```
Monomer('Bid', ['b', 'S'], {'S':['u', 't']})
```

Having added the state site we can now further specify the state of the Bid protein whe it undergoes rule-based interactions and explicitly indicate the changes of the protein state.

With this state site added, we can now go ahead and write the rules that will account for the binding step and the unbinding step as follows:

```
Rule('C8_Bid_bind', C8(b=None) + Bid(b=None, S='u') <>C8(b=1) % Bid(b=1, S='u'), kf, kr)
Rule('tBid_from_C8Bid', C8(b=1) % Bid(b=1, S='u') >> C8(b=None) % Bid(b=None, S='t'), kc)
```

As shown, the initial reactants, *C8* and *Bid* initially in the unbound state and, for Bid, in the 'u' state, undergo a complexation reaction and further a dissociation reaction to return the original *C8* protein and the *Bid* protein but now in the 't' state, indicating its truncation. Make these additions to your `mymodel.py` file. After you are done, your file should look like this:

```
# import the pysb module and all its methods and functions
from pysb import *

# instantiate a model
Model()

# declare monomers
Monomer('C8', ['b'])
Monomer('Bid', ['b', 'S'], {'S':['u', 't']})

# input the parameter values
Parameter('kf', 1.0e-07)
Parameter('kr', 1.0e-03)
Parameter('kc', 1.0)

# now input the rules
Rule('C8_Bid_bind', C8(b=None) + Bid(b=None, S=None) <> C8(b=1) % Bid(b=1, S=None), kf, kr)
Rule('tBid_from_C8Bid', C8(b=1) % Bid(b=1, S='u') >> C8(b=None) + Bid(b=None, S='t'), kc)
```

Once you are done editing your file, start your *ipython* (or *python*) interpreter and type the commands at the prompts below. Once you load your model you should be able to probe and check that you have the correct monomers, parameters, and rules. Your output should be very similar to the one presented (output shown below the `'>>>'` python prompts).:

```
>>> import mymodel as m
>>> m.model.monomers
   {'C8': Monomer(name='C8', sites=['b'], site_states={}),
   'Bid': Monomer(name='Bid', sites=['b', 'S'], site_states={'S': ['u', 't']})}
>>> model.parameters
   {'kf': Parameter(name='kf', value=1.0e-07),
    'kr': Parameter(name='kr', value=1.0e-03),
    'kc': Parameter(name='kc', value=1.0    )}
>>> m.model.rules
   {'C8_Bid_bind': Rule(name='C8_Bid_bind', reactants=C8(b=None) +
   Bid(b=None, S='u'), products=C8(b=1) % Bid(b=1, S='u'),
   rate_forward=Parameter(name='kf', value=1.0e-07),
   rate_reverse=Parameter(name='kr', value=1.0e-03)),
   'tBid_from_C8Bid': Rule(name='tBid_from_C8Bid', reactants=C8(b=1) %
   Bid(b=1, S='u'u), products=C8(b=None) + Bid(b=None, S=t),
   rate_forward=Parameter(name='kc', value=1.0))}
```

With this we are almost ready to run a simulation, all we need now is to specify the initial conditions of the system.

### 2.5.5 Observables

In our model we have two initial species (*C8* and *Bid*) and one output species (*tBid*). As shown in the (**??**) derived from the reactions above, there are four mathematical species needed to describe the evolution of the system (i.e. *C8*, *Bid*, *tBid*, and *C8:Bid*). Although this system is rather small, there are situations when we will have many more species than we care to monitor or characterize throughout the time evolution of the (**??**). In addition, it will often happen that the desirable species are combinations or sums of many other species. For this reason the rules-based engines we currently employ implemented the *Observables* call which automatically collects the necessary information and returns the desired species. In our case, we will monitor the amount of free *C8*, unbound *Bid*, and active *tBid*. To specify the observables enter the following lines in your `mymodel.py` file as follows:

```
Observable('obsC8', C8(b=None))
Observable('obsBid', Bid(b=None, S='u'))
Observable('obstBid', Bid(b=None, S='t'))
```

As shown,the observable can be a species. As we will show later the observable can also contain wild-cards and given the "don't care don't write" approach to rule-writing it can be a very powerful approach to observe activated complexes.

## 2.6 Initial conditions

Having specified the *monomers*, the *parameters* and the *rules* we have the basics of what is needed to generate a set of ODEs and run a model. From a mathematical perspective a system of ODEs can only be solved if a bound is placed on the ODEs for integration. In our case, these bounds are the initial conditions of the system that indicate how much non-zero initial species are present at time *t=0s* in the system. In our system, we only have two initial species, namely *C8* and *Bid* so we need to specify their initial concentrations. To do this we enter the following lines of code into the `mymodel.py` file:

```
Parameter('C8_0', 1000)
Parameter('Bid_0', 10000)
Initial(C8(b=None), C8_0)
Initial(Bid(b=None, S='u'), Bid_0)
```

A parameter object must be declared to specify the initial condition rather than just giving a value as shown above. Once the parameter object is declared (i.e. *C8_0* and *Bid_0*) it can be fed to the *Initial* definition. Now that we have specified the initial conditions we are basically ready to run simulations. We will add an *observables* call in the next section prior to running the simulation.

## 2.7 Simulation and analysis

By now your `mymodel.py` file should look something like this:

```
# import the pysb module and all its methods and functions
from pysb import *

# instantiate a model
Model()

# declare monomers
Monomer('C8', ['b'])
Monomer('Bid', ['b', 'S'], {'S':['u', 't']})

# input the parameter values
```

```
Parameter('kf', 1.0e-07)
Parameter('kr', 1.0e-03)
Parameter('kc', 1.0)

# now input the rules
Rule('C8_Bid_bind', C8(b=None) + Bid(b=None, S=None) <> C8(b=1) % Bid(b=1, S=None), *[kf, kr])
Rule('tBid_from_C8Bid', C8(b=1) % Bid(b=1, S='u') >> C8(b=None) + Bid(b=None, S='t'), kc)

# initial conditions
Parameter('C8_0', 1000)
Parameter('Bid_0', 10000)
Initial(C8(b=None), C8_0)
Initial(Bid(b=None, S='u'), Bid_0)

# Observables
Observable('obsC8', C8(b=None))
Observable('obsBid', Bid(b=None, S='u'))
Observable('obstBid', Bid(b=None, S='t'))
```

You can use a few commands to check that your model is defined properly. Start your *ipython* (or *python*) interpreter and enter the commands as shown below. Notice the output should be similar to the one shown (output shown below the '>>>' ` prompts):

```
>>> import mymodel as m
>>> m.model.monomers
   {'C8': Monomer(name='C8', sites=['b'], site_states={}),
    'Bid': Monomer(name='Bid', sites=['b', 'S'], site_states={'S': ['u', 't']})}
>>> m.model.parameters
   {'kf': Parameter(name='kf', value=1.0e-07),
    'kr': Parameter(name='kr', value=1.0e-03),
    'kc': Parameter(name='kc', value=1.0    ),
    'C8_0': Parameter(name='C8_0', value=1000),
    'Bid_0': Parameter(name='Bid_0', value=10000)}
>>> m.model.observables
   {'obsC8': <pysb.core.Observable object at 0x104b2c4d0>,
    'obsBid': <pysb.core.Observable object at 0x104b2c5d0>,
    'obstBid': <pysb.core.Observable object at 0x104b2c6d0>}
>>> m.model.initial_conditions
   [(C8(b=None), Parameter(name='C8_0', value=1000)), (Bid(b=None, S=u), Parameter(name='Bid_0', valu
>>> m.model.rules
   {'C8_Bid_bind': Rule(name='C8_Bid_bind', reactants=C8(b=None) +
   Bid(b=None, S=None), products=C8(b=1) % Bid(b=1, S=None),
   rate_forward=Parameter(name='kf', value=1.0e-07),    rate_reverse=Parameter(name='kr', value=1.0e-
    'tBid_from_C8Bid': Rule(name='tBid_from_C8Bid', reactants=C8(b=1)
    % Bid(b=1, S=u), products=C8(b=None) + Bid(b=None, S=t),    rate_forward=Parameter(name='kc', val
```

With this we are now ready to run a simulation! The parameter values for the simulation were taken directly from typical values in the paper about extrinsic apoptosis signaling. To run the simulation we must use a numerical integrator. Common examples include LSODA, VODE, CVODE, Matlab's ode15s, etc. We will use two *python* modules that are very useful for numerical manipulation. We have adapted the integrators in the *SciPy*[#sp]_ module to function seamlessly with PySB for integration of ODE systems. We will also be using the *PyLab*[2] package for graphing and plotting from the command line.

We will begin our simulation by loading the model from the *ipython* (or *python*) interpreter as shown below:

[2] PyLab: http://www.scipy.org/PyLab

```
>>> import mymodel as m
```

You can check that your model imported correctly by typing a few commands related to your model as shown:

```
>>> m.mymodel.monomers
>>> m.mymodel.rules
```

Both commands should return information about your model. (Hint: If you are using iPython, you can press tab twice after "m.mymodel" to tab complete and see all the possible options).

Now, we will import the *PyLab* and PySB integrator module. Enter the commands as shown below:

```
>>> from pysb.integrate import odesolve
>>> import pylab as pl
```

We have now loaded the integration engine and the graph engine into the interpreter environment. You may get some feedback from the program as some functions can be compiled at runtime for speed, depending on your operating system.Next we need to tell the integrator the time domain over which we wish to integrate the equations. For our case we will use $20000s$ of simulation time. To do this we generate an array using the *linspace* function from *PyLab*. Enter the command below:

```
>>> t = pl.linspace(0, 20000)
```

This command assigns an array in the range $[0..20000]$ to the variable $t$. You can type the name of the variable at any time to see the content of the variable. Typing the variable $t$ results in the following:

```
>>> t
array([     0.        ,    408.16326531,    816.32653061,   1224.48979592,
         1632.65306122,   2040.81632653,   2448.97959184,   2857.14285714,
         3265.30612245,   3673.46938776,   4081.63265306,   4489.79591837,
         4897.95918367,   5306.12244898,   5714.28571429,   6122.44897959,
         6530.6122449 ,   6938.7755102 ,   7346.93877551,   7755.10204082,
         8163.26530612,   8571.42857143,   8979.59183673,   9387.75510204,
         9795.91836735,  10204.08163265,  10612.24489796,  11020.40816327,
        11428.57142857,  11836.73469388,  12244.89795918,  12653.06122449,
        13061.2244898 ,  13469.3877551 ,  13877.55102041,  14285.71428571,
        14693.87755102,  15102.04081633,  15510.20408163,  15918.36734694,
        16326.53061224,  16734.69387755,  17142.85714286,  17551.02040816,
        17959.18367347,  18367.34693878,  18775.51020408,  19183.67346939,
        19591.83673469,  20000.        ])
```

These are the points at which we will get data for each ODE from the integrator. With this, we can now run our simulation. Enter the following commands to run the simulation:

```
>>> yout = odesolve(m.model, t)
```

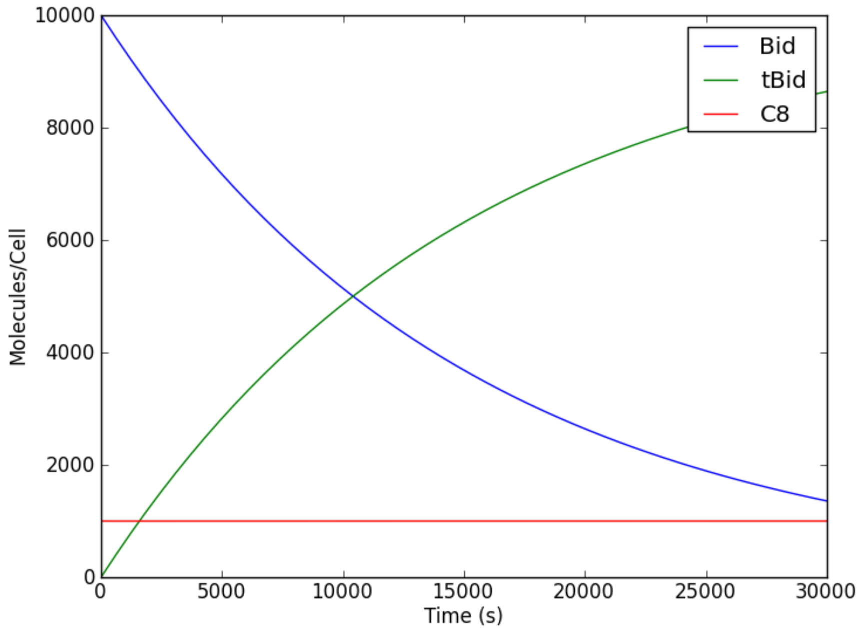
To verify that the simulation run you can see the content of the *yout* object. For example, check for the content of the *Bid* observable defined previously:

```
>>> yout['obsBid']
array([ 10000.        ,   9601.77865674,   9224.08135988,   8868.37855506,
         8534.45591732,   8221.19944491,   7927.08884234,   7650.48970981,
         7389.81105408,   7143.5816199 ,   6910.47836131,   6689.32927828,
         6479.10347845,   6278.89607041,   6087.91189021,   5905.45001654,
         5730.89003662,   5563.68044913,   5403.32856328,   5249.39176146,
         5101.47069899,   4959.20384615,   4822.26262101,   4690.34720441,
         4563.18294803,   4440.51745347,   4322.11815173,   4207.77021789,
         4097.27471952,   3990.44698008,   3887.11517373,   3787.11923497,
         3690.30945136,   3596.54594391,   3505.69733323,   3417.64025401,
         3332.25897699,   3249.44415872,   3169.09326717,   3091.10923365,
         3015.40034777,   2941.87977234,   2870.4652525 ,   2801.07879018,
         2733.64632469,   2668.09744369,   2604.36497901,   2542.38554596,
         2482.09776367,   2423.44473279])
```

As you may recall we named some observables in the *Observables* section above. The variable *yout* contains an array of all the ODE outputs from the integrators along with the named observables (i.e. *obsBid*, *obstBid*, and *obsC8*) which can be called by their names. We can therefore plot this data to visualize our output. Using the commands imported from the *PyLab* module we can create a graph interactively. Enter the commands as shown below:

```
>>> pl.ion()
>>> pl.figure()
>>> pl.plot(t, yout['obsBid'], label="Bid")
>>> pl.plot(t, yout['obstBid'], label="tBid")
>>> pl.plot(t, yout['obsC8'], label="C8")
>>> pl.legend()
>>> pl.xlabel("Time (s)")
>>> pl.ylabel("Molecules/cell")
>>> pl.show()
```

You should now have a figure in your screen showing the number of *Bid* molecules decreaing from the initial amount decreasing over time, the number of *tBid* molecules increasing over time, and the number of free *C8* molecules decrease to about half. For help with the above commands and to see more commands related to *PyLab* check the documentation [2]. Your figure should look something like the one below:



Congratulations! You have created your first model and run a simulation!

## 2.8 Visualization

It is useful to visualize the species and reactions that make a model. We have provided two methods to visualize species and reactions. We recommend using the tools in Kappa and BioNetGen for other visualization tools such as contact maps and stories.

The simplest way to visualize a model is to generate the graph file using the programs available from the command line. The files are located in the `.../pysb/tools` directory. The files to visualize reactions and species are `render_reactions.py` and `render_species.py`. These python scripts will generate `.dot` graph files that can be visualized using several tool such as OmniGraffle in `OS X` or GraphViz in all major platforms. For this tutorial we will use the GraphViz renderer. For this example will visualize the `mymodel.py` file that was created earlier. Issue the following command, replacing the comments inside square brackets‘‘[]‘‘ with the correct paths. We will first generate the `.dot` from the command line as follows:

```
[path-to-pysb]/pysb/tools/render_reactions.py [path-to-pysb-model-file]/mymodel.py > mymodel.dot
```

If your model can be properly visualized you should have gotten no errors and should now have a file called `mymodel.dot`. You can now use this file as an input for any visualization tool as described above. You can follow the same procedures with the `render_species.py` script to visualize the species generated by your models.

