Modeling with programs: cell signaling models using the PySB framework

Carlos F. Lopez
Vanderbilt University
School of Medicine
Outline

1. Bio modeling
   1. How we express Bio signaling
   2. How we model Bio Signaling

2. Modeling with Programs
   1. A rules-based approach (BioNetGen/Kappa)
   2. Programming biology
      1. Inheritance, encapsulation, modularization, polymorphism, reuse, transparency, etc...

3. Keeping track of bio models as programs
   1. Using programming tools for bio modeling
      1. Git
      2. GitHub
      3. Sphinx
Outline

1. Bio modeling
   1. How we express Bio signaling
   2. How we model Bio Signaling

2. Modeling with Programs
   1. A rules-based approach (BioNetGen/Kappa)
   2. Programming biology
      1. Inheritance, encapsulation, modularization, polymorphism, reuse, transparency, etc...

3. Keeping track of bio models as programs
   1. Using programming tools for bio modeling
      1. Git
      2. GitHub
      3. Sphinx
Why hasn’t Systems Biology lived up to the hype?

Biological systems are:
1. Highly stochastic
2. Highly complex
3. Highly multiscale

*Methods developed in other fields (e.g., combustion chemistry) are not directly transferable to biology without modification.

Biological Complexity

\[ R + L \xrightleftarrows_{k_a}^{k_d} RL \]
“Complex” and “complicated” aren’t the same thing

Complex

VS.

Complicated

How to represent biology

- Systems Biology

- lamellar L_\alpha d-spacing
- lipid rafts
- vesicle size
- Cell
- bonds
- viruses
- mesoscale
- fluid mechanics
- membrane fusion
- DNA replication
- protein folding
- ion channels
- lipid diffusion
- light harvesting
- bond vibrations

- time scale:
  - s
  - ms
  - \mu s
  - ns
  - ps
  - fs

- length scale:
  - nm
  - \mu m
  - mm


**Combinatorial complexity in proteins**

Contact Map

- Protein Domain Structure
- Site Modifications
- Protein-Protein contacts

\[ N_R = 2 \times 4 \times 6 = 48 \]

\[ N_{RR} = N_R(N_R - 1)/2 + N_R = 1176 \]
Levels of resolution

What we mean vs. what we write

We mean:

- BID --- BIM | PUMA
- NOXA
- BCL-2 | BCL-xL | MCL-1
- BAX | BAK
- DIMERS
- TETRAMERS
- PORES
Writing models “by hand”

\[ A + B \leftrightarrow A:B \]

\[ A:B \rightarrow A^* + B \]

\[
\begin{align*}
dA/dt &= -k_1 [A][B] + -k_{-1} [A:B] \\
 dB/dt &= -k_1 [A][B] + -k_{-1} [A:B] + k_2 [A:B]
\end{align*}
\]
Writing models “by hand”

\[ A + B \overset{k_1}{\leftrightarrow} A:B \overset{k_{-1}}{\rightarrow} A* + B \]

\[ B + C \overset{k_3}{\leftrightarrow} B:C \overset{k_{-3}}{\rightarrow} \]

\[ \frac{dA}{dt} = -k_1[A][B] + -k_{-1}[A:B] \]
\[ \frac{dB}{dt} = -k_1[A][B] + -k_{-1}[A:B] + k_2[A:B] -k_3[B][C] + k_{-3}[B:C] \]
\[ \frac{dC}{dt} = -k_3[B][C] + -k_{-3}[B:C] \]
Biology as equations…
Issues with traditional modeling approaches

- Large models require large amounts of bookkeeping, often carried out by hand.
  - Error Prone
  - Not easily Shareable
  - Not easy to edit/add/remove components.

- Models not easily transferable or modifiable.
  - Biological Complexity $\leftrightarrow$ Mathematics
  - C, FORTRAN, MATLAB, not easily shareable.

- Rules-based programs great tools to represent the biochemistry of a system.

- Use rules-approaches to develop a language to describe bio knowledge at multiple resolutions with the goal to reuse, share, and expand models routinely.
Outline

1. Bio modeling
   1. How we express Bio signaling
   2. How we model Bio Signaling

2. Modeling with Programs
   1. A rules-based approach
   2. Programming biology
      1. Inheritance, encapsulation, modularization, polymorphism, reuse, transparency, etc...

