We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

6,900
Open access books available

185,000
International authors and editors

200M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities



WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com



2

From Discrete to Continuous Gene Regulation Models – A Tutorial Using the Odefy Toolbox

Jan Krumsiek[1,2], Dominik M. Wittmann[1,3] and Fabian J. Theis[1,3]
[1]*Institute of Bioinformatics and Systems Biology, Helmholtz Zentrum München*
[2]*Center of Life and Food Sciences, Technische Universität München*
[3]*Department of Mathematics, Technische Universität München*
Germany

1. Introduction

Vital functions of living organisms, such as immune responses or the metabolism, are controlled by complex regulatory networks. These networks comprise, amongst others, regulatory genes called transcription factors and cascades of information-processing proteins such as enzymes. The ultimate goal of the increasingly popular systems biology approach is to set-up extensive computer models that closely reflect the real-life behavior of these biological networks (Kitano, 2002; Werner, 2007). With a reasonable in silico implementation at hand, novel predictions, e.g. about the effect of gene mutations, can be generated by the computer.
The two basic modes of regulation we concentrate on here, are inhibition and activation between two factors. Figure 1A visualizes the relation between the concentrations of e.g. two transcription factors, which are linked by an activation (left-hand figure) or inhibition (right-hand figure). Figure 1B shows a network of interacting activations and inhibitions as it might be found in living cells. While a single regulatory interaction can easily be understood, the complex wiring of several interactions, even for a medium-scale model as depicted here, renders the manual investigation of the system's dynamics unfeasible. For further information on the concepts of regulation, we refer biologically interested readers to Alon (2006).
Classical computational modeling approaches attempt to describe biochemical reaction networks as systems of ordinary differential equations (ODEs) (Klipp et al., 2005; Tyson et al., 2002). This requires detailed knowledge about the molecular mechanisms in order to implement precise kinetic rate laws for each biochemical reaction. However, for many biological systems, and especially gene-regulatory networks, only qualitative information about interactions, like "A inhibits B", is available. A well-established workaround for this lack of information is the application of discrete modeling approaches. In Boolean methodology we abstract from actual molecule quantities and assign each player in the system the state on or off (e.g. active or inactive). Despite their simplicity, Boolean models have been shown to provide valuable information about the general dynamics and capabilities of the underlying system (Albert & Othmer, 2003; Fauré et al., 2006; Samaga et al., 2009).
To bridge the gap between discrete and fully quantitative models, we developed Odefy, a MATLAB- and Octave-compatible toolbox for the automated transformation of Boolean

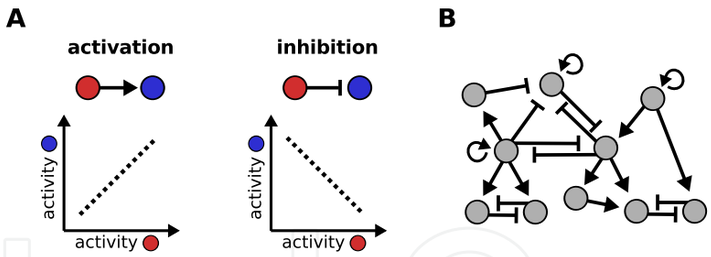


Fig. 1. **A** Two basic modes of regulation, e.g. between two genes and their proteins. If the regulatory factor (red) constitutes an activatory influence towards another factor (blue), it will increase the activity of the blue factor, whereby the magnitude of this activation is dependent on the expression of the red factor itself. Inhibition acts analogously, but the expression of both factors is anti-correlated. **B** Regulatory interactions are part of complex gene-regulatory networks which can be analyzed only by means of computational tools.

models into systems of ODEs (Krumsiek et al., 2010; Wittmann et al., 2009a). Odefy implements a canonical way of transforming Boolean into continuous models, where the use of multivariate polynomial interpolation allows transformation of logic operations into a system of ODEs. Furthermore, we optionally apply sigmoidal Hill functions to get reasonable approximations of real gene regulation dynamics. The Odefy software provides convenient access to different model sources, the conversion process itself and various analysis and export methods. After generating the ODEs, the user can easily adjust model parameters and perform time-course simulations using Odefy's graphical user interface. The ODE systems can be exported to MATLAB script files for further usage in MATLAB programs, to ODE script files for the R computing platform, to the SBML format, or to the well-established MATLAB Systems Biology Toolbox (Schmidt & Jirstrand, 2006). Due to the nice mathematical properties of the produced ODEs and the integration with state-of-the-art modeling tools, a variety of analysis methods can be immediately applied to the models generated by Odefy, including bifurcation analysis, parameter estimation, parameter sensitivity analysis, and the like.
This chapter is organized as follows. First, we will review the theoretical background of the Odefy method by introducing Boolean models, the interpolation process and Hill functions as a generalization of Michaelis-Menten kinetics. Next, the general structure of the Odefy toolbox as well as details about model representation and input formats are discussed. In the major part of this chapter, we will guide the reader through four sample applications of our toolbox, which include both regular Boolean modeling as well as Odefy-converted ODE models, see Table 1 for a detailed overview. The Odefy toolbox can be freely downloaded from `http://hmgu.de/cmb/odefy`. All codes and additional files used in the examples throughout this chapter are located at `http://hmgu.de/cmb/odefymaterials`.

2. Mathematical backgrounds

In the following, we provide a brief introduction to Boolean models in general, and the automatic conversion of Boolean models into continuous systems of ordinary differential equations. For more detailed information on these topics, we refer the reader to the papers Krumsiek et al. (2010); Thomas (1991); Wittmann et al. (2009a).

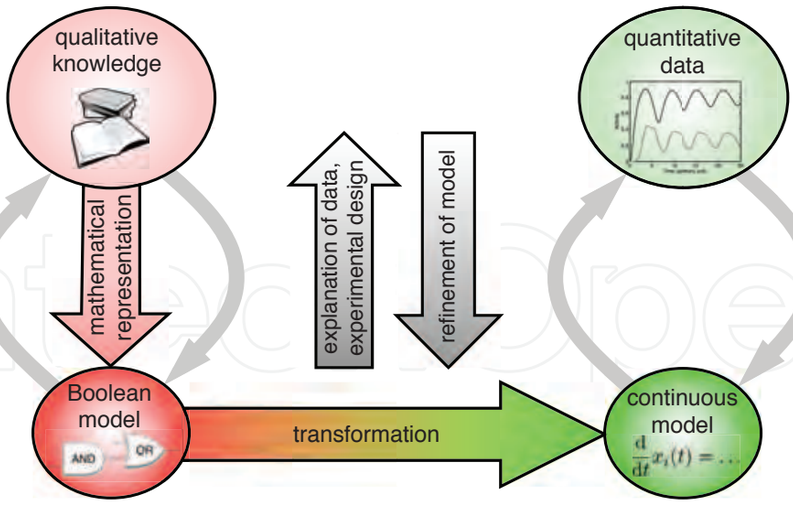


Fig. 2. Qualitative knowledge of regulatory interactions can readily be transformed into Boolean models. The models are then automatically converted to a continuous ODE model by our approach, making them suitable for quantitative analysis and comparison to real data. Figure taken from Wittmann et al. (2009a)

Section	Biological system	Odefy techniques
4	Toy example	the graphical user interface, the toolbox's main functionalities, definition of Boolean models in the yEd graph editor, adjustment of initial values and parameters, Boolean and ODE time-course simulations
5	The genetic toggle switch	advanced model input, advanced functionalities from the MATLAB command line, generating the Boolean state-transition graph, finding Boolean steady-states, phase-plane visualizations
6	Differentiation of mid- and hindbrain	automated model selection
7	Large-scale model of T-cells	export options, connecting Odefy to the SB toolbox, model to .mex compilation

Table 1. Overview of the biological systems and Odefy techniques explained in sections 4–6.

2.1 Boolean models

In a Boolean model, the actual concentration or activity of each factor is abstracted to be either 'on' or 'off', active or inactive, 1 or 0. In a system of N factors with discretized time, regulatory interactions can be described by a set of Boolean update rules that determine the value of each factor x_i at the next time step $t+1$, dependent on all factors in the current time step:

$$x_i(t+1) := B_i\left(x_{i1}(t), x_{i2}(t), \ldots, x_{iN_i}(t)\right) \in \{0,1\} \ , \quad i = 1,2,\ldots N \ .$$

Boolean update functions B_i could, for instance, be represented as multidimensional truth tables, containing an assignment of zero or one for each combination of input factors. A

more convenient and intuitive way of representing Boolean update functions is the usage of symbolic equations with logical operators. For example,

$$A(t+1) = (B(t) \vee C(t)) \wedge \neg D(t)$$

represents a regulatory interaction where A will be 'on' in the next time step if and only if at least one of the activators B and C is present and the inhibitor D is absent. For simplicity, we leave out time dependencies in the Boolean equations:

$$A = (B \vee C) \wedge \neg D$$

Some exemplary evaluations for this equation: (i) if B=1,C=0,D=0 then A will be 1; (ii) likewise, if B=0,C=1,D=0 then A=1; (iii) if D=1, then A=0 regardless of the values of B and C.