## 2.9 Higher-order rules

For this section we will show the power working in a programming environment by creating a simple function called “catalyze”. Catalysis happens quite often in models and it is one of the basic functions we have found useful in our model development. Rather than typing many lines such as:

```
Rule("association",  Enz(b=None) + Sub(b=None, S="i") <> Enz(b=1)%Sub(b=1,S="i"), kf, kr)
Rule("dissociation", Enz(b=1)%Sub(b=1,S="i") >> Enz(b=None) + Sub(b=None, S="a"), kc)
```

multiple times, we find it more powerful, transparent and easy to instantiate/edit a simple, one-line function call such as:

```
catalyze(Enz, Sub, "S", "i", "a", kf, kr, kc)
```

We find that the functional form captures what we mean to write: a chemical species (the substrate) undergoes catalytic activation (by the enzyme) with a given set of parameters. We will now describe how a function can be written in PySB to automate the scripting of simple concepts into a programmatic format. Examine the function below:

```
def catalyze(enz, sub, site, state1, state2, kf, kr, kc):   # (0) function call
    """2-step catalytic process"""                          # (1) reaction name
    r1_name = '%s_assoc_%s' % (enz.name, sub.name)          # (2) name of association reaction for ru
    r2_name = '%s_diss_%s' % (enz.name, sub.name)           # (3) name of dissociation reaction for r
    E = enz(b=None)                                         # (4) define enzyme state in function
    S = sub({'b': None, site: state1})                      # (5) define substrate state in function
    ES = enz(b=1) % sub({'b': 1, site: state1})             # (6) define state of enzyme:substrate co
    P = sub({'b': None, site: state2})                      # (7) define state of product
    Rule(r1_name, E + S <> ES, kf, kr)                      # (8) rule for enzyme + substrate associa
    Rule(r2_name, ES >> E + P, kc)                          # (9) rule for enzyme:substrate dissociat
```

As shown it takes about ten lines to write the catalyze function (shorter variants are certainly possible with more advanced *Python* statements). The skeleton of every function in *Python*

As shown, *Monomers*, *Parameters*, *Species*, and pretty much anything related to rules-based modeling are instantiated as objects in *Python*. One could write functions to interact with these objects and they could be instantiated and inherit methods from a class. The limits to programming biology with PySB are those enforced by the *Python* language itself. We can now go ahead and embed this into a model. Go back to your `mymodel.py` file and modify it to look something like this:

```
# import the pysb module and all its methods and functions
from pysb import *


def catalyze(enz, sub, site, state1, state2, kf, kr, kc):   # function call
```

```
    """2-step catalytic process"""                             # reaction name
    r1_name = '%s_assoc_%s' % (enz.name, sub.name)              # name of association reaction for rule
    r2_name = '%s_diss_%s' % (enz.name, sub.name)              # name of dissociation reaction for rule
    E = enz(b=None)                                            # define enzyme state in function
    S = sub({'b': None, site: state1})                         # define substrate state in function
    ES = enz(b=1) % sub({'b': 1, site: state1})                # define state of enzyme:substrate comple
    P = sub({'b': None, site: state2})                         # define state of product
    Rule(r1_name, E + S <> ES, kf, kr)                         # rule for enzyme + substrate associatio
    Rule(r2_name, ES >> E + P, kc)                             # rule for enzyme:substrate dissociation

# instantiate a model
Model()

# declare monomers
Monomer('C8', ['b'])
Monomer('Bid', ['b', 'S'], {'S':['u', 't']})

# input the parameter values
Parameter('kf', 1.0e-07)
Parameter('kr', 1.0e-03)
Parameter('kc', 1.0)

# OLD RULES
# Rule('C8_Bid_bind', C8(b=None) + Bid(b=None, S=None) <> C8(b=1) % Bid(b=1, S=None), *[kf, kr])
# Rule('tBid_from_C8Bid', C8(b=1) % Bid(b=1, S='u') >> C8(b=None) + Bid(b=None, S='t'), kc)
#
# NEW RULES
# Catalysis
catalyze(C8, Bid, 'S', 'u', 't', kf, kr, kc)


# initial conditions
Parameter('C8_0', 1000)
Parameter('Bid_0', 10000)
Initial(C8(b=None), C8_0)
Initial(Bid(b=None, S='u'), Bid_0)

# Observables
Observable('obsC8', C8(b=None))
Observable('obsBid', Bid(b=None, S='u'))
Observable('obstBid', Bid(b=None, S='t'))
```

With this you should be able to execute your code and generate figures as described in the previous sections.

## 2.10 Using provided macros

For further reference we invite the users to explore the `macros.py` file in the `.../pysb/` directory. Based on our experience with modeling signal transduction pathways we have identified a set of commonly-used constructs that can serve as building blocks for more complex models. In addition to some meta-macros useful for instantiating user macros, we provide a set of macros such as `equilibrate`. `bind`, `catalyze`, `catalyze_one_step`, `catalyze_one_step_reversible`, `synthesize`, `degrade`, `assemble_pore_sequential`, and `pore_transport`. In addition to these basic macros we also provide the higher-level macros `bind_table` and `catalyze_table` which we have found useful in instantiating the interactions between families of models.

In what follows we expand our previous model example of `Caspase-8` by adding a few more species. The initiator caspase, as was described earlier, catalytically cleaves `Bid` to create truncated `Bid` (`tBid`) in this model. This `tBid` then catalytically activates Bax and Bak which eventually go on to form pores at the mitochondria leading to mitochondrial outer-membrane permeabilization (MOMP) and eventual cell death. To introduce the concept of higher-level macros we will show how the `bind_table` macro can be used to show how a family of inhibitors, namely `Bcl-2`, `Bcl-xL`, and `Mcl-1` inhibits a family of proteins, namely `Bid`, `Bax`, and `Bak`.

In your favorite editor, go ahead and create a file (I will refer to it as ::file::*mymodel_fxns*). Many rules that dictate the interactions among species depend on a single binding site. We will begin by creating our model and declaring a generic binding site. We will also declare some functions, using the `PySB` macros and tailor them to our needs by specifying the binding site to be passed to the function. The first thing we do is import PySB and then import PySB macros. Then we declare our generic site and redefine the `pysb.macros` for our model as follows:

```
# import the pysb module and all its methods and functions
from pysb import *
from pysb.macros import *

# some functions to make life easy
site_name = 'b'
def catalyze_b(enz, sub, product, klist):
    """Alias for pysb.macros.catalyze with default binding site 'b'.
    """
    return catalyze(enz, site_name, sub, site_name, product, klist)
def bind_table_b(table):
    """Alias for pysb.macros.bind_table with default binding sites 'bf'.
    """
    return bind_table(table, site_name, site_name)
```

The first two lines just import the necessary modules from PySB. The `catalyze_b`` function, tailored for the model, takes four inputs but feeds six inputs to the `pysb.macros.catalyze` function, hence making the model more clean. Similarly the `bind_table_b` function takes only one entry, a list of lists, and feeds the entries needed to the `pysb.macros.bind_table` macro. Note that these entries could be contained in a header file to be hidden from the user at model time.

With this technical work out of the way we can now actually start our mdoel building. We will declare two sets of rates, the `bid_rates` that we will use for all the `Bid` interactions and the `bcl2_rates` which we will use for all the Bcl-2 interactions. Thesevalues could be specified individually as desired as desired but it is common practice in models to use generic values for the reaction rate parameters of a model and determine these in detail through some sort of model calibration. We will use these values for now for illustrative purposes.

The next entries for the rates, the model declaration, and the Monomers follow:

```
# Bid activation rates
bid_rates = [        1e-7, 1e-3, 1] #

# Bcl2 Inhibition Rates
bcl2_rates = [1.428571e-05, 1e-3] # 1.0e-6/v_mito

# instantiate a model
Model()

# declare monomers
Monomer('C8',     ['b'])
Monomer('Bid',    ['b', 'S'], {'S':['u', 't', 'm']})
Monomer('Bax',    ['b', 'S'], {'S':['i', 'a', 'm']})
Monomer('Bak',    ['b', 'S'], {'S':['i', 'a']})
Monomer('BclxL', ['b', 'S'], {'S':['c', 'm']})
Monomer('Bcl2', ['b'])
Monomer('Mcl1', ['b'])
```

As shown, the generic rates are declared followed by the declaration of the monomers. We have the `C8` and `Bid` monomers as we did in the initial part of the tutorial, the MOMP effectors `Bid`, `Bax`, `Bak`, and the MOMP inhibitors `Bcl-xL`, `Bcl-2`, and `Mcl-1`. The `Bid`, `Bax`, and `BclxL` monomers, in addition to the active and inactive terms also have a `'m'` term indicating that they can be in a membrane, which in this case we indicate as a state. We will have a translocation to the membrane as part of the reactions.

We can now begin to write some checmical procedures. The first procedure is the catalytic activation of `Bid` by `C8`. This is followed by the catalytic activation of Bax and Bak.

```
# Activate Bid
catalyze_b(C8, Bid(S='u'), Bid(S='t'), [KF, KR, KC])

# Activate Bax/Bak
catalyze_b(Bid(S='m'), Bax(S='i'), Bax(S='m'), bid_rates)
catalyze_b(Bid(S='m'), Bak(S='i'), Bak(S='a'), bid_rates)
```

As shown, we simply state the soecies that acts as an *enzyme* as the first function argument, the species that acts as the *reactant* with the enzyme as the second argument (along with any state specifications) and finally the *product* species. The `bid_rates` argument is the list of rates that we declared earlier.

You may have noticed a problem with the previous statements. The `Bid` species undergoes a transformation from state `S='u'` to `S='t'` but the activation of `Bax` and `Bak` happens only when `Bid` is in state `S='m'` to imply that these events only happen at the membrane. In order to transport `Bid` from the `'t'` state to the `'m'` state we need a transporf function. We achieve this by using the *equilibrate* macro in PySB between these states. In addition we use this same macro for the transport of the `Bax` species and the `BclxL` species as shown below.

```
# Bid, Bax, BclxL "transport" to the membrane
equilibrate(Bid(b=None, S='t'),   Bid(b=None, S='m'), [1e-1, 1e-3])
equilibrate(Bax(b=None, S='m'),   Bax(b=None, S='a'), [1e-1, 1e-3])
equilibrate(BclxL(b=None, S='c'), BclxL(b=None, S='m'), [1e-1, 1e-3])
```

According to published experimental data, the Bcl-2 family of inhibitors can inhibit the initiator `Bid` and the effector `Bax` and `Bak`. These family has complex interactions with all these proteins. Given that we have three inhibitors, and three molecules to be inhibited, this indicates nine interactions that need to be specified. This would involve writing nine reversible reactions in a rules language or at least eighteen reactions for each direction if we were writing the ODEs. Given that we are simply stating that these species *bind* to inhibit interactions, we can take advantage of two things. In the first case we have already seen that there is a *bind* macro specified in PySB. We can further functionalize this into a higher level macro, naemly the *bind_table* macro, which takes a table of interactions as an argument and generates the rules based on these simple interactions. We specify the bind table for the inhibitors (top row) and the inhibited molecules (left column) as follows.

```
bind_table_b([[                  Bcl2,  BclxL(S='m'),       Mcl1],
              [Bid(S='m'), bcl2_rates,  bcl2_rates,   bcl2_rates],
              [Bax(S='a'), bcl2_rates,  bcl2_rates,         None],
              [Bak(S='a'),       None,  bcl2_rates,   bcl2_rates]])
```

As shown the inhibitors interact by giving the rates of interactions or the *"None"* Python keyword to indicate no interaction. The only thing left to run this simple model is to declare some initial conditions and some observables. We declare the following:

```
# initial conditions
Parameter('C8_0',    1e4)
Parameter('Bid_0',   1e4)
Parameter('Bax_0',  .8e5)
Parameter('Bak_0',  .2e5)
Parameter('BclxL_0', 1e3)
Parameter('Bcl2_0',  1e3)
Parameter('Mcl1_0',  1e3)
```

```
Initial(C8(b=None), C8_0)
Initial(Bid(b=None, S='u'), Bid_0)
Initial(Bax(b=None, S='i'), Bax_0)
Initial(Bak(b=None, S='i'), Bak_0)
Initial(BclxL(b=None, S='c'), BclxL_0)
Initial(Bcl2(b=None), Bcl2_0)
Initial(Mcl1(b=None), Mcl1_0)

# Observables
Observable('obstBid', Bid(b=None, S='m'))
Observable('obsBax', Bax(b=None, S='a'))
Observable('obsBak', Bax(b=None, S='a'))
Observable('obsBaxBclxL', Bax(b=1, S='a')%BclxL(b=1, S='m'))
```

By now you should have a file with all the components that looks something like this:

```
# import the pysb module and all its methods and functions
from pysb import *
from pysb.macros import *

# some functions to make life easy
site_name = 'b'
def catalyze_b(enz, sub, product, klist):
    """Alias for pysb.macros.catalyze with default binding site 'b'.
    """
    return catalyze(enz, site_name, sub, site_name, product, klist)

def bind_table_b(table):
    """Alias for pysb.macros.bind_table with default binding sites 'bf'.
    """
    return bind_table(table, site_name, site_name)

# Default forward, reverse, and catalytic rates
KF = 1e-6
KR = 1e-3
KC = 1

# Bid activation rates
bid_rates = [        1e-7, 1e-3, 1] #

# Bcl2 Inhibition Rates
bcl2_rates = [1.428571e-05, 1e-3] # 1.0e-6/v_mito

# instantiate a model
Model()

# declare monomers
Monomer('C8',    ['b'])
Monomer('Bid',   ['b', 'S'], {'S':['u', 't', 'm']})
Monomer('Bax',   ['b', 'S'], {'S':['i', 'a', 'm']})
Monomer('Bak',   ['b', 'S'], {'S':['i', 'a']})
Monomer('BclxL', ['b', 'S'], {'S':['c', 'm']})
Monomer('Bcl2', ['b'])
Monomer('Mcl1', ['b'])

# Activate Bid
catalyze_b(C8, Bid(S='u'), Bid(S='t'), [KF, KR, KC])

# Activate Bax/Bak
```

```
catalyze_b(Bid(S='m'), Bax(S='i'), Bax(S='m'), bid_rates)
catalyze_b(Bid(S='m'), Bak(S='i'), Bak(S='a'), bid_rates)

# Bid, Bax, BclxL "transport" to the membrane
equilibrate(Bid(b=None, S='t'),   Bid(b=None, S='m'), [1e-1, 1e-3])
equilibrate(Bax(b=None, S='m'),   Bax(b=None, S='a'), [1e-1, 1e-3])
equilibrate(BclxL(b=None, S='c'), BclxL(b=None, S='m'), [1e-1, 1e-3])


bind_table_b([[                  Bcl2,  BclxL(S='m'),       Mcl1],
              [Bid(S='m'), bcl2_rates,  bcl2_rates,   bcl2_rates],
              [Bax(S='a'), bcl2_rates,  bcl2_rates,         None],
              [Bak(S='a'),       None,  bcl2_rates,   bcl2_rates]])

# initial conditions
Parameter('C8_0',    1e4)
Parameter('Bid_0',   1e4)
Parameter('Bax_0',  .8e5)
Parameter('Bak_0',  .2e5)
Parameter('BclxL_0', 1e3)
Parameter('Bcl2_0',  1e3)
Parameter('Mcl1_0',  1e3)

Initial(C8(b=None), C8_0)
Initial(Bid(b=None, S='u'), Bid_0)
Initial(Bax(b=None, S='i'), Bax_0)
Initial(Bak(b=None, S='i'), Bak_0)
Initial(BclxL(b=None, S='c'), BclxL_0)
Initial(Bcl2(b=None), Bcl2_0)
Initial(Mcl1(b=None), Mcl1_0)

# Observables
Observable('obstBid', Bid(b=None, S='m'))
Observable('obsBax', Bax(b=None, S='a'))
Observable('obsBak', Bax(b=None, S='a'))
Observable('obsBaxBclxL', Bax(b=1, S='a')%BclxL(b=1, S='m'))
```

With this you should be able to run the simulations and generate figures as described in the basic tutorial sections.

## 2.11 Compartments

We will continue building on your `mymodel_fxns.py` file and add one more species and a compartment. In extrinsic apoptosis, once *tBid* is activated it translocates to the outer mitochondrial membrane where it interacts with the protein *Bak* (residing in the membrane).

## 2.12 Model calibration

## 2.13 Modules

## 2.14 Miscellaneous

### 2.14.1 Self-export

For anyone who feels a little queasy about self-export, this section will try to explain the rationale behind it.

In order to make model definition feel like a domain-specific language specially designed for model construction, the mechanism for component definition needs to provide three things:

- It must provide an internal name so that components can be usefully distinguished when inspected interactively, or translated into various output file formats such as BNGL.
- The component object must be assigned to a local variable so that subsequent component declarations can reference it by name using normal Python syntax (including operator overloading).
- The object must also be inserted into the data structures of the model object itself.

Without self-export, every component definition would need to manage these points explicitly:

```
A = Monomer('A')
model.add_component(A)
B = Monomer('B')
model.add_component(B)
```

This pattern introduces several opportunities for error, for example a `name` argument and the corresponding variable name may end up out of sync or the modeler may forget an `add_component` call. The redundancy also introduces visual noise which makes the code harder to read. Furthermore, self-export makes model modularization much simpler, as components may be defined within functions without forcing the function to explicitly return them or requiring extra code in the caller to deal with the returned components.

In addition to `Component` and its subclasses, the `Model` constructor also utilizes self-export, with two differences: The local variable is always named `model`, and the `name` argument is optional and defaults to the full hierarchical name of the module from which `Model()` is called, e.g. `pysb.examples.tutorial_a`.

CHAPTER 3

# PySB Modules Reference

## 3.1 PySB core (`pysb.core`)

**class** `pysb.core.`**`ANY`**

Site must have a bond, but identity of binding partner is irrelevant.

Use ANY in a MonomerPattern site_conditions dict to indicate that a site must have a bond without specifying what the binding partner should be.

Equivalent to the "+" bond modifier in BNG.

**class** `pysb.core.`**`Compartment`**(*name*, *parent=None*, *dimension=3*, *size=None*, *_export=True*)

Model component representing a bounded reaction volume.

**Parameters** **parent** : Compartment, optional

Compartment which contains this one. If not specified, this will be the outermost compartment and its parent will be set to None.

**dimension** : integer, optional

The number of spatial dimensions in the compartment, either 2 (i.e. a membrane) or 3 (a volume).

**size** : Parameter, optional

A parameter object whose value defines the volume or area of the compartment. If not specified, the size will be fixed at 1.0.

