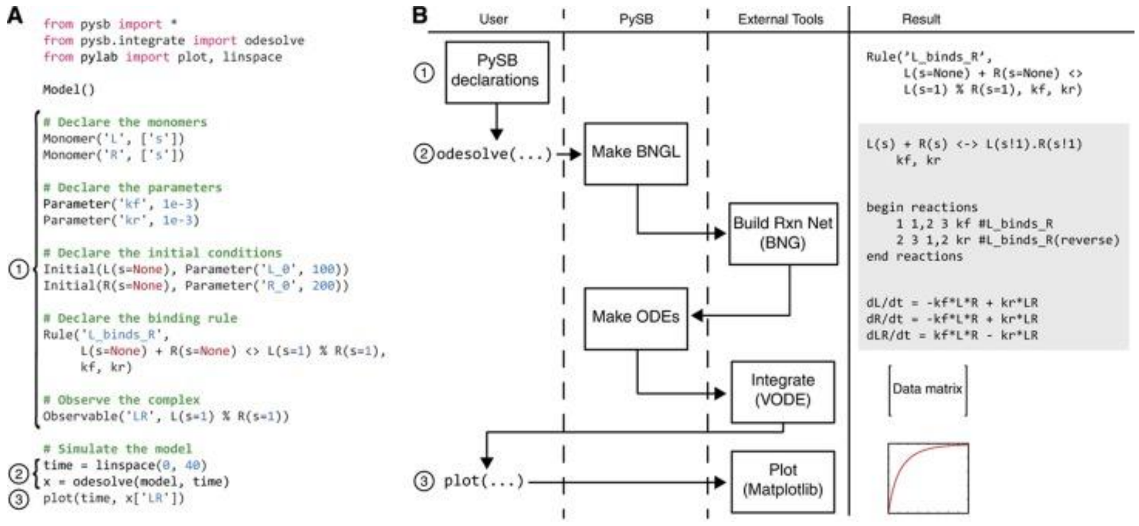
Rule-based modeling in PySB

Chemical kinetics on a computer

Martina Prugger, PhD, Carlos F. Lopez, PhD
Vanderbilt University

PySB: rule-based modeling in Python

- interface accessed from Python; communicates with various external tools (BNG network generation from the rules, ODE solvers, plotting libraries)
- fuses rule-based modeling with coding algorithms in Python



```
from pysb import *
from pysb.integrate import odesolve
from pylab import plot, linspace

Model()

# Declare the monomers
Monomer('L', ['s'])
Monomer('R', ['s'])

# Declare the parameters
Parameter('kf', 1e-3)
Parameter('kr', 1e-3)

# Declare the initial conditions
Initial(L(s=None), Parameter('L_0', 100))
Initial(R(s=None), Parameter('R_0', 200))

# Declare the binding rule
Rule('L_binds_R',
     L(s=None) + R(s=None) <> L(s=1) % R(s=1),
     kf, kr)

# Observe the complex
Observable('LR', L(s=1) % R(s=1))

# Simulate the model
time = linspace(0, 40)
x = odesolve(model, time)
plot(time, x['LR'])
```

```
Rule('L_binds_R',
     L(s=None) + R(s=None) <>
     L(s=1) % R(s=1), kf, kr)
```

```
L(s) + R(s) <-> L(s!1).R(s!1)
    kf, kr


begin reactions
    1 1,2 3 kf #L_binds_R
    2 3 1,2 kr #L_binds_R(reverse)
end reactions


dL/dt = -kf*L*R + kr*LR
dR/dt = -kf*L*R + kr*LR
dLR/dt = kf*L*R - kr*LR
```

Source: *Lopez, C. F., Muhlich, J. L., Bachman, J. A., & Sorger, P. K. (2013). Programming biological models in Python using PySB. Molecular systems biology, 9(1), 646. `doi:10.1038/msb.2013.1`*

PySB introduction: creating a model

```
from pysb import *

Model()
```

- create an instance of the model class
- model definition
    - write a model in a *.py* file (not created to be used interactively)
- model usage
    - model files are to be called by python files for analysis and simulation (this can be done interactively)
- `from pysb import *`
    - brings in all of the Python classes needed to define a model
- `Model()`
    - creates an instance of the *Model* class and implicitly assigns this object to the variable *model*

PySB introduction: model components

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

Parameter('C8_0', 1000)
Parameter('Bid_0', 10000)