When computing the follow-up state for the next time point $t+1$ from the current time point t, two principal updating schemes can be employed (cf. Fauré et al. (2006)). When following a *synchronous* updating policy, the states of all factors are updated at the same time. On the other hand, when performing *asynchronous* updating, the value of only one factor is changed during each time step. For the latter case, an update order for the players in the system is required. In Odefy, one can either provide a predefined update order (e.g. B, A, D, C) that will be followed, or one can let the toolbox randomly select a player at each new time step.

2.2 From Boolean models to ordinary differential equations

We now describe how to generate a system of ordinary differential equations (ODEs), given a set of Boolean update functions B_i. The main idea is to convert the above discrete model into a continuous ODE model, where each species x_i is allowed to take values $x_i \in [0,1]$, and its temporal development is described by the ordinary differential equation

$$\dot{\overline{x}}_i = \frac{1}{\tau_i} \left(\overline{B}_i \left(\overline{x}_{i1}, \overline{x}_{i2}, \ldots, \overline{x}_{iN_i} \right) - \overline{x}_i \right) \ .$$

The right-hand side of this equation consists of two parts, an activation function $\overline{B}_i$ describing the production of species x_i and a first-order decay term. An additional parameter τ_i is introduced to the system, which can be understood as the life-time of species x_i. $\overline{B}_i$ can be considered a continuous homologue of the Boolean update function. The key point is how it can be obtained from B_i in a computationally efficient manner. We present here three concrete approaches of extending a Boolean function to the continuous interval [0,1]. The basis of all three transformation methods are the so-called *BooleCubes*:

$$\overline{B}^{\mathrm{I}} \left(\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_N \right) := \sum_{x_1=0}^{1} \sum_{x_2=0}^{1} \cdots \sum_{x_N=0}^{1} \left[B(x_1, x_2, \ldots, x_N) \cdot \prod_{i=1}^{N} \left(x_i \overline{x}_i + (1-x_i)\left(1-\overline{x}_i\right) \right) \right]$$

which we obtain by multilinear interpolation of the Boolean function B, cf. Figure 3A. The functions $\overline{B}^{\mathrm{I}}$ are affine multilinear. Many molecular interactions, however, are known to show a switch-like behavior, which can be modeled using sigmoidal *Hill functions* $f\left(\overline{x}\right) = \overline{x}^n / \left(\overline{x}^n + k^n\right)$, see Figure 3B. Hill functions are a generalization of the well-known

Michaelis-Menten kinetics assuming multiple binding sites (Alon, 2006). The two parameters n and k have a direct biological meaning. The Hill coefficient n determines the slope of the curve and is a measure of the cooperativity of the interaction. The parameter k corresponds to the threshold in the Boolean model, above which one defines the state of a species as 'on'. Mathematically speaking, it is the value at which the activation is half maximal, i.e. equal to 0.5. If not otherwise specified, Odefy assumes the default parameters n=3, k=0.5 and τ=1 for all equations.

We now introduce a Hill function f_i with parameters (n_i, k_i) for every interaction and define a new continuous function

$$\overline{B}^{\mathrm{H}}\left(\overline{x}_1, \ldots, \overline{x}_N\right) := \overline{B}^{\mathrm{I}}\left(f_1\left(\overline{x}_1\right), \ldots, f_N\left(\overline{x}_N\right)\right) ,$$

which we call *HillCubes*, see Figure 3C. One can show that for sufficiently large Hill exponents n, there will be a steady state of the continuous system in the neighborhood of each Boolean steady state Wittmann et al. (2009a). In other words, the continuous model is capable of reproducing the Boolean steady states, but of course displays a much richer dynamical behavior.

Note that Hill functions never assume the value 1, but rather approach it asymptotically. Hence, the HillCubes are not perfect homologues of the Boolean update function B. If this is desired a simple solution is to normalize the Hill functions to the unit interval. This yields another continuous (perfect) homologue of the Boolean function B

$$\overline{B}^{\mathrm{Hn}}\left(\overline{x}_1, \ldots, \overline{x}_N\right) := \overline{B}^{\mathrm{I}}\left(\frac{f_1\left(\overline{x}_1\right)}{f_1\left(1\right)}, \ldots, \frac{f_N\left(\overline{x}_N\right)}{f_N\left(1\right)}\right) ,$$

which we call *normalized HillCube*, see Figure 3D.

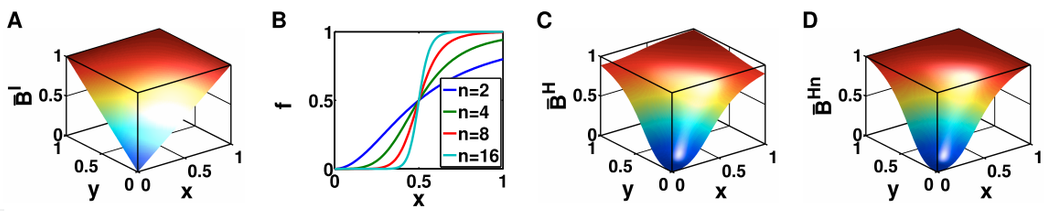


Fig. 3. **A** Multilinear interpolation of a two-variable OR gate (BooleCube) as the continuous homologues of Boolean functions. **B** Hill functions with Hill coefficients n = 2, 4, 8, 16 and k = 0.5 as continuous relaxation of a Boolean step function. **C** Composition of BooleCube from A with Hill functions (HillCube). **D** Normalized HillCube which actually assumes a value of 1 at the corners of the cube. Figure taken from Krumsiek et al. (2010).

3. The Odefy toolbox

This section explains how to start up the Odefy toolbox, and how Boolean models are represented as MATLAB structure variables. Note that the non-GUI functionality of Odefy

is compatible with the freely available Octave toolbox[1]. After downloading and unpacking Odefy[2], we navigate to the respective directory in MATLAB and call

```
InitOdefy
```

which should display a startup message like this:

```
Odefy initialized
For detailed usage instructions type 'OdefyHelp'
or open /home/jan/work/odefy/doc/index.html in your webbrowser.
```

Odefy is now ready-to-use. The core object in the toolbox is a Boolean model which can be defined in various ways (Figure 4), two of which we will get to know in the example section 4. We will here discuss a small example, namely an incoherent feed-forward loop (Alon, 2006) defined by a set of Boolean equations:

```
model = ExpressionsToOdefy(...
    {'A=<>', 'B=A', 'C=A&&~B'});
```

In this model, A is defined as an input species without regulators (denoted by the <>), which never changes its current activity value. B is directly activated by A and will thus closely follow A's expression. Finally, C is activated by A and inhibited by B. Note the use of MATLAB Boolean operators in the Boolean expression. The equation reads "C will be active if A is active and B is not active". The command generates a MATLAB structure variable containing the Boolean model:

```
>> model

model =

    species: {'A'  'B'  'C'}
     tables: [1x3 struct]
       name: 'odefymodel'
```

The `tables` field contains the actual Boolean update functions encoded as multidimensional arrays, that is as hypercubes of edge length two:

```
>> model.tables(3).inspecies

ans =

     1     2

>> model.tables(3).truth

ans =

     0     0
     1     0
```

The third factor, C, has two input species 1 and 2, that is A and B. The four-element truth table precisely describes the above-mentioned "A and not B" logic defined in the symbolic Boolean equation. In order to update the state of a factor, Odefy looks up the corresponding value in the truth table, based on the current system state, and returns the new expression value.
With the defined Boolean model variable we can now access the full functionality of the toolbox, like simulations of the Boolean model and, of course, the conversion of Boolean

[1] http://www.gnu.org/software/octave/
[2] http://hmgu.de/cmb/odefy

models to ODE systems as described in section 2. The rest of this chapter provides various sample applications of the toolbox based on both Odefy's GUI dialogs and MATLAB command-line programming. For a complete reference of all Odefy features, we refer the reader to the Odefy online documentation at `http://hmgu.de/cmb/odefydocs`.