3. Keeping track of bio models as programs
   1. Using programming tools for bio modeling
      1. Git
      2. GitHub
      3. Sphinx
Rule-based Modeling:
Domain structure, site modifications, and PPI

● Motivation
  - Modular domain structure of macromolecules
  - Complexity of signaling pathways

● Basic elements
  - Protein domain structure
  - Protein-protein interactions
  - Site modification

R(l,d,Y1068~U~P,Y1148~U~P)
Reaction Rules

BNGL syntax:

\[ R(Y_{1068}P) + Grb2(SH2) \iff R(Y_{1068}P) . Grb2(SH2) \]
Embed rules-type syntax into Python

Monomers:

```
DEFINE: Monomer('Csp8', ['b', 'state'], {'state': ['I', 'A']})
```

- Binding (as many as needed)
- State (any number of states)
- Name (any name)

Species:

```
USE: Csp8(b=None, state="I")
```

- Unbound
- Inactive
- Monomer name

```
USE: Csp8(b=1, state="A")
```

- Bound
- Active
- Monomer name
Describe interactions w native Python syntax

**DISC + Csp8 ↔ DISC:Csp8**

\[
\text{Rule(‘Csp8\_Bind\_DISC’,} \\
\text{DISC}(b=\text{None, state=‘A’}) + \text{Csp8}(b=\text{None, state=‘I’}) \leftrightarrow \text{DISC}(b=1, state=‘A’) \% \text{Csp8}(b=1, state=‘I’),} \\
\text{KDISCCSP8F, KDISCCSP8R)}
\]

**DISC:Csp8 → DISC + Csp8**

\[
\text{Rule(‘Csp8\_Activation’,} \\
\text{DISC}(b=1, state=‘A’) \% \text{Csp8}(b=1, state=‘I’) >} \\
\text{DISC}(b=\text{None, state=‘I’}) + \text{Csp8}(b=\text{None, state=‘I’})} \\
\text{KDISCCSP8C)}
\]
Multiple resolutions of biology

- **Knowledge Level Representation**
  - (biocyc, ingenuity, biopax)

- **Model Specification Languages**
  - (BNG, kappa, SBML)

- **Analytic/Mathematical Languages**
  - (Matlab, Mathematica, C, FORTRAN, etc)

High-level, biological knowledge search / inference.

Mid-level, model generation using biological knowledge.

Low-level, execution of mathematical model representation.
Outline

1. Bio modeling
   1. How we express Bio signaling
   2. How we model Bio Signaling

2. Modeling with Programs
   1. The rules-based approach
   2. Modeling with programs
      1. Inheritance, encapsulation, modularization, polymorphism, reuse, transparency, etc...

3. Keeping track of bio models as programs
   1. Using programming tools for bio modeling
      1. Git
      2. GitHub
      3. Sphinx
Introduce flexibility through programming

- Working on a multi-resolution approach
  - ODEs
  - Rules-based modeling
  - Simple macros
  - High level macros
  - Multi-model inference
- Model transparency
- Model tracking
- Model sharing
- Key goal: write biological knowledge
- Avoid obfuscating lists of parameters/formalisms
Models as Python Objects

class ComplexPattern(object):
    ""
    A bound set of MonomerPatterns, i.e. a pattern to match a complex.
    In BNG terms, a list of patterns combined with the "." operator.
    ""

    def __init__(self, monomer_patterns, compartment, match_once=False):
        # ensure compartment is a Compartment
        if compartment and not isinstance(compartment, Compartment):
            raise Exception("compartment is not a Compartment object")

        self.monomer_patterns = monomer_patterns
        self.compartment = compartment
        self.match_once = match_once

    def is_concrete(self):
        """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
        ""
        # 1.
        mp_concrete_ok = all(mp.is_concrete() for mp in self.monomer_patterns)
        # 2.
        compartment_ok = self.compartment is not None and \
                        all(mp.is_site_concrete() for mp in self.monomer_patterns)
        return mp_concrete_ok or compartment_ok

    def is_equivalent_to(self, other):
        """Checks for equality with another ComplexPattern""
        # Didn't implement __eq__ to avoid confusion with __ne__ operator used for Rule building

        # FIXME the literal site_conditions comparison requires bond numbering to be identical,
        # so some sort of canonicalization of that numbering is necessary.
        if not isinstance(other, ComplexPattern):
            raise Exception("Can only compare ComplexPattern to another ComplexPattern")
        return \
            sorted((mp.monomer, mp.site_conditions) for mp in self.monomer_patterns) == \