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

- `Monomer`
  - indivisible elements that will make up molecules and complexes in the model (specific protein; other biomolecule)
  - consist of
    - *name* - e.g.: monomer representing the protein `'C8'` or `'Bid'`
    - list of *sites* (locations on which monomers can *bind* to the site of another monomer and/or take on a *state*) - e.g.: `['b', 'S']` for the binding site `b` and the possible states `S`
    - *dict* for specification of allowable states for the sites - e.g.: `{'S':['u', 't']}` can be untruncated `u` or truncated `t`
- `Parameter`
  - constant numerical values that represent biological constants
    - reaction rate
    - compartment volume
    - initial (boundary) condition for a molecular species
  - consists of *name* & *numerical value* (default = 0)
- `Initials`
  - initial state of the system: species that are present at time *t = 0*

PySB introduction: model rules

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

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)
```

- `Rule`
    - define the chemical reactions between molecules and complexes
    - consists of
        - *name* (any string, enclosed in quotation marks) - e.g.: `'C8_Bid_bind'`; `'tBid_from_C8Bid'`
        - pattern describing which molecular species (*instances* of monomers in a specific state) should act as the *reactants*
          - e.g.: `C8(b=None) + Bid(b=None, S='u')`; `C8(b=1) % Bid(b=1, S='u')`
        - pattern describing how reactants should be transformed into *products*
          - e.g.: `| C8(b=1) % Bid(b=1, S='u'), kf, kr)`; `>> C8(b=None) % Bid(b=None, S='t'), kc)`
        - parameters denoting the *rate constants* (which have to be declared as parameters like the model components) - e.g.: `kf`, `kr`, `kc`
- rule interaction operators
    - `+` operator to represent complexation (left side of rule; tells the two species that are undergoing a transition)
    - `|` operator to represent backward/forward reaction (reaction is reversible; separates left and right side of rule)
    - `>>` operator to represent forward-only reaction (reaction is only one way; separates left and right side of rule)
    - `%` operator to represent a binding interaction between two species (indicates that a bond is formed between two or more species by the matching integer as identifiers - e.g.: 1)

PySB introduction: model rules, example

```
# 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)
```

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

The chemical reactions $C8 + Bid \underset{kr}{\overset{kf}{\rightleftarrows}} C8\text{-}Bid$ translate into the rule

name

unbound species identifier = None

unbound species in un-truncated state

bound species with identifying bound 1

bound species identifying bound 1 in un-truncated state

```
Rule('C8_Bid_bind', C8(b=None) + Bid(b=None, S='u') | C8(b=1) % Bid(b=1, S='u'), kf, kr)
```

complexation/ addition operator

forward/backward operator

binding operator

forward rate, backward rate for the | operator

The truncation of Bid when it is bound to C8 $C8\text{-}Bid \xrightarrow{kc} C8, tBid$ into (unbound) C8 and tBid translates into

```
Rule('tBid_from_C8Bid', C8(b=1) % Bid(b=1, S='u') >> C8(b=None) % Bid(b=None, S='t'), kc)
```

C8 and Bid (in the untruncated state ‘u’) are bound together (denoted by the same identifier 1)

after the catalysis, C8 is unbound in the solution

after the catalysis, Bid is unbound in the solution, but its state changed from untruncated u to truncated t

PySB introduction: model rules, macros

- due to the power of working in the programming language Python, higher-order rules can be created by simple functions
- macros are commonly used higher-order rules that have been pre-defined
- to make use of macros, the library *pysb.macros* has to be imported
- examples for pre-defined rules provided by the library are

    equilibrate(S1,S2,[kf,kr]) generate the unimolecular reversible equilibrium reaction S1 <-> S2

    encodes
    ```
    Rule('equilibrate_S1_to_S2', S1() | S2(), kf, kr)
    ```

    bind(S1,site1,S2,site2,[kf,kr]) generate the reversible binding reaction S1 + S2 | S1:S2

    encodes
    ```
    Rule('bind_S1_S2', S1(x=None) + S2(y=None) | S1(x=1) % S2(y=1), kf, kr)
    ```

    catalyze(Enzyme,e_site,Substrate,s_site,product,[kf,kr,kc]) generate the two-step catalytic reaction E+S | E:S >> E+P

    encodes
    ```
    Rule('bind_E_S_to_ES', E(b=None) + S(b=None) | E(b=1) % S(b=1), kf, kr)
    Rule('catalyze_ES_to_E_P', E(b=1) % S(b=1) >> E(b=None) + P(), kc)
    ```

- for a full list of available macros as well as their detailed description and usage, please refer to the PySB documentation for macros: https://pysb.readthedocs.io/en/stable/modules/macros.html

PySB introduction: model observables

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

- `Observable`

  monitors the declared monomer

  can be a specific species, a combination or sum of various species

  - e.g.: `C8(b=None)` (free C8), `Bid(b=None, S='u')`(unbound Bid), `Bid(b=None, S='t')`(active Bid)

  no specifier has to be declared (`Observable('C8', C8`) will observe every `C8`, no matter the state)

PySB introduction: model full model-file example