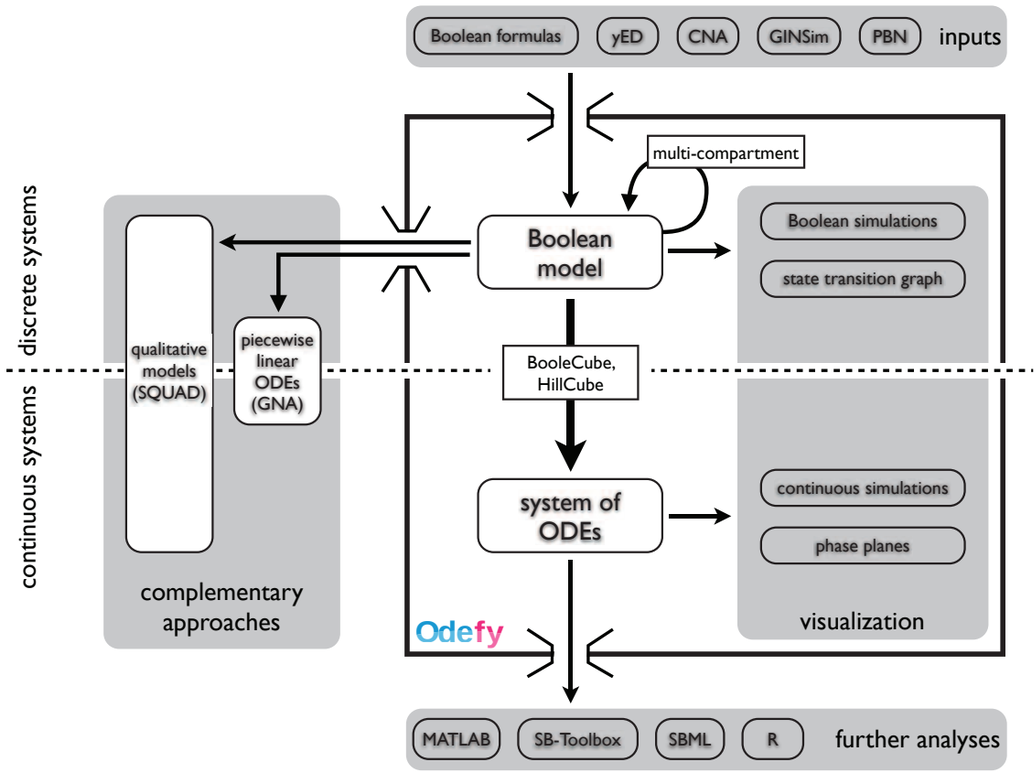


Fig. 4. General structure of the Odefy toolbox. Boolean models can be defined by various input formats. In this chapter, we will introduce both the usage of Boolean formulas and graphs created in the yEd graph editor. Once a Boolean model is created, the user can either concentrate on analyzing the Boolean model, or convert it to a system of ODEs and perform continuous analyses. The resulting models can be exported to several external formats. Figure taken from Krumsiek et al. (2010).

4. A toy example: First steps in Odefy

We will now familiarize the reader with the graphical user interface of the Odefy toolbox, which provides convenient access to the toolbox's main functionalities. In particular, we show how Boolean models can be defined in the yEd graph editor, how initial values and parameters can be adjusted and how Boolean and ODE time-course simulations are run. Note that the Boolean model employed here does not have a real biological background, but was rather constructed to contain important features of gene regulatory networks, like negative feedback, positive feedback, and different wirings of AND and OR logics.

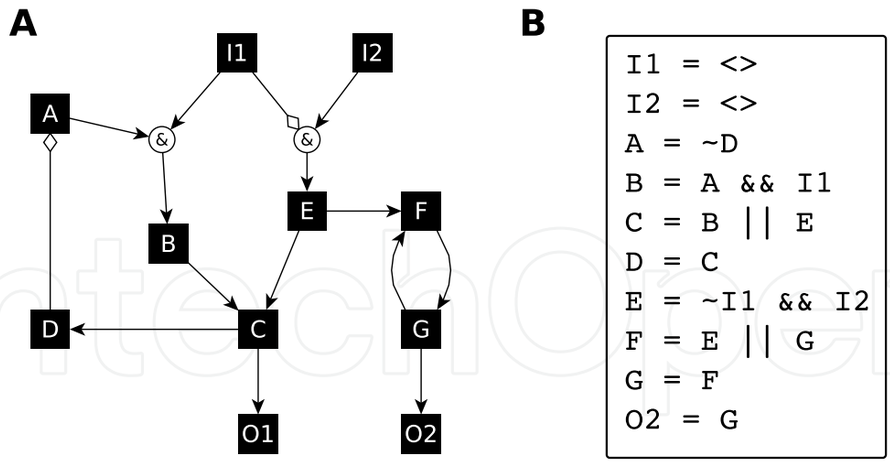


Fig. 5. Two ways of defining the same Boolean model. **A** Graphical representation of the regulatory interactions created in the yEd graph editor. Note the usage of "&" labeled nodes in order to create AND gates. Regular arrows represent activation whereas diamond head arrows stand for inhibition. **B** Boolean equations for the same model. We use <> to indicate input species with no regulators, and MATLAB Boolean operators `||`, `&&` and ∼ to define the Boolean equations.

4.1 Definition of the Boolean model

The most convenient methods to define Boolean models in the Odefy toolbox are Boolean equations and the yEd graph editor[3]. A simple graph, where each node represents a factor of the system and each edge represents a regulatory interaction, is not sufficient to define a Boolean model, since we cannot distinguish between AND and OR gates of different inputs. Therefore, we adapted the intuitive hypergraph representation proposed by Klamt et al. (2006), as exemplarily demonstrated in Figure 5A. All incoming edges into a factor are interpreted as OR gates; for instance, C will be active when B or E is present. AND gates are created by using a special node labeled "&", e.g. E will be active when I2 is present and I1 is not present. We now load this model from a pre-created .graphml file which is contained in the Odefy materials download package. Ensure that Odefy is initialized first:

```
InitOdefy;
```

We can now call the `LoadModelFile` command, which automatically detects the underlying file format:

```
model = LoadModelFile('cnatoy.graphml');
```

As mentioned previously in this chapter, Boolean equations are a convenient alternative for constructing a Boolean model. While obviously the graphical depiction of the network is lost, Boolean equations can be rapidly setup and altered (Figure 5B). We can either load them from a text file containing one equation per line, or directly enter them into the MATLAB command line:

```
model = LoadModelFile('cnatoy.txt');
```

or

```
model = ExpressionsToOdefy({'I1 = <>', 'I2 = <>',
    'A = ~D', 'B = A && I1', ...
    'C = B || E', 'D = C', 'E = ~I1 && I2', 'F = E || G',
    'G = F', 'O2 = G'});
```

[3] http://www.yworks.com/en/products_yed_about.html

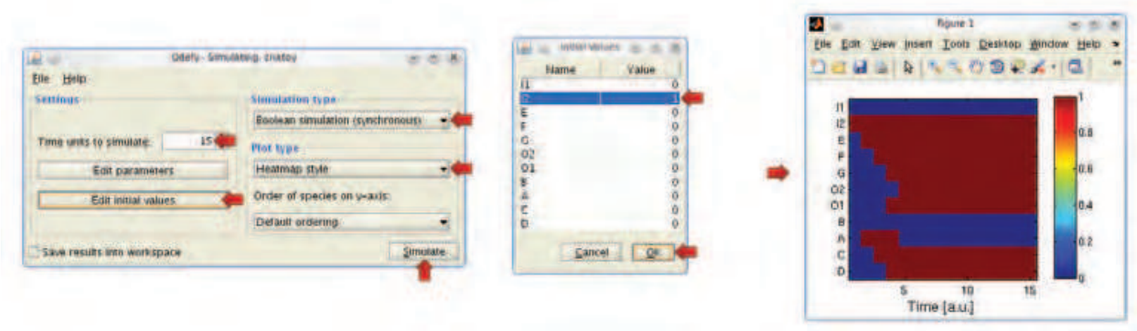
At this point, the `model` variable contains the full Boolean model depicted in Figure 5, stored as an Odefy-internal representation in a MATLAB structure.

