A Pycellerator Tutorial

Bruce E. Shapiro[1], Eric Mjolsness[2]

[1]Dept. of Mathematics, California State University, Northridge; [2]Depts. of Computer Science and Mathematics, and Institute of Genomics and Bioinformatics, University of California, Irvine.

Summary

We present a tutorial on using Pycellerator for biomolecular simulations. Models are described in human readable (and editable) text files (UTF8 or ASCII) containing collections of reactions, assignments, initial conditions, function definitions, and rate constants. These models are then converted into a Python program that can optionally solve the system, e.g., as a system of differential equations using ODEINT, or be run by another program. The input language implements an extended version of the Cellerator arrow notation, including mass action, Hill functions, S-Systems, MWC, and reactions with user-defined kinetic laws. Simple flux balance analysis is also implemented. We will demonstrate the implementation and analysis of progressively more complex models, starting from simple mass action through indexed cascades. Pycellerator can be used as a library that is integrated into other programs, run as a command line program, or in iPython notebooks. It is implemented in Python 2.7 and available under an open source GPL license.

Keywords: Cellerator, SBML, Systems Biology, Python

1 Introduction

Using Pycellerator typically involves the following steps steps: (a) model preparation; (b) model instantiation; (c) model execution (simulation); and (d) simulation analysis. Model preparation requires creation of a model in the Pycellerator model description language (henceforth called model files). This are human-readable files that are typically hand-written in a text-editor. A Pycellerator model consists of some combination of reactions, assignment rules and functions, along with specifications of initial conditions, parameter values, and constant variables. [1] In addition, the modeling language incorporates an extended text-based version of the full Cellerator modeling language. [2] (However, Pycellerator is a completely separate program from Cellerator and, unlike the latter, does not depend on or use any features of Mathematica.) Where features are compatible (e.g., events and level 3 packages are not supported), models can also be read from SBML files [3] and Cellerator Mathematica models.

Model instantiation means conversion of a Pycellerator model into a Python program. This auto-generated Python program includes two parts: a main driver program that invokes the `scipy.odeint` numerical solver [4], and a function that instantiates the model as a systems of differential equations. This function is in the format that is typically expected by `odeint`. The program is saved to the file system, and modelers may choose to modify the program and/or use it independently of the remainder of Pycellerator. This program does not depend on any Pycellerator libraries.

Model execution and analysis involves performing simulations with the instantiated model and interpreting the results. Pycellerator provides functionality to execute an instantiated model and to plot time courses of the results. In addition, the results of the numerical integration may be saved to `numpy` arrays [5], and any of the analysis functions available in Python are subsequently available for use.

The main features of Pycellerator that differentiate it from other modeling language based simulators are (a) conversion to accessible, user-modifiable, executable Python model descriptions; and (b) the ability to incorporate standard Python expressions (such as ternary conditionals) into assignment rules.

With Pycellerator a modeler may perform any of the the following tasks automatically:

1. Generate a Python code implementation of a model.
2. Generate a stand-alone program that can be used to run the code produced in step 1.
3. Run a deterministic simulation of the model using the code generated in the step 2.
4. Generate Python code wrapper to perform a parametric variation (e.g., of a rate constant or initial condition over an interval) of the code implemented in step 2.
5. Run the parametric evaluation written in step 4.
6. Plot the results (e.g., state variable time courses or parametric variation) from steps 2 or 4.
7. Solve simple flux models for unknown fluxes.
8. Export results from steps 2, 4, or 7 into numpy arrays for further analysis within Python.

These functions may be performed either in iPython notebook or from the command shell. Auto-generated code can be modified by users and incorporated into user programs without restriction.

2 Model Files

Models are composed of primarily of lists of chemical reactions and their associated rate constants and initial conditions. An example of a simple model for an enzymatic reaction is given in listing 1. Models may also include equations that specify species values, mathematical functions, and to some extent, simple Python expressions. The reactions are specified using standard arrow-like keyboard symbols, and equations resemble standard Python expressions. In general, a model file is divided into six sections: reactions, parameter values, initial conditions, function definitions, assignment rules, and a list of constant species. Each of these sections begins with a special keyword: `$REACTIONS`, `$RATES`, `$IC`, `$FUNCTIONS`, `$ASSIGNMENTS`, `$FROZEN`. The keywords are not case sensitive. In its simplest form, a model would consist of one or more reactions, initial conditions, and rate constants.

2.1 Arrows

The canonical arrow form is

`[[`*`reactants`*`] `*`arrow`*` [`*`products`*`], mod[`*`modifier`*`], `*`keyword`*`[`*`parameters`*`]]` (1)

where *`reactants`* and *`products`* are comma-delimited sequences of one or more species names; *`arrow`* is a text arrow (see table 1, column 1); *`keyword`* is a keyword that indicates how the arrow is to use the list of *`parameters`* (table 1, column 2); and *`modifier`* is a one or more species names that are optionally allowed with some arrow/keyword combinations. When there is only one reactant (or only one product), the square brackets around the corresponding sequence (of *`reactants`* or *`products`*) is omitted. In certain cases (e.g., mass action reactions), the plus-symbol ("+") is used in place of commas to delimit *`reactants`* or *`products`*, and the brackets are also omitted in these situations. The entire reaction must be enclosed in square brackets. Each arrow contributes terms to the system of differential equations that describe the model. The following section describe how each type of arrow is understood by Pycellerator.

2.1.1 Mass Action

The most basic reaction arrows in Pycellerator use mass action kinetics (see table 2). A numerical stoichiometry may be specified and there is no limit to either the number of reactants or products in a reaction. The standard syntax is

`[e1*X1+e2*X2+...en*Xn -> f1*Y1 + f2*Y2 + ...  + fm*Ym, k]` (2)

This means that reactants `X1, X2, ..., Xn` are combined with stoichiometries `e1, e2, ... en` to produce products `Y1, Y2, ..., Ym` with stoichiometries `f1, f2, ... fm`. The asterisks are optional but the numerical stoichiometries must precede the symbols. For each reaction in the model, a differential equation term is generated for each species (by species we mean reactant or product) in that reaction. Let Z be be some species that appears with (possibly zero) stoichiometries e_j and f_j on the left hand side and right hand side of the arrow in reaction j. Then if species $X_1, \ldots, X_n$ appear on the left hand side of reaction j with stoichiometries $e_{j1}, \ldots, e_{jn}$,

$$\frac{dZ}{dt} = \sum_{j \in \text{Reactions}} k_j (f_j - e_j) \prod_{a=1}^{n} X_a^{e_{ja}} \tag{3}$$

where k_j is the rate constant of reaction j. [6]

Although stoichiomety is normally integer, there is nothing preventing a modeller from using any non-integer floating point value. For example, consider the following system of biochemical reactions from the Field-Noyes (Oregonator) model, in which the original model authors use a stoichimetry of 1/2 for for the last reaction. [7]

$$\left.\begin{array}{l}
\mathrm{Br} + \mathrm{BrO_3} \xrightarrow{k_1} \mathrm{HBrO_2} + \mathrm{HOBr} \\
\mathrm{Br} + \mathrm{HBrO_2} \xrightarrow{k_2} 2\mathrm{HOBr} \\
\mathrm{BrO_3} + \mathrm{HBrO_2} \xrightarrow{k_3} 2\mathrm{Ce} + 2\mathrm{HBrO_2} \\
2\mathrm{HBrO_2} \xrightarrow{k_4} \mathrm{BrO_3} + \mathrm{HOBr} \\
\mathrm{Ce} \xrightarrow{k5} 0.5\mathrm{Br}
\end{array}\right\} \tag{4}$$

The `Reactions` section of the corresponding model file might look like this:

$$\left.\begin{array}{l}
\texttt{[Br + BrO3 -> HBrO2 + HOBr, k1]} \\
\texttt{[Br + HBrO2 -> 2*HOBr, k2]} \\
\texttt{[BrO3 + HBrO2 -> 2*Ce + 2*HBrO2, k3]} \\
\texttt{[2*HBrO2-> BrO3 + HOBr, k4]} \\
\texttt{[Ce -> 0.5*Br, k5]}
\end{array}\right\} \tag{5}$$

Each species in this system will automatically be converted to differential equations according to equation (3). If we also include `BrO3` in the `Frozen` section of the model file (to keep its value constant), the resulting mass action equations (in Python form) are

$$\left.\begin{array}{l}
\texttt{Ce' = 2*BrO3*HBrO2*k3 - Ce*k5} \\
\texttt{HOBr' = Br*BrO3*k1 + 2*Br*HBrO2*k2 + HBrO2**2*k4} \\
\texttt{HBrO2' = Br*BrO3*k1 - Br*HBrO2*k2 + BrO3*HBrO2*k3} \\
\qquad\texttt{- 2*HBrO2**2*k4} \\
\texttt{BrO3' = 0} \\
\texttt{Br' = -Br*BrO3*k1 - Br*HBrO2*k2 + 0.5*Ce*k5}
\end{array}\right\} \tag{6}$$

While it is unlikely for n nor m to be larger than two, nothing precludes modelers from incorporating higher order reactions in Pycellerator models.