Models as native programming objects
Defining macros w programmatic syntax

```
def catalyze(enz, sub, prod, kf, kr, kc):
    """2-step catalytic process""
    r1_name = 'bind_%s_%s' % (sub.name, enz.name)
    r2_name = 'produce_%s_via_%s' % (prod.name, enz.name)
    E = enz(b=None)
    S = sub(b=None)
    ES = enz(b=1) * sub(b=1)
    P = prod(b=None)
    Rule(r1_name, E + S <> ES, kf, kr)
    Rule(r2_name, ES >> E + P, kc)

def catalyze_convert(s1, s2, p, kf, kr, kc):
    """2-step catalytic-type process, but the "catalyst" is effectively consumed""
    r1_name = 'bind_%s_%s' % (s1.name, s2.name)
    r2_name = 'produce_%s' % p.name
    A = s1(b=None)
    B = s2(b=None)
    AB = s1(b=1) * s2(b=1)
    C = p(b=None)
    Rule(r1_name, A + B <> AB, kf, kr)
    Rule(r2_name, AB >> C, kc)

def inhibit(targ, inh, kf, kr):
    """inhibition by complexation/sequestration""
    r_name = 'inhibit_%s_by_%s' % (targ.name, inh.name)
    T = targ(b=None)
    I = inh(b=None)
    TI = targ(b=1) * inh(b=1)
    Rule(r_name, T + I <> TI, kf, kr)
```

- Express biochemistry concepts in straightforward functions
- Readable
- Functions reused as needed: avoid errors
A Model with Simple Macros

- We write what we mean
- Easily shareable
- Easy to annotate/track
More complex macros: oligomerization

Make model functional rather than prescriptive:

def pore_assembly(Subunit, size, rates):
    for poresize in range(2, size + 1):
        basepore = pore_species(Subunit, 1)
        addunit = pore_species(Subunit, i-1)
        newpore = pore_species(Subunit, i)
        rules.append(Rule('%s_pore_assembly_%d'
                           % (Subunit.monomer.name, poresize),
                           basepore + addunit <> newpore, *rates))
    return rules

Call for rule generation:

pore_assembly(Bax(bf=None, state='A'), 4, kd['BAX_PORE'])
One line $\rightarrow$ Tens of rules $\rightarrow$ Hundreds of ODEs

Rules for Bax/Bak tetramer generation ($\sim30$ rules)

Equations… ($\sim100$ equations)
Update models as new insights are discovered

- Explore at different resolutions.
- Reuse previous knowledge.
- Update/correct previous modeling efforts.
- Biology view is “modular”
Simple macros $\rightarrow$ Complex macros

```python
def simple_bind(name(opt), sub1, sub2, kf, kr):
    rule_name="complex_formation_"+sub1.name+sub2.name
    A = sub1(site=None)
    B = sub2(site=None)
    AB = sub1(site=1) % sub2(site=1)
    return Rule(rule_name, A + B <> AB, kf, kf)

def simple_bind_table(table, paramlist):
    rules = []
    reactants0 = table(first-row)
    reactants1 = table(first-column)
    interactions = table(rest-row, rest-column)
    for pair(i,j) in interactions:
        if intrxn[i,j] is True:
            kf = paramlist[i,j]
            kr = paramlist[i,j]
            rule_name = "complex_"+ i.name + j.name
            rules.append(simple_bind(rule_name, i, j, kf, kr))
    return rules
```
Simple instantiation of model updates

9 interactions

```python
simple_bind_table([[Bcl2, BclxL, Mcl1], [{}], [{}], [{}], [Bid, {'state': 'A'}, True, False, False], [Bax, {'state': 'A'}, True, True, False], [Bak, {'state': 'A'}, False, True, True]], BCL-2_params)
```

15 interactions

```python
simple_bind_table([[Bcl2, BclxL, Mcl1, Bclw, A1], [{}], [{}], [{}], [{}], [{}], [Bid, {'state': 'A'}, True, False, False, False, False, True], [Bax, {'state': 'A'}, True, True, False, True, False], [Bak, {'state': 'A'}, False, True, True, False, True]], BCL-2_params)
```