4.2 Boolean simulation using the Odefy GUI

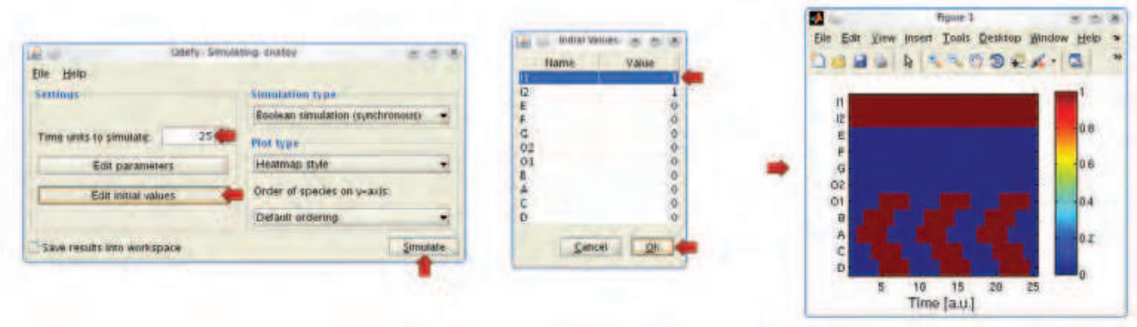
After defining the Boolean model within the Odefy toolbox, we now start analyzing the underlying system using Boolean simulations. We open the Odefy simulation GUI by entering:

```
Simulate(model);
```

A simulation window appears, in which we now setup a synchronous Boolean update policy, change some initial values and finally run the simulation (red arrows indicate required user actions):



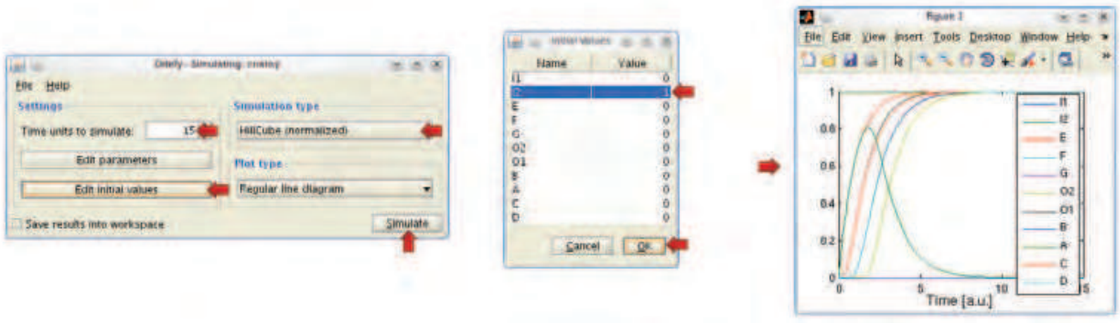
When the input species I2 is active while I1 is inactive, the signal can steadily propagate through the system due to the absent inhibition of E. All species, except for B and A, eventually reach an active steady state after a few simulation steps. A displays an interesting pulsing behavior induced by the negative regulation from C towards A. Initially, A is turned on since its inhibitor D is absent, but is then downregulated once the signal passes through the system. The system produces a substantially different behavior when both input species are active:



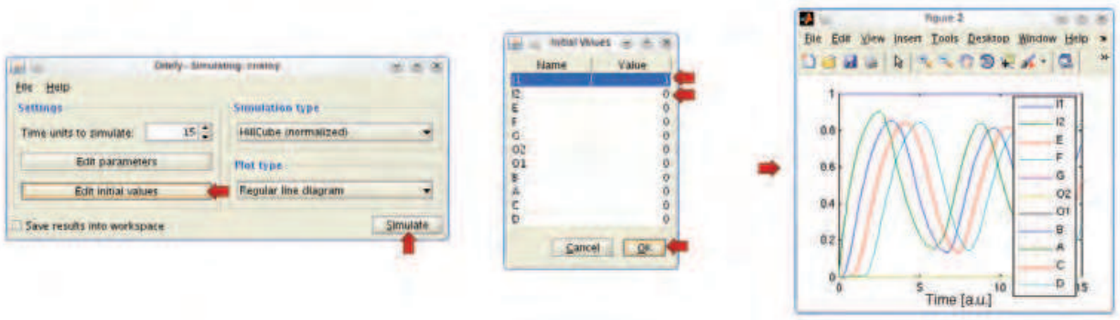
Interestingly, we now observe oscillations in the central part of the network, while the right-hand part with E, F, G and O2 stays deactivated. The oscillations are due to a negative feedback loop in the system along A, B, C and D. Negative feedback basically denotes a regulatory wiring where a player acts as its own inhibitor. In our setup, for example, A indirectly induces D via B and C, which in turn inhibits A. Our obtained results demonstrate that already a simple model can give rise to entirely different behaviors when certain parts of the system are activated or deactivated - here simulated via the initial values of the input species I1 and I2.

4.3 Continuous simulation

In the next steps we will learn how the automatic conversion of Boolean models to ODE systems allows us to quantitatively investigate the pulsing and oscillation effects observed in the Boolean simulation from the previous section. Again, we use the simulation GUI of Odefy, but this time we choose the normalized HillCube variant. In the GUI variant of Odefy, the conversion to an ODE system is automatically performed prior to the simulation.



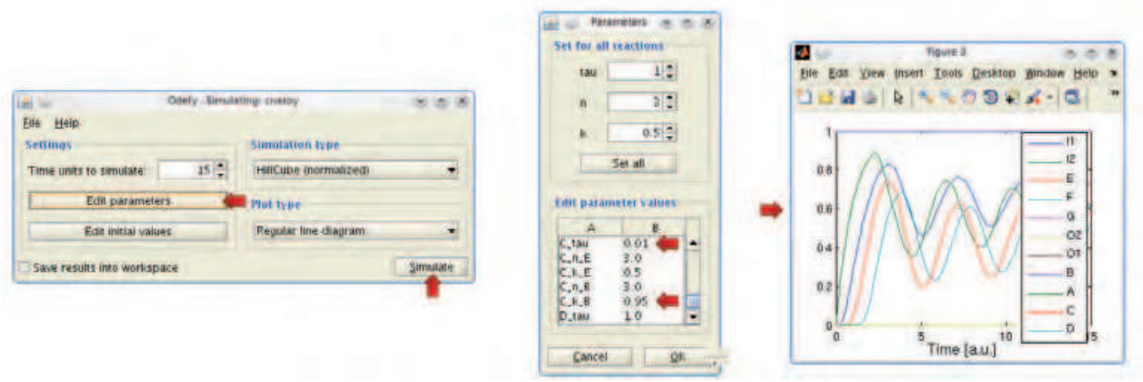
Note that the simulation runs with a set of default parameters for the regulatory interactions: n=3, k=0.5, tau=1. Similarly to the Boolean variant, we observe that all factors are successively activated except for A, which in the continuous version generates a smooth expression pulse lasting around 10 time steps. We also get quantitative insights now, since A does not go up to a full expression of 1.0, but reaches a maximum of only 0.8 before being deactivated. Next, we simulate the oscillatory scenario where both input species are present:



Again, the simulation trajectories show oscillations of the central model factors A, B, C, D and subsequently O1. Note that - in contrast to the Boolean version - the oscillations here display a specific frequency and amplitude. As will be seen in the next section, such quantitative features of the system are heavily dependent on the actual parameters chosen.

4.4 Adjusting the system parameters

As described at the beginning of this chapter, the ODE-converted version of our Boolean networks contain different parameters that control how strong and sensitive each regulatory interaction reacts, and how quick each species in the system responds to regulatory changes. In the following, we will exemplarily change some of the parameters in the oscillatory toy model scenario (the following GUI steps assume you already have performed the quantitative simulations from the previous sections):



In this example, we changed two system parameters: (i) the tau parameter of C was set to a very small value, rendering C very responsive to regulatory changes, (ii) the k threshold parameter from B towards E is set to 0.95, and thus the activation of E by B is only constituted for very high values of B. The resulting simulation still shows the expected oscillatory behavior, but the amplitude, frequency and synchronicity of the recurring patterns are altered in comparison to the previous variants. This is an example for a behavior that could not have been investigated by using pure Boolean models alone, but actually required the incorporation of a quantitative modeling approach.

5. The genetic toggle switch: Advanced model input and analysis techniques

While the last section focused on achieving quick results using the Odefy graphical user interface, we now focus on actual MATLAB programming. This provides far more power and flexibility during analysis than the fixed set of options implemented in a GUI. Furthermore, we now focus on a real biological system, namely the mutual inhibition of two genes (Figure 6). Intuitively, only one of the two antagonistic factors can be fully active at any given time. This simple wiring thus provides an elegant way for a cell to robustly decide between two different states. Consequently, mutual inhibition is a frequently found regulatory motif in cell differentiation processes. For example, the differentiation of the erythroid and myeloid lineages in hematopoiesis, that is the production of blood cells in higher organisms, is governed by the two transcription factors PU.1 and GATA-1, which are known to repress each other's expression (Cantor & Orkin, 2001). Once the cell has decided to become an erythroid cell, the myeloid program is blocked, and vice versa.

The switch model will be implemented in MATLAB by specifying the regulatory logic between the two genes as sets of Boolean rules and subsequent automatic conversion into a set of ODEs. The resulting model state space is analyzed for the discrete as well as the continuous case (for the latter one we use the common phase-plane visualization technique). We particularly investigate how different parameters affect the multistationarity of the system, and whether the system obtains distinct behaviors when combining regulatory inputs either with an AND or an OR gate.