### Notes

The compartments of a model must form a tree via their *parent* attributes with a three-dimensional (volume) compartment at the root. A volume compartment may have any number of two-dimensional (membrane) compartments as its children, but never another volume compartment. A membrane compartment may have a single volume compartment as its child, but nothing else.

### Examples

Compartment('cytosol', dimension=3, size=cyto_vol, parent=ec_membrane)

### Attributes

| | |
|---|---|
| Identical to Parameters (see above). | |

**class** `pysb.core.`**`ComplexPattern`**(*monomer_patterns*, *compartment*, *match_once=False*)

A bound set of MonomerPatterns, i.e. a pattern to match a complex.

In BNG terms, a list of patterns combined with the '.' operator.

**Parameters** **monomer_patterns** : list of MonomerPatterns

MonomerPatterns that make up the complex.

**compartment** : Compartment

Location restriction. None means don't care.

**match_once** : bool, optional

If True, the pattern will only count once against a species in which the pattern can match the monomer graph in multiple distinct ways. If False (default), the pattern will count as many times as it matches the monomer graph, leading to a faster effective reaction rate.

### Attributes

| | |
|---|---|
| Identical to Parameters (see above). | |

**`copy`**()

Implement our own brand of shallow copy.

The new object will have references to the original compartment, and copies of the monomer_patterns.

**`is_concrete`**()

Return a bool indicating whether the pattern is 'concrete'.

'Concrete' means the pattern satisfies ANY of the following: 1. All monomer patterns are concrete 2. The compartment is specified AND all monomer patterns are site-concrete

**`is_equivalent_to`**(*other*)

Checks for equality with another ComplexPattern

**class** `pysb.core.`**`Component`**(*name*, *_export=True*)

The base class for all the named things contained within a model.

**Parameters** **name** : string

Name of the component. Must be unique within the containing model.

### Attributes

| | |
|---|---|
| name | (string) Name of the component. |
| model | (weakref(Model)) Containing model. |

**`rename`**(*new_name*)

Change component's name.

This is typically only needed when deriving one model from another and it would be desirable to change a component's name in the derived model.

**exception** `pysb.core.`**`ComponentDuplicateNameError`**
A component was added with the same name as an existing one.

**class** `pysb.core.`**`ComponentSet`**(*iterable=[]*)
An add-and-read-only container for storing model Components.

It behaves mostly like an ordered set, but components can also be retrieved by name *or* index by using the [] operator (like a combination of a dict and a list). Components can not be removed or replaced, but they can be renamed. Iteration returns the component objects.

**Parameters** **iterable** : iterable of Components, optional

Initial contents of the set.

**`rename`**(*c*, *new_name*)
Change the name of component *c* to *new_name*.

**exception** `pysb.core.`**`ConstantExpressionError`**
Expected a constant Expression but got something else.

**class** `pysb.core.`**`Expression`**(*name*, *expr*, *_export=True*)
Model component representing a symbolic expression of other variables.

**Parameters** **expr** : sympy.Expr

Symbolic expression.

**Attributes**

| | |
|---|---|
| expr | (sympy.Expr) See Parameters. |

**`expand_expr`**()
Return expr rewritten in terms of terminal symbols only.

**`is_constant_expression`**()
Return True if all terminal symbols are Parameters or numbers.

**exception** `pysb.core.`**`ExpressionError`**
Expected an Expression but got something else.

`pysb.core.`**`Initial`**(**args*)
Declare an initial condition (see Model.initial).

**exception** `pysb.core.`**`InvalidComplexPatternException`**
Expression can not be cast as a ComplexPattern.

**exception** `pysb.core.`**`InvalidComponentNameError`**(*name*)
Inappropriate component name.

**exception** `pysb.core.`**`InvalidInitialConditionError`**
Invalid initial condition pattern.

**exception** `pysb.core.`**`InvalidReactionPatternException`**
Expression can not be cast as a ReactionPattern.

**exception** `pysb.core.`**`InvalidReversibleSynthesisDegradationRule`**
Synthesis or degradation rule defined as reversible.

`pysb.core.`**`MatchOnce`**(*pattern*)
Make a ComplexPattern match-once.

**class** `pysb.core.`**`Model`**(*name=None*, *base=None*, *_export=True*)
A rule-based model containing monomers, rules, compartments and parameters.

**Parameters** **name** : string, optional

Name of the model. If not specified, will be set to the name of the file from which the constructor was called (with the .py extension stripped).

**base** : Model, optional

If specified, the model will begin as a copy of *base*. This can be used to achieve a simple sort of model extension and enhancement.

### Attributes

| | |
|---|---|
| name | (string) Name of the model. See Parameter section above. |
| base | (Model or None) See Parameter section above. |
| monomers, compartments, parameters, rules, observables | (ComponentSet) The Component objects which make up the model. |
| initial_conditions | (list of tuple of (ComplexPattern, Parameter)) Specifies which species are present in the model's starting state (t=0) and how much there is of each one. The ComplexPattern defines the species identity, and it must be concrete (see ComplexPattern.is_concrete). The Parameter defines the amount or concentration of the species. |
| species | (list of ComplexPattern) List of all complexes which can be produced by the model, starting from the initial conditions and successively applying the rules. Each ComplexPattern is concrete. |
| odes | (list of sympy.Expr) Mathematical expressions describing the time derivative of the amount of each species, as generated by the rules. |
| reactions | (list of dict) Structures describing each possible unidirectional reaction that can be produced by the model. Each structure stores the name of the rule that generated the reaction ('rule'), the mathematical expression for the rate of the reaction ('rate'), tuples of species indexes for the reactants and products ('reactants', 'products'), and a bool indicating whether the reaction is the reverse component of a bidirectional reaction ('reverse'). |
| reactions_bidirectional | (list of dict) Similar to *reactions* but with only one entry for each bidirectional reaction. The fields are identical except 'reverse' is replaced by 'reversible', a bool indicating whether the reaction is reversible. The 'rate' is the forward rate minus the reverse rate. |
| annotations | (list of Annotation) Structured annotations of model components. See the Annotation class for details. |

**`add_annotation`**(*annotation*)
Add an annotation to the model.

**`add_component`**(*other*)
Add a component to the model.

**`all_component_sets`**()
Return a list of all ComponentSet objects.

**`all_components`**()
Return a ComponentSet containing all components in the model.

**`enable_synth_deg`**()
Add components needed to support synthesis and degradation rules.

**expressions_constant**()
Return a ComponentSet of constant expressions.

**expressions_dynamic**()
Return a ComponentSet of non-constant expressions.

**get_annotations**(*subject*)
Return all annotations for the given subject.

**get_species_index**(*complex_pattern*)
Return the index of a species.

**Parameters** **complex_pattern** : ComplexPattern

A concrete pattern specifying the species to find.

**has_synth_deg**()
Return true if model uses synthesis or degradation reactions.

**initial**(*pattern*, *value*)
Add an initial condition.

An initial condition is made up of a species and its amount or concentration.

**Parameters** **pattern** : ComplexPattern

A concrete pattern defining the species to initialize.

**value** : Parameter

Amount of the species the model will start with.

**parameters_compartments**()
Return a ComponentSet of compartment size parameters.

**parameters_initial_conditions**()
Return a ComponentSet of initial condition parameters.

**parameters_rules**()
Return a ComponentSet of the parameters used in rules.

**parameters_unused**()
Return a ComponentSet of unused parameters.

**reload**()
Reload a model after its source files have been edited.

This method does not yet reload the model contents in-place, rather it returns a new model object. Thus the correct usage is `model = model.reload()`.

If the model script imports any modules, these will not be reloaded. Use python's reload() function to reload them.

**reset_equations**()
Clear out fields generated by bng.generate_equations or the like.

**update_initial_condition_pattern**(*before_pattern*, *after_pattern*)
Update the pattern associated with an initial condition.

Leaves the Parameter object associated with the initial condition unchanged while modifying the pattern associated with that condition. For example this is useful for changing the state of a site on a monomer or complex associated with an initial condition without having to create an independent initial condition, and parameter, associated with that alternative state.

**Parameters** **before_pattern** : ComplexPattern

The concrete pattern specifying the (already existing) initial condition. If the model does not contain an initial condition for the pattern, a ValueError is raised.

**after_pattern** : ComplexPattern

The concrete pattern specifying the new pattern to use to replace before_pattern.

**exception** `pysb.core.`**`ModelExistsWarning`**

A second model was declared in a module that already contains one.

**class** `pysb.core.`**`Monomer`**(*name*, *sites=[]*, *site_states={}*, *_export=True*)

Model component representing a protein or other molecule.

**Parameters** **sites** : list of strings, optional

Names of the sites.

**site_states** : dict of string => string, optional

Allowable states for sites. Keys are sites and values are lists of states. Sites which only take part in bond formation and never take on a state may be omitted.

## Notes

A Monomer instance may be "called" like a function to produce a MonomerPattern, as syntactic sugar to approximate rule-based modeling language syntax. It is typically called with keyword arguments where the arg names are sites and values are site conditions such as bond numbers or states (see the Notes section of the *`MonomerPattern`* documentation for details). To help in situations where kwargs are unwieldy (for example if a site name is computed dynamically or stored in a variable) a dict following the same layout as the kwargs may be passed as the first and only positional argument instead.

## Attributes

| | |
|---|---|
| Identical to Parameters (see above). | |

**class** `pysb.core.`**`MonomerPattern`**(*monomer*, *site_conditions*, *compartment*)

A pattern which matches instances of a given monomer.

**Parameters** **monomer** : Monomer

The monomer to match.

**site_conditions** : dict

The desired state of the monomer's sites. Keys are site names and values are described below in Notes.

**compartment** : Compartment or None

The desired compartment where the monomer should exist. None means "don't-care".

## Notes

The acceptable values in the *site_conditions* dict are as follows:

- `None` : no bond
- *str* : state
- *int* : a bond (to a site with the same number in a ComplexPattern)

- *list of int* : multi-bond (not valid in Kappa)
- `ANY` : "any" bond (bound to something, but don't care what)
- `WILD` : "wildcard" bond (bound or not bound)
- *tuple of (str, int)* : state with bond
- *tuple of (str, WILD)* : state with wildcard bond

If a site is not listed in site_conditions then the pattern will match any state for that site, i.e. "don't write, don't care".

### Attributes

| Identical to Parameters (see above). | |
|---|---|

**is_concrete**()

Return a bool indicating whether the pattern is 'concrete'.

'Concrete' means the pattern satisfies ALL of the following:

1. All sites have specified conditions
2. If the model uses compartments, the compartment is specified.

**is_site_concrete**()

Return a bool indicating whether the pattern is 'site-concrete'.

'Site-concrete' means all sites have specified conditions.

**class** pysb.core.**Observable**(*name*, *reaction_pattern*, *match='molecules'*, *_export=True*)

Model component representing a linear combination of species.

Observables are useful in correlating model simulation results with experimental measurements. For example, an observable for "A()" will report on the total number of copies of Monomer A, regardless of what it's bound to or the state of its sites. "A(y='P')" would report on all instances of A with site 'y' in state 'P'.

**Parameters** **reaction_pattern** : ReactionPattern

The list of ComplexPatterns to match.

**match** : 'species' or 'molecules'

Whether to match entire species ('species') or individual fragments ('molecules'). Default is 'molecules'.

### Notes

ReactionPattern is used here as a container for a list of ComplexPatterns, solely so users could utilize the ComplexPattern '+' operator overload as syntactic sugar. There are no actual "reaction" semantics in this context.

### Attributes

| | |
|---|---|
| reaction_pattern | (ReactionPattern) See Parameters. |
| species | (list of integers) List of species indexes for species matching the pattern. |
| coefficients | (list of integers) List of coefficients by which each species amount is to be multipled to correct for multiple pattern matches within a species. |

**class** `pysb.core.`**`Parameter`**(*name*, *value=0.0*, *_export=True*)

Model component representing a named constant floating point number.

Parameters are used as reaction rate constants, compartment volumes and initial (boundary) conditions for species.

**Parameters** **value** : number, optional

The numerical value of the parameter. Defaults to 0.0 if not specified. The provided value is converted to a float before being stored, so any value that cannot be coerced to a float will trigger an exception.

### Attributes

| | |
|---|---|
| Identical to Parameters (see above). | |

**class** `pysb.core.`**`ReactionPattern`**(*complex_patterns*)

A pattern for the entire product or reactant side of a rule.

Essentially a thin wrapper around a list of ComplexPatterns. In BNG terms, a list of complex patterns combined with the '+' operator.

**Parameters** **complex_patterns** : list of ComplexPatterns

ComplexPatterns that make up the reaction pattern.

### Attributes

| | |
|---|---|
| Identical to Parameters (see above). | |

**class** `pysb.core.`**`Rule`**(*name*, *rule_expression*, *rate_forward*, *rate_reverse=None*, *delete_molecules=False*, *move_connected=False*, *_export=True*)

Model component representing a reaction rule.

**Parameters** **rule_expression** : RuleExpression

RuleExpression containing the essence of the rule (reactants, products, reversibility).

**rate_forward** : Parameter

Forward reaction rate constant.

**rate_reverse** : Parameter, optional

Reverse reaction rate constant (only required for reversible rules).

**delete_molecules** : bool, optional

If True, deleting a Monomer from a species is allowed to fragment the species into multiple pieces (if the deleted Monomer was the sole link between those pieces). If False (default) then fragmentation is disallowed and the rule will not match a reactant species if applying the rule would fragment a species.

**move_connected** : bool, optional

If True, a rule that transports a Monomer between compartments will co-transport anything connected to that Monomer by a path in the same compartment. If False (default), connected Monomers will remain where they were.

### Attributes

| | |
|---|---|
| Identical to Parameters (see above), plus the component elements of | |
| *rule_expression*: reactant_pattern, product_pattern and is_reversible. | |

**is_deg**()
Return a bool indicating whether this is a degradation rule.

**is_synth**()
Return a bool indicating whether this is a synthesis rule.

**class** pysb.core.**RuleExpression**(*reactant_pattern*, *product_pattern*, *is_reversible*)
A container for the reactant and product patterns of a rule expression.

Contains one ReactionPattern for each of reactants and products, and a bool indicating reversibility. This is a temporary object used to implement syntactic sugar through operator overloading. The Rule constructor takes an instance of this class as its first argument, but simply extracts its fields and discards the object itself.

**Parameters** **reactant_pattern, product_pattern** : ReactionPattern

The reactants and products of the rule.

**is_reversible** : bool

If True, the reaction is reversible. If False, it's irreversible.

### Attributes

| | |
|---|---|
| Identical to Parameters (see above). | |

**class** pysb.core.**SelfExporter**
Make model components appear in the calling module's namespace.

This class is for pysb internal use only. Do not construct any instances.

**static cleanup**()
Delete previously exported symbols.

**static export**(*obj*)
Export an object by name and add it to the default model.

**static rename**(*obj*, *new_name*)
Rename a previously exported symbol

**exception** pysb.core.**SymbolExistsWarning**
A component declaration or rename overwrote an existing symbol.

**class** pysb.core.**WILD**
Site may be bound or unbound.

Use WILD as part of a (state, WILD) tuple in a MonomerPattern site_conditions dict to indicate that a site must have the given state, irrespective of the presence or absence of a bond. (Specifying only the state implies there must not be a bond). A bare WILD in a site_conditions dict is also permissible, but as this has the same meaning as the much simpler option of leaving the given site out of the dict entirely, this usage is deprecated.

Equivalent to the "?" bond modifier in BNG.

pysb.core.**as_complex_pattern**(*v*)
Internal helper to 'upgrade' a MonomerPattern to a ComplexPattern.

pysb.core.**as_reaction_pattern**(*v*)
Internal helper to 'upgrade' a Complex- or MonomerPattern or None to a complete ReactionPattern.

`pysb.core.`**`build_rule_expression`**(*reactant*, *product*, *is_reversible*)
: Internal helper for operators which return a RuleExpression.

`pysb.core.`**`extract_site_conditions`**(*conditions=None*, ***kwargs*)
: Parse MonomerPattern site conditions.

`pysb.core.`**`validate_const_expr`**(*obj*, *description*)
: Raises an exception if the argument is not a constant expression.

`pysb.core.`**`validate_expr`**(*obj*, *description*)
: Raises an exception if the argument is not an expression.

## 3.2 ODE integrators (`pysb.integrate`)

**class** `pysb.integrate.`**`Solver`**(*model*, *tspan*, *integrator='vode'*, ***integrator_options*)
: An interface for numeric integration of models.

**Parameters** **model** : pysb.Model

Model to integrate.

**tspan** : vector-like

Time values over which to integrate. The first and last values define the time range, and the returned trajectories will be sampled at every value.

**integrator** : string, optional

Name of the integrator to use, taken from the list of integrators known to `scipy.integrate.ode`.

**integrator_params**

Additional parameters for the integrator.

### Notes

The expensive step of generating the code for the right-hand side of the model's ODEs is performed during initialization. If you need to integrate the same model repeatedly with different parameters then you should build a single Solver object and then call its `run` method as needed.