• Model expansion is straightforward
• Model query
• Transparent, reusable, shareable.
Real-world example... EGFR

```plaintext
## Receptor activation
Rule('EGFRActivate', EGFR(b1=None, S='i') ↔ EGFR(b1=None, S='a'), kact, kinact)
Rule('EGFRTATPActivate', EGFR(b1=1, S='i') % ATP(b=1) ↔ EGFR(b1=1, S='a') % ATP(b=1), kactATP, kinact)

## ATP binding to EGFR
Rule('iEGFRATPBind', EGFR(b1=None, S='i') + ATP(b=None) ↔ EGFR(b1=1, S='i') % ATP(b=1), koniATP, koffATP)
Rule('aEGFRATPBind', EGFR(b1=None, S='a') + ATP(b=None) ↔ EGFR(b1=1, S='a') % ATP(b=1), konaATP, koffATP)

## Gefitinib binding
Rule('iEGFRGefBind', EGFR(b1=None, S='i') + Gef(b=None) ↔ EGFR(b1=1, S='i') % Gef(b=1), koniGef, koffGef)
Rule('aEGFRGefBind', EGFR(b1=None, S='a') + Gef(b=None) ↔ EGFR(b1=1, S='a') % Gef(b=1), konGef, koffGef)
Rule('EGFRGefActivate', EGFR(b1=1, S='i') % Gef(b=1) ↔ EGFR(b1=1, S='a') % Gef(b=1), kactGef, kinact)

## Lapatinib binding
Rule('EGFRLapBind', EGFR(b1=None, S='i') + Lap(b=None) ↔ EGFR(b1=1, S='i') % Lap(b=1), konLap, koffLap)

## Erlotinib binding
Rule('iEGFRErlBind', EGFR(b1=None, S='i') + Erl(b=None) ↔ EGFR(b1=1, S='i') % Erl(b=1), koniErl, koffErl)
Rule('aEGFRErlBind', EGFR(b1=None, S='a') + Erl(b=None) ↔ EGFR(b1=1, S='a') % Erl(b=1), konErl, koffErl)
Rule('EGFRErlActivate', EGFR(b1=1, S='i') % Erl(b=1) ↔ EGFR(b1=1, S='a') % Erl(b=1), kactErl, kinact)

## Canertinib binding
Rule('iEGFRCanBind', EGFR(b1=None, S='i') + Can(b=None, c='u') ↔ EGFR(b1=1,S='i') % Can(b=1, c='u'), koniCan, koffCan)
Rule('aEGFRCanBind', EGFR(b1=None, S='a') + Can(b=None, c='u') ↔ EGFR(b1=1,S='a') % Can(b=1, c='u'), konCan, koffCan)
Rule('EGFRCanActivate', EGFR(b1=1, S='i') % Can(b=1) ↔ EGFR(b1=1, S='a') % Can(b=1), kactCan, kinact)
Rule('EGFRCovalent', EGFR(b1=1) % Can(b=1, c='u') ↔ EGFR(b1=1) % Can(b=1, c='b'), kcatCan)

## Neratinib binding
Rule('iEGFRNerBind', EGFR(b1=None, S='i') + Ner(b=None, c='u') ↔ EGFR(b1=1,S='i') % Ner(b=1, c='u'), koniNer, koffNer)
Rule('aEGFRNerBind', EGFR(b1=None, S='a') + Ner(b=None, c='u') ↔ EGFR(b1=1,S='a') % Ner(b=1, c='u'), konNer, koffNer)
Rule('EGFRNerActivate', EGFR(b1=1, S='i') % Ner(b=1) ↔ EGFR(b1=1, S='a') % Ner(b=1), kactNer, kinact)
Rule('EGFRNerCovalent', EGFR(b1=1) % Ner(b=1, c='u') ↔ EGFR(b1=1) % Ner(b=1, c='b'), kcatNer)

## P elitinib binding
Rule('iEGFRPelBind', EGFR(b1=None, S='i') + Pel(b=None, c='u') ↔ EGFR(b1=1,S='i') % Pel(b=1, c='u'), koniPel, koffPel)
Rule('aEGFRPelBind', EGFR(b1=None, S='a') + Pel(b=None, c='u') ↔ EGFR(b1=1,S='a') % Pel(b=1, c='u'), konPel, koffPel)
Rule('EGFRPelActivate', EGFR(b1=1, S='i') % Pel(b=1) ↔ EGFR(b1=1, S='a') % Pel(b=1), kactPel, kinact)
Rule('EGFRPelCovalent', EGFR(b1=1) % Pel(b=1, c='u') ↔ EGFR(b1=1) % Pel(b=1, c='b'), kcatPel)
```
Simplifying a model w macros

```r
## ATP and inhibitor binding
bind_table([[
    [ATP, (9.7338e00, 1.0876e00), (4.7879e05, 1.0876e00)],
    [Gef, (1.2490e06, 2.0609e-01), (2.0609e08, 2.0609e-01)],
    [Lap, (2.5466e03, 7.6397e-06), None],
    [Er1, (1.2490e06, 2.0609e-01), (2.0609e08, 2.0609e-01)],
    [Can(c='u'), (1.2490e06, 2.0609e-01), (2.0609e08, 2.0609e-01)],
    [Ner(c='u'), (1.2490e06, 2.0609e-01), (2.0609e08, 2.0609e-01)],
    [Pel(c='u'), (1.2490e06, 2.0609e-01), (2.0609e08, 2.0609e-01)]
]])
```
Further abstraction
Multi-model inference