5.1 Model definition

We have already seen that defining a Boolean model from the MATLAB command line is straightforward, since we can directly enter Boolean equations into the code. We will generate

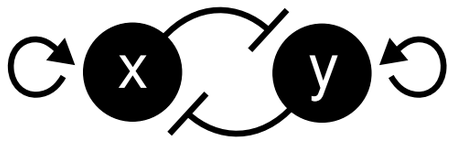


Fig. 6. Mutual inhibition and self-activation between two transcription factors.

two versions of the mutual switch model, one with an AND gate combining self-activation and the inhibition, and one with an OR gate:

```
switchAND = ExpressionsToOdefy({'x = x && ~y', 'y = y && ~x'});
switchOR  = ExpressionsToOdefy({'x = x || ~y', 'y = y || ~x'});
```

Similar to the GUI variant, we could also define the model in a file (yEd or Boolean expressions text file) and load the models from these files. While the definition directly within the code allows for rapid model alteration and prototypic analyses, the saving of the model in a file is the more convenient variant once model generation is finished.

5.2 Simulations from the command line

We want again to perform both Boolean and continuous simulations, but this time we control the entire computation from the MATLAB command line. First, we need to generate a simulation structure that holds all information required for the simulation, like initial states, simulation type and parameters (if applicable):

```
simstruct = CreateSimstruct(switchAND);
```

Within this simulation structure, we define a Boolean simulation for 5 time steps with asynchronous updating in random order (cf. section 2.1), starting from an initial value of x=1 and y=1:

```
simstruct.timeto = 5;
simstruct.type = 'boolrandom';
simstruct.initial = [1 1];
```

The actual simulation is now performed by calling the `OdefySimulation` function:

```
y = OdefySimulation(simstruct);
```

resulting, for example, in:

```
y =

     1     1     1     1     1
     1     0     0     0     0
```

While this result might not look to be very exciting, it actually reflects the main functionality of this regulatory network. The system falls into one of two follow-up states and stably stays within this state (→ a steady state). The player being expressed at the end of the simulation is randomly determined here, another simulation might result in this trajectory:

```
y =

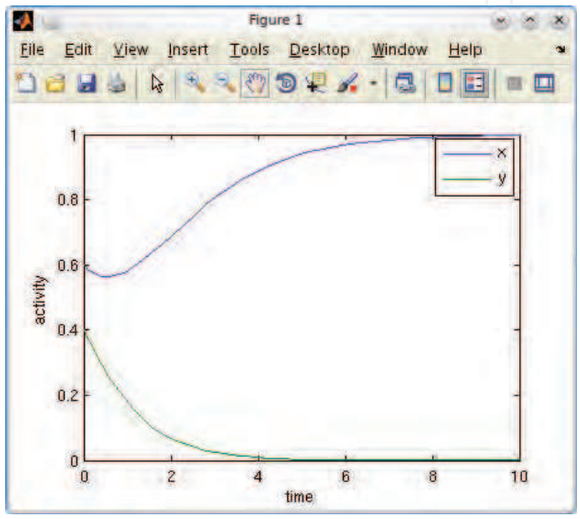
     1     0     0     0     0
     1     1     1     1     1
```

Obviously, this very sharp switching is an effect of the Boolean discretization. For comparison, we will now create a continuous simulation of the same system:

```
simstruct.timeto = 10;
simstruct.type='hillcubenorm';
simstruct.initial = [0.6 0.4];
[t y] = OdefySimulation(simstruct);
```

We employed the normalized HillCube variant with 10 simulated time steps. Note that we could now use real-valued initial values instead of just 0 and 1. The simulated trajectory looks like this:

```
plot(t,y)
legend(switchAND.species);
xlabel('time');
ylabel('activity');
```



We observe a similar decision effect as for the Boolean variant, but this time in a fully quantitative fashion. Although both factors have similar activity values at the beginning of the simulation, the small excess of X is sufficient to drive the system to a steady state where X is present and Y is not. With reversed initial values, X would have gone to 0 and Y would have been fully expressed.

5.3 Exploring the Boolean state space

In the previous sections we learned how Boolean and continuous simulations of a regulatory model can be interpreted. However, it is important to understand that such simulations merely represents single trajectories through the space of possible spaces, and do not reflect the full capabilities of the system. Therefore, it is often desirable to calculate the full set of possible trajectories of the system, the so-called *state-transition graph* (STG) in the case of a discrete model. We will now learn how to calculate the Boolean steady states of a given model along with its STG using Odefy. The primary calculation consists of a single call:

```
[s g] = BooleanStates(switchAND);
```

The variable `s` now contains the set of steady states of this system where as the STG is represented a sparse matrix in g. Steady states are encoded as decimal representations of their Boolean counterparts and can be conveniently displayed using the `PrettyPrintStates` function:

```
>> PrettyPrintStates(switchAND,s)
x        0 1 0
y        0 0 1
3 states
```

We see that the system has three steady states which are intuitively explainable. If one of the factors is on, the activation of the respective other factor is prohibited, so the state is stable (second and third column). Furthermore, if no player is active then the system is dead, which also represents a stable state (first column). Instead of `PrettyPrintStates` you can also use the `StateMatrix` function which stores the same results in a matrix variable for further working steps:

```
>> m = StateMatrix(switchAND,s)

m =

     0     1     0
     0     0     1
```

The variable g contains the STG encoded as a sparse adjacency matrix of states, which can be readably displayed using the `PrettyPrintSTGraph` function:

```
>> PrettyPrintSTGraph(switchAND,g)
11 => 10
11 => 01
```

That is, from the state where both factors are active, either one of the two exclusive steady states can be reached. No further state transitions are possible in this system. If we repeat the procedure of `BooleanStates` calculation and printing of steady states and STG for the `switchOR` variant, we get the result displayed in Figure 7. Both variants are capable of switch-like decisions that end in a certain steady state. Whereas in the AND variant the 00 state is steady, the same holds true for the 11 state in the OR variant. At this point, we could compare these observations to results from a real biological system, that is evaluating whether the system switches from an activated or inactivated basal state, and thus select one of the two variants as "closer" to biological reality.

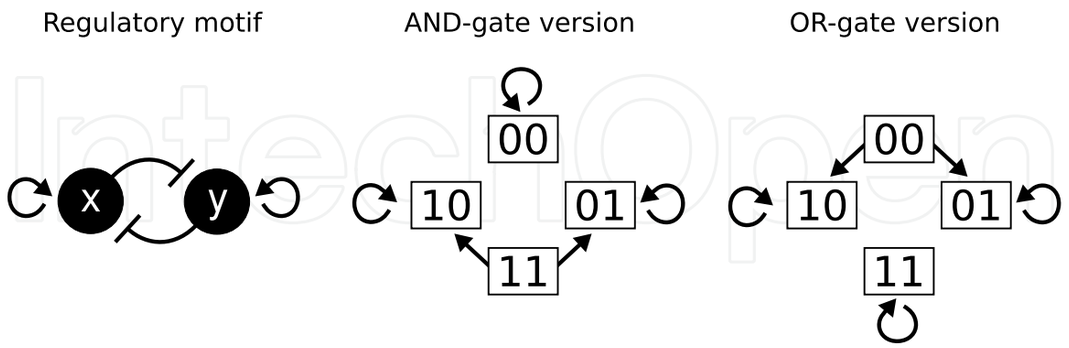


Fig. 7. State-transition graphs for the AND and OR variants of the mutual inhibition motif. Note that states without transitions going towards other states are the steady states of the system.

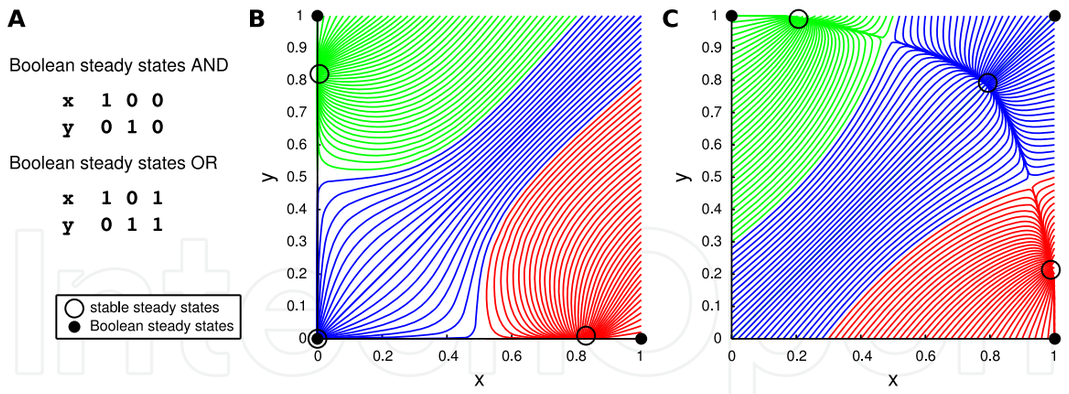


Fig. 8. **A** Boolean steady states of the OR and AND version of the mutual inhibitory switch model. **B,C** Phase planes visualizing the attractor landscapes of the AND and OR variants, respectively. The plots display trajectories of both dynamical systems from various initial concentrations. Trajectories with the same color fall into the same stable steady state. Both systems comprise three stable continuous steady states, each of which belongs to one Boolean steady state. Adapted from Krumsiek et al. (2010)

