# Modeling Cheatsheet[1]

What is a model?

A mathematical model is a description of a system using mathematical concepts and language. The process of developing a mathematical model is termed mathematical modeling[2]. Models are used frequently to describe physical systems and are used to make predictions about the future state of the system. A model is not however a replica of a system but an approximation that allows the essential features of the system to be studied without concern for unnecesaary and distracting details.

Chemical Kinetics

Chemical kinetics describes the factors that influence the rate of chemical reactions. Such factors include concentration, temperature, light, catalysts or the presence of ions or other components present in the reaction mixture.

What is a mass-action rate law?

The law of mass-action states that the rate of an elementary chemical reaction is proportional to the product of the concentrations of the reacting chemical species. For the reaction:

$$aA + bB + \ldots \rightarrow pP + qQ + \ldots$$
$$v = k_1 A^a B^b \ldots - k_2 P^p Q^q \ldots$$
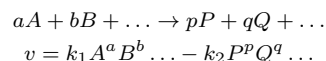
SBML

The Systems Biology Markup Language (SBML) is a representation format, based on XML, for communicating and storing computational models of biological processes. It is a free and open standard with widespread software support. SBML can represent many different classes of biological phenomena, including metabolic networks, cell signaling pathways, regulatory networks, infectious diseases, and many others. As an XML format, SBML is not meant to be read or written by Humans.

Antimony

The Antimony language provides a way for researchers to use simple text statements to create, import, and combine biological models, allowing complex models to be built from simpler models. It is fully compatible with SBML such that SBML and Antimony can be converted from one to the other.

Examples of Antimony Models

1.

```
# Simple Decay process
S1 -> S2; k1*S1;

k1 = 0.1; S1 = 10; S2 = 0
```

2.

___

[1] Version 1.0
[2] Wikipedia

```
# Consecutive reactions
S1 -> S2; k1*S1;
S2 -> S3; k2*S2;
k1 = 0.1; k2 = 0.2;
S1 = 10; S2 = 0; S3 = 0;
```

3.

```
# Bimolecular reactions
S1 + S2 -> S3;  k1*S1*S2;
S3 -> S1 + S2;  k2*S3;
# Branched System
S1 -> S2; k1*S1;
S2 -> S3; k2*S2;
S2 -> S4; k3*S2;
```

4.

```
# Open system with fixed boundaries
# '$' indicates a fixed species
$S1 -> S2;  k1*S1;
 S2 -> S3;  k2*S2;
 S3 -> $S4; k3*S3;
```

5.

```
# Open system with empty boundaries
   -> S2;  k1*S1;
 S2 -> S3; k2*S2;
 S3 -> ;   k3*S3;
```

6.

```
# Simple Feedback System
   -> S2;  k1*S1/(k2 + S1 + S3/Ki);
 S2 -> S3; k2*S2 - k3*S3;
 S3 -> ;   k4*S3;
```

7.

```
# Named reactions and events
J1: $S1 -> S2;  k1*S1;
J2:  S2 -> $S3; k2*S2;
k1 = 0.1; k2 = 0.3;
S1 = 10;

at (time > 10): k2 = 0.6
```

8.

```
# Modeling Gene expression

# Modeling an activator P0 that
# results in expression of P1
G1:   -> P1;  Vm1*P0^n/(K1 + P0^n);

# Modeling a repressor P0
J2:  S2 -> S3;  Vm1/(K1 + P0^n);
```

Loading SBML Models

There are two identical ways to load SBML models into libRoadRunner:

```
r = roadrunner.RoadRunner ('mymodel.xml')
r = roadrunner.RoadRunner (sbmlString)

r = roadrunner.loadSBMLModel ('mymodel.xml')
r = roadrunner.loadSBMLModel (sbmlString)
```

The methods return a reference to a copy of libRoadRunner.

Loading Antimony Models

There are two identical ways to load Antimony models into libRoadRunner:

```
# From a file
r = te.loadAntimonyModel ('mymodel.ant')

# From a string
antStr = '''
    S1 -> S2; k1*S1;
    k1 = 0.1;
    S1 = 10; S2 = 0;
'''
r = te.loadAntimonyModel (antStr)

# Short-cut to loadAntimonyModel
r = te.loada ('mymodel.xml')
r = te.loada (antStr)
```

`loada` is simply a short-cut for `loadAntimonyModel`.

Simulation

libRoadRunner supports two kinds of simulation, differential equation based and stochastic based simulations. After a model has been loaded into libRoadRunner, the user has the option to either carry out a deterministic simulation based on solving differential equations, or a stochastic simulation based on the Gillespie Algorithm.