### Attributes

| | |
|---|---|
| model | (pysb.Model) Model passed to the constructor |
| tspan | (vector-like) Time values passed to the constructor. |
| y | (numpy.ndarray) Species trajectories. Dimensionality is `(len(tspan), len(model.species))`. |
| yobs | (numpy.ndarray with record-style data-type) Observable trajectories. Length is is `len(tspan)` and record names follow `model.observables` names. |
| yobs_view | (numpy.ndarray) An array view (sharing the same data buffer) on `yobs`. Dimensionality is `(len(tspan), len(model.observables))`. |
| inte-grator | (scipy.integrate.ode) Integrator object. |

**run** (*param_values=None*, *y0=None*)

Perform an integration.

Returns nothing; access the Solver object's `y`, `yobs`, or `yobs_view` attributes to retrieve the results.

**Parameters** **param_values** : vector-like, optional

Values to use for every parameter in the model. Ordering is determined by the order of model.parameters. If not specified, parameter values will be taken directly from model.parameters.

**y0** : vector-like, optional

Values to use for the initial condition of all species. Ordering is determined by the order of model.species. If not specified, initial conditions will be taken from model.initial_conditions (with initial condition parameter values taken from *param_values* if specified).

`pysb.integrate.`**odesolve** (*model*, *tspan*, *param_values=None*, *y0=None*, *integrator='vode'*, ***integrator_options*)

Integrate a model's ODEs over a given timespan.

This is a simple function-based interface to integrating (a.k.a. solving or simulating) a model. If you need to integrate a model repeatedly with different parameter values or initial conditions (as in parameter estimation), using the Solver class directly will provide much better performance.

**Parameters** **model** : pysb.Model

Model to integrate.

**tspan** : vector-like

Time values over which to integrate. The first and last values define the time range, and the returned trajectories will be sampled at every value.

**param_values** : vector-like, optional

Values to use for every parameter in the model. Ordering is determined by the order of model.parameters. If not specified, parameter values will be taken directly from model.parameters.

**y0** : vector-like, optional

Values to use for the initial condition of all species. Ordering is determined by the order of model.species. If not specified, initial conditions will be taken from model.initial_conditions (with initial condition parameter values taken from *param_values* if specified).

**integrator** : string, optional

Name of the integrator to use, taken from the list of integrators known to `scipy.integrate.ode`.

**integrator_params**

Additional parameters for the integrator.

**Returns** **yfull** : record array

The trajectories calculated by the integration. The first dimension is time and its length is identical to that of *tspan*. The second dimension is species/observables and its length is the sum of the lengths of model.species and model.observables. The dtype of the array specifies field names: '__s0', '__s1', etc. for the species and observable names for the observables. See Notes below for further explanation and caveats.

### Notes

This function was the first implementation of integration support and accordingly it has a few warts:

- It performs expensive code generation every time it is called.
- The returned array, with its record-style data-type, allows convenient selection of individual columns by their field names, but does not permit slice ranges or indexing by integers for columns. If you only need access to your model's observables this is usually not a problem, but sometimes it's more convenient to have a "regular" array. See Examples below for code to do this.

The actual integration code has since been moved to the Solver class and split up such that the code generation is only performed on initialization. The model may then be integrated repeatedly with different parameter values or initial conditions with much better performance. Additionally, Solver makes the species trajectories available as a simple array and only uses the record array for the observables where it makes sense.

This function now simply serves as a wrapper for creating a Solver object, calling its `run` method, and building the record array to return.

### Examples

Simulate a model and display the results for an observable:

```
>>> from pysb.examples.robertson import model
>>> from numpy import linspace
>>> numpy.set_printoptions(precision=4)
>>> yfull = odesolve(model, linspace(0, 40, 10))
>>> print yfull['A_total']
[ 1.      0.899   0.8506  0.8179  0.793   0.7728  0.7557  0.7408  0.7277
0.7158]
```

Obtain a view on a returned record array which uses an atomic data-type and integer indexing (note that the view's data buffer is shared with the original array so there is no extra memory cost):

```
>>> print yfull.shape
(10,)
>>> print yfull.dtype
[('__s0', '<f8'), ('__s1', '<f8'), ('__s2', '<f8'), ('A_total', '<f8'),
('B_total', '<f8'), ('C_total', '<f8')]
>>> print yfull[0:4, 1:3]
Traceback (most recent call last):
  ...
IndexError: too many indices for array
>>> yarray = yfull.view(float).reshape(len(yfull), -1)
>>> print yarray.shape
(10, 6)
>>> print yarray.dtype
float64
>>> print yarray[0:4, 1:3]
[[  0.0000e+00   0.0000e+00]
 [  2.1672e-05   1.0093e-01]
 [  1.6980e-05   1.4943e-01]
 [  1.4502e-05   1.8209e-01]]
```

`pysb.integrate.`**`setup_module`**(*module*)

Doctest fixture for nose.

## 3.3 BioNetGen integration (`pysb.bng`)

**exception** `pysb.bng.`**`GenerateNetworkError`**

BNG reported an error when trying to generate a network for a model.

**exception** `pysb.bng.`**`NoInitialConditionsError`**

Model initial_conditions is empty.

**exception** `pysb.bng.`**`NoRulesError`**

Model rules is empty.

`pysb.bng.`**`generate_equations`**(*model*)

Generate math expressions for reaction rates and species in a model.

This fills in the following pieces of the model:

- odes
- species
- reactions
- reactions_bidirectional
- observables (just *coefficients* and *species* fields for each element)

`pysb.bng.`**`generate_network`**(*model*, *cleanup=True*, *append_stdout=False*)

Return the output from BNG's generate_network function given a model.

The output is a BNGL model definition with additional sections 'reactions' and 'groups', and the 'species' section expanded to contain all possible species. BNG refers to this as a 'net' file.

**Parameters** **model** : Model

Model to pass to generate_network.

**cleanup** : bool, optional

If True (default), delete the temporary files after the simulation is finished. If False, leave them in place (in *output_dir*). Useful for debugging.

**append_stdout** : bool, optional

If True, provide BNG.pl's standard output stream as comment lines appended to the net file contents. If False (default), do not append it.

`pysb.bng.`**`run_ssa`**(*model*, *t_end=10*, *n_steps=100*, *output_dir='/tmp'*, *output_file_basename=None*, *cleanup=True*)

Simulate a model with BNG's SSA simulator and return the trajectories.

**Parameters** **model** : Model

Model to simulate.

**t_end** : number, optional

Final time point of the simulation.

**n_steps** : int, optional

Number of steps in the simulation.

**output_dir** : string, optional

Location for temporary files generated by BNG. Defaults to '/tmp'.

**output_file_basename** : string, optional

The basename for the .bngl, .gdat, .cdat, and .net files that are generated by BNG. If None (the default), creates a basename from the model name, process ID, and a random integer in the range (0, 100000).

**cleanup** : bool, optional

If True (default), delete the temporary files after the simulation is finished. If False, leave them in place (in *output_dir*). Useful for debugging.

## 3.4 Kappa integration (`pysb.kappa`)

Wrapper functions for running the Kappa programs *Kasim* and *complx*.

In general only the following three functions will be needed for typical use:

- `run_simulation()`
- `influence_map()`
- `contact_map()`

The other functions are used internally and manage the execution of the Kappa software and the parsing of the data into a Numpy format.

`pysb.kappa.`**`contact_map`**(*model*, *output_dir='.'*, *base_filename=None*, *do_open=False*, ***kwargs*)
Runs complx with arguments for generating the contact map.

**Parameters** **model** : pysb.core.Model

The model for generating the contact map.

**output_dir** : string

The subdirectory in which to generate the Kappa (.ka) file for the model and all output files produced by complx. Default value is '.' Note that only relative paths can be specified; paths are relative to the directory where the current Python instance is running. If the specified directory does not exist, an Exception is thrown.

**base_filename** : string

The base filename to be used for generation of the Kappa (.ka) file and all output files produced by complx. Defaults to a string of the form:

```
'%s_%d_%d_temp' % (model.name, os.getpid(), random.randint(0,10000))
```

The contact map filenames append '_cm.jpg' and '_cm.dot' to this base filename; the reachable complexes filename appends '_rch.dot'.

**do_open** : boolean

If do_open is set to True, then calls the `open_file()` method to display the contact map using the default program for opening .jpg files.

****kwargs** : other keyword arguments

Any other keyword arguments are passed through to the function `run_complx()`.

`pysb.kappa.`**`influence_map`**(*model*, *do_open=False*, ***kwargs*)
Generates the influence map.

Runs KaSim with no events or points and sets the dump_influence_map argument to True.

**Parameters** **model** : pysb.core.Model

The model for generating the influence map.

**do_open** : boolean

If do_open is set to True, then calls the `open_file()` method to display the influence map using the default program for opening .gv files (e.g., GraphViz).

****kwargs** : other keyword arguments

Any other keyword arguments are passed to the function `run_kasim()`.

**Returns** string

Returns the name of the .gv (GraphViz) file where the influence map has been stored. This can subsequently be used to build a networkx or PyGraphViz graph.

`pysb.kappa.`**`open_file`**`(`*filename*`)`

Utility function for opening files for display on Mac OS X.

Uses the 'open' command to open the given file using the default program associated with the file's filetype. Ultimately this should be rewritten to auto-detect the operating system and use the appropriate system call.

`pysb.kappa.`**`parse_kasim_outfile`**`(`*out_filename*`)`

Parses the KaSim .out file into a Numpy ndarray.

**Parameters** **out_filename** : string

String specifying the location of the .out filename produced by KaSim.

**Returns** numpy.ndarray

Returns the KaSim simulation data as a Numpy ndarray. Data is accessed using the syntax:

```
results[index_name]
```

The index 'time' gives the data for the time coordinates of the simulation. Data for the observables can be accessed by indexing the array with the names of the observables.

`pysb.kappa.`**`run_complx`**`(`*gen*, *kappa_filename*, *args*`)`

Generalized method for passing arguments to the complx executable.

**Parameters** **gen** : `pysb.generator.KappaGenerator`

A KappaGenerator object that is used to produce the Kappa content for writing to a file.

**kappa_filename** : string

The name of the file to write the generated Kappa to.

**args** : list of strings

List of command line arguments to pass to complx, with one entry for each argument, for example:

```
['--output-high-res-contact-map-jpg', jpg_filename]
```

`pysb.kappa.`**`run_kasim`**`(`*model*, *time=10000*, *points=200*, *output_dir='.'*, *cleanup=False*, *base_filename=None*, *dump_influence_map=False*, *perturbation=None*`)`

Run KaSim on the given model with the provided arguments.

**Parameters** **model** : pysb.core.Model

The model to simulate/analyze using KaSim.

**time** : number

The amount of time (in arbitrary units) to run a simulation. Identical to the -t argument when using KaSim at the command line. Default value is 10000. If set to 0, no simulation will be run, but the influence map will be generated (if dump_influence_map is set to True).

**points** : integer

The number of data points to collect for plotting. Identical to the -p argument when using KaSim at the command line. Default value is 200.

**output_dir** : string

The subdirectory in which to generate the Kappa (.ka) file for the model and all output files produced by KaSim. Default value is '.' Note that only relative paths can be specified; paths are relative to the directory where the current Python instance is running. If the specified directory does not exist, an Exception is thrown.

**cleanup** : boolean

Specifies whether output files produced by KaSim should be deleted after execution is completed. Default value is False.

**base_filename** : The base filename to be used for generation of the Kappa

(.ka) file and all output files produced by KaSim. Defaults to a string of the form:

```
'%s_%d_%d_temp' % (model.name, program id, random.randint(0,10000))
```

The influence map filename appends '_im.dot' to this base filename; the flux map filename appends '_fm.dot'; and the simulation output file appends '.out'

**dump_influence_map** : boolean

Specifies whether or not to produce the influence map. Default value is False.

**perturbation** : string or None

Optional perturbation language syntax to be appended to the Kappa file. See KaSim manual for more details. Default value is None (no perturbation).

**Returns** A dict with three entries giving the filenames for the files produced:

- output_dict['out'] gives the .out filename
- output_dict['im'] gives the influence map filename
- output_dict['fm'] gives the flux map filename

`pysb.kappa.`**`run_simulation`**(*model*, ***kwargs*)

Runs the given model using KaSim and returns the parsed results.

**Parameters** ****kwargs** : List of keyword arguments

All keyword arguments specifying conditions for the simulation are passed to the function `run_kasim()` (see documentation associated with that function for more information).

**Returns** numpy.ndarray

Returns the kasim simulation data as a Numpy ndarray. Data is accessed using the syntax:

```
results[index_name]
```

The index 'time' gives the data for the time coordinates of the simulation. Data for the observables can be accessed by indexing the array with the names of the observables.

## 3.5 Macros (`pysb.macros`)

A collection of generally useful modeling macros.

These macros are written to be as generic and reusable as possible, serving as a collection of best practices and implementation ideas. They conform to the following general guidelines:

- All components created by the macro are implicitly added to the current model and explicitly returned in a ComponentSet.
- Parameters may be passed as Parameter objects, or as plain numbers for which Parameter objects will be automatically created using an appropriate naming convention.
- Arguments which accept a MonomerPattern should also accept Monomers, which are to be interpreted as MonomerPatterns on that Monomer with an empty condition list. This is typically implemented by having the macro apply the "call" (parentheses) operator to the argument with an empty argument list and using the resulting value instead of the original argument when creating Rules, e.g. `arg = arg()`. Calling a Monomer will return a MonomerPattern, and calling a MonomerPattern will return a copy of itself, so calling either is guaranteed to return a MonomerPattern.

The _macro_rule helper function contains much of the logic needed to follow these guidelines. Every macro in this module either uses _macro_rule directly or calls another macro which does.

Another useful function is _verify_sites which will raise an exception if a Monomer or MonomerPattern does not possess every one of a given list of sites. This can be used to trigger such errors up front rather than letting an exception occur at the point where the macro tries to use the invalid site in a pattern, which can be harder for the caller to debug.

`pysb.macros.`**`equilibrate`**`(`*s1*, *s2*, *klist*`)`

Generate the unimolecular reversible equilibrium reaction S1 <-> S2.

**Parameters s1, s2** : Monomer or MonomerPattern

S1 and S2 in the above reaction.

**klist** : list of 2 Parameters or list of 2 numbers

Forward (S1 -> S2) and reverse rate constants (in that order). If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the names and states of S1 and S2 and these parameters will be included at the end of the returned component list.

**Returns components** : ComponentSet

The generated components. Contains one reversible Rule and optionally two Parameters if klist was given as plain numbers.

### Examples

Simple two-state equilibrium between A and B:

```
Model()
Monomer('A')
Monomer('B')
equilibrate(A(), B(), [1, 1])
```

Execution:

```
>>> Model()
<Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
>>> Monomer('A')
Monomer('A')
>>> Monomer('B')
Monomer('B')
>>> equilibrate(A(), B(), [1, 1])
ComponentSet([
 Rule('equilibrate_A_to_B', A() <> B(), equilibrate_A_to_B_kf, equilibrate_A_to_B_kr),
 Parameter('equilibrate_A_to_B_kf', 1.0),
 Parameter('equilibrate_A_to_B_kr', 1.0),
 ])
```

`pysb.macros.`**`bind`**(*s1*, *site1*, *s2*, *site2*, *klist*)

Generate the reversible binding reaction S1 + S2 <> S1:S2.

**Parameters** **s1, s2** : Monomer or MonomerPattern

Monomers participating in the binding reaction.

**site1, site2** : string

The names of the sites on s1 and s2 used for binding.

**klist** : list of 2 Parameters or list of 2 numbers

Forward and reverse rate constants (in that order). If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the names and states of S1 and S2 and these parameters will be included at the end of the returned component list.

**Returns** **components** : ComponentSet

The generated components. Contains the bidirectional binding Rule and optionally two Parameters if klist was given as numbers.

## Examples

Binding between A and B:

```
Model()
Monomer('A', ['x'])
Monomer('B', ['y'])
bind(A, 'x', B, 'y', [1e-4, 1e-1])
```

Execution:

```
>>> Model()
<Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
>>> Monomer('A', ['x'])
Monomer('A', ['x'])
>>> Monomer('B', ['y'])
Monomer('B', ['y'])
>>> bind(A, 'x', B, 'y', [1e-4, 1e-1])
ComponentSet([
 Rule('bind_A_B', A(x=None) + B(y=None) <> A(x=1) % B(y=1), bind_A_B_kf, bind_A_B_kr),
 Parameter('bind_A_B_kf', 0.0001),
 Parameter('bind_A_B_kr', 0.1),
 ])
```

pysb.macros.**bind_table**(*bindtable*, *row_site*, *col_site*, *kf=None*)

Generate a table of reversible binding reactions.

Given two lists of species R and C, calls the *bind* macro on each pairwise combination (R[i], C[j]). The species lists and the parameter values are passed as a list of lists (i.e. a table) with elements of R passed as the "row headers", elements of C as the "column headers", and forward / reverse rate pairs (in that order) as tuples in the "cells". For example with two elements in each of R and C, the table would appear as follows (note that the first row has one fewer element than the subsequent rows):

```
[[              C1,           C2],
 [R1, (1e-4, 1e-1), (2e-4, 2e-1)],
 [R2, (3e-4, 3e-1), (4e-4, 4e-1)]]
```

Each parameter tuple may contain Parameters or numbers. If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the names and states of the relevant species and these parameters will be included at the end of the returned component list. To omit any individual reaction, pass None in place of the corresponding parameter tuple.