Explore model biochemistry.
Update/correct previous models.
Generate model topologies for numerical testing.

Use model calibration, muti-model inference, etc. (not today)
Actual code: indirect hypothesis

```python
from earm_indirect_mem_parms import parameter_dict as kd
import earm_1_0modules

translocate('Bax_to_mem', Bax, state = 'T', state = 'M'), kbaxtf, kbaxtr)
translocate('Bclxl_to_mem', Bclxl, state = 'T', state = 'M'), kbcclxl, kbcclxltr)
pore_assembly(Bax(bf=None, state='A'), 4, kd['BAX_PORE'])
pore_assembly(Bak(bf=None, state='A'), 4, kd['BAK_PORE'])

inhib_table([[[],
              {'state': 'M'}, {},
              [Bid, {'state': 'T'}, True, True],
              [Bax, {'state': 'A'}, True, False],
              [Bak, {'state': 'A'}, True, True],
              kd['BID_BAX_BAK_inh'], model])

sensi_table([[[],
               {'state': 'M'}, {}],
               [Bad, {}, True, False],
               [NOXA, {}, True, True],
               kd['BCLs_sens'], model])

earm_1_0modules.rec_to_bid(model, kd)
earm_1_0modules.pore_to_parp(model, kd)
```

6 lines of function calls
Reusable modules / parameters called from main program
(actual python code)
from earm_direct_mem_Parms import parameter_dict as kd
import earm_1_0modules

translocate('Bid_to_mem', Bid, state = 'T', state = 'M'), kbidtf, kbidtr)
translocate('Bax_to_mem', Bax, state = 'T', state = 'M'), kbaxtf, kbaxtr)
translocate('Bclxl_to_mem', Bclxl, state = 'T', state = 'M'), kbclxltr, kbclxltr)

activate(Bid(state = 'T'), Bax(state = 'M'), Bax(bf = None, state = 'A'), kd[ 'BIDt_BAX' ])
activate(Bid(state = 'M'), Bax(state = 'M'), Bax(bf = None, state = 'A'), kd[ 'BIDm_BAX' ])
activate(Bid(state = 'T'), Bak(state = 'M'), Bak(bf = None, state = 'A'), kd[ 'BIDt_BAK' ])
pore.Assembly(Bax(bf=None, state='A'), 4, kd[ 'BAX_PORE' ])
pore.Assembly(Bak(bf=None, state='A'), 4, kd[ 'BAK_PORE' ])

inhib_table([[Bcl2, Bclxl, Mcl1], [Bid, {'state': 'M'}, {}, {}], [Bid, {'state': 'M'}, True, True, False], kd[ 'BID_BAX_BAK_inh' ], model])
sensi_table([[Bcl2, Bclxl, Mcl1], [Bad, {}, True, True, False], [NOXA, {}, False, True, True], kd[ 'BCLs_sens' ], model])
earm_1_0modules.rec_to_bid(model, kd)
earm_1_0modules.pore_to_par_parp(model, kd)

10 lines of function calls
Reusable modules / parameters called from main program
(actual python code)
12 lines of function calls
Reusable modules / parameters called from main program
(actual python code)
Multi-model topology exploration

[Graphs showing different models such as IC-RP, tBid, IMS-RP, Smac, EC-RP, and cParp with various curves plotted against time or iterations]
Hypothesis testing

Indirect

Direct parameters ~2-3 orders of magnitude from EARM1.0 values to fit data...