Differential Equation Based Simulations

To simulate model based on differential equations use the `simulate` command:

```
result = r.simulate (0, 10, 100)
```

In its basic form, simulate takes three arguments, time start, time end and the number of points to generate. If the arguments are omitted default values are chosen. The simulate command returns an array that contains the results of the simulation. The first column holds data that represents the time axis, all subsequent columns represent the floating species in the model. The particular columns in the result array can be changed with a selection list, for example:

```
result = r.simulate (0, 10, 100, ['Time', 'S1', 'k1'])
```

means that the first column will hold time, the second column S1 and the third column k1.

Stochastic Based Simulations

To simulate a model based on Gillespie stochastic method use the `gillespie` command:

```
result = r.gillespie (0, 10)
```

In its basic form, gillespie takes two arguments, time start, and time end. If the arguments are omitted default values are chosen. The gillespie command returns an array that contains the results of the simulation. The first column hold data that represents the time axis, all subsequent columns represent the floating species in the model. The particular columns in the result array can be changed with a selection list, for example:

```
result = r.gillespie (0, 10, ['Time', 'S1', 'S4'])
```

means that the first column will hold time, the second column S1 and the third column S4. The Gillespie method by its nature returns values in the time column at irregular intervals. We provide a variant where the output is returned on a regular time grid. This method can be used by supplying a third argument after time end that represents the number of points on the grid:

```
result = r.gillespie (0, 10, 100)
result = r.gillespie (0, 10, 100, ['Time', 'S1', 'S4'])
```

Changing Values

In a simulation it is often necessary to make changes to values in a model and rerun the simulation. Assume the following model was loaded into libRoadRunner:

```
r = te.loada ('''
    S1 -> S2; k1*S1;
    S2 -> S3; k2*S2;
    S3 -> S4; k3*S3;
    k1 = 0.1; k2 = 0.2; k3 = 0.3;
    S1 = 10
''')
result1 = r.simulate (0, 10, 50)
# Modify the k1 rate constant
r.k1 = 12.0
result2 = r.simulate (0, 10, 50)

# Changing species levels
r.S1 = 24.5
```

Plotting

A common need is to be able to plot simulation results. To do this we use the `plot` command

```
r = te.loada ('''
    S1 -> S2; k1*S1;
    S2 -> S3; k2*S2;
    S3 -> S4; k3*S3;
    k1 = 0.1; k2 = 0.2; k3 = 0.3;
    S1 = 10
''')
result = r.simulate (0, 50, 50)
r.plot (result)
```

Resetting the Model

A common operation when doing interactive simulation is resetting a model back to some initial state.

```
# Reset a model back to the state it was
# when it was first loaded or created

r.resetToOrigin()


# Reset the current species values back
# to their initial conditions

r.reset()


# Reset the current species values back
# to their initial conditions and reset
# all parameters back to when the model was
# first loaded or created

r.resetAll()
```

Exporting a Model

Models can be exported in threes different formats: SBML, Antimony or Matlab

```
# Export the model as SBML
print r.getCurrentSBML()

# Export the model as Matlab
print r.getMatlab()

# Export the mode as Antimony
print r.getAntimony()
```

Computing the Steady State

To compute the steady state for a model use:

```
r.steadyState()
print r.getFloatingSpeciesConcentrations()
# Short-cut:
print r.sv()  # Species vector
```

The steadyState function returns a value indicating how close the the solution is to the steady state. The smaller the value the better. Values less than 10^{-4} usually indicate that the steady state was found.

Useful Matrices

There are a variety of matrices that can be obtained from the model, only two will be described here.
The stoichiometry matrix:

```
print r.getFullStoichiometryMatrix()
# Short-cut:
print r.sm()
```

The Jacobian matrix:

```
print r.getFullJacobian()
# Short-cut:
print r.fjac()
```

Useful Vectors

The reaction rate vector:

```
print r.getReactionRates()
# Short-cut:
print r.rv()
```

The rates of change vector:

```
print r.getRatesOfChange()
# Short-cut:
print r.dv()
```

The species concentration vector:

```
print r.getFloatingSpeciesConcentrations()
# Short-cut:
print r.sv()
```

The names of all floating species

```
print r.getFloatingSpeciesIds()
# Short-cut:
print r.fs()
```

The names of all kinetic parameters in the model

```
print r.getGlobalParameterIds
# Short-cut:
print r.ps()
```

Resources

http://libroadrunner.org/
http://antimony.sourceforge.net/
http://tellurium.analogmachine.org/