2.1.2 Enzymatic Expansion

We define a number of catalyzed mass action reactions that are expanded into standard enzymatic reactions, e.g., simple conversion via creation of an intermediate complex. Each of the these enzymatic reactions is indicated by a single line of code in the model. Consider the following biochemical reaction:

$$\mathrm{S} + \mathrm{E} \underset{k_2}{\overset{k_1}{\rightleftarrows}} \mathrm{SE} \underset{k_4}{\overset{k_3}{\rightleftarrows}} \mathrm{P} + \mathrm{E} \tag{7}$$

This is represented by a single arrow

$$\texttt{[S=>P, mod[E], rates[k1,k2,k3,k4]]} \tag{8}$$

Pycellerator allows users to omit rate constants from the end of the list, defaulting their values to zero. Thus

$$\texttt{[S=>P, mod[E], rates[k1,k2,k3]]} \tag{9}$$

represents the reaction

$$\mathrm{S} + \mathrm{E} \underset{k_2}{\overset{k_1}{\rightleftarrows}} \mathrm{SE} \overset{k_3}{\rightarrow} \mathrm{P} + \mathrm{E} \tag{10}$$

Pycellerator automatically reinterprets arrow (8) as the following system of arrows. The name of the intermediate complex is automatically generated from the names of the substrate and the catalyst.

$$\left.\begin{array}{l} \texttt{[S+E->S\_E,k1]} \\ \texttt{[S\_E->S+E,k2]} \\ \texttt{[S\_E->P+E,k3]} \\ \texttt{[P+E->S\_E,k4]} \end{array}\right\} \tag{11}$$

These arrows (as well as the other forms in table 2) are converted into differential equations as per equation (3). For example, `[S<=>P,mod[F,R], rates[k1,..,k8]]` represents the pair of reactions

$$\left.\begin{array}{l} \mathrm{S} + \mathrm{E} \underset{k_2}{\overset{k_1}{\rightleftarrows}} \mathrm{SE} \underset{k_4}{\overset{k_3}{\rightleftarrows}} \mathrm{P} + \mathrm{E} \\ \mathrm{P} + \mathrm{R} \underset{k_6}{\overset{k_5}{\rightleftarrows}} \mathrm{PR} \underset{k_8}{\overset{k_7}{\rightleftarrows}} \mathrm{S} + \mathrm{R} \end{array}\right\} \tag{12}$$

Internally, this would be expanded first into two arrows of the form reaction (8) and then eight arrows of the form reaction (11). Four of these arrows would correspond to the first reaction in reactions (12) and four in the second reaction in reactions (12).

2.1.3 Michaelis-Menten-Henri-Briggs-Haldane Approximation

Henri, Michaelis and Menten, and Briggs and Haldane all obtained the following formula for reaction 10, but with different assumptions.

$$\frac{dP}{dt} = \frac{vS}{K+S} = -\frac{dS}{dt} \tag{13}$$

Since they made different assumtions, the actual chemistry should be interpreted differently in each case. Henri (in 1903) and Michaelis and Menten (in 1913) assumed fast equilibrium of the of the catalyst/substrate reaction to form SE, which subsequently dissociates. [8, 9] Briggs and Haldane (in 1925), on the other hand, obtained the same result by assuming that SE in quasi-steady state. [10] The Briggs and Haldane method is usually used in elementary biology classes. However it is interpreted, the same equation is obtained.

The canonical Pycellerator arrow

```
[S :-> P,MMH[K, v]]
```
(14)

is used to produce the rate law in equation (13), which in general only has two parameters: where v is the maximum reaction rate and K is the substrate concentration at half maximum. The actual enzyme concentration E does not (normally) come into the rate law as it is absorbed into the constant v. The rate law equation (13) will be used to produce differential equation terms for the variables P and S, which will be added to other differential equation terms in the model.

Additional versions of this model implemented in Pycellerator allow K and v to be replaced by (a)K, v, and E (this replaces v in equation (13) with vE)

```
[S :-> P,mod[E],MMH[K, v]]
```
(15)

(b) k_1, k_2, and k_3 (this sets $v = k_3$ and $K = (k_2 + k_3)/k_1$ in equation (13))

```
[S :-> P,MMH[k1,k2,k3]]
```
(16)

and (c) E, k_1, k_2, and k_3 (this sets $v = k_3E$ and $K = (k_2 + k_3)/k_1$ in equation (13)):

```
[S :--> P,mod[E],MMH[k1,k2,k3]]
```
(17)

In (a) and (c) the variable E may be any other species in the model that is controlled by its own dynamics, including other reactions or assignments, or it may be a fixed parameter. See table 3 for details.

2.1.4 Hill Functions

Pycellerator includes several regulatory arrows (table 4). In a regulatory arrow, only the species on the right hand side of the arrow are affected by the resulting new differential equation terms. The species listed on the left hand side (LHS) of the arrow contribute information to the system, in the sense that they define how these terms are constructed, but the LHS species are not consumed. Regulatory arrows include Hill functions, GRN (Genetic Regulatory Network Arrows), S-Systems, and Rational functions.

Hill functions frequently arise as approximations of the cooperative binding of ligands. Because of their sigmoidal shape, Hill functions can sometimes be numerically optimized to accurately described bistable switches, where the amount, concentration, or rate of production of one species (say Y) depends on the corresponding amount or concentration of a second species (say X).[11] The canonical form in Pycellerator is

```
[X|->Y,Hill[v,n,K,a,T]]                    (18)
```

which is described by the differential equations term

$$\frac{dY}{dt} = \frac{v(a+TA)^n}{K^n + (a+TX)^n} \tag{19}$$

Here v, n, K, a and T are constants that are allowed to take on any floating point or integer value. In particular, the exponent (n) is not restricted to a positive integer, and may take on negative, or even fractional values. A traditional hill function with cooperativity n and concentration and half-maximum K is obtained by setting $a = 0$ and $T = v = 1$. Multiple inducers $X_1, X_2, \ldots, X_n$ can be combined in a single arrow,

```
[[X1,X2,..,Xn]|->Y,Hill[v,n,K,a,[T1,T2,..Tn]]]                    (20)
```

This is described by differential equation terms

$$\frac{dY}{dt} = \frac{v(a + \sum T_j X_j)^n}{K^n + (a + \sum T_j X_j)^n} \tag{21}$$

A facilitated version of the Hill arrow is also available, which multiplies the corresponding differential equation terms by an optional modifier.

```
[[X1,X2,..,Xn]|-->Y,mod[E],Hill[v,n,K,a,[T1,T2,..Tn]]]          (22)
```

The differential equation terms produced by equation (19) for Y and equation (21) for $Y_1, \ldots, Y_n$ will be added to the other differential equations for those variables.

2.1.5 GRN Arrows

Genetic Regulatory Network (GRN) arrows are useful for modeling transcriptional networks, gene regulation, and any interactions involving bistability or switching. They can be numerically fit to molecular sub-networks to describe overall input-output behavior without actually describing the specific molecular mechanisms occurring within the sub-network. The GRN functions used in Pycellerator are logistic functions. The slope and location of the decision/threshold boundary can be optimized to fit available data. [12] Logistic functions are commonly used in machine learning to solve decision problems and as threshold functions in neural network models. The probability distribution described by a logistic function can be related to a two state Boltzmann distribution or softmax process. [13]

GRN arrows are summarized in Table 4. The basic GRN arrow in Pycellerator is

```
[X|->Y,GRN[v,T,n,h]]          (23)
```

where v, T n, and h are constants that may be set to any floating point value. In particular, there is no restriction that exponent n be integer, and it may take on fractional or negative values. This produces the differential equation term

$$\frac{dY}{dt} = \frac{v}{1+e^{-(h+TX^n)}} \tag{24}$$

The GRN arrow does not affect the differential equation for the variables on the left side of the equation (X in this case). A standardized logistic function $1/(1+e^{-x})$ is obtained by setting $v = T = n = 1$ and $h = 0$. Extended forms of the GRN arrow include an optional modifier species that multiplies the rate v and the use of multiple input species.

```
[[X1,X2,..,Xk]|->Y,GRN[v,[T1,T2,..,Tk],n,h]]
[[X1,X2,..,Xk]|-->Y,mod[E],GRN[v,[T1,T2,..,Tk],n,h]]
```
(25)

This changes the differential equation term to

$$\frac{dY}{dt} = \frac{vE}{1+e^{-(h+\sum T_j X_j^n)}} \tag{26}$$

If the `mod[E]` is omitted in the arrow then the E is omitted from the equation.

2.1.6 Rational Functions

Rational functions produce rate laws that are described by quotients of polynomials. Each term in the polynomial may be a product of species raised to a power. Only the species on the right hand side of the arrow are affected by the reaction. The simple form of the rational arrow is

```
[[[X1,X2,..,Xp],[Y1,..,Yq]]==>Z,rational[
[a0,a1,..,ap],[d0,d1,..,d1],[m0,..,mp],[n0,..,nq]]]
```
(27)

The corresponding contribution to the rate law is

$$\frac{dZ}{dt} = \frac{a_0^{m_0} + a_1 X_1^{m_1} + a_2 X_2^{m_2} + \cdots + a_p X_p^{m_p}}{d_0^{n_0} + d_1 Y_1^{n_1} + d_2 Y_2^{n_2} + \cdots + d_q Y_q^{n_q}} \tag{28}$$

In the more general case, each X_i or Y_j can be replaced by a product of species. For example, the arrow

```
[[[A,B*C,D*E*F],[A, B,B*C,B*C*D,B*C,B*D]]==>A
 rational[[a0,a1,a2,a3],[b0,b1,b2,b3,b4,b5,b6],
 [m0,m1,m2,m3],[n0,n1,n2,n3,n4,n5,n6]]]
```
(29)

contributes the single differential equation term (in Python)

```
A′  =  (A**m1*a1 + a0**m0 + a2*(B*C)**m2 + a3*(D*E*F)**m3)/
       (A**n1*b1 + B**n2*b2 + b0**n0 + b3*(B*C)**n3
      + b4*(B*C*D)**n4 + b5*(B*C)**n5 + b6*(B*D)**n6)
```
(30)

An example that includes the use of rational functions is given by the implementation of plant stem cell lineage in the distribution folder (file `chickarmane.model` in the models folder).