5.4 Exploring the continuous state space

Analogously to the Boolean state space described above, it is oftentimes desirable to investigate the behavior of the whole system for various internal states rather than concentrating on a single trajectory through the system. Since in the continuous case the system does not consist of a finite set of discrete states, we need a complementary approach to the state transition graphs introduced above. One possibility is the simulation of the continuous system from a variety of initial values and subsequent visualization in a two-dimensional phase plane (cf. Vries et al. (2006)):

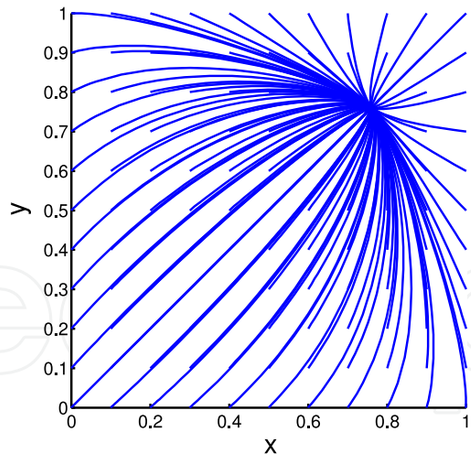
```
simstruct = CreateSimstruct(switchAND);
figure;
OdefyPhasePlane(simstruct, 1, 0:0.1:1, 2, 0:0.1:1);
```

This code produces the phase plane plot displayed in Figure 8B. Depending on the initial values, the system falls into one of three stable steady states, where either one of the two factors is active while the other one is turned off, or where both players are inactive. Importantly, the three steady states are qualitatively identical to the three Boolean steady states (again shown in 8A). If we think of these trajectories as possible state trajectories in a living cell, this phase plane could describe for which expression levels of the two transcription factors the system will turn into either on of the two opposing differentiation lineages. Furthermore, by observing if in the third state real cells rather have both factors active or inactive, we could determine whether the AND or the OR variant is a more suitable model of the underlying system.

We now change the Hill exponent n in all regulatory functions from the standard value of 3 to 1, and recalculate the phase-plane for the OR version:

```
simstruct = CreateSimstruct(switchOR);
simstruct = SetParameters(simstruct, [], [], 'n', 1);
figure;
OdefyPhasePlane(simstruct, 1, 0:0.1:1, 2, 0:0.1:1);
```

producing the following phase plane plot:



Interestingly, with this parameter configuration the system is not able to constitute a multistable behavior anymore. All trajectories fall into a single, central steady state with medium expression of both factors, regardless of the actual initial values of the simulation. This result is in line with findings from Glass & Kauffman (1973), who showed the requirement of cooperativity ($n \geq 2$) in order to generate multistationarity. Again, by comparing the system behavior with the real biological system we gain insights into the possibly correct parameter ranges. For our example here, since we assume stem cells to be able to obtain multistationarity, an n value below 2 seems rather unlikely.

5.5 Advanced command line usage: simulations using MATLAB's numerical ODE solvers

The continuous simulations shown above used Odefy's internal `OdefySimulation` function. However, in order to get full control of our ODE simulations the usage of MATLAB ODE .m files is desirable. We can generate such script files using the `SaveMatlabODE` function:

```
SaveMatlabODE(switchAND, 'myode.m', 'hillcubenorm');
rehash;
```

Note that `rehash` might be required so that the following code immediately finds the newly created function. The newly created file `myode.m` contains an ODE compatible with MATLAB's numerical solving functions. Next we set the initial values and change some parameters:

```
initial = zeros(2,1);
initial = SetInitialValue(initial, switchAND, 'x', 0.6);
initial = SetInitialValue(initial, switchAND, 'y', 0.4);

params = DefaultParameters(switchAND);
params = SetParameters(params,switchAND, [], [], 'n', 1);
```

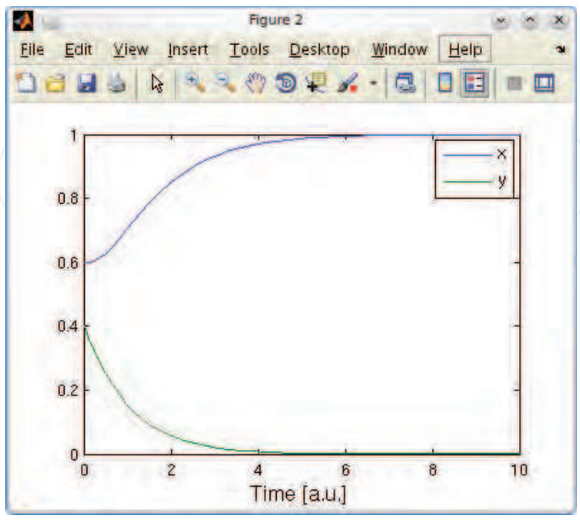
The `SetInitialValue` and `SetParameters` function can not only work on a simulation structure, but can also be used to edit raw value and parameter matrices directly. Finally, we run the simulation by calling:

```
paramvec = ParameterVector(switchAND,params);
time = 10;
r = ode15s(@(t,y)myode(t,y,paramvec), [0 time], initial);
```

For further information on the result variable `r`, we refer the reader to the documentation of `ode15s`. Odefy's `Visualize` method facilitates plot generation by taking care of drawing and labeling:

```
Visualize(r.x,r.y,switchAND.species);
```

resulting in the following trajectories, which we have already analyzed several times throughout this example:



6. The differentiation of mid- and hindbrain: automatic model selection

A common problem in the modeling of biological systems is the existence of a plethora of possible models that could explain the observed behavior. Therefore, methods for the automatic evaluation of features on a whole series of models are often required. In our third example of dynamic modeling using Odefy we investigate a multicellular system from developmental biology. During vertebrate development, the differentiation of mid- and hindbrain is determined by several transcription and secreted factors, which are expressed in a well-defined spatial pattern (Prakash & Wurst, 2004), the mid-hindbrain boundary (MHB, see Figure 9, left). While transcription factors control the regulation of genes within the same cell, secreted factors are transported through the cell membrane in order to induce signaling cascades in surrounding cells. The gene expression pattern is again maintained by a tightly regulated regulatory network between the respective factors (Wittmann et al., 2009b). We will here focus on four major factors from the MHB system: the transcription factors Otx2 and Gbx2, as well as the secreted proteins Fgf8 and Wnt1.
From the technical point-of-view, we will learn how to create a whole ensemble of different regulatory models, and subsequently how to iterate over all models in order to check whether each regulatory wiring is capable of maintaining the sharp expression patterns at the MHB.

6.1 Modeling a multi-compartment system using Odefy

A substantial difference to the models we worked with in previous sections of this chapter is the presence of multiple, linearly arranged cells in the modeled biological system (recall Figure 9). Each of these cells contains the identical regulatory machinery which needs to be connected and replicated as visualized in Figure 10. Note that this regulatory wiring corresponds to the results published in Wittmann et al. (2009b); below we will discuss the existence of further compatible models. The transcription factors Otx2 and Gbx2 inhibit each other's expression and control the expression of the secreted factors Fgf8 and Wnt1. The latter

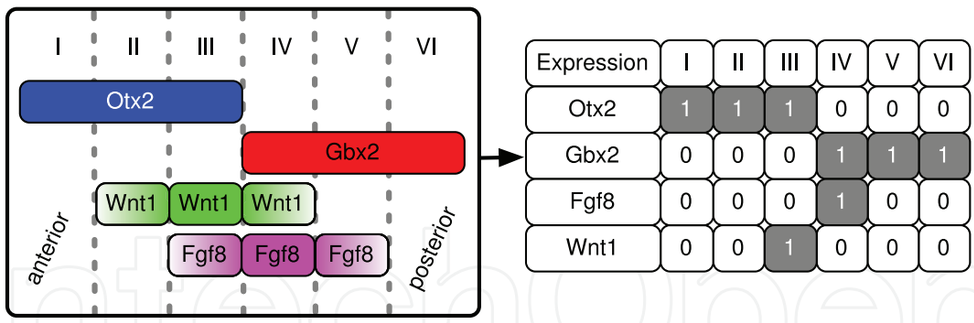


Fig. 9. Expression patterns at the mid-hindbrain boundary. While the anterior part of the developing brain is dominated by Otx2 expression and Wnt1 signaling at the boundary, the posterior part shows Gbx2 expression and Fgf8 signaling. Note that in the left panel fading colors indicate secreted factors that do not translate into the discretized expression pattern on the right. Adapted from Krumsiek et al. (2010)

ones in turn enhance each others activity in the neighboring cells, simulating the secretion and diffusion of these proteins in the multicellular context. For our analysis, we will focus on only 6 "cells" – which could also represent a whole region during development at the MHB – linearly arranged next to each other.