Alternately, single kd values (dissociation constant, kr/kf) may be specified instead of (kf, kr) tuples. If kds are used, a single shared kf Parameter or number must be passed as an extra *kf* argument. kr values for each binding reaction will be calculated as kd*kf. It is important to remember that the forward rate constant is a single parameter shared across the entire bind table, as this may have implications for parameter fitting.

**Parameters** **bindtable** : list of lists

Table of reactants and rates, as described above.

**row_site, col_site** : string

The names of the sites on the elements of R and C, respectively, used for binding.

**kf** : Parameter or number, optional

If the "cells" in bindtable are given as single kd values, this is the shared kf used to calculate the kr values.

**Returns** **components** : ComponentSet

The generated components. Contains the bidirectional binding Rules and optionally the Parameters for any parameters given as numbers.

## Examples

Binding table for two species types (R and C), each with two members:

```
Model()
Monomer('R1', ['x'])
Monomer('R2', ['x'])
Monomer('C1', ['y'])
Monomer('C2', ['y'])
bind_table([[               C1,           C2],
            [R1,  (1e-4, 1e-1),  (2e-4, 2e-1)],
            [R2,  (3e-4, 3e-1),          None]],
           'x', 'y')
```

Execution:

```
>>> Model()
<Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
```

```
>>> Monomer('R1', ['x'])
Monomer('R1', ['x'])
>>> Monomer('R2', ['x'])
Monomer('R2', ['x'])
>>> Monomer('C1', ['y'])
Monomer('C1', ['y'])
>>> Monomer('C2', ['y'])
Monomer('C2', ['y'])
>>> bind_table([[               C1,           C2],
...             [R1,  (1e-4, 1e-1),  (2e-4, 2e-1)],
...             [R2,  (3e-4, 3e-1),          None]],
...            'x', 'y')
ComponentSet([
 Rule('bind_R1_C1', R1(x=None) + C1(y=None) <> R1(x=1) % C1(y=1),
     bind_R1_C1_kf, bind_R1_C1_kr),
 Parameter('bind_R1_C1_kf', 0.0001),
 Parameter('bind_R1_C1_kr', 0.1),
 Rule('bind_R1_C2', R1(x=None) + C2(y=None) <> R1(x=1) % C2(y=1),
     bind_R1_C2_kf, bind_R1_C2_kr),
 Parameter('bind_R1_C2_kf', 0.0002),
 Parameter('bind_R1_C2_kr', 0.2),
 Rule('bind_R2_C1', R2(x=None) + C1(y=None) <> R2(x=1) % C1(y=1),
     bind_R2_C1_kf, bind_R2_C1_kr),
 Parameter('bind_R2_C1_kf', 0.0003),
 Parameter('bind_R2_C1_kr', 0.3),
 ])
```

`pysb.macros.`**`catalyze`**(*enzyme*, *e_site*, *substrate*, *s_site*, *product*, *klist*)

Generate the two-step catalytic reaction E + S <> E:S >> E + P.

**Parameters** **enzyme, substrate, product** : Monomer or MonomerPattern

E, S and P in the above reaction.

**e_site, s_site** : string

The names of the sites on *enzyme* and *substrate* (respectively) where they bind each other to form the E:S complex.

**klist** : list of 3 Parameters or list of 3 numbers

Forward, reverse and catalytic rate constants (in that order). If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the names and states of enzyme, substrate and product and these parameters will be included at the end of the returned component list.

**Returns** **components** : ComponentSet

The generated components. Contains two Rules (bidirectional complex formation and unidirectional product dissociation), and optionally three Parameters if klist was given as plain numbers.

### Notes

When passing a MonomerPattern for *enzyme* or *substrate*, do not include *e_site* or *s_site* in the respective patterns. The macro will handle this.

### Examples

Using distinct Monomers for substrate and product:

```
Model()
Monomer('E', ['b'])
Monomer('S', ['b'])
Monomer('P')
catalyze(E(), 'b', S(), 'b', P(), (1e-4, 1e-1, 1))
```

Execution:

```
>>> Model()
<Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
>>> Monomer('E', ['b'])
Monomer('E', ['b'])
>>> Monomer('S', ['b'])
Monomer('S', ['b'])
>>> Monomer('P')
Monomer('P')
>>> catalyze(E(), 'b', S(), 'b', P(), (1e-4, 1e-1, 1))
ComponentSet([
 Rule('bind_E_S_to_ES', E(b=None) + S(b=None) <> E(b=1) % S(b=1),
     bind_E_S_to_ES_kf, bind_E_S_to_ES_kr),
 Parameter('bind_E_S_to_ES_kf', 0.0001),
 Parameter('bind_E_S_to_ES_kr', 0.1),
 Rule('catalyze_ES_to_E_P', E(b=1) % S(b=1) >> E(b=None) + P(),
     catalyze_ES_to_E_P_kc),
 Parameter('catalyze_ES_to_E_P_kc', 1.0),
 ])
```

Using a single Monomer for substrate and product with a state change:

```
Monomer('Kinase', ['b'])
Monomer('Substrate', ['b', 'y'], {'y': ('U', 'P')})
catalyze(Kinase(), 'b', Substrate(y='U'), 'b', Substrate(y='P'),
         (1e-4, 1e-1, 1))
```

Execution:

```
>>> Model()
<Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
>>> Monomer('Kinase', ['b'])
Monomer('Kinase', ['b'])
>>> Monomer('Substrate', ['b', 'y'], {'y': ('U', 'P')})
Monomer('Substrate', ['b', 'y'], {'y': ('U', 'P')})
>>> catalyze(Kinase(), 'b', Substrate(y='U'), 'b', Substrate(y='P'), (1e-4, 1e-1, 1))
ComponentSet([
 Rule('bind_Kinase_SubstrateU_to_KinaseSubstrateU',
     Kinase(b=None) + Substrate(b=None, y='U') <> Kinase(b=1) % Substrate(b=1, y='U'),
     bind_Kinase_SubstrateU_to_KinaseSubstrateU_kf,
     bind_Kinase_SubstrateU_to_KinaseSubstrateU_kr),
 Parameter('bind_Kinase_SubstrateU_to_KinaseSubstrateU_kf', 0.0001),
 Parameter('bind_Kinase_SubstrateU_to_KinaseSubstrateU_kr', 0.1),
 Rule('catalyze_KinaseSubstrateU_to_Kinase_SubstrateP',
      Kinase(b=1) % Substrate(b=1, y='U') >> Kinase(b=None) + Substrate(b=None, y='P'),
      catalyze_KinaseSubstrateU_to_Kinase_SubstrateP_kc),
 Parameter('catalyze_KinaseSubstrateU_to_Kinase_SubstrateP_kc', 1.0),
 ])
```

pysb.macros.**catalyze_state**(*enzyme*, *e_site*, *substrate*, *s_site*, *mod_site*, *state1*, *state2*, *klist*)

Generate the two-step catalytic reaction E + S <> E:S >> E + P. A wrapper around catalyze() with a signature specifying the state change of the substrate resulting from catalysis.

**Parameters** **enzyme** : Monomer or MonomerPattern

E in the above reaction.

**substrate** : Monomer or MonomerPattern

S and P in the above reaction. The product species is assumed to be identical to the substrate species in all respects except the state of the modification site. The state of the modification site should not be specified in the MonomerPattern for the substrate.

**e_site, s_site** : string

The names of the sites on *enzyme* and *substrate* (respectively) where they bind each other to form the E:S complex.

**mod_site** : string

The name of the site on the substrate that is modified by catalysis.

**state1, state2** : strings

The states of the modification site (mod_site) on the substrate before (state1) and after (state2) catalysis.

**klist** : list of 3 Parameters or list of 3 numbers

Forward, reverse and catalytic rate constants (in that order). If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the names and states of enzyme, substrate and product and these parameters will be included at the end of the returned component list.

**Returns** **components** : ComponentSet

The generated components. Contains two Rules (bidirectional complex formation and unidirectional product dissociation), and optionally three Parameters if klist was given as plain numbers.

## Notes

When passing a MonomerPattern for *enzyme* or *substrate*, do not include *e_site* or *s_site* in the respective patterns. In addition, do not include the state of the modification site on the substrate. The macro will handle this.

## Examples

Using a single Monomer for substrate and product with a state change:

```
Monomer('Kinase', ['b'])
Monomer('Substrate', ['b', 'y'], {'y': ('U', 'P')})
catalyze_state(Kinase, 'b', Substrate, 'b', 'y', 'U', 'P',
         (1e-4, 1e-1, 1))
```

Execution:

```
>>> Model()
<Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
>>> Monomer('Kinase', ['b'])
Monomer('Kinase', ['b'])
>>> Monomer('Substrate', ['b', 'y'], {'y': ('U', 'P')})
Monomer('Substrate', ['b', 'y'], {'y': ('U', 'P')})
>>> catalyze_state(Kinase, 'b', Substrate, 'b', 'y', 'U', 'P', (1e-4, 1e-1, 1))
ComponentSet([
 Rule('bind_Kinase_SubstrateU_to_KinaseSubstrateU',
     Kinase(b=None) + Substrate(b=None, y='U') <> Kinase(b=1) % Substrate(b=1, y='U'),
     bind_Kinase_SubstrateU_to_KinaseSubstrateU_kf,
     bind_Kinase_SubstrateU_to_KinaseSubstrateU_kr),
 Parameter('bind_Kinase_SubstrateU_to_KinaseSubstrateU_kf', 0.0001),
 Parameter('bind_Kinase_SubstrateU_to_KinaseSubstrateU_kr', 0.1),
 Rule('catalyze_KinaseSubstrateU_to_Kinase_SubstrateP',
     Kinase(b=1) % Substrate(b=1, y='U') >> Kinase(b=None) + Substrate(b=None, y='P'),
     catalyze_KinaseSubstrateU_to_Kinase_SubstrateP_kc),
 Parameter('catalyze_KinaseSubstrateU_to_Kinase_SubstrateP_kc', 1.0),
 ])
```

`pysb.macros.`**`catalyze_one_step`**`(`*enzyme*, *substrate*, *product*, *kf*`)`

Generate the one-step catalytic reaction E + S >> E + P.

**Parameters** **enzyme, substrate, product** : Monomer or MonomerPattern

E, S and P in the above reaction.

**kf** : a Parameter or a number

Forward rate constant for the reaction. If a Parameter is passed, it will be used directly in the generated Rules. If a number is passed, a Parameter will be created with an automatically generated name based on the names and states of the enzyme, substrate and product and this parameter will be included at the end of the returned component list.

**Returns** **components** : ComponentSet

The generated components. Contains the unidirectional reaction Rule and optionally the forward rate Parameter if klist was given as a number.

### Notes

In this macro, there is no direct binding between enzyme and substrate, so binding sites do not have to be specified. This represents an approximation for the case when the enzyme is operating in its linear range. However, if catalysis is nevertheless contingent on the enzyme or substrate being unbound on some site, then that information must be encoded in the MonomerPattern for the enzyme or substrate. See the examples, below.

### Examples

Convert S to P by E:

```
Model()
Monomer('E', ['b'])
Monomer('S', ['b'])
Monomer('P')
catalyze_one_step(E, S, P, 1e-4)
```

If the ability of the enzyme E to catalyze this reaction is dependent on the site 'b' of E being unbound, then this macro must be called as

> catalyze_one_step(E(b=None), S, P, 1e-4)

and similarly if the substrate or product must be unbound.

Execution:

```
>>> Model()
<Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
>>> Monomer('E', ['b'])
Monomer('E', ['b'])
>>> Monomer('S', ['b'])
Monomer('S', ['b'])
>>> Monomer('P')
Monomer('P')
>>> catalyze_one_step(E, S, P, 1e-4)
ComponentSet([
 Rule('one_step_E_S_to_E_P', E() + S() >> E() + P(), one_step_E_S_to_E_P_kf),
 Parameter('one_step_E_S_to_E_P_kf', 0.0001),
 ])
```

`pysb.macros.`**`catalyze_one_step_reversible`**(*enzyme*, *substrate*, *product*, *klist*)

Create fwd and reverse rules for catalysis of the form:

```
E + S -> E + P
    P -> S
```

**Parameters** **enzyme, substrate, product** : Monomer or MonomerPattern

E, S and P in the above reactions.

**klist** : list of 2 Parameters or list of 2 numbers

A list containing the rate constant for catalysis and the rate constant for the conversion of product back to substrate (in that order). If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the names and states of S1 and S2 and these parameters will be included at the end of the returned component list.

**Returns** **components** : ComponentSet

The generated components. Contains two rules (the single-step catalysis rule and the product reversion rule) and optionally the two generated Parameter objects if klist was given as numbers.

### Notes

Calls the macro catalyze_one_step to generate the catalysis rule.

### Examples

One-step, pseudo-first order conversion of S to P by E:

```
Model()
Monomer('E', ['b'])
Monomer('S', ['b'])
```

```
    Monomer('P')
    catalyze_one_step_reversible(E, S, P, [1e-1, 1e-4])
```

Execution:

```
    >>> Model()
    <Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
    >>> Monomer('E', ['b'])
    Monomer('E', ['b'])
    >>> Monomer('S', ['b'])
    Monomer('S', ['b'])
    >>> Monomer('P')
    Monomer('P')
    >>> catalyze_one_step_reversible(E, S, P, [1e-1, 1e-4])
    ComponentSet([
     Rule('one_step_E_S_to_E_P', E() + S() >> E() + P(), one_step_E_S_to_E_P_kf),
     Parameter('one_step_E_S_to_E_P_kf', 0.1),
     Rule('reverse_P_to_S', P() >> S(), reverse_P_to_S_kr),
     Parameter('reverse_P_to_S_kr', 0.0001),
     ])
```

`pysb.macros.`**`synthesize`**`(`*species*, *ksynth*`)`

Generate a reaction which synthesizes a species.

Note that *species* must be "concrete", i.e. the state of all sites in all of its monomers must be specified. No site may be left unmentioned.

**Parameters** **species** : Monomer, MonomerPattern or ComplexPattern

The species to synthesize. If a Monomer, sites are considered as unbound and in their default state. If a pattern, must be concrete.

**ksynth** : Parameters or number

Synthesis rate. If a Parameter is passed, it will be used directly in the generated Rule. If a number is passed, a Parameter will be created with an automatically generated name based on the names and site states of the components of *species* and this parameter will be included at the end of the returned component list.

**Returns** **components** : ComponentSet

The generated components. Contains the unidirectional synthesis Rule and optionally a Parameter if ksynth was given as a number.

### Examples

Synthesize A with site x unbound and site y in state 'e':

```
    Model()
    Monomer('A', ['x', 'y'], {'y': ['e', 'f']})
    synthesize(A(x=None, y='e'), 1e-4)
```

Execution:

```
    >>> Model()
    <Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
    >>> Monomer('A', ['x', 'y'], {'y': ['e', 'f']})
    Monomer('A', ['x', 'y'], {'y': ['e', 'f']})
    >>> synthesize(A(x=None, y='e'), 1e-4)
    ComponentSet([
     Rule('synthesize_Ae', None >> A(x=None, y='e'), synthesize_Ae_k),
     Parameter('synthesize_Ae_k', 0.0001),
     ])
```

pysb.macros.**degrade**(*species*, *kdeg*)

Generate a reaction which degrades a species.

Note that *species* is not required to be "concrete".

**Parameters** **species** : Monomer, MonomerPattern or ComplexPattern

The species to synthesize. If a Monomer, sites are considered as unbound and in their default state. If a pattern, must be concrete.

**kdeg** : Parameters or number

Degradation rate. If a Parameter is passed, it will be used directly in the generated Rule. If a number is passed, a Parameter will be created with an automatically generated name based on the names and site states of the components of *species* and this parameter will be included at the end of the returned component list.

**Returns** **components** : ComponentSet

The generated components. Contains the unidirectional degradation Rule and optionally a Parameter if ksynth was given as a number.

### Examples

Degrade all B, even bound species:

```
Model()
Monomer('B', ['x'])
degrade(B(), 1e-6)
```

Execution:

```
>>> Model()
<Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
>>> Monomer('B', ['x'])
Monomer('B', ['x'])
>>> degrade(B(), 1e-6)
ComponentSet([
 Rule('degrade_B', B() >> None, degrade_B_k),
 Parameter('degrade_B_k', 1e-06),
 ])
```

pysb.macros.**synthesize_degrade_table**(*table*)

Generate a table of synthesis and degradation reactions.

Given a list of species, calls the *synthesize* and *degrade* macros on each one. The species and the parameter values are passed as a list of lists (i.e. a table) with each inner list consisting of the species, forward and reverse rates (in that order).

Each species' associated pair of rates may be either Parameters or numbers. If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the names and states of the relevant species and these parameters will be included in the returned component list. To omit any individual reaction, pass None in place of the corresponding parameter.

Note that any *species* with a non-None synthesis rate must be "concrete".

**Parameters** **table** : list of lists

Table of species and rates, as described above.

**Returns** **components** : ComponentSet

The generated components. Contains the unidirectional synthesis and degradation Rules and optionally the Parameters for any rates given as numbers.