2.1.7 Generalized MWC

The Monod-Wyman-Changeaux (MWC) model [14] describes allosteric enzymes with multiple binding sites that influence one another's affinities. In addition, such an enzyme is typically composed of multiple sub-units that may exist in different states or conformations. We follow the "generalized" MWC model of [15], which also accounts for for multiple activator and inhibitor factors in allosteric enzymes. The basic generalized MWC arrow is

```
[S==>P,mod[E],MWC[k,n,c,L,K]]
```
(31)

where k,n,c,L, and K are constants. This produces differential equation terms for both S and P.

$$\frac{dP}{dt} = E\frac{s\,(1+s)^{n-1} + Lsc\,(1+sc)^{n-1}}{(1+s)^{n} + L\,(1+sc)^{n-1}} = -\frac{dS}{dt} \tag{32}$$

where $s = S/K$. The generalized arrow is

$$\texttt{[[S1,..] ==> P,mod[E,[A1,..],[I1,..],[[CI1,..],[CA1,..]],MWC[k,n,c,L,K]]} \tag{33}$$

Here A_i, I_i, C_{ij} are optional sequences of activators, inhibitors and competitive inhibitors at the substrate and activator site, and K is a list of constants

$$[K_{S1}, K_{S2}, .., K_{A1}, K_{A2}, .., K_{I1}, K_{I2}, .., K_{CI1}, .., K_{CA1}, ..] \tag{34}$$

Let $s_j = S_j/K_{Sj}$, $a_j = A_j/K_{Aj}$, $i_j = I_j/K_{Ij}$, $\overline{s_j} = c\sum_k C_{jk}/K_{C_{jk}}$, and $\overline{a_j} = c\sum_k C_{jk}/K_{CA_{jk}}$. Define the intermediate terms

$$\mathcal{A} = \prod(1 + a_j + \overline{a_j})^n \tag{35}$$

$$\mathcal{I} = \prod(1 + i_j)^n \tag{36}$$

$$\mathcal{S} = \prod(1 + s_j + \overline{s_j})^{n-1} \tag{37}$$

$$\mathcal{S}_c = \prod(1 + cs_j + \overline{s_j})^{n-1} \tag{38}$$

Then the generalized model generates terms

$$\frac{dP}{dt} = -\frac{dS_i}{dt} = E\frac{\mathcal{A}\mathcal{S}\prod s_j + L\mathcal{I}\mathcal{S}_c\prod(cs_j)}{\mathcal{A}\prod(1+s_j)^n + L\mathcal{I}\mathcal{S}_c\prod(1+cs_j)} \tag{39}$$

in the system of differential equations.

2.1.8 NHCA

The basic form for Non-hierarchical cooperative activation (NHCA)[16, 17] is

```
[[X1,X2,..]|-->Y, mod[E], NHCA[v,[TP1,..],[TM1,..],[n1,..],m,k]]   (40)
```

where `X1,..` are one or more reactants, `Y` is the product, `E` is a modifier, `TP1, TP2, ...` `TM1, TM2, ...`, `n1,n2,..`, `v`, `m` and `k` are numeric parameters. The corresponding rate law is

$$\frac{dY}{dt} = vE\frac{\prod_i \left(1 + T_{Pi}X_i^{n_i}\right)^m}{k\prod_i \left(1 + T_{Mi}X_i^{n_i}\right)^m + \prod_i \left(1 + T_{Pi}X_i^{n_i}\right)^m} \tag{41}$$

2.1.9 User Defined Arrows

Users can define arrows with their own rate laws. Let `X1`, `X2`, ... and `Y1`, `Y2`, ... be reactants and products, respectively, with numeric stoichiometries `e1`, `e2`, ..., and `f1`, `f2`, ... Then the basic arrow form

```
[e1*X1 + e2*X2 + ...  -> f1*Y1+f2*Y2+...,using[expr]]          (42)
```

Here `using` is a Pycellerator keyword and *`expr`* represents any evaluatable Python expression involving model species. The user arrow contributes differential equation terms

$$\frac{dZ}{dt} = (f_z - e_z) \times (expr) \tag{43}$$

for each model species Z that appears in a user reaction, where f_z and e_z are the stoichiometry of Z on the right and left hand sides of the arrow.

Similarly, a user-defined regulatory arrow takes the form

```
[[X1,X2,..,Xk]|->Y,USER[v,[T1,T2,..,Tk],[n1,n2,..,nk],h,f]]      (44)
```

Here `X1`, `X2`, ..., `Xk` are the input variables, whose values are not affected by the arrow; `Y` is the output variable; `v`, `T1`, ..., `Tk`, `n1`, ..., `nk`, `h` are numeric parameters; and `f` is a function defined in the `$Functions` section of the model file. The arrow contributes the following term to the differential equation for Y:

$$\frac{dY}{dt} = vf\left(h - \sum_i T_i X_i^{n_i}\right) \tag{45}$$

As a simple example, the following partial model file:

```
$Reactions
 [ Nil <-> X, rates[a,d] ]
 [ X |->Y, USER[v,T,n,h,f] ]
 [ Y->Nil, k ]
$Functions
 f(x)=1/(1+exp(-x))
```

This be converted to the following equations (in Python):

```
f = lambda x :1/(1 + exp(-x))
Y′ = -Y*k + 1.0*v*f(-T*X**n - h)
X′ = -X*d + a
```

The lambda expansion gives a rate law term for Y of

$$\frac{dY}{dt} = -kY + \frac{1}{1 + e^{TX^n+h}} \tag{46}$$

This is similar to a generalized GRN expansion with arbitrary exponent.

2.1.10 Cascades

A Pycellerator cascade is sequence of repeated reactions with the same arrow. For example, the enzymatic arrows

```
[MAPK => MAPKp, mod[KKpp], rates[a,d,k]]
[MAPKp => MAPKpp, mod[KKpp], rates[a,d,k]]          (47)
```

can be combined into a single arrow

```
[MAPK => MAPKp => MAPKpp, mod[KKpp], rates[a,d,k]]          (48)
```

as shown in listing 2.

Any mass action, `MMH`, `Hill`, `GRN`, `SSystem`, or `NHCA` reaction can be written as a cascade to reduce the number of arrows in the model. The reduction can be significant, especially when cascades are combined with enzymatic expansion. A three stage MAPK cascade might be written as

$$\left.\begin{array}{l}
\texttt{[KKK <=> KKKp, mod[S, KKKph], rates[a1,d1,k1,0,a4,d4,k4]]} \\
\texttt{[KK <=> KKp <=> KKpp, mod[KKKp,KKph], rates[a3,d3,k3,0,a5,d5,k5]]} \\
\texttt{[MAPK <=> Kp <=> Kpp, mod[KKpp,Kph], rates[a3,d3,k3,0,a6,d6,k6]]}
\end{array}\right\} \tag{49}$$

This can be further simplified with an indexed notation:

$$\left.\begin{array}{l}
\texttt{[K(3,0) <=> K(3,1), mod[S, KKKph], rates[a1,d1,k1,0,a4,d4,k4]]} \\
\texttt{[K(2,0) <=> K(2,1) <=> K(2,2), mod[K(3,1),KKph], rates[a3,d3,k3,0,a5,d5,k5]]} \\
\texttt{[K(1,0) <=> K(1,1) <=> K(1,2), mod[K(2,2),Kph], rates[a3,d3,k3,0,a6,d6,k6]]}
\end{array}\right\} \tag{50}$$

A three stage MAPK cascade including stimulation and feedback can be written with only five arrows using <=> cascades. Using only the forward arrow cascades (=>) this can be done in eight reactions. Without any cascades, but still using enzymatic => arrows, twelve reactions are required. Each of these models would expand to the same system of 34 of simple mass action reactions, which would have to be typed in manually without using any enzymatic expansion.

2.2 Flux Models

A Pycellerator model may be composed either entirely of kinetic arrows as described above) or entirely of flux arrows. The two may not be combined. Reactions that represent fluxes are a fundamentally different type of entity than reactions used in kinetic models. This is because flux reactions do not (necessarily) have a rate law or differential equation (of the same sort) associated with them. What they do normally have is a total rate, given by the product of a velocity and a stoichiometry.

The format of a flux arrow is

$$\left.\begin{array}{r}\texttt{[e1*X1 + e2*X2 + ...-> f1*Y1+f2*Y2+ ...,}\\ \texttt{Flux[low < var < up, obj, fluxvalue]}\end{array}\right\} \tag{51}$$

where `X1,X2,...` and `Y1,Y2,..` are species; `e1,e2,..` and `f1,f2,..` are stoichiometries; `var` is an identifier used to refer to the flux variable for the reaction; `low` and `up` are numeric lower and upper bounds for optimization; `obj` is a numeric objective coefficient; and `fluxvalue` is an optional numeric flux value.

Users should be cautioned that the symbol "<" used in the `Flux` arrow is inclusive and really corresponds to the mathematical inclusive "less than or equal to" symbol, "$\leqslant$." Optimization is inclusive, not exclusive. There is no less than or equal to symbol in Pycellerator, only the single "<" is used.