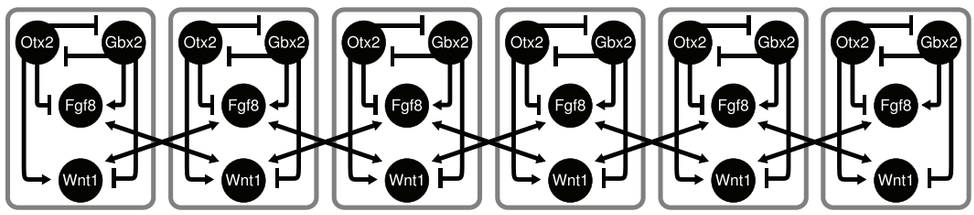


Fig. 10. Six-compartment model representing the different areas of the developing brain. Each unit contains the same regulatory network, neighboring cells are connected via the secreted protein Fgf8 and Wnt1.

In Odefy, we first need to define the core model, again using simple Boolean formulas for the representation of the regulatory wiring:

```
mhb = ExpressionsToOdefy({'Otx2=~Gbx2','Gbx2=~Otx2',...
  'Fgf8=~Otx2&&Gbx2&&Wnt1','Wnt1=~Gbx2&&Otx2&&Fgf8'});
```

Now, in order to automatically generate a connected six cell system, we make use of the Odefy `MultiModel` function:

```
multiMHB=MultiModel(mhb, [3 4], 6);
```

From the regulatory model `single` we generate 6 cells, whereas the third and fourth factors of the system are considered to be connected between neighboring cells. The variable `multiMHB` now contains the complete multi-cellular model comprising of a total of 24 factors:

```
multiMHB =

    tables: [1x24 struct]
      name: 'odefymodel_x_6'
   species: {24x1 cell}
```

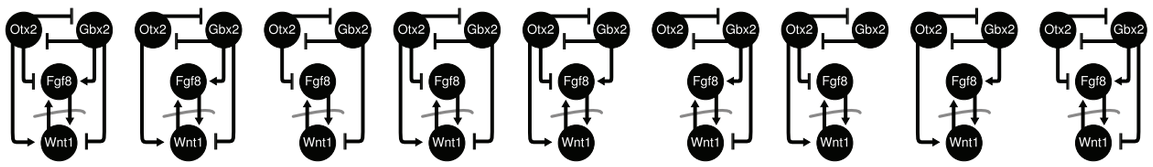


Fig. 11. All network variants known to give rise to a stable MHB boundary. For all networks we observe a mutual inhibition of Otx2 and Gbx2 and have antagonistic effects of these two factors on Fgf8 and Wnt1 expression. Moreover, we find that Fgf8 and Wnt1 require each other for their stable maintenance. Adapted from Krumsiek et al. (2010)

6.2 Automatic model selection procedure

In the following we will assemble a set over 100 distinct models between the four factors in our MHB system. We will have nine variants in total which indeed give rise to the correct behavior and are compatible to biological reality, and 100 randomly assembled networks which will obviously fail to produce a stable MHB. The following networks are the nine "positive" variants, cf. Krumsiek et al. (2010):

```
eqs = {};
eqs{end+1} = {'Otx2=~Gbx2','Gbx2=~Otx2','Fgf8=~Otx2&&Gbx2&&Wnt1',
    'Wnt1=~Gbx2&&Otx2&&Fgf8'};
eqs{end+1} = {'Otx2=~Gbx2','Gbx2=~Otx2','Fgf8=Gbx2&&Wnt1',
    'Wnt1=~Gbx2&&Otx2&&Fgf8'};
eqs{end+1} = {'Otx2=~Gbx2','Gbx2=~Otx2','Fgf8=~Otx2&&Gbx2&&Wnt1',
    'Wnt1=~Gbx2&&Fgf8'};
eqs{end+1} = {'Otx2=~Gbx2','Gbx2=~Otx2','Fgf8=~Otx2&&Wnt1',
    'Wnt1=~Gbx2&&Otx2&&Fgf8'};
eqs{end+1} = {'Otx2=~Gbx2','Gbx2=~Otx2','Fgf8=~Otx2&&Gbx2&&Wnt1',
    'Wnt1=Otx2&&Fgf8'};
eqs{end+1} = {'Otx2=~Gbx2','Gbx2=~Otx2','Fgf8=Gbx2&&Wnt1',
    'Wnt1=~Gbx2&&Fgf8'};
eqs{end+1} = {'Otx2=~Gbx2','Gbx2=~Otx2','Fgf8=~Otx2&&Wnt1',
    'Wnt1=Otx2&&Fgf8'};
eqs{end+1} = {'Otx2=~Gbx2','Gbx2=~Otx2','Fgf8=Gbx2&&Wnt1',
    'Wnt1=Otx2&&Fgf8'};
eqs{end+1} = {'Otx2=~Gbx2','Gbx2=~Otx2','Fgf8=~Otx2&&Wnt1',
    'Wnt1=~Gbx2&&Fgf8'};
```

The initial network we discussed in Figure 10 is the first one in this list, while all other networks represent subsets of the first one (Figure 11). Note that for now we only create single-compartment variants, the `MultiModel` function comes into play later on. Next, we need to generate actual Boolean models from these equations:

```
models={};
for i=1:numel(eqs)
    models{i} = ExpressionsToOdefy(eqs{i});
end
```

Next, we add a thousand randomly generated networks by using the `GraphToOdefy` function. This function takes the adjacency matrix of a regulatory network, interpreting 1 as activatory, -1 as inhibitory and 0 as no influence, and automatically generates an Odefy model structure:

```
for i=1:100
    models{end+1} =  GraphToOdefy(randi(3,4,4)-2);
end
```

The expression `randi(3,4,4)-2` creates a 4x4 matrix of values between -1 and 1. Note that if not explicitly specified, Odefy employs a standard logic to combine multiple inputs, where a player will be active whenever at least one activator and no inhibitors are present. Our `models` cell array now contains a total of 109 Boolean models, each of which we will test for its capability to create the MHB expression pattern. The general idea is to first convert each model to a multicompartment variant, and then let an ODE simulation run from the known stable MHB expression pattern in order to check whether the system departs from this required state. First, we need to define an initial state corresponding to the stable expression pattern from Figure 9:

```
init = [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 1 1 0 0 0 0];
```

Next, we iterate over all networks and perform the actual testing:

```
for i=1:numel(models)
    multi = MultiModel(models{i}, [3 4], 6);
    simstruct = CreateSimstruct(multi);
    simstruct.initial = knownstate;
    simstruct.type = 'hillcubenorm';
    [t,y] = OdefySimulation(simstruct, 0);
    if all(y(end,:)>0.5 == knownstate)
        fprintf('Valid: Model %d\n', i);
    end
end
```

Note the usage of `CreateSimstruct` and `OdefySimulation` to create a continuous ODE simulation of the converted Boolean model, as previously described in this chapter. The final validation statement `if all(y(end,:)>0.5 == knownstate)` determines whether each player still fits to the known MHB expression state, considering each player above a value of 0.5 to be active. Be aware that the execution of the model selection code might take a few minutes, depending on your machine. Since it is very unlikely that any of the randomly generated models is actually capable of obtaining the desired behavior, the final command line result should look like this:

```
Valid: Model 1
Valid: Model 2
Valid: Model 3
Valid: Model 4
Valid: Model 5
Valid: Model 6
Valid: Model 7
Valid: Model 8
Valid: Model 9
```

Taken together, we demonstrated how to automatically test for a specific feature in a set of models. For illustration purposes and in order to actually get a positive result here, we added a set of models known to give rise to the desired behavior.

7. A large-scale model of T-cell signaling: connecting Odefy to the SB toolbox

In our final example we focus on a model of T-cell activation processes, which play a pivotal role in the immune system. The model employed here has been previously described in the literature and consists of 40 factors and 55 pairwise regulatory interactions (Wittmann et al., 2009a). We will demonstrate how to convert the Boolean model to its ODE version and export

the result to the popular MATLAB Systems Biology toolbox[4]. From within this toolbox we can then conveniently perform simulations, steady state analysis as well as parameter sensitivity analysis. Furthermore, we will see how the compilation of an SB toolbox model to a .mex file MATLAB function dramatically increases the simulation speed of ODE systems.