### Examples

Specify synthesis and degradation reactions for A and B in a table:

```
Model()
Monomer('A', ['x', 'y'], {'y': ['e', 'f']})
Monomer('B', ['x'])
synthesize_degrade_table([[A(x=None, y='e'), 1e-4, 1e-6],
                          [B(),              None, 1e-7]])
```

Execution:

```
>>> Model()
<Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
>>> Monomer('A', ['x', 'y'], {'y': ['e', 'f']})
Monomer('A', ['x', 'y'], {'y': ['e', 'f']})
>>> Monomer('B', ['x'])
Monomer('B', ['x'])
>>> synthesize_degrade_table([[A(x=None, y='e'), 1e-4, 1e-6],
...                           [B(),              None, 1e-7]])
ComponentSet([
    Rule('synthesize_Ae', None >> A(x=None, y='e'), synthesize_Ae_k),
    Parameter('synthesize_Ae_k', 0.0001),
    Rule('degrade_Ae', A(x=None, y='e') >> None, degrade_Ae_k),
    Parameter('degrade_Ae_k', 1e-06),
    Rule('degrade_B', B() >> None, degrade_B_k),
    Parameter('degrade_B_k', 1e-07),
    ])
```

`pysb.macros.`**`assemble_pore_sequential`**(*subunit*, *site1*, *site2*, *max_size*, *ktable*)

Generate rules to assemble a circular homomeric pore sequentially.

The pore species are created by sequential addition of *subunit* monomers, i.e. larger oligomeric species never fuse together. The pore structure is defined by the *pore_species* macro.

**Parameters** **subunit** : Monomer or MonomerPattern

The subunit of which the pore is composed.

**site1, site2** : string

The names of the sites where one copy of *subunit* binds to the next.

**max_size** : integer

The maximum number of subunits in the pore.

**ktable** : list of lists of Parameters or numbers

Table of forward and reverse rate constants for the assembly steps. The outer list must be of length *max_size* - 1, and the inner lists must all be of length 2. In the outer list, the first element corresponds to the first assembly step in which two monomeric subunits bind to form a 2-subunit complex, and the last element corresponds to the final step in which the *max_size'th subunit is added. Each inner list contains the forward and* *reverse rate constants (in that order) for the corresponding assembly reaction, and each of these pairs must comprise solely Parameter objects or solely numbers (never one of each). If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on 'subunit*, *site1*, *site2* and the pore sizes and these parameters will be included at the end of the returned component list.

### Examples

Assemble a three-membered pore by sequential addition of monomers, with the same forward/reverse rates for monomer-monomer and monomer-dimer interactions:

```
Model()
Monomer('Unit', ['p1', 'p2'])
assemble_pore_sequential(Unit, 'p1', 'p2', 3, [[1e-4, 1e-1]] * 2)
```

Execution:

```
>>> Model()
<Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
>>> Monomer('Unit', ['p1', 'p2'])
Monomer('Unit', ['p1', 'p2'])
>>> assemble_pore_sequential(Unit, 'p1', 'p2', 3, [[1e-4, 1e-1]] * 2)
ComponentSet([
 Rule('assemble_pore_sequential_Unit_2',
      Unit(p1=None, p2=None) + Unit(p1=None, p2=None) <>
          Unit(p1=None, p2=1) % Unit(p1=1, p2=None),
      assemble_pore_sequential_Unit_2_kf,
      assemble_pore_sequential_Unit_2_kr),
 Parameter('assemble_pore_sequential_Unit_2_kf', 0.0001),
 Parameter('assemble_pore_sequential_Unit_2_kr', 0.1),
 Rule('assemble_pore_sequential_Unit_3',
      Unit(p1=None, p2=None) + Unit(p1=None, p2=1) % Unit(p1=1, p2=None) <>
          MatchOnce(Unit(p1=3, p2=1) % Unit(p1=1, p2=2) % Unit(p1=2, p2=3)),
      assemble_pore_sequential_Unit_3_kf,
      assemble_pore_sequential_Unit_3_kr),
 Parameter('assemble_pore_sequential_Unit_3_kf', 0.0001),
 Parameter('assemble_pore_sequential_Unit_3_kr', 0.1),
 ])
```

`pysb.macros.`**`pore_transport`**(*subunit*, *sp_site1*, *sp_site2*, *sc_site*, *min_size*, *max_size*, *csource*, *c_site*, *cdest*, *ktable*)

Generate rules to transport cargo through a circular homomeric pore.

The pore structure is defined by the *pore_species* macro – *subunit* monomers bind to each other from *sp_site1* to *sp_site2* to form a closed ring. The transport reaction is modeled as a catalytic process of the form pore + csource <> pore:csource >> pore + cdest

**Parameters subunit** : Monomer or MonomerPattern

Subunit of which the pore is composed.

**sp_site1, sp_site2** : string

Names of the sites where one copy of *subunit* binds to the next.

**sc_site** : string

Name of the site on *subunit* where it binds to the cargo *csource*.

**min_size, max_size** : integer

Minimum and maximum number of subunits in the pore at which transport will occur.

**csource** : Monomer or MonomerPattern

Cargo “source”, i.e. the entity to be transported.

**c_site** : string

Name of the site on *csource* where it binds to *subunit*.

**cdest** : Monomer or MonomerPattern

Cargo “destination”, i.e. the resulting state after the transport event.

**ktable** : list of lists of Parameters or numbers

Table of forward, reverse and catalytic rate constants for the transport reactions. The outer list must be of length *max_size* - *min_size* + 1, and the inner lists must all be of length 3. In the outer list, the first element corresponds to the transport through the pore of size *min_size* and the last element to that of size *max_size*. Each inner list contains the forward, reverse and catalytic rate constants (in that order) for the corresponding transport reaction, and each of these pairs must comprise solely Parameter objects or solely numbers (never some of each). If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the subunit, the pore size and the cargo, and these parameters will be included at the end of the returned component list.

## Examples

Specify that a three-membered pore is capable of transporting cargo from the mitochondria to the cytoplasm:

```
Model()
Monomer('Unit', ['p1', 'p2', 'sc_site'])
Monomer('Cargo', ['c_site', 'loc'], {'loc':['mito', 'cyto']})
pore_transport(Unit, 'p1', 'p2', 'sc_site', 3, 3,
               Cargo(loc='mito'), 'c_site', Cargo(loc='cyto'),
               [[1e-4, 1e-1, 1]])
```

Generates two rules–one (reversible) binding rule and one transport rule–and the three associated parameters.

Execution:

```
>>> Model()
<Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
>>> Monomer('Unit', ['p1', 'p2', 'sc_site'])
Monomer('Unit', ['p1', 'p2', 'sc_site'])
>>> Monomer('Cargo', ['c_site', 'loc'], {'loc':['mito', 'cyto']})
Monomer('Cargo', ['c_site', 'loc'], {'loc': ['mito', 'cyto']})
>>> pore_transport(Unit, 'p1', 'p2', 'sc_site', 3, 3,
...                Cargo(loc='mito'), 'c_site', Cargo(loc='cyto'),
...                [[1e-4, 1e-1, 1]])
ComponentSet([
 Rule('pore_transport_complex_Unit_3_Cargomito',
     MatchOnce(Unit(p1=3, p2=1, sc_site=None) %
         Unit(p1=1, p2=2, sc_site=None) %
         Unit(p1=2, p2=3, sc_site=None)) +
         Cargo(c_site=None, loc='mito') <>
     MatchOnce(Unit(p1=3, p2=1, sc_site=4) %
         Unit(p1=1, p2=2, sc_site=None) %
```

```
            Unit(p1=2, p2=3, sc_site=None) %
            Cargo(c_site=4, loc='mito')),
        pore_transport_complex_Unit_3_Cargomito_kf,
        pore_transport_complex_Unit_3_Cargomito_kr),
    Parameter('pore_transport_complex_Unit_3_Cargomito_kf', 0.0001),
    Parameter('pore_transport_complex_Unit_3_Cargomito_kr', 0.1),
    Rule('pore_transport_dissociate_Unit_3_Cargocyto',
        MatchOnce(Unit(p1=3, p2=1, sc_site=4) %
            Unit(p1=1, p2=2, sc_site=None) %
            Unit(p1=2, p2=3, sc_site=None) %
            Cargo(c_site=4, loc='mito')) >>
        MatchOnce(Unit(p1=3, p2=1, sc_site=None) %
            Unit(p1=1, p2=2, sc_site=None) %
            Unit(p1=2, p2=3, sc_site=None)) +
            Cargo(c_site=None, loc='cyto'),
        pore_transport_dissociate_Unit_3_Cargocyto_kc),
    Parameter('pore_transport_dissociate_Unit_3_Cargocyto_kc', 1.0),
    ])
```

`pysb.macros.`**`pore_bind`**`(`*subunit*, *sp_site1*, *sp_site2*, *sc_site*, *size*, *cargo*, *c_site*, *klist*`)`

Generate rules to bind a monomer to a circular homomeric pore.

The pore structure is defined by the *pore_species* macro – *subunit* monomers bind to each other from *sp_site1* to *sp_site2* to form a closed ring. The binding reaction takes the form pore + cargo <> pore:cargo.

**Parameters** **subunit** : Monomer or MonomerPattern

Subunit of which the pore is composed.

**sp_site1, sp_site2** : string

Names of the sites where one copy of *subunit* binds to the next.

**sc_site** : string

Name of the site on *subunit* where it binds to the cargo *cargo*.

**size** : integer

Number of subunits in the pore at which binding will occur.

**cargo** : Monomer or MonomerPattern

Cargo that binds to the pore complex.

**c_site** : string

Name of the site on *cargo* where it binds to *subunit*.

**klist** : list of Parameters or numbers

List containing forward and reverse rate constants for the binding reaction (in that order). Rate constants should either be both Parameter objects or both numbers. If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the subunit, the pore size and the cargo, and these parameters will be included at the end of the returned component list.

## Examples

Specify that a cargo molecule can bind reversibly to a 3-membered pore:

```
Model()
Monomer('Unit', ['p1', 'p2', 'sc_site'])
Monomer('Cargo', ['c_site'])
pore_bind(Unit, 'p1', 'p2', 'sc_site', 3,
          Cargo(), 'c_site', [1e-4, 1e-1, 1])
```

Execution:

```
>>> Model()
<Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
>>> Monomer('Unit', ['p1', 'p2', 'sc_site'])
Monomer('Unit', ['p1', 'p2', 'sc_site'])
>>> Monomer('Cargo', ['c_site'])
Monomer('Cargo', ['c_site'])
>>> pore_bind(Unit, 'p1', 'p2', 'sc_site', 3,
...           Cargo(), 'c_site', [1e-4, 1e-1, 1])
ComponentSet([
 Rule('pore_bind_Unit_3_Cargo',
     MatchOnce(Unit(p1=3, p2=1, sc_site=None) %
         Unit(p1=1, p2=2, sc_site=None) %
         Unit(p1=2, p2=3, sc_site=None)) +
         Cargo(c_site=None) <>
     MatchOnce(Unit(p1=3, p2=1, sc_site=4) %
         Unit(p1=1, p2=2, sc_site=None) %
         Unit(p1=2, p2=3, sc_site=None) %
         Cargo(c_site=4)),
     pore_bind_Unit_3_Cargo_kf, pore_bind_Unit_3_Cargo_kr),
 Parameter('pore_bind_Unit_3_Cargo_kf', 0.0001),
 Parameter('pore_bind_Unit_3_Cargo_kr', 0.1),
 ])
```

`pysb.macros.`**`assemble_chain_sequential_base`**(*base*, *basesite*, *subunit*, *site1*, *site2*, *max_size*, *ktable*, *comp=1*)

Generate rules to assemble a homomeric chain sequentially onto a base complex (only the subunit creates repeating chain, not the base).

The chain species are created by sequential addition of *subunit* monomers. The chain structure is defined by the *pore_species_base* macro.

**Parameters** **base** : Monomer or MonomerPattern

The base complex to which the chain is attached.

**basesite** : string

The name of the site on the complex to which chain attaches.

**subunit** : Monomer or MonomerPattern

The subunit of which the chain is composed.

**site1, site2** : string

The names of the sites where one copy of *subunit* binds to the next; the first will also be the site where the first subunit binds the base.

**max_size** : integer

The maximum number of subunits in the chain.

**ktable** : list of lists of Parameters or numbers

Table of forward and reverse rate constants for the assembly steps. The outer list must be of length *max_size* + 1, and the inner lists must all be of length 2. In the outer list, the first element corresponds to the first assembly step in which the complex binds the first subunit. The next corresponds to a bound subunit binding to form a 2-subunit complex, and the last element corresponds to the final step in which the *max_size'th subunit is added. Each inner list contains the forward and reverse rate constants (in that order) for the corresponding assembly reaction, and each of these pairs must comprise solely Parameter objects or solely numbers (never one of each). If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on 'subunit*, *site1*, *site2* and the chain sizes and these parameters will be included at the end of the returned component list.

**comp** : optional; a ComplexPattern to which the base molecule is attached.

### Examples

Assemble a three-membered chain by sequential addition of monomers to a base, which is in turn attached to a complex, with the same forward/reverse rates for monomer-monomer and monomer-dimer interactions:

```
Model()
Monomer('Base', ['b1', 'b2'])
Monomer('Unit', ['p1', 'p2'])
Monomer('Complex1', ['s1'])
Monomer('Complex2', ['s1', s2'])
assemble_chain_sequential(Base(b2=ANY), 'b1', Unit, 'p1', 'p2', 3, [[1e-4, 1e-1]] * 2, Complex1(
```

Execution:

```
>>> Model()
<Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
>>> Monomer('Base', ['b1', 'b2'])
Monomer('Base', ['b1', 'b2'])
>>> Monomer('Unit', ['p1', 'p2'])
Monomer('Unit', ['p1', 'p2'])
>>> Monomer('Complex1', ['s1'])
Monomer('Complex1', ['s1'])
>>> Monomer('Complex2', ['s1', 's2'])
Monomer('Complex2', ['s1', 's2'])
>>> assemble_chain_sequential_base(Base(b2=ANY), 'b1', Unit, 'p1', 'p2', 3, [[1e-4, 1e-1]] * 2,
ComponentSet([
 Rule('assemble_chain_sequential_base_Unit_2', Unit(p1=None, p2=None) + Complex1(s1=<class 'pysb
 Parameter('assemble_chain_sequential_base_Unit_2_kf', 0.0001),
 Parameter('assemble_chain_sequential_base_Unit_2_kr', 0.1),
 Rule('assemble_chain_sequential_base_Unit_3', Unit(p1=None, p2=None) + Complex1(s1=<class 'pysb
 Parameter('assemble_chain_sequential_base_Unit_3_kf', 0.0001),
 Parameter('assemble_chain_sequential_base_Unit_3_kr', 0.1),
 ])
```

`pysb.macros.`**`bind_complex`**(*s1*, *site1*, *s2*, *site2*, *klist*, *m1=None*, *m2=None*)

Generate the reversible binding reaction S1 + S2 <> S1:S2, with optional complexes attached to either S1 (C1:S1 + S2 <> C1:S1:S2), S2 (S1 + C2:S2 <> C2:S2:S1), or both (C1:S1 + C2:S2 <> C1:S1:S2:C2).

**Parameters s1, s2** : Monomer, MonomerPattern, or ComplexPattern

Monomers or complexes participating in the binding reaction.

**site1, site2** : string

The names of the sites on s1 and s2 used for binding.

**klist** : list of 2 Parameters or list of 2 numbers

Forward and reverse rate constants (in that order). If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the names and states of S1 and S2 and these parameters will be included at the end of the returned component list.

**m1, m2** : Monomer or MonomerPattern

If either binding site is present in multiple monomers within a complex, the specific monomer desired for binding must be specified.

**Returns components** : ComponentSet

The generated components. Contains the bidirectional binding Rule and optionally two Parameters if klist was given as numbers.

### Examples

Binding between A:B and C:D:

```
Model()
Monomer('A', ['a', 'b'])
Monomer('B', ['c', 'd'])
Monomer('C', ['e', 'f'])
Monomer('D', ['g', 'h'])
bind_complex(A(a=1) % B(c=1), 'b', C(e=2) % D(g=2), 'h', [1e-4, 1e-1])
```

Execution:

```
>>> Model()
<Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
>>> Monomer('A', ['a', 'b'])
Monomer('A', ['a', 'b'])
>>> Monomer('B', ['c', 'd'])
Monomer('B', ['c', 'd'])
>>> Monomer('C', ['e', 'f'])
Monomer('C', ['e', 'f'])
>>> Monomer('D', ['g', 'h'])
Monomer('D', ['g', 'h'])
>>> bind(A, 'a', B, 'c', [1e4, 1e-1])
ComponentSet([
 Rule('bind_A_B', A(a=None) + B(c=None) <> A(a=1) % B(c=1), bind_A_B_kf, bind_A_B_kr),
 Parameter('bind_A_B_kf', 10000.0),
 Parameter('bind_A_B_kr', 0.1),
 ])
>>> bind(C, 'e', D, 'g', [1e4, 1e-1])
ComponentSet([
 Rule('bind_C_D', C(e=None) + D(g=None) <> C(e=1) % D(g=1), bind_C_D_kf, bind_C_D_kr),
 Parameter('bind_C_D_kf', 10000.0),
 Parameter('bind_C_D_kr', 0.1),
 ])
>>> bind_complex(A(a=1) % B(c=1), 'b', C(e=2) % D(g=2), 'h', [1e-4, 1e-1])
ComponentSet([
 Rule('bind_AB_DC', A(a=1, b=None) % B(c=1) + D(g=3, h=None) % C(e=3) <> A(a=1, b=50) % B(c=1) %
 Parameter('bind_AB_DC_kf', 0.0001),
 Parameter('bind_AB_DC_kr', 0.1),
 ])
```

pysb.macros.**bind_table_complex**(*bindtable*, *row_site*, *col_site*, *m1=None*, *m2=None*, *kf=None*)

Generate a table of reversible binding reactions when either the row or column species (or both) have a complex bound to them.