For example,

$$\texttt{[ES -> E + S, Flux[0 < v < 1, 1, 0]} \tag{52}$$

The flux arrow means that the optimization is performed so that $0 \leqslant v \leqslant 1$, even though only the "<" is used. Attempting to include an equal sign in the expression will lead to a syntax error.

To force equality, say $v = 1$, in a constraint, one would use `Flux[1<v<1]`. The flux optimization process will solve the linear programming problem

$$\left.\begin{array}{rl}\text{maximize} & \mathbf{v^T f}\\ \text{subject to} & \mathbf{Nv} = \mathbf{0}\\ \text{and} & low_1 \leqslant v_1 \leqslant up_1\\ \text{and} & low_2 \leqslant v_2 \leqslant up_2\\ & \vdots\end{array}\right\} \tag{53}$$

where $\mathbf{v}$ is the vector of fluxes $(v_1, v_2, \dots)^{\mathtt{T}}$, $\mathbf{N}$ is the stoichiometry matrix, and $\mathbf{f}$ is the vector of objective coefficients. The optimization is performed using Python's `pulp` package. [18]

2.3 Functions

The `$Functions` section of the model file contains a list of function definitions in standard algebraic notation. A function may have multiple arguments and these are treated as dummy parameters. When the function is instantiated the parameters are replaced with the arguments used in the function invocation. Within the function any other global parameters (such as rate constants) may be referenced. The general format is

$$\texttt{f(v1,v2,...vn) = } \textit{expr} \tag{54}$$

where `f` is the function name, as it is used elsewhere in the model; `v1`, ..., `vn` are the function dummy arguments, as they are referenced on the right hand side of the function definition; and *`expr`* is a standard arithmetic expression that is evaluable in Python. All variables and parameters must be conform to the rules for permissible Python identifiers (e.g., case sensitive, alphanumeric, must start with with a letter). When the model is converted to Python, each function is converted to a Python lambda expression.

An example with two functions is given in the minimal cascade model for a mitotic oscillator (listing 3, [19]). The function `f(m,x)` has two arguments,

$$\texttt{f(m,x) = m * (1-x)/(K3+1-x)} \tag{55}$$

This function is instantiated as `f(M,X)` to define a concentration dependent rate constant for the reaction `[Nil -> X]`. The second function `g(m)` has one argument,

$$\texttt{g(m) = (1-m)/(K1+1-m)} \tag{56}$$

and is instantiated as `g(M)` to produce a concentration dependent rate for the hill function. In this particular model, the functions use variable names that are simular to (e.g., lower case versions) of the variables used in their instantiations. No such restriction is actually placed on the user, and equation (57) could just as well have been implemented as

$$\texttt{f(foo, bar) = foo* (1-bar)/(K3+1-bar)} \tag{57}$$

2.4 Assignments

The optional `$Assignments` section of the model file contains a list of species definitions as set equal to statements. The general format is

$$\texttt{X = } \textit{expr} \tag{58}$$

where `X` is the species name, as it is used elsewhere in the model, and *`expr`* is any Python expression. These assignments hold at all times throughout a simulation. If `X` is a species that would otherwise be define by a differential equation, then it should also be listed in the `$FROZEN` section to ensure that the differential calculation is inhibited.

2.5 Initial Conditions

Initial conditions are defined in the `$IC` section. A species is not required to have an initial condition, but if an initial condition is omitted, it is assumed to be zero. The `$IC` section contains a sequence of statements of the form

$$\texttt{X = value} \tag{59}$$

where `X` is the species name, and value is the numeric value of the species at $t = 0$. If a variable is specified by an assignment rule then it should not be given an initial condition.

2.6 Parameter Values

Parameter values are defined in the `$Rates` section. All constants and parameters that are defined using an identifier in the model must be given a value in this section. The `$Rates` section contains a sequence of statements of the form

$$\texttt{identifier = value} \tag{60}$$

where `identifier` is the parameter name, and value is the numeric value of the parameter. Parameters can also be replaced with algebraic (Python) expressions in the `$ASSIGNMENT` sections. If a parameter is listed on the left-hand side of an assignment statement then its parameter value will be ignored.

2.7 Frozen Variables

Frozen species do not contribute terms to the system of differential equations. However, if a variable is frozen but is given an assignment rule, it may still change as a function of time. This provides a convenient way to provide a time-dependent input. Frozen species are listed in the `$Frozen` section of the model file. This section contains a list of the frozen species, one species per line. Only species may be frozen, not other parameters in the model.

2.8 Identifiers and Symbols

Identifiers in the model, i.e., species representing reactants, products and modifiers; function names; and parameters (rate constants), must start with a letter and may contain any number of alphanumeric characters in them. Identifiers are case sensitive, and may also contain the underscore character. Users should beware that the underscore character is also used by Pycellerator to join species names when auto-generating new species names (for example, see reactions (11)).

The special identifier `Nil` is used to represent the empty set and it is not converted into a differential equation term. Thus reactions such as `[Nil -> X]` and `[X -> Nil]` represent *cretio ex nihilo* and removal from the system, respectively.

The special identifier `t` may be used in functions and assignments to define explicit time dependent expressions. For example, a constant stimulation of $S = 1$ may be turned on from $t = 100$ to $t = 200$ by setting S as a frozen variable and using an assignment:

```
$Frozen
 S
$Assignments
 S = (0.0 if t<100 else (stim if t<200 else 0.0))
```

```
        $Rates
         stim=1
```

Species may be indexed using parenthesis, e.g., `[X(1)->Y(2)]` or `[K(2,0) => K(2,1) => K(2,2)]`. When the model is instantiated the index numbers are embedded into the variable name; they are not implemented as either Python arrays or lists.

If more than one statement is placed on a single line in the model file, the statements should be separated by a semicolon.

3 Protocols

This section describes how to install the necessary software; instantiate models (generate Python functions); and run simulations using Pycellerator.

3.1 Requirements

3.1.1 Install Python

1. Install Python 2.7 if it is not already present. You will probably need to run the instllation in adminstrator mode (Windows) or sudo (Linux).

   On Windows, it is generally easier to install a complete scientific version. Links for several complete scientific packages are given at `https://www.scipy.org/install.html`.

   On Macs, Python is already installed by default, and it is not normally necessary to reinstall it.

   Linux users should be able to install Python using their package manager.

2. Install `setuptools` using `pip`. The program `pip` should automatically be installed when Python is installed.

   In a Windows command shell (it is called `cmd.exe`), type the following:

```
python -m pip install -U pip setuptools
```

From the terminal program on a Mac or in Linux, type

```
pip install -U pip setuptools
```

If `pip` fails to run in this manner, follow the instructions at `https://packaging.python.org/installing` to download and install `get-pip`. Then repeat this step.

3. Install the required packages: `numpy`, `scipy`, `sympy`, `matplotlib` and `pyparsing`. From the command shell (any operating system),

```
pip install numpy scipy sympy matplotlib pyparsing
```

Linux users may prefer to install these packages from their package repositories, but it does not matter whether you use the repository or `pip`

4. (Optional) Install the optional packages `pulp`, `ipython` and `jupyter`. If `pulp` is not installed, then flux models cannot be solved. If `ipython` and `jupyter` are not installed, then the notebook interface will not be available. From the command shell (any operating system),

```
pip install pulp ipython jupyter
```

5. (Optional) Install libsbml for Python 2.7. To be able to either read or write SBML files you must install libsbml. For most operating systems, type the following in the command shell

```
pip install python-libstall
```

Before you do this, check for operating-system-specific instructions at `http://sbml.org/Software/libSBML/docs/python-api/`.

3.1.2 Pycellerator Installation

It is not necessary to use administrator or superuser mode to install Pycellerator.

1. Download Pycellerator from the github repository at `https://github.com/biomathman/pycellerator/releases`. Look for the file `install-pycellerator-v-X.zip` (where `X` is some number) and download that file to your computer. Advanced users may be more interested in the source but you don't need that to run to Pycellerator.

2. Unzip and create working folder. Unzip the download, which will probably be in your Downloads folder. Look for a folder called `pycellerator` in that unzipped file. Copy this entire folder anywhere you want on your disk drive, such as your home folder or your desktop. This is going to be your working folder for Pycellerator. It is not necessary to modify your Python path so long as you run models from this folder.

   You should see two folders inside the `pycellerator` folder: `cellerator` and `models`. The `cellerator` folders contains code needed to run the program and should not be modified. The `models` folder contains sample model files.

3.2 Model Instantiation and Simulation

3.2.1 Plot Time Course in A Notebook

Here we consider simulation and plotting of the basic model shown in listing 1. This model contains a single arrow representing reaction (**??**) using enzymatic expansion (arrow (8)). In the model file, only the first three of the four rate constant are specified, so the fourth rate constant defaults to zero.

1. Set your current working directory to the `pycellerator` folder that you created during installation. You can do this, e.g., by opening the folder in your desktop manager.

2. Using a text editor open a new text file and copy or type the contents of listing 1 into it. Note that if you cut and paste from an electronic version of this paper, the fonts will most likely generate a few incompatible characters. It is best to verify that only valid text

characters (e.g., UTF-8) are in your file. Then save your file as `basicmodel.model` in the current working directory.

3. Open the `jupyter` notebook interface. To do this, open a command shell (`cmd.exe` in Windows, `terminal` in MacOS or Linux) and type

   ```
   ipython notebook
   ```

   This will open the `jupyter` notebook interface in your default browser.