```
from pysb import *

Model()

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

Parameter('C8_0', 1000)
Parameter('Bid_0', 10000)

Initial(C8(b=None), C8_0)
Initial(Bid(b=None, S='u'), Bid_0)

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

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)

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

PySB introduction: sim, load model

```
import mymodel as m
from pysb.simulator import ScipyOdeSimulator
import pylab as pl
```

- the rules created in PySB are sent to BioNetGen (BNG) to create a reaction network
- the reaction network is translated into ordinary differential equations (ODEs), which need to be integrated using a numerical integrator
- for convenience, simulators have been included into PySB
- for plotting and further analysis, one of the various libraries provided by Python can be used
- `import mymodel as m`

  loads the model `m` from the file `mymodel` (replace with the actual name of the model file created above)
- `from pysb.simulator import ScipyOdeSimulator`

  loads the integration engine provided by PySB

  the integrators in the PySB package are versions of the integrators from SciPy, adapted to function seamlessly with PySB
- `import pylab as pl`

  loads one of the graph engines provided by the Python library pylab for plotting

PySB introduction: simulation

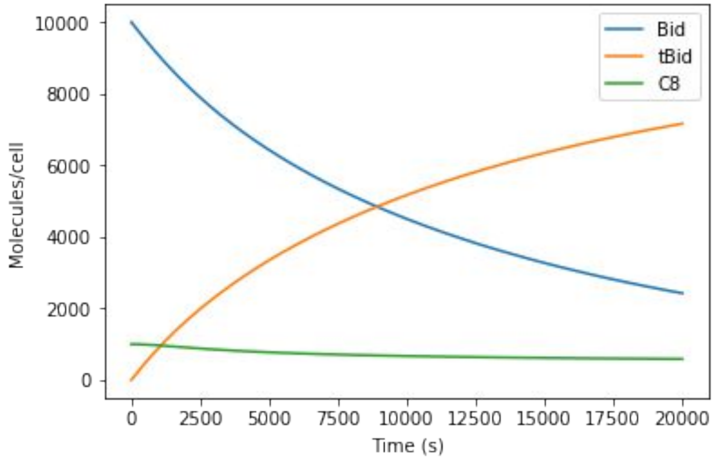
```
t = pl.linspace(0, 20000)

simres = ScipyOdeSimulator(m.model, tspan=t).run()
yout = simres.all
```

- `t = pl.linspace(0, 20000)`
  creates an array from 0 to 20000
  the entries of `t` are the integration points at which the ODEs are solved
  this can be interpreted as the time points over which the system is evaluated
- `simres = ScipyOdeSimulator(m.model, tspan=t).run()`
  calls the integrator to actually solve the system
- `yout = simres.all`
  saves the results from the integration in `yout`
  this variable can now be used for plotting and further analysis
  note that the variables that have been integrated and can now be analysed are corresponding with the observables that are defined in the model file

PySB introduction: sim, plotting

```
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()
```

- this is an example for an interactive plot (for the command line)
- the integration points `t` are used as x-axis
- on the y-axis, the three observables from the model file `C8(b=None)`, `Bid(b=None, S='u')`, `Bid(b=None, S='t')`



In the resulting figure, we can see the number of Bid molecules decreasing over time from the initial amount, the number of active Bid increasing over time and the number of free C8 molecules decreasing to about half

Try it yourself

Go to

pysbdemo.lolab.xyz

and use one of the usernames and passwords provided

Exercise

Use the provided Jupyter Notebook to

- create a model for a simple Michaelis-Menten two-step enzyme catalysis in PySB
- simulate the model
- plot the results of the model