Given two lists of species R and C (each with an optional attached complex), calls the *bind* macro on each pairwise combination (R[i], C[j]). The species lists and the parameter values are passed as a list of lists (i.e. a table) with elements of R passed as the "row headers", elements of C as the "column headers", and forward / reverse rate pairs (in that order) as tuples in the "cells". For example with two elements in each of R and C, the table would appear as follows (note that the first row has one fewer element than the subsequent rows):

```
[[              C1,           C2],
 [R1, (1e-4, 1e-1), (2e-4, 2e-1)],
 [R2, (3e-4, 3e-1), (4e-4, 4e-1)]]
```

Each parameter tuple may contain Parameters or numbers. If Parameters are passed, they will be used directly in the generated Rules. If numbers are passed, Parameters will be created with automatically generated names based on the names and states of the relevant species and these parameters will be included at the end of the returned component list. To omit any individual reaction, pass None in place of the corresponding parameter tuple.

Alternately, single kd values (dissociation constant, kr/kf) may be specified instead of (kf, kr) tuples. If kds are used, a single shared kf Parameter or number must be passed as an extra *kf* argument. kr values for each binding reaction will be calculated as kd*kf. It is important to remember that the forward rate constant is a single parameter shared across the entire bind table, as this may have implications for parameter fitting.

**Parameters** **bindtable** : list of lists

Table of reactants and rates, as described above.

**row_site, col_site** : string

The names of the sites on the elements of R and C, respectively, used for binding.

**m1, m2** : Monomer or MonomerPattern

If either binding site is present in multiple monomers within a complex, the specific monomer desired for binding must be specified.

**kf** : Parameter or number, optional

If the "cells" in bindtable are given as single kd values, this is the shared kf used to calculate the kr values.

**Returns** **components** : ComponentSet

The generated components. Contains the bidirectional binding Rules and optionally the Parameters for any parameters given as numbers.

## Examples

Binding table for two species types (R and C, each with an attached complex), each with two members:

```
Model()
Monomer('R1', ['x', 'c1'])
Monomer('R2', ['x', 'c1'])
Monomer('C1', ['y', 'c2'])
Monomer('C2', ['y', 'c2'])
bind(C1, 'y', C2, 'y', [1e4, 1e-1])
bind(R1, 'c1', R2, 'x', [1e3, 1e-1])
bind_table_complex([[C1(y=1) % C2(y=1),           C2],
                   [R1,  (1e-4, 1e-1),  (2e-4, 2e-1)],
                   [R1(c1=1) % R2(x=1),  (3e-4, 3e-1), None]], 'x', 'c2', m1=R1, m2=C2)
```

Execution:

```
>>> Model()
<Model '_interactive_' (monomers: 0, rules: 0, parameters: 0, expressions: 0, compartments: 0) a
>>> Monomer('R1', ['x', 'c1'])
Monomer('R1', ['x', 'c1'])
>>> Monomer('R2', ['x', 'c1'])
Monomer('R2', ['x', 'c1'])
>>> Monomer('C1', ['y', 'c2'])
Monomer('C1', ['y', 'c2'])
>>> Monomer('C2', ['y', 'c2'])
Monomer('C2', ['y', 'c2'])
>>> bind(C1, 'y', C2, 'y', [1e4, 1e-1])
ComponentSet([
 Rule('bind_C1_C2', C1(y=None) + C2(y=None) <> C1(y=1) % C2(y=1), bind_C1_C2_kf, bind_C1_C2_kr),
 Parameter('bind_C1_C2_kf', 10000.0),
 Parameter('bind_C1_C2_kr', 0.1),
 ])
>>> bind(R1, 'c1', R2, 'x', [1e3, 1e-1])
ComponentSet([
 Rule('bind_R1_R2', R1(c1=None) + R2(x=None) <> R1(c1=1) % R2(x=1), bind_R1_R2_kf, bind_R1_R2_kr
 Parameter('bind_R1_R2_kf', 1000.0),
 Parameter('bind_R1_R2_kr', 0.1),
 ])
>>> bind_table_complex([[                    C1(y=1) % C2(y=1), C2],
ComponentSet([
 Rule('bind_C2C1_R1', C2(y=1, c2=None) % C1(y=2) + R1(x=None) <> C2(y=1, c2=50) % C1(y=2) % R1(x
 Parameter('bind_C2C1_R1_kf', 0.0001),
 Parameter('bind_C2C1_R1_kr', 0.1),
 Rule('bind_R1_C2', R1(x=None) + C2(c2=None) <> R1(x=1) % C2(c2=1), bind_R1_C2_kf, bind_R1_C2_kr
 Parameter('bind_R1_C2_kf', 0.0002),
 Parameter('bind_R1_C2_kr', 0.2),
 Rule('bind_R1R2_C2C1', R1(x=None, c1=1) % R2(x=1) + C2(y=2, c2=None) % C1(y=2) <> R1(x=50, c1=1
 Parameter('bind_R1R2_C2C1_kf', 0.0003),
 Parameter('bind_R1R2_C2C1_kr', 0.3),
 ])
```

## 3.6 Visualizing model structure

PySB currently includes a handful of tools for visualizing the structure of models. These can be supplemented with existing tools for visualizing the structure of rule-based models (e.g., contact maps and influence maps for Kappa models).

### 3.6.1 Render a model's reaction network (`pysb.tools.render_reactions`)

#### Usage

Usage: `python -m pysb.tools.render_reactions mymodel.py > mymodel.dot`

Renders the reactions produced by a model into the "dot" graph format which can be visualized with Graphviz.

To create a PDF from the .dot file, use the "dot" command from Graphviz:

```
dot mymodel.dot -T pdf -O
```

This will create mymodel.dot.pdf. You can also change the "dot" command to one of the other Graphviz drawing tools for a different type of layout. Note that you can pipe the output of render_reactions straight into Graphviz without creating an intermediate .dot file, which is especially helpful if you are making continuous changes to the model and need to visualize your changes repeatedly:

```
python -m pysb.tools.render_reactions mymodel.py | dot -T pdf -o mymodel.pdf
```

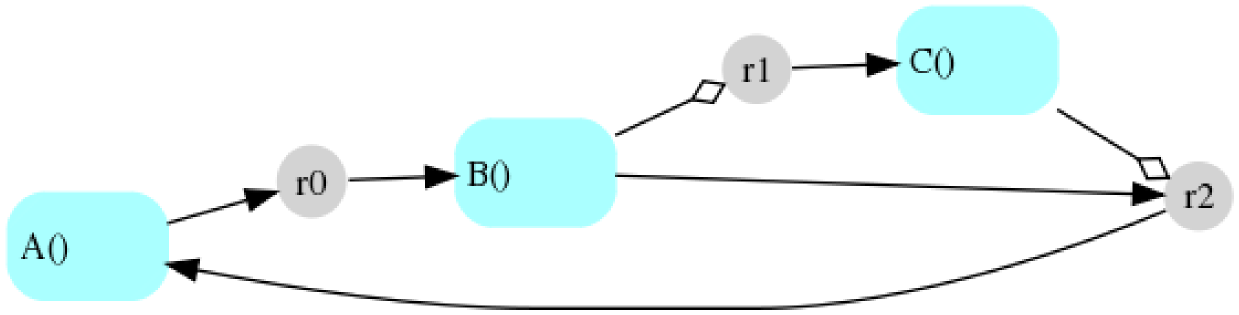
Note that some PDF viewers will auto-reload a changed PDF, so you may not even need to manually reopen it every time you rerun the tool.

### Output for Robertson example model

The Robertson example model (in `pysb.examples.robertson`) contains the following three reactions:

- A -> B
- B + B -> B + C
- C + B -> C + A

The reaction network diagram for this system as generated by this module and rendered using `dot` is shown below:



Circular nodes (`r0`, `r1` and `r2`) indicate reactions; square nodes (`A()`, `B()` and `C()`) indicate species. Incoming arrows from a species node to a reaction node indicate that the species is a reactant; outgoing arrows from a reaction node to a species node indicate that the species is a product. A hollow diamond-tipped arrow from a species to a reaction indicates that the species is involved as both a reactant and a product, i.e., it serves as a "modifier" (enzyme or catalyst).

`pysb.tools.render_reactions.`**`run`**(*model*)

Render the reactions produced by a model into the "dot" graph format.

**Parameters** **model** : pysb.core.Model

The model to render.

**Returns** string

The dot format output.

## 3.6.2 Render a model's species (`pysb.tools.render_species`)

Usage: `python -m pysb.tools.render_species mymodel.py > mymodel.dot`

Renders the species from a model into the "dot" graph format which can be visualized with Graphviz.

To create a PDF from the .dot file, use the Graphviz tools in the following command pipeline:

```
ccomps -x mymodel.dot | dot | gvpack -m0 | neato -n2 -T pdf -o mymodel.pdf
```

You can also change the "dot" command to "circo" or "sfdp" for a different type of layout. Note that you can pipe the output of render_species straight into a Graphviz command pipeline without creating an intermediate .dot file, which is especially helpful if you are making continuous changes to the model and need to visualize your changes repeatedly:

```
python -m pysb.tools.render_species mymodel.py | ccomps -x | dot |
  gvpack -m0 | neato -n2 -T pdf -o mymodel.pdf
```

Note that some PDF viewers will auto-reload a changed PDF, so you may not even need to manually reopen it every time you rerun the tool.

`pysb.tools.render_species.`**`run`**`(`*model*`)`

Render the species from a model into the "dot" graph format.

**Parameters** **model** : pysb.core.Model

The model to render.

**Returns** string

The dot format output.

## 3.7 Exporting to other formats (`pysb.export`)

Tools for exporting PySB models to a variety of other formats.

Exporting can be performed at the command-line or programmatically/interactively from within Python.

### 3.7.1 Command-line usage

At the command-line, run as follows:

```
python -m pysb.export model.py <format>
```

where `model.py` is a file containing a PySB model definition (i.e., contains an instance of `pysb.core.Model` instantiated as a global variable). `[format]` should be the name of one of the supported formats:

- `bngl`
- `bng_net`
- `kappa`
- `potterswheel`
- `sbml`
- `python`
- `mathematica`
- `matlab`

In all cases, the exported model code will be printed to standard out, allowing it to be inspected or redirected to another file.

### 3.7.2 Interactive usage

Export functionality is implemented by this module's top-level function `export`. For example, to export the "Robertson" example model as SBML, first import the model:

```
from pysb.examples.robertson import model
```

Then import the `export` function from this module:

```
from pysb.export import export
```

Call the `export` function, passing the model instance and a string indicating the desired format, which should be one of the ones indicated in the list in the "Command-line usage" section above:

```
sbml_output = export(model, 'sbml')
```

The output (a string) can be inspected or written to a file, e.g. as follows:

```
with open('robertson.sbml', 'w') as f:
    f.write(sbml_output)
```

### 3.7.3 Implementation of specific exporters

Information on the implementation of specific exporters can be found in the documentation for the exporter classes in the package *`pysb.export`*:

#### Export SBML (`pysb.export.sbml`)

Module containing a class for exporting a PySB model to SBML.

For information on how to use the model exporters, see the documentation for *`pysb.export`*.

**class** `pysb.export.sbml.`**`MathMLContentPrinter`**`(`*settings=None*`)`
Prints an expression to MathML without presentation markup.

**class** `pysb.export.sbml.`**`SbmlExporter`**`(`*model*, *docstring=None*`)`
A class for returning the SBML for a given PySB model.

Inherits from *`pysb.export.Exporter`*, which implements basic functionality for all exporters.

**`export`**`()`
Export the SBML for the PySB model associated with the exporter.

**Returns** string

String containing the SBML output.

`pysb.export.sbml.`**`indent`**`(`*text*, *n=0*`)`
Re-indent a multi-line string, stripping leading newlines and trailing spaces.

#### Export ODEs to MATLAB (`pysb.export.matlab`)

A class for converting a PySB model to a set of ordinary differential equations for integration in MATLAB.

Note that for use in MATLAB, the name of the `.m` file must match the name of the exported MATLAB class (e.g., `robertson.m` for the example below).

For information on how to use the model exporters, see the documentation for *`pysb.export`*.

#### Output for the Robertson example model

Information on the form and usage of the generated MATLAB class is contained in the documentation for the MATLAB model, as shown in the following example for `pysb.examples.robertson`:

```
classdef robertson
    % A simple three-species chemical kinetics system known as "Robertson's
    % example", as presented in:
    %
    % H. H. Robertson, The solution of a set of reaction rate equations, in Numerical
    % Analysis: An Introduction, J. Walsh, ed., Academic Press, 1966, pp. 178-182.
    %
    % A class implementing the ordinary differential equations
    % for the robertson model.
    %
    % Save as robertson.m.
    %
    % Generated by pysb.export.matlab.MatlabExporter.
    %
    % Properties
    % ----------
    % observables : struct
    %     A struct containing the names of the observables from the
    %     PySB model as field names. Each field in the struct
    %     maps the observable name to a matrix with two rows:
    %     the first row specifies the indices of the species
    %     associated with the observable, and the second row
    %     specifies the coefficients associated with the species.
    %     For any given timecourse of model species resulting from
    %     integration, the timecourse for an observable can be
    %     retrieved using the get_observable method, described
    %     below.
    %
    % parameters : struct
    %     A struct containing the names of the parameters from the
    %     PySB model as field names. The nominal values are set by
    %     the constructor and their values can be overriden
    %     explicitly once an instance has been created.
    %
    % Methods
    % -------
    % robertson.odes(tspan, y0)
    %     The right-hand side function for the ODEs of the model,
    %     for use with MATLAB ODE solvers (see Examples).
    %
    % robertson.get_initial_values()
    %     Returns a vector of initial values for all species,
    %     specified in the order that they occur in the original
    %     PySB model (i.e., in the order found in model.species).
    %     Non-zero initial conditions are specified using the
    %     named parameters included as properties of the instance.
    %     Hence initial conditions other than the defaults can be
    %     used by assigning a value to the named parameter and then
    %     calling this method. The vector returned by the method
    %     is used for integration by passing it to the MATLAB
    %     solver as the y0 argument.
    %
    % robertson.get_observables(y)
    %     Given a matrix of timecourses for all model species
    %     (i.e., resulting from an integration of the model),
    %     get the trajectories corresponding to the observables.
    %     Timecourses are returned as a struct which can be
    %     indexed by observable name.
    %
    % Examples
    % --------
    % Example integration using default initial and parameter
    % values:
    %
    % >> m = robertson();
    % >> tspan = [0 100];
    % >> [t y] = ode15s(@m.odes, tspan, m.get_initial_values());
    %
    % Retrieving the observables:
    %
    % >> y_obs = m.get_observables(y)
    %
    properties
        observables
        parameters
    end

    methods
        function self = robertson()
            % Assign default parameter values
            self.parameters = struct( ...
                'k1', 0.040000000000000001, ...
                'k2', 30000000, ...
                'k3', 10000, ...
                'A_0', 1, ...
                'B_0', 0, ...
                'C_0', 0);

            % Define species indices (first row) and coefficients
            % (second row) of named observables
            self.observables = struct( ...
                'A_total', [1; 1], ...
                'B_total', [2; 1], ...
                'C_total', [3; 1]);
        end

        function initial_values = get_initial_values(self)
            % Return the vector of initial conditions for all
            % species based on the values of the parameters
            % as currently defined in the instance.

            initial_values = zeros(1,3);
            initial_values(1) = self.parameters.A_0; % A()
            initial_values(2) = self.parameters.B_0; % B()
            initial_values(3) = self.parameters.C_0; % C()
        end

        function y = odes(self, tspan, y0)
            % Right hand side function for the ODEs

            % Shorthand for the struct of model parameters
            p = self.parameters;

            % A();
            y(1,1) = -p.k1*y0(1) + p.k3*y0(2)*y0(3);
            % B();
            y(2,1) = p.k1*y0(1) - p.k2*power(y0(2), 2) - p.k3*y0(2)*y0(3);
            % C();
            y(3,1) = p.k2*power(y0(2), 2);
        end

        function y_obs = get_observables(self, y)
            % Retrieve the trajectories for the model observables
            % from a matrix of the trajectories of all model
            % species.

            % Initialize the struct of observable timecourses
            % that we will return
            y_obs = struct();

            % Iterate over the observables;
            observable_names = fieldnames(self.observables);
            for i = 1:numel(observable_names)
                obs_matrix = self.observables.(observable_names{i});
                species = obs_matrix(1, :);
                coefficients = obs_matrix(2, :);
                y_obs.(observable_names{i}) = ...
                                y(:, species) * coefficients';
            end
        end
    end
end
```

**class** `pysb.export.matlab.`**`MatlabExporter`**(*model*, *docstring=None*)

A class for returning the ODEs for a given PySB model for use in MATLAB.

Inherits from *`pysb.export.Exporter`*, which implements basic functionality for all exporters.

**`export`**()

Generate a MATLAB class definition containing the ODEs for the PySB model associated with the exporter.