4. Create a new notebook. From the drop down menu near the top right of the `jupyter` window, select `New > Python 2`. This will open a new window labeled "untitled." From the drop down menu on the top left of the window select `File > Rename`. Type a name for the notebook, such as "Basic-Model" in the pop-up window and click OK. Your file will be named `Basic-Model.ipynb`. The file extension `ipynb` is required.

   If you click back on the tab `Home` in your browser you should see a list of files. One of those files will be the file `Basic-Model.ipynb` that you just created. If you go to your desktop and open a folder, you will also see the file. (Note that some operating systems may suppress visibility of the file extension (the letters after the dot in the file name) when you look at your list of files this way.) You will not be able to edit or modify this file except using the `jupyter` interface because it is written in a special format that is called JSON. If you open it up and look at it in any other format it will probably look like nonsense to you.

5. Click on the tab for you notebook. In the first cell type in the required Python includes:

   ```
   from cellerator import cellerator as c
   import matplotlib.pyplot as plt
   import numpy as np
   %matplotlib inline
   ```

   After entering code in any cell, click on enter to ensure that the code is executed.

   Note that `pyplot` and `numpy` are not strictly necessary as separate imports. They are used inside the program, but not directly accessible by user. If you want to make modifications using `pyplot` or `numpy` features you may need to import them.

6. To determine the differential equations for the model and print them to the screen in Python form,

```
model="basicmodel.model"
c.PrintODES(model)
```

For this model, the output should be

```
E′ = −E⋆S⋆a + S_E⋆d + S_E⋆k
S′ = −E⋆S⋆a + S_E⋆d
S_E′ = E⋆S⋆a − S_E⋆d − S_E⋆k
P′ = S_E⋆k
```

7. Solve the model. The basic function is `c.Solve`:

```
t,v,s=c.Solve(model)
```

Here `model` is the file name as before, and the return value is a Python tuple `(t,v,s)`.

- `t` is a numpy array of times at which the solution is returned in `s`. it is the return value of `odeint`. The default setting for `t=[0,1,2,..,100]`. These can be changed with the keywords `step` and `duration`. Note that `step` only controls what is returned, and is not related to the integration step size.
- `v` is a list of variable names (as strings); in this case the return value would be `['E', 'S', 'S_E', 'P']`
- `s` is a numpy array of solution vectors, one vector per time point, as returned by `odeint`.

8. Plot the results. The basic function is `c.PlotAll`, which takes the the three variables returned by `c.Solve` and returns a Pyplot `axis` object. As long as the line `%matplotlib inline` was executed prior to this step (see step 5), the plot will be displayed in the next cell of your notebook.

```
ax=c.PlotAll(t,v,s)
```

9. Tweak the run and plot parameters. To get a more precise plot, we can re-run the simulation with an output step of 0.1. Since the interesting stuff happens early (with the given values of the parameters) will also only need to run for a short time. Then we can use `pyplot` to add axis labels, change scales, etc.

```
t,v,s=c.Solve(model, step=.1,duration=15)
ax=c.PlotAll(t,v,s)
ax.set_yticks(np.arange(0,1.,.2))
ax.set_yticklabels(np.arange(0,1.01,.2),fontsize=12)
ax.set_xlim(0,15)
ax.set_xticks(np.arange(0,15.1,5))
ax.set_xticklabels(np.arange(0,15.1,5),fontsize=12)
ax.set_xlabel("Time", fontsize=14)
ax.set_ylabel("Value", fontsize=14)
ax.set_title("The Results of My Simulation", fontsize=16)
fig=plt.gcf()
fig.set_size_inches(6,3)
fig.tight_layout()
fig.savefig("basicmodel.pdf")
```

The resulting plot is shown in figure 1.

3.2.2 Run Auto-generated Code as Stand-Alone Program

1. Locate the auto-generated code produced by Pycellerator. The default file name is `solver_for_model_timecode.py`, where `model` is the model name (e.g., `basicmodel` in the previous section); and `timecode` is a time code to uniquely identify the file. The default file name can be overridden in `c.Solve` with the keyword `solverfile`:

```
model="basicmodel.model"
t,v,s=c.Solve(model,solverfile="foo",step=.1,duration=15)
```

2. Locate and examine the auto-generated code (e.g., `foo.py`). The code produced in this model is shown in listing 4. As the program stands right now, nothing would normally be output. The code can be modified with any standard text editor.

3. Modify the autogenerated code (e.g., `foo.py`). For example, to print a comma-separated value listing of the result to the screen, add the following code before the `return` statement of `thesolver()`, between lines 41 and 42,

```
print "t,"+",".join(variables)+","
for t,v in zip (times,sol):
    print ",".join(map(str,list([t])+list(v)))
```

4. Run the program. Type the following in a command shell.

```
python foo.py
```

3.2.3 Run and Plot a Model From the Command Shell

1. To run the basic model with a step size of 0.1 and duration 15, plot the results, and save the results to a CSV file, type the following in the command shell on a single line

```
python pycellerator.py solve -run 15 .2 -in basicmodel.model
   -plot -pyfile spam.py -out eggs.csv
```

   The plot should pop up as a separate window. In some cases it might be hidden behind existing windows. The auto-generated code is written to `spam.py` and the results of the simulation are saved to `eggs.csv`. Additional options are described in the users guide.

2. Optionally modify and rerun the code. To re-run the code generated in step 1, enter

```
python spam.py
```

   from the command shell. The auto-generated code for the model function is identical to the code generated in the notebook. The code generated for the driver is different, since it includes a wrapper for output. If this code is run from the command line, the plot will automatically pop up, and a new CSV file will be generated.

3.2.4 Perform Parametric Tweaks and Scans

In listing 5 we show a toy example of the spread of disease based on the SIRS model, implementing the following system of differential equations, based on the Kermack-McKendrick model with feedback, birth, and death. [22].

$$\left.\begin{aligned} I' &= kIS - (1+d)I \\ S' &= -kIS + bI + R(b+f) + (b-d)S \\ R' &= I - R(d+f) \end{aligned}\right\} \tag{61}$$

The populations of S (susceptible), I (infected), and R (recovered) are dimensionless; k is the ratio of infection to recovery rate (hence non-dimensional); f is some fraction of the recovered population that returns to the susceptible population; and b and d are the population birth and death rates. All newborns are assumed to be susceptible.

1. Tweak individual parameters using `c.Solve`. For example, to override the initial conditions for R and S, and the value for f in the model file, set them at run time. The options `IC` and `RATES` are Python dictionaries.

```
model="SIRS.model"
t, v, s  = c.Solve(model, step=.1, duration=100,
    IC={"R":.5,"S":.5},
    RATES={"f":.5})
c.PlotAll(t,v,s)
```

2. Do a parametric scan. To determine the values of all the state variables at, say, $t = 200$, as a function f for $0.5 \leqslant 5 \leqslant 1$ in steps of 0.5, use the `scan` keyword.

```
variables, pscan=c.Solve(model,
  scan=["f",0.05,1,0.05], duration=200)
```

In this case `c.Solve` returns a 2-tuple rather than a 3-tuple. The first item is a list of the variables in the model, as strings. For this model, it will return the list `['I', 'S',` 'R']. The second item is a `numpy` array of vectors, where each vector has the form (in this case) `[f, I, S, R]`. The state variables (i.e., I, S, and R) are evaluated at the very end of the simulation. Typically this would be when the simulation reaches steady state, but some knowledge of the model is required to verify this. Pyellerator does not verify steady state; it merely returns the values at the time requested.

3. Plot the parametric scan. This can be done using `pyplot`, or the data can be exported to a spreadsheet or plotting program. For example, to plot the fraction recovered (R) as a function of f,

```
fvals=pscan[:,0]; RVals=pscan[:,3]
plt.plot(fvals, RVals,marker="o")
plt.xlabel("f",fontsize=14)
plt.ylabel("Fraction Recovered", fontsize=14)
plt.title("SIRS model", fontsize=16)
```

   The resulting parametric scan is show in figure 2.

3.2.5 Include a Time Dependent Stimulation

The easiest way to include a time-dependent stimulation in a model is as follows.

1. Add a dummy reaction that would normally create a steady state value for your stimulation, such as

$$\texttt{[Nil<->S, rates[a0, d0]]} \tag{62}$$

   This tells the system to treat `S` as a species. If you omit this reaction, `S` will be considered an unknown variable during the simulation and the program will terminate with an error. Normally reaction (62) would lead to a differential equation of the form `S'=a0-d0*S`. However, this is overridden in the following step.

2. Make `S` a frozen variable by adding a line containing `S` to the `$Frozen` section in the model file. This tells the Pycellerator to replace the differential equation in step 1 with `S'=0`.

3. Define an assignment rule that explicitly gives the value of `S` as a function of time using standard Python. This tells Pycellerator to replace the differential equation for `S` with an algebraic expressio for `S`. For a square pulse, use a Python ternary operator:

```
     S=(0.0 if t<t1 else (K if t<t2 else 0.0))
```

   in the `$Assignments` block. The values of `t1`, `t2`, and `K` can be initialized in the `$Rates` block. This way you can manually override the values during a simulation without editing the file. To use a more complex stimulation and make the model file more readable, use a function in the model file.

```
    $Assignments
     S=(0.0 if t<t1 else (f(t) if t<t2 else 0.0))
    $Functions
      f(t)=sin((t-1000.0)*pi/(3000))
```

   Any function in the Python math library may be referenced.

4. Run the simulation. For a model with a large number of variables, plot the results in a grid.

```
    t, v, s  = c.Solve(model, step=.1, duration=5000)
    c.PlotColumns(t,v,s,ncols=4,bg="white",colors=23*["black"])
```

   Verify the stimulation on the plots (e.g., figure 3).