Embedded???? Direct?

Outline

1. Bio modeling
   1. How we express Bio signaling
   2. How we model Bio Signaling

2. Modeling with Programs
   1. A rules-based approach (BioNetGen/Kappa)
   2. Programming biology
      1. Inheritance, encapsulation, modularization, polymorphism, reuse, transparency, etc...

3. Keeping track of bio models as programs
   1. Using programming tools for bio modeling
      1. Git
      2. GitHub
      3. Sphinx
PySB brings tools together

PySB Model

```python
# Instantiate the model
Model()

Monomer('tBid', ['b'])
Monomer('Bax', ['b', 'state'], {'state': ['I', 'A']})

two_step_mod(tBid(b=None), Bax(b=None, state='I'),
    Bax(b=None, state='A'), site='b')

Rule('Bax_inactivation',
    Bax(b=None, state='A') >> Bax(b=None, state='I'),
    Parameter('k_rev', 1e-1))

# Set the initial conditions
Initial(tBid(b=None), Parameter('tBid_0', 1000))
Initial(Bax(b=None, state='I'), Parameter('Bax_0', 1000))

# Model output: activated Bax
Observe('aBax', Bax(state='A'))
```

# Run simulation and calibrate output
output = odesolve(model, 20000)
annealout = annlodesolve(model...)

# Solve for st. state aBax using SymPy
cons_eqns = parse_expr('s0 + s1 ...')
solve(model.odes + cons_eqns, s5)

# Run a deterministic ODE simulation
det_time = linspace(0, tmax, numpoints)
det_data = odesolve(model, det_time)

# Run a stochastic simulation using kappa
stoch_data = kappa.get_kasim_data(model,
    time=tmax, points=numpoints)

Sundials/CVODE Integrators

BNG,
SymPy

Kappa
Stochastic Simulator
Program tools for free: Git
Program tools: GitHub

Software version control to share, revise, and maintain MODELS efficiently.

Model tracking: EARM V2

Code tracking: PySB
Welcome to EARM’s documentation!

Contents:

- Introduction
  - What is EARM?
  - Goals
  - Installation
  - How to use the documentation
  - Staying current
- Overview of the models
  - MOMP module “boundaries”
  - The models in EARM
  - How to use the models
  - Parameter values
  - The code is meant to be read!
- EARM reference
  - Lopez modules
  - Albeck modules
  - Shen models
  - Shared macros

Indices and tables

- Index
- Module Index
- Search Page
Hands-on demonstration.

**Biological model:**
mechanistic hypotheses
  e.g. extrinsic apoptosis

**Mathematical representation:**
chemical kinetics
  e.g. ODE, Gillespie
Biological model: mechanistic hypotheses  
e.g. extrinsic apoptosis

Rules: protein-protein interactions  
e.g. A + B <-> A:B

Mathematical representation: chemical kinetics  
e.g. ODE, Gillespie
Hands-on demonstration.

**Biological model:**
mechanistic hypotheses
  e.g. extrinsic apoptosis

**Macros:**
common biochemical processes
  e.g. catalyze, oligomerize, translocate

**Rules:**
protein-protein interactions
  e.g. A + B <-> A:B

**Mathematical representation:**
chemical kinetics
  e.g. ODE, Gillespie
Hands-on demonstration.

**Biological model:**
mechanistic hypotheses
  e.g. extrinsic apoptosis

**Motif:**
phrase or sentence in a word model
  e.g. tBid activates Bax and Bak

**Macros:**
common biochemical processes
  e.g. catalyze, oligomerize, translocate

**Rules:**
protein-protein interactions
  e.g. A + B <> A:B

**Mathematical representation:**
chemical kinetics
  e.g. ODE, Gillespie
Hands-on demonstration.

**Biological model:**
mechanistic hypotheses
e.g. extrinsic apoptosis

**Module:**
self-contained pathway in a model
e.g. direct, indirect, embedded

**Motif:**
phrase or sentence in a word model
e.g. tBid activates Bax and Bak

**Macros:**
common biochemical processes
e.g. catalyze, oligomerize, translocate

**Rules:**
protein-protein interactions
e.g. $A + B \leftrightarrow A:B$

**Mathematical representation:**
chemical kinetics
e.g. ODE, Gillespie
Summary

1. Bio modeling
2. Modeling with Programs
3. Keeping track of multiple resolutions in models