**Returns** string

String containing the MATLAB code for an implementation of the model's ODEs.

## Export ODEs to Mathematica (`pysb.export.mathematica`)

Module containing a class for converting a PySB model to a set of ordinary differential equations for integration or analysis in Mathematica.

For information on how to use the model exporters, see the documentation for *`pysb.export`*.

### Output for the Robertson example model

The Mathematica code produced will follow the form as given below for `pysb.examples.robertson`:

```
(*
A simple three-species chemical kinetics system known as "Robertson's
example", as presented in:

H. H. Robertson, The solution of a set of reaction rate equations, in Numerical
Analysis: An Introduction, J. Walsh, ed., Academic Press, 1966, pp. 178-182.

Mathematica model definition file for model robertson.
Generated by pysb.export.mathematica.MathematicaExporter.

Run with (for example):
tmax = 10
soln = NDSolve[Join[odes, initconds], slist, {t, 0, tmax}]
Plot[s0[t] /. soln, {t, 0, tmax}, PlotRange -> All]
*)

(* Parameters *)
k1 = 0.040000000000000001;
k2 = 30000000;
k3 = 10000;
A0 = 1;
B0 = 0;
C0 = 0;

(* List of Species *)
(* s0[t] = A() *)
(* s1[t] = B() *)
(* s2[t] = C() *)

(* ODEs *)
odes = {
s0'[t] == -k1*s0[t] + k3*s1[t]*s2[t],
s1'[t] == k1*s0[t] - k2*s1[t]^2 - k3*s1[t]*s2[t],
s2'[t] == k2*s1[t]^2
}

(* Initial Conditions *)
initconds = {
s0[0] == A0,
s1[0] == B0,
s2[0] == C0
}

(* List of Variables (e.g., as an argument to NDSolve) *)
solvelist = {
s0[t],
s1[t],
s2[t]
}

(* Run the simulation -- example *)
tmax = 100
soln = NDSolve[Join[odes, initconds], solvelist, {t, 0, tmax}]

(* Observables *)
Atotal = (s0[t] * 1) /. soln
Btotal = (s1[t] * 1) /. soln
Ctotal = (s2[t] * 1) /. soln
```

The output consists of a block of commands that define the ODEs, parameters, species and other variables for the model, along with a set of descriptive comments. The sections are as follows:

- The header comments identify the model and show an example of how to integrate the ODEs in Mathematica.
- The parameters block defines the numerical values of the named parameters.
- The list of species gives the mapping between the indexed species (`s0`, `s1`, `s2`) and their representation in PySB (`A(), B(), C()`).
- The ODEs block defines the set of ordinary differential equations and assigns the set of equations to the variable `odes`.
- The initial conditions block defines the initial values for each species and assigns the set of conditions to the variable `initconds`.
- The "list of variables" block enumerates all of the species in the model (`s0[t]`, `s1[t]`, `s2[t]`) and assigns them to the variable `solvelist`; this list can be passed to the Mathematica command `NDSolve` to indicate the variables to be solved for.
- This is followed by an example of how to call `NDSolve` to integrate the equations.
- Finally, the observables block enumerates the observables in the model, expressing each one as a linear combination of the appropriate species in the model. The interpolating functions returned by `NDSolve` are substituted in from the solution variable `soln`, allowing the observables to be plotted.

Note that Mathematica does not permit underscores in variable names, so any underscores used in PySB variables will be removed (e.g., `A_total` will be converted to `Atotal`).

**class** `pysb.export.mathematica.`**`MathematicaExporter`**`(`*model*, *docstring=None*`)`

A class for returning the ODEs for a given PySB model for use in Mathematica.

Inherits from *`pysb.export.Exporter`*, which implements basic functionality for all exporters.

**`export`**`()`

Generate the corresponding Mathematica ODEs for the PySB model associated with the exporter.

**Returns** string

String containing the Mathematica code for the model's ODEs.

## Export ODEs to PottersWheel (`pysb.export.potterswheel`)

Module containing a class for converting a PySB model to an equivalent set of ordinary differential equations for integration or analysis in PottersWheel.

For information on how to use the model exporters, see the documentation for *`pysb.export`*.

### Output for the Robertson example model

The PottersWheel code produced will follow the form as given below for `pysb.examples.robertson`:

```
% A simple three-species chemical kinetics system known as "Robertson's
% example", as presented in:
%
% H. H. Robertson, The solution of a set of reaction rate equations, in Numerical
% Analysis: An Introduction, J. Walsh, ed., Academic Press, 1966, pp. 178-182.
%
% PottersWheel model definition file
% save as robertson.m
function m = robertson()
```

```
m = pwGetEmptyModel();

% meta information
m.ID          = 'robertson';
m.name        = 'robertson';
m.description = '';
m.authors     = {''};
m.dates       = {''};
m.type        = 'PW-1-5';

% dynamic variables
m = pwAddX(m, 's0', 1.000000e+00);
m = pwAddX(m, 's1', 0.000000e+00);
m = pwAddX(m, 's2', 0.000000e+00);

% dynamic parameters
m = pwAddK(m, 'k1', 4.000000e-02);
m = pwAddK(m, 'k2', 3.000000e+07);
m = pwAddK(m, 'k3', 1.000000e+04);
m = pwAddK(m, 'A_0', 1.000000e+00);
m = pwAddK(m, 'B_0', 0.000000e+00);
m = pwAddK(m, 'C_0', 0.000000e+00);

% ODEs
m = pwAddODE(m, 's0', '-k1*s0 + k3*s1*s2');
m = pwAddODE(m, 's1', 'k1*s0 - k2*power(s1, 2) - k3*s1*s2');
m = pwAddODE(m, 's2', 'k2*power(s1, 2)');

% observables
m = pwAddY(m, 'A_total', '1.000000 * s0');
m = pwAddY(m, 'B_total', '1.000000 * s1');
m = pwAddY(m, 'C_total', '1.000000 * s2');

% end of PottersWheel model robertson
```

**class** `pysb.export.potterswheel.`**`PottersWheelExporter`**(*model*, *docstring=None*)

A class for returning the PottersWheel equivalent for a given PySB model.

Inherits from *`pysb.export.Exporter`*, which implements basic functionality for all exporters.

**`export`**()

Generate the PottersWheel code for the ODEs of the PySB model associated with the exporter.

**Returns** string

String containing the PottersWheel code for the ODEs.

## Export BNGL (`pysb.export.bngl`)

Module containing a class for exporting a PySB model to BNGL.

Serves as a wrapper around `pysb.generator.bng.BngGenerator`.

For information on how to use the model exporters, see the documentation for *`pysb.export`*.

**class** `pysb.export.bngl.`**`BnglExporter`**(*model*, *docstring=None*)

A class for returning the BNGL for a given PySB model.

Inherits from *`pysb.export.Exporter`*, which implements basic functionality for all exporters.

**export**()

Generate the corresponding BNGL for the PySB model associated with the exporter. A wrapper around `pysb.generator.bng.BngGenerator`.

**Returns** string

The BNGL output for the model.

## Export BNGL NET file (`pysb.export.bng_net`)

Module containing a class for getting the BNGL NET file for a given PySB model.

Serves as a wrapper around *`pysb.bng.generate_network()`*, which generates the BNGL for the model and then invokes BNG to generate the NET file.

For information on how to use the model exporters, see the documentation for *`pysb.export`*.

**class** `pysb.export.bng_net.`**`BngNetExporter`**(*model*, *docstring=None*)

A class for generating the BNG NET file for a given PySB model.

Inherits from `pysb.export.Export`, which implements basic functionality for all exporters.

**export**()

Generate the BNGL NET file for the PySB model associated with the exporter. A wrapper around *`pysb.bng.generate_network()`*.

**Returns** string

The NET file output for the model, generated by BNG.

## Export Kappa (`pysb.export.kappa`)

Module containing a class for returning the Kappa equivalent for a given PySB model.

Serves as a wrapper around `pysb.generator.kappa.KappaGenerator`.

For information on how to use the model exporters, see the documentation for *`pysb.export`*.

**class** `pysb.export.kappa.`**`KappaExporter`**(*model*, *docstring=None*)

A class for returning the Kappa for a given PySB model.

Inherits from *`pysb.export.Exporter`*, which implements basic functionality for all exporters.

**export**(*dialect='kasim'*)

Generate the corresponding Kappa for the PySB model associated with the exporter. A wrapper around `pysb.generator.kappa.KappaGenerator`.

**Parameters** **dialect** : (optional) string, either 'kasim' (default) or 'complx'

The Kappa file syntax for the Kasim simulator is slightly different from that of the complx analyzer. This argument specifies which type of Kappa to produce ('kasim' is the default).

**Returns** string

The Kappa output.

## Export standalone Python (`pysb.export.python`)

A module containing a class that produces Python code for simulating a PySB model without requiring PySB itself (note that NumPy and SciPy are still required). This offers a way of distributing a model to those who do not have PySB.

For information on how to use the model exporters, see the documentation for *pysb.export*.

### Structure of the standalone Python code

The standalone Python code defines a class, `Model`, with a method `simulate` that can be used to simulate the model.

As shown in the code for the Robertson model below, the `Model` class defines the fields `parameters`, `observables`, and `initial_conditions` as lists of `collections.namedtuple` objects that allow access to the features of the model.

The `simulate` method has the following signature:

```
def simulate(self, tspan, param_values=None, view=False):
```

with arguments as follows:

- `tspan` specifies the array of timepoints
- `param_values` is an optional vector of parameter values that can be used to override the nominal values defined in the PySB model
- `view` is an optional boolean argument that specifies if the simulation output arrays are returned as copies (views) of the original. If True, returns copies of the arrays, allowing changes to be made to values in the arrays without affecting the originals.

`simulate` returns a tuple of two arrays. The first array is a matrix with timecourses for each species in the model as the columns. The second array is a numpy record array for the model's observables, which can be indexed by name.

### Output for the Robertson example model

Example code generated for the Robertson model, `pysb.examples.robertson`:

```
"""A simple three-species chemical kinetics system known as "Robertson's
example", as presented in:

H. H. Robertson, The solution of a set of reaction rate equations, in Numerical
Analysis: An Introduction, J. Walsh, ed., Academic Press, 1966, pp. 178-182.
"""

# exported from PySB model 'robertson'

import numpy
import scipy.weave, scipy.integrate
import collections
import itertools
import distutils.errors


_use_inline = False
# try to inline a C statement to see if inline is functional
try:
    scipy.weave.inline('int i;', force=1)
    _use_inline = True
except distutils.errors.CompileError:
    pass

Parameter = collections.namedtuple('Parameter', 'name value')
Observable = collections.namedtuple('Observable', 'name species coefficients')
Initial = collections.namedtuple('Initial', 'param_index species_index')


class Model(object):

    def __init__(self):
        self.y = None
        self.yobs = None
        self.integrator = scipy.integrate.ode(self.ode_rhs)
        self.integrator.set_integrator('vode', method='bdf',
                                       with_jacobian=True)
        self.y0 = numpy.empty(3)
        self.ydot = numpy.empty(3)
        self.sim_param_values = numpy.empty(6)
        self.parameters = [None] * 6
        self.observables = [None] * 3
        self.initial_conditions = [None] * 3

        self.parameters[0] = Parameter('k1', 0.040000000000000001)
        self.parameters[1] = Parameter('k2', 30000000)
        self.parameters[2] = Parameter('k3', 10000)
        self.parameters[3] = Parameter('A_0', 1)
        self.parameters[4] = Parameter('B_0', 0)
        self.parameters[5] = Parameter('C_0', 0)

        self.observables[0] = Observable('A_total', [0], [1])
        self.observables[1] = Observable('B_total', [1], [1])
        self.observables[2] = Observable('C_total', [2], [1])

        self.initial_conditions[0] = Initial(3, 0)
        self.initial_conditions[1] = Initial(4, 1)
        self.initial_conditions[2] = Initial(5, 2)

    if _use_inline:

        def ode_rhs(self, t, y, p):
            ydot = self.ydot
            scipy.weave.inline(r'''
                ydot[0] = -p[0]*y[0] + p[2]*y[1]*y[2];
                ydot[1] = p[0]*y[0] - p[1]*pow(y[1], 2) - p[2]*y[1]*y[2];
                ydot[2] = p[1]*pow(y[1], 2);
                ''', ['ydot', 't', 'y', 'p'])
            return ydot

    else:

        def ode_rhs(self, t, y, p):
            ydot = self.ydot
            ydot[0] = -p[0]*y[0] + p[2]*y[1]*y[2]
            ydot[1] = p[0]*y[0] - p[1]*pow(y[1], 2) - p[2]*y[1]*y[2]
            ydot[2] = p[1]*pow(y[1], 2)
            return ydot


    def simulate(self, tspan, param_values=None, view=False):
        if param_values is not None:
            # accept vector of parameter values as an argument
            if len(param_values) != len(self.parameters):
                raise Exception("param_values must have length %d" %
                                len(self.parameters))
            self.sim_param_values[:] = param_values
        else:
            # create parameter vector from the values in the model
            self.sim_param_values[:] = [p.value for p in self.parameters]
        self.y0.fill(0)
        for ic in self.initial_conditions:
            self.y0[ic.species_index] = self.sim_param_values[ic.param_index]
        if self.y is None or len(tspan) != len(self.y):
            self.y = numpy.empty((len(tspan), len(self.y0)))
            if len(self.observables):
                self.yobs = numpy.ndarray(len(tspan),
                                zip((obs.name for obs in self.observables),
                                    itertools.repeat(float)))
            else:
                self.yobs = numpy.ndarray((len(tspan), 0))
            self.yobs_view = self.yobs.view(float).reshape(len(self.yobs),
                                                           -1)
        # perform the actual integration
        self.integrator.set_initial_value(self.y0, tspan[0])
        self.integrator.set_f_params(self.sim_param_values)
        self.y[0] = self.y0
        t = 1
        while self.integrator.successful() and self.integrator.t < tspan[-1]:
            self.y[t] = self.integrator.integrate(tspan[t])
            t += 1
        for i, obs in enumerate(self.observables):
            self.yobs_view[:, i] = \
                (self.y[:, obs.species] * obs.coefficients).sum(1)
        if view:
            y_out = self.y.view()
            yobs_out = self.yobs.view()
            for a in y_out, yobs_out:
                a.flags.writeable = False
        else:
            y_out = self.y.copy()
            yobs_out = self.yobs.copy()
        return (y_out, yobs_out)
```

### Using the standalone Python model

An example usage pattern for the standalone Robertson model, once generated:

```
# Import the standalone model file
import robertson_standalone
import numpy
from matplotlib import pyplot as plt
```

```
# Instantiate the model object (the constructor takes no arguments)
model = robertson_standalone.Model()

# Simulate the model
tspan = numpy.linspace(0, 100)
(species_output, observables_output) = model.simulate(tspan)

# Plot the results
plt.figure()
plt.plot(tspan, observables_output['A_total'])
plt.show()
```

**class** `pysb.export.python.`**`PythonExporter`**(*model*, *docstring=None*)

A class for returning the standalone Python code for a given PySB model.

Inherits from *`pysb.export.Exporter`*, which implements basic functionality for all exporters.

**`export`**()

Export Python code for simulation of a model without PySB.

**Returns** string

String containing the standalone Python code.

**class** `pysb.export.`**`Exporter`**(*model*, *docstring=None*)

Base class for all PySB model exporters.

Export functionality is implemented by subclasses of this class. The pattern for model export is the same for all exporter subclasses: a model is passed to the exporter constructor and the `export` method on the instance is called.

**Parameters** **model** : pysb.core.Model

The model to export.

**docstring** : string (optional)

The header comment to include at the top of the exported file.

### Examples

Exporting the "Robertson" example model to SBML using the `SbmlExporter` subclass:

```
>>> from pysb.examples.robertson import model
>>> from pysb.export.sbml import SbmlExporter
>>> e = SbmlExporter(model)
>>> sbml_output = e.export()
```

**`docstring`** = **None**

Header comment to include at the top of the exported file.

**`export`**()

The export method, which must be implemented by any subclass.

All implementations of this method are expected to return a single string containing the representation of the model in the desired format.

**`model`** = **None**

The model to export.

`pysb.export.`**`export`**`(`*model*, *format*, *docstring=None*`)`

Top-level function for exporting a model to a given format.

**Parameters** **model** : pysb.core.Model

The model to export.

**format** : string

A string indicating the desired export format.

**docstring** : string (optional)

The header comment to include at the top of the exported file.

`pysb.export.`**`pad`**`(`*text*, *depth=0*`)`

Dedent multi-line string and pad with spaces.

CHAPTER 4

# Useful References

A collection of links for learning more about Python and other tools used by PySB.

## 4.1 Python Language

For those unfamiliar with Python or programming there are several resources available online. We have found the ones below useful to learn Python in a practical and straightfoward manner.

**Quick Python Overview:**

- Instant Python

**Python tutorials for beginners, experienced users, or if you want a refresher:**

- Official Python tutorial
- Python for non-programmers
- Dive into Python
- Thinking in Python

## 4.2 NumPy and SciPy

**NumPy:**

- NumPy for Matlab
- Also the Mathesaurus
- Matlab commands in Numerical Python cheatsheet

**SciPy:**

- Scientific Python

## 4.3 BioNetGen

- BioNetGen tutorial
- Compartmental BNGL

CHAPTER 5

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

# Index