4 Remarks

Pycellerator can be freely downloaded from github. A public respository is located at `https://github.com/biomathman/pycellerator/releases`. All software is covered by a GPL version 3 license.

The complete syntax and all options are detailed in the user manual that is included with the download package.

Pycellerator is implemented in Python 2.7. There are no plans at the present time to implement the program in Python 3.

Table 1: Arrows and keywords used in Pycellerator arrows expressions.

Arrow	Keyword	modifer	Description Typical usages
`->`	N/A	no	Mass action
`-->`	N/A	yes	Mass action with modifier
`<->`	`rates`	no	Mass action
`=>`	`rates`	yes	Mass action expansion, SE complex
`:=>`	`rates`	yes	Mass action expansion, SE and PE complex
`<=>`	`rates`	yes	Mass action expansion, SE complex
`|->`	`Hill`	no	Hill Function
	`GRN`	no	Generalized logistic rate function
	`SSystem`	no	S-System
	`USER`	no	User defined rate law
`|-->`	`Hill`	yes	Hill Function
	`GRN`	yes	Generalized logistic rate function
	`NHCA`	yes	Non-hierarchical cooperative activation.
	`USER`	yes	User defined rate law
`:->`	`MMH`	No	Michaelis-Menten-Henri-Briggs-Haldane
`:-->`		Yes	
`==>`	`MWC`	yes	Monod-Wyman-Changeaux model.
	`Rational`	no	Rational Function

Table 2: Mass Action Reactions

Pycellerator Syntax	Biochemical Notation	Note
`[ A + B -> C, k ]`	$\mathrm{A + B} \xrightarrow{k} \mathrm{C}$	(a)
`[ A + B <-> C, rates[k1, k2] ]`	$\begin{cases} \mathrm{A + B} \xrightarrow{k_1} \mathrm{C} \\ \mathrm{C} \xrightarrow{k_2} \mathrm{A} + B \end{cases}$	(a)
`[ e1*X1 + e2*X2 + ··· -> f1*Y1 + f2*Y2 + ···, k]`	$\sum_i e_i \mathrm{X}_i \xrightarrow{k} \sum_j f_j \mathrm{Y}_j$	(b,c)
`[ e1*X1 + e2*X2 + ··· <-> f1*Y1 + f2*Y2 + ···, rates[k1,k2]]`	$\sum_i e_i \mathrm{X}_i \underset{k2}{\overset{k_1}{\rightleftarrows}} \sum_j f_j \mathrm{Y}_j$	(b,c)
`[ S=>P, mod[E], rates[k1,k2,k3,k4] ]`	$\mathrm{S + E} \underset{k_2}{\overset{k_1}{\rightleftarrows}} \mathrm{SE} \underset{k_4}{\overset{k_3}{\rightleftarrows}} \mathrm{P + E}$ or: $\begin{cases} \mathrm{S + E} \xrightarrow{k_1} \mathrm{SE} \\ \mathrm{SE} \xrightarrow{k_2} \mathrm{S + E} \\ \mathrm{SE} \xrightarrow{k_3} \mathrm{P + E} \\ \mathrm{P + E} \xrightarrow{k_4} \mathrm{PE} \end{cases}$	
`[S<=>P,mod[F,R],rates[k1,k2,k3,k4, k5,k6,k7,k8]]`	$\begin{cases} \mathrm{S + F} \underset{k_2}{\overset{k_1}{\rightleftarrows}} \mathrm{SF} \underset{k_4}{\overset{k_3}{\rightleftarrows}} \mathrm{P + F} \\ \mathrm{P + R} \underset{k_6}{\overset{k_5}{\rightleftarrows}} \mathrm{PR} \underset{k_8}{\overset{k_7}{\rightleftarrows}} \mathrm{S + R} \end{cases}$ or: $\begin{cases} \mathrm{S + F} \xrightarrow{k_1} \mathrm{SF} & \mathrm{SF} \xrightarrow{k_2} \mathrm{S + F} \\ \mathrm{SF} \xrightarrow{k_3} \mathrm{P + F} & \mathrm{P{+}F} \xrightarrow{k_4} \mathrm{SF} \\ \mathrm{P + R} \xrightarrow{k_5} \mathrm{PR} & \mathrm{PR} \xrightarrow{k_6} \mathrm{P + R} \\ \mathrm{PR} \xrightarrow{k_7} \mathrm{S + R} & \mathrm{S{+}R} \xrightarrow{k_8} \mathrm{SR} \end{cases}$	
`[S:=>P,mod[E],rates[k1,k2,k3, k4,k5,k6]]`	$\mathrm{S + E} \underset{k_2}{\overset{k_1}{\rightleftarrows}} \mathrm{SE} \underset{k_4}{\overset{k_3}{\rightleftarrows}} \mathrm{PE} \underset{k_6}{\overset{k_5}{\rightleftarrows}} \mathrm{P + E}$ or: $\begin{cases} \mathrm{S + E} \xrightarrow{k_1} \mathrm{SE} & \mathrm{SE} \xrightarrow{k_2} \mathrm{S + E} \\ \mathrm{SE} \xrightarrow{k_3} \mathrm{PE} & \mathrm{PE} \xrightarrow{k_4} \mathrm{SE} \\ \mathrm{PE} \xrightarrow{k_5} \mathrm{P + E} & \mathrm{P + E} \xrightarrow{k_6} \mathrm{PE} \end{cases}$	

(a) May be multiple reactants and products. (b) The stoichiometries `ei` and `fj` are numeric. (c) The multiplication symbol (asterisk, "*") between the stoichiomety and species is optional; however, the stoichiometry must come first, and be numeric. If the stoichiometry is equal to one, it may be omitted.

Table 3: Michaelis Menten type arrows in Pycellerator.

Pycellerator Syntax	Rate Law
`[S:->P,MMH[K, v]]`	$\frac{dP}{dt} = -\frac{dS}{dt} = \frac{vS}{K+S}$
`[S:->P,mod[E],MMH[K, v]]`	$\frac{dP}{dt} = -\frac{dS}{dt} = \frac{vSE}{K+S}$
[S:->P,MMH[k$_1$,k$_2$,k$_3$]]	$\frac{dP}{dt} = -\frac{dS}{dt} = \frac{k_3 S}{(k_2+k_3)/k_1 + S}$
[S:->P,mod[E],MMH[k$_1$,k$_2$,k$_3$]]	$\frac{dP}{dt} = -\frac{dS}{dt} = \frac{k_3 SE}{(k_2+k_3)/k_1 + S}$

Table 4: Regulatory arrows in Pycellerator. Regulatory arrows only affect the variables on the right-hand side of the arrow symbol; they do not contribute differentiatial equation terms to variables on the left hand side.

Type	Pycellerator arrow	Differential equation term
Hill	`[X|->Y,Hill[v,n,K,a,T]]`	$Y' = \dfrac{v(a+TX)^n}{K^n+(a+TX)^n}$
	`[[X1,X2,..,Xn]|->Y, Hill[v,n,K,a,[T1,T2,..,Tn]]]`	$Y' = \dfrac{v(a+\sum T_jX_j)^n}{K^n+(a+\sum T_jX_j)^n}$
	`[[X1,X2,..,XN]|-->Y, mod[E],Hill[v,n,K,a, [T1,T2,..,Tn]]`	$Y' = \dfrac{vE(a+\sum T_jX_j)^n}{K^n+(a+\sum T_jX_j)^n}$
GRN	`[X|->Y,GRN[v,T,n,h]]`	$Y' = \dfrac{v}{1+e^{-(h+TX^n)}};$
	`[[X1,X2,..,Xn]|->Y, GRN[v,[T,..],n,h]]`	$Y' = \dfrac{v}{1+e^{-(h+\sum T_jX_j^n)}}$
	`[[X1,X2,..,Xn]|-->Y, mod[E],GRN[v,[T,..],n,h]]`	$Y' = \dfrac{vE}{1+e^{-(h+\sum T_jX_j^n)}}$
S-System	`[[X1,..,Xn]|->Y,SSystem[tau, a,b,[g1,..,gn],[h1,..,hn]]`	$Y' = \dfrac{1}{\tau}\left(a\prod_i X_i^{g_i} - b\prod_i X_i^{h_i}\right)$
Rational	`[[[X1,X2,..],[Y1,Y2,..]]==>Z, rational[[a0,a1,a2,..], [d0,d1,d2,..],[m0,m1,m2,..], [n0,n1,n2,..]]]`	$Z' = \dfrac{a_0^{m_0}+\sum_i a_iX_i^{m_i}}{d_0^{m_0}+\sum_i d_iY_i^{n_i}}$
	`[[[X11*X12*..., X21*X22*..., ], [Y11*Y12*..., Y21*Y22*... ]]] ==>Z,rational[[a0,a1,a2,..], [d0,d1,d2,..],[m0,m1,m2,..], [n0,n1,n2,..]]]`	$Z' = \dfrac{a_0^{m_0}+\sum_i a_i(X_{i1}X_{i2}\cdots)^{m_i}}{d_0^{m_0}+\sum_i d_i(Y_{i1}Y_{i2}\cdots)^{n_i}}$

Listing 1: A basic model describing the reaction $S+E \underset{d}{\overset{a}{\rightleftarrows}} SE \xrightarrow{k} P+E$ using three elementary reactions.

```
$REACTIONS
 [S+E -> SE, a]
 [SE -> S+E, d]
 [SE -> P+E, k]
$IC
 S = 1
 E = 1
 P = 0
$Rates
 a = 1
 d = 1
 k = 1
```

Listing 2: Model of MAPK oscillation demonstrating the use of cascades and an external stimulation. Stimulation is provided by species S.[20, 21].

```
$Reactions
 [Nil<->S, rates[a0, d0]]
 [KKK <=> KKKp, mod[S, KKKph], rates[a1,d1,k1,0,a4,d4,k4]]
 [KK <=> KKp <=> KKpp, mod[KKKp,KKph], rates[a3,d3,k3,0,a5,d5,k5]]
 [MAPK <=> Kp <=> Kpp, mod[KKpp,Kph], rates[a3,d3,k3,0,a6,d6,k6]]
 [KKK_S + Kpp <-> KKK_S_Kpp, rates[a7, d7]]
$IC
  KKK =  100;  KKKp =  0
  KK =  300;   KKp =  0;  KKpp =  0
  MAPK =  300; Kp =  0;   Kpp =  0
  Kph =  1;    KKph =  1; KKKph =  10
$Frozen
 S
$Assignments
 S=0. if t<1000 else (1.0 if t<4000 else 0.0)
$Rates
 a0 = 1; d0 = 1
 a1 = 1; d1 = 7.5; k1 = 2.5
 a3 = 1; d3 = 10;  k3 = 0.025
 a4 = 1; d4 = 1;   k4 = 1
 a5 = 1; d5 = 1;   k5 = 1
 a6 = 1; d6 = 1;   k6 = 1
 a7 = 1; d7 =  1
```

Listing 3: Goldbeter's Minimal cascade model for a mitotic oscillator.[19] This file is included in the distribution as sample model Gold1; an alternative version called Goldbeter does not use functions.

```
$REACTIONS
 [C <-> Nil, rates[kd, vi]]
 [C |--> Nil, mod[X], Hill[vd, 1, Kd, 0,1 ]]
 [M |--> Nil, mod[Nil], Hill[v2, 1, K2, 0, 1]]
 [X |--> Nil, mod[Nil], Hill[v4, 1, K4, 0, 1]]
 [Nil -> X, "vm3*f(M,X)"]
 [C |-> M, Hill["vm1*g(M)", 1, Kc, 0, 1]]
$IC
 C = 0.1
 M = 0.2
 X = 0.3
$FUNCTIONS
 f(m,x) =  m * (1-x)/(K3+1-x)
 g(m) = (1-m)/(K1+1-m)
$RATES
 vd = 0.1;   vi = 0.023; v2 = 0.167;   v4 = 0.1
 vm1 = 0.5;  vm3 = 0.2;  kd = 0.00333; K1 = 0.1
 K2 = 0.1;   K3 = 0.1;   K4 = 0.1;     Kc = 0.3
 Kd = 0.02
```

Listing 4: Auto-generated Python code for the model shown in listing 1.

```
import numpy as np
from scipy.integrate import odeint

from math import *
def ode_function_rhs(y,t):
  #
  # this odeint(..) compatible function was
  # automatically generated by Cellerator 2016-07-31 12:27:17
  # 2.7.6 (default, Jun 22 2015, 17:58:13)  [GCC 4.8.2]
  # linux2
  #
  # ============================================================
  # Model:
  #
  # [S => P, mod[E], rates[a,d,k]]
  # ============================================================
  # rate constants
  a = 1.0
  d = 1.0
  k = 1.0
# pick up values from previous iteration
  E = max(0, y[0])
  S = max(0, y[1])
  S_E = max(0, y[2])
  P = max(0, y[3])
# calculate derivatives of all variables
  yp=[0 for i in range(4)]
  yp[0] = -E*S*a + S_E*d + S_E*k
  yp[1] = -E*S*a + S_E*d
  yp[2] = E*S*a - S_E*d - S_E*k
  yp[3] = S_E*k
  return yp

def thesolver():
    filename ="/home/mathman/Desktop/pycellerator/basicmodel.model"
    variables=['E', 'S', 'S_E', 'P']
    runtime = 15
    stepsize = 0.1
    times = np.arange(0,runtime+stepsize,stepsize)
    y0 = [1.0, 1.0, 0.0, 0.0]
    sol = odeint(ode_function_rhs, y0, times, mxstep=50000)
    return sol

if __name__=="__main__":
    thesolver()
```

Listing 5: Simple SIRS disease model described in equations (61).

```
$REACTIONS
 [S + I -> I + I, k]
 [R -> Nil, d]; [I -> Nil, d]; [S -> Nil, d]
 [R -> R+S, b]; [I -> I + S, b]; [S -> S+S, b]
 [I -> R, 1]
 [R -> S, f]
$IC
 S = 0.9999999
 I = 1.0E-7
 R = 0
$RATES
 k = 5.0
 d = 0.0005
 b = 0.0005
 f = 0.05
```

References

[1] Shapiro, BE, Mjolsess E (2016)Pycellerator: an arrow-based reaction-like modelling language for biological simulations. Bioinformatics.32(4):629-31. doi: 10.1093/bioinformatics/btv596.

[2] Shapiro, BE,Levchenko, A, Meyerowitz, EM, Wold, BJ, Mjolsness, ED (2003) Cellerator: extending a computer algebra system to include biochemical arrows for signal transduction simulations. Bioinformatics 19:677-678, doi: 10.1093/bioinformatics/btg042.

[3] Hucka, M, Finney, A, Sauro, HM, et al. (2003) The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. Bioinformatics, 19:513-523. doi: 10.1093/bioinformatics/btg015

[4] Jones E, Oliphant E, Peterson P, et al. (2001) SciPy: Open Source Scientific Tools for Python. `http://www.scipy.org/` (Online; accessed 2016-08-01).

[5] van der Walt, S, Colbert, SC, Varoquaux g. (2011) The NumPy Array: A Structure for Efficient Numerical Computation Computing in Science & Engineering. 13:22-30. doi:10.1109/MCSE.2011.37

[6] Waage, P, Guldherg,CM (1864) Forhandlinger i Videnskabs Selskabet i Christiania, 37.

[7] Field, RJ, Noyes, RM (1974) Oscillations in chemical systems. IV. Limit cycle behavior in a model of a real chemical reaction. J. Chem. Phys. 60: 1877-1884. doi:10.1064/1.1681288

[8] Henri, V (1903) Lois Générales de l'action des Distases. Hermann, Paris.

[9] Michaelis, L, and Menten, M L (1913) Die Kinetik der Invertinwirkung, Biochemische Zeitschrift 49:333-369.

[10] Briggs, GE, Haldane, JBS (1925) A Note on the Kinetics of Enzyme Action. Biochemical Journal 19(2):338-339. doi: 10.1042/bj0190338

[11] Hill, AV (1910) The possible effects of the aggregation of the molecules of haemoglobin on its dissociation curve. Proc. Phsyiol. Soc. 40 (suppl): 4-7. doi: 10.1113/jphysiol.1910.sp001386

[12] Mjolsness, E, Sharp, DH, Reinitz, J (1991) A Connectionist Model of Development. J. theor. Biol. 152: 429-453. doi:10.1016/S0022-5193(05)80391-1

[13] Murphy, KP (2012) Machine Learning, A Probabilistic Perspective. MIT Press, Cambridge.

[14] Monod, J, Wyman, J, Changeux,JP (1965) On the Nature Of Allosteric Transitions: A Plausible Model. J. Mol. Biol. 12:88-118. doi:10.1016/S0022-2836(65)80285-6

[15] Najdi, T, Yang, C-R, Shapiro, BE, Hatfield, W, Mjolsness, E(2006) Application of a Generalized MWC Model for the Mathematical Simulation of Metabolic Pathways Regulated by Allosteric Enzymes. J Bioinform Comput Biol 4(2):335-55. doi :10.1142/S0219720006001862

[16] Mjolsness, E. (2000) Trainable gene regulation networks with applications to Drosophila pattern formation. In: Bower,JM, Bolouri, H (ed) Computational Models of Genetic and Biochemical Networks, MIT Press, Cambridge.

[17] Shapiro BE, Mjolsness ED (2001) Developmental Simulations with Cellerator. Paper presented at Second International Conference on Systems Biology, Pasadena, 4-7 Nov. 2001

[18] Mitchell, S, O'Sulivan, M, Dunning, I (2011) PuLP: A Linear Programming Toolkit for Python. `http://www.optimization-online.org/DB_FILE/2011/09/3178.pdf`. Accessed 31 July 2016.

[19] Goldbeter, A (991) A minimal cascade model for the mitotic oscillator involving cyclin and cdc2 kinase. Proc. Natl. Acad. Sci. USA 88:9107-1101 (1991). doi 10.1073/pnas.88.20.9107

[20] Huang, CY, Ferrell, JE Jr (1996) Ultrasensitivity in the mitogen-activated protein kinase cascade. Proc Natl Acad Sci U S A. 93(19):10078-83.

[21] Kholodenko BN (2000) Negative feedback and ultrasensitivity can bring about oscillations in the mitogen-activated protein kinase cascades. Eur J Biochem. 267(6):1583-8 doi 10.1046/j.1432-1327.2000.01197.x

[22] Kermack, WO, McKendrick, AG (1927) A Contribution to the Mathematical Theory of Epidemics. Proc. Royal Soc. A. 115(772):700-721. doi 10.1098/rspa.1927.0118

Figure 1: Time course of simulation of basic model shown in listing 1.

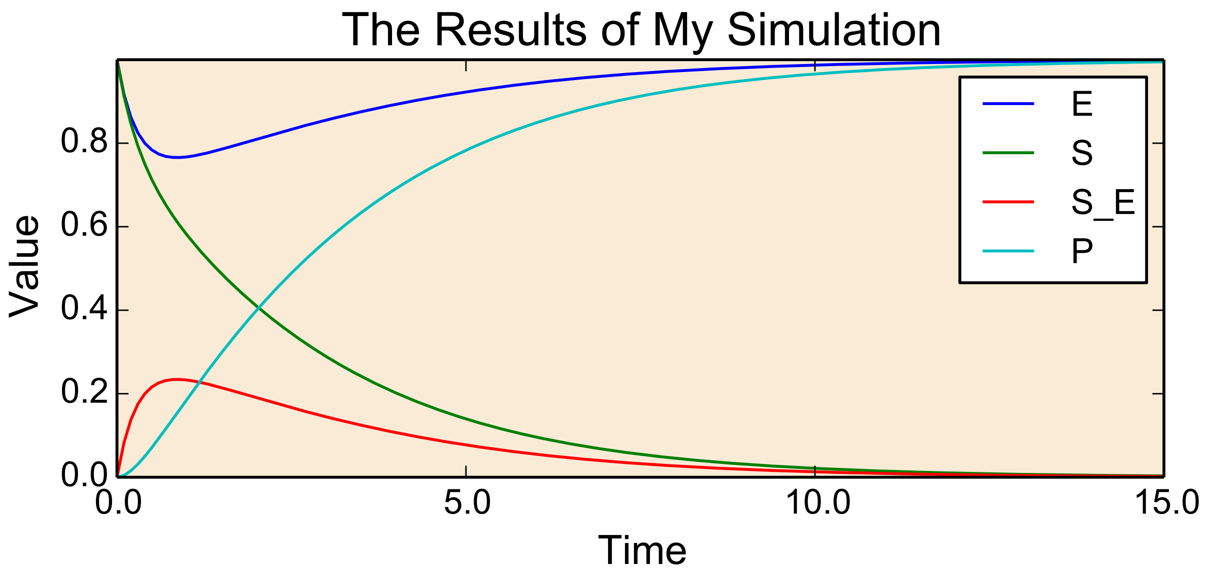


Figure 2: Parametric scan of the SIRS model (listing 5).

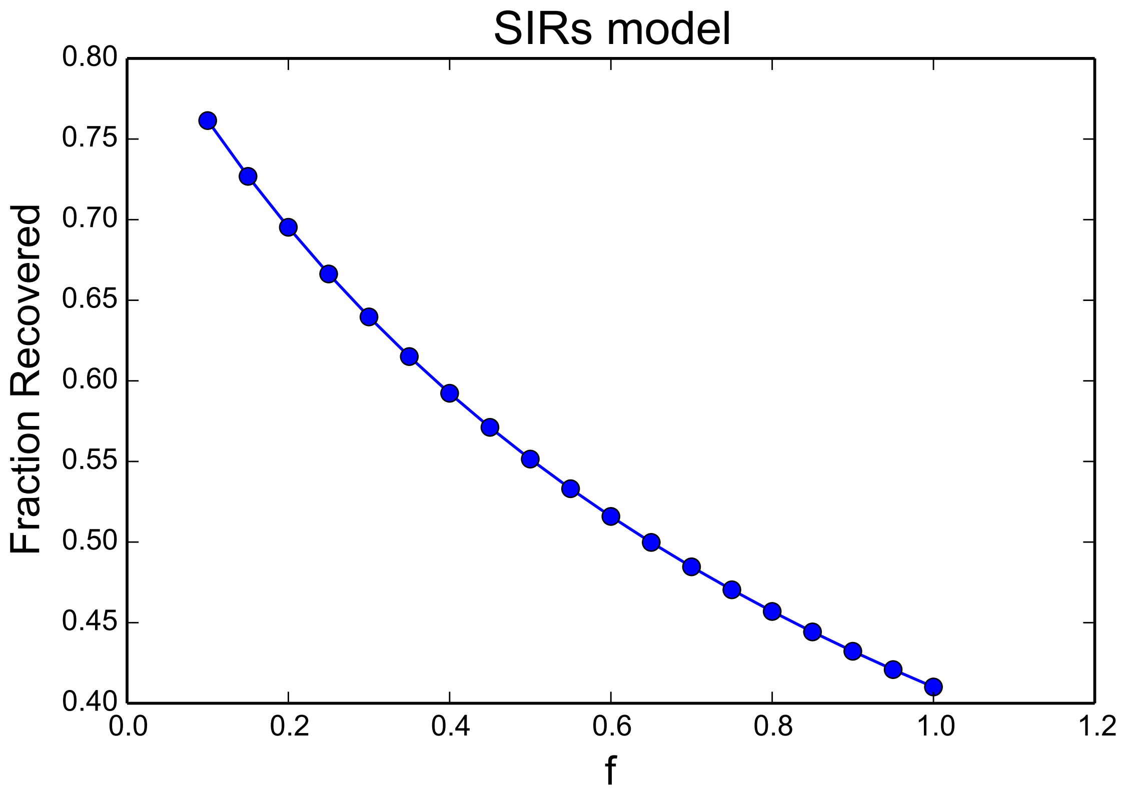
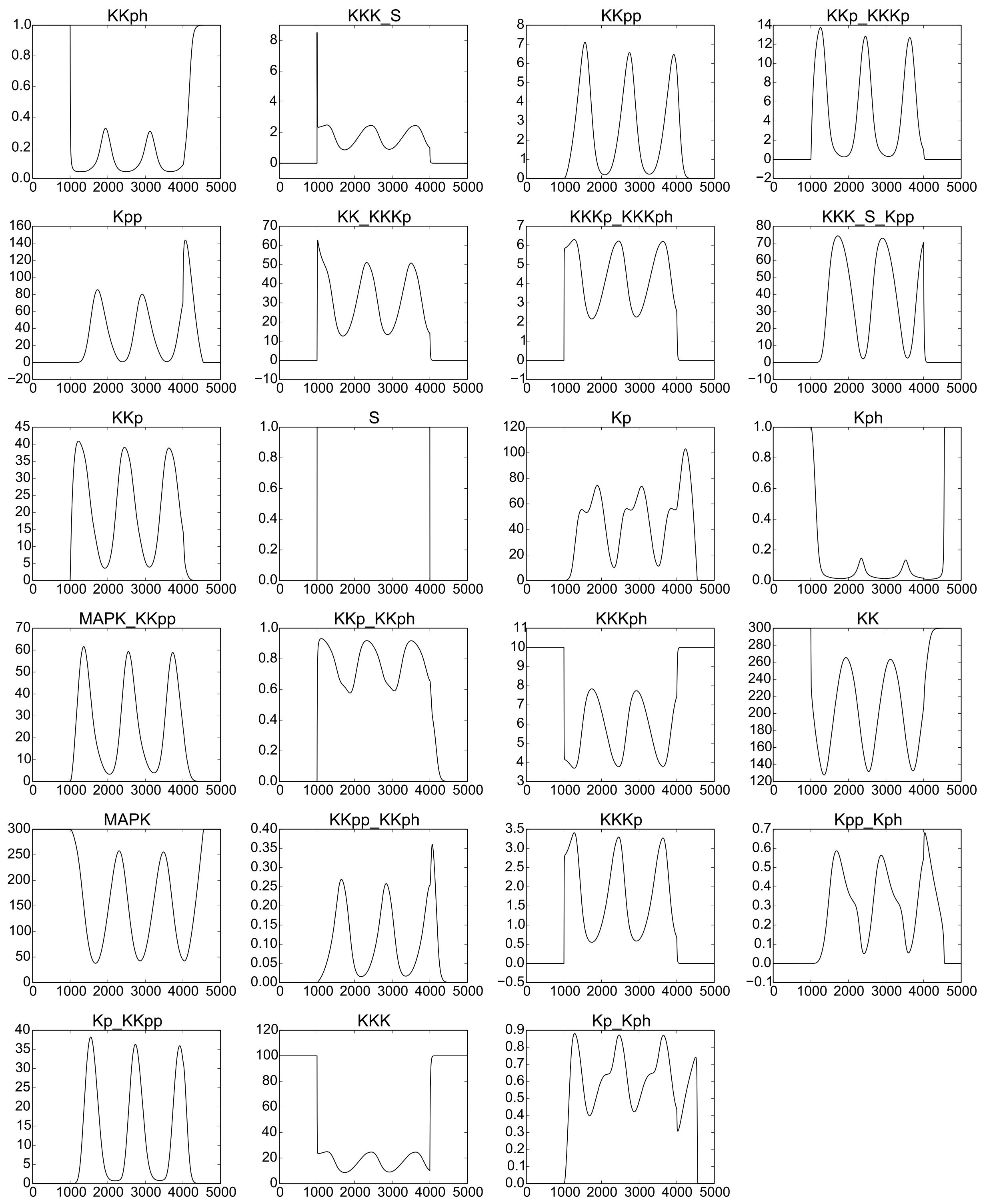


Figure 3: Oscillations in MAPK cascade with feedback and square wave stimulation. The model is shown in listing 2.


KKph
KKK_S
KKpp
KKp_KKKp
Kpp
KK_KKKp
KKKp_KKKph
KKK_S_Kpp
KKp
S
Kp
Kph
MAPK_KKpp
KKp_KKph
KKKph
KK
MAPK
KKpp_KKph
KKKp
Kpp_Kph
Kp_KKpp
KKK
Kp_Kph