7.1 The model

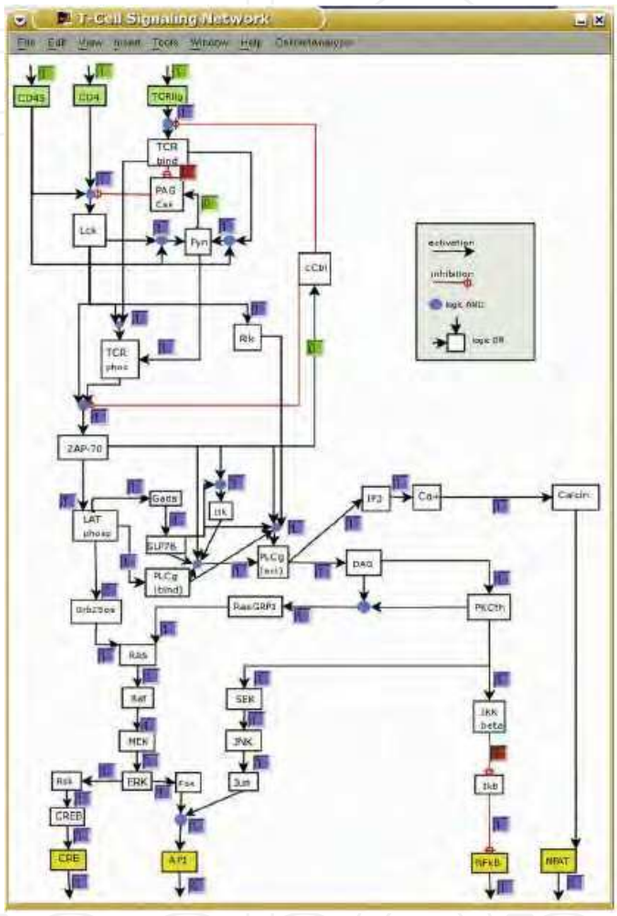


Fig. 12. Logical model of T-cell activation. The model contains a total of 40 factors and 49 regulatory interactions, with three input species - resembling T-cell receptors - and four output species - the activated transcription factors. Screenshot from CellNetAnalyzer (Klamt et al., 2006)

T-cells are part of the lymphoid immune system in higher eukaryotes. When foreign antigens, like bacterial cell surface markers, bind to certain receptors these cells, signaling cascades are triggered within the T-cell triggering the expression of several transcription factors in the nucleus. Ultimately, this leads to the initiation of a specific immune response aimed at eliminating the targeted foreign antigens (Klamt et al., 2006). The logical structure of the T-cell signaling model is shown in Figure 12. There are three inputs to the system: the T-cell receptor TCR, the coreceptor CD4 and an input for CD45; as well as four outputs:

[4] `http://www.sbtoolbox2.org/`

the transcription factors CRE, AP1, NFkB and NFAT. In total, the model comprises of 40 factors with 49 regulatory interactions. We will not provide a list of all Boolean formulas in this system here. The model can either be downloaded from the Odefy materials page[5], or obtained along with the CellNetAnalyzer toolbox[6]. In the following, we assume the Odefy model variable `tcell` to be existent in the current MATLAB workspace:

```
>> load tcell.mat
>> tcell

tcell =

    species: {1x40 cell}
     tables: [1x40 struct]
       name: 'Tcellsmall'
```

7.2 Exporting the ODE version to SB toolbox

At this point we require a working copy of the SBTOOLBOX2 package which can be freely obtained from the web[7]. We translate the Boolean T-cell model into its HillCube ODE counterpart and convert the resulting differential equation system into an SB toolbox internal representation:

```
sbmodel = CreateSBToolboxModel(tcell, 'hillcube', 1)
```

The third argument indicates whether to directly create an `SBmodel` object, or whether to generate an internal MATLAB structure representation of the model. Both variants should be compatible with the other SB toolbox functions. The result should now look like this:

```
SBmodel
        =======
        Name: Tcellsmall
        Number States:                  40
        Number Variables:               0
        Number Parameters:              147
        Number Reactions:               0
        Number Functions:               0
```

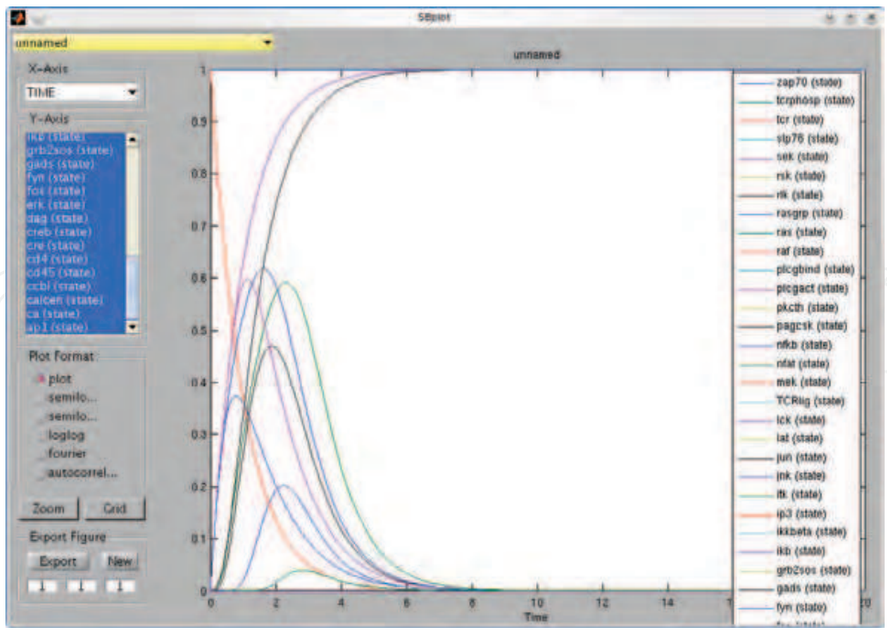
We successfully created a HillCube ODE version of the Boolean T-cell model in SB toolbox. This allows us to make use of the full functionality of this toolbox, like regular simulations and steady state calculations for example:

```
init=zeros(numel(tcell.species),1);
init(strcmp(SBstates(sbmodel),'tcr'))=1;
init(strcmp(SBstates(sbmodel),'cd4'))=1;
init(strcmp(SBstates(sbmodel),'cd45'))=1;
sbmodel = SBinitialconditions(sbmodel,init);

SBsimulate(sbmodel);
ss=SBsteadystate(sbmodel);
```

We first set the initial values of the input factors TCR, CD4 and CD45 to 1 and then call the `SBsteadystate` function. The `ss` vector now contains steady states for all 40 factors in the system given the current initial states and parameters. `SBsimulate` will open the interactive simulation dialog of SB toolbox:

[5] http://hmgu.de/cmb/odefymaterials
[6] http://www.mpi-magdeburg.mpg.de/projects/cna/cna.html
[7] http://www.sbtoolbox2.org/



In addition to these simple functionalities we could also have achieved with the Odefy toolbox, we could now apply advanced dynamic model analysis techniques implemented in the SB toolbox. This includes, amongst others, local and global parameter sensitivity analysis (Zhang et al., 2010), bifurcation analysis (Waldherr et al., 2007) and parameter fitting methods (Lai et al., 2009).

7.3 Compiling the model to .mex format – fast model simulations

As our final example of connecting Odefy with the SB Toolbox, we will compile the T-cell model into the MATLAB .mex format. For this purpose we also need a copy of the SBPD Toolbox[8] in addition to the regulatory SB Toolbox. The compilation is performed in a single function call as follows:

```
SBPDmakeMEXmodel(sbmodel);
```

which will create a file called `Tcellsmall.mexa64` (the file extension might differ depending on the operating system and architecture) in the current working directory. Since the compiled SB toolbox functions employ a special numeric ODE integrator optimized for compiled models, the compiled version outperforms the regular simulation by far. To verify this, we let the system run from the initial state defined above and measure the elapsed time for the calculation:

```
tic;
for i=1:10
    r = SBsimulate(sbmodel,0:0.01:20);
end
toc;
```

yielding

```
Elapsed time is 13.585409 seconds.
```

on a Intel(R) Core(TM)2 Duo CPU P9700, 2.8 GHz. In contrast, the compiled model simulation is substantially faster:

[8] can also be obtained from `http://www.sbtoolbox2.org/`

```
tic;
for i=1:10
    r=Tcellsmall(0:0.01:20, init);
end
toc;
```

producing

```
Elapsed time is 0.100033 seconds.
```

That is, for the T-cell model the compiled version runs approximately 140 times faster than a regular simulation employing MATLAB built-in numerical ODE solvers. This feature can be particularly useful when a large number of simulations is required, e.g. for parameter optimization by fitting the simulated curves to measured experimental data.

8. Conclusion

In this tutorial we learned how to use the Odefy toolbox to model and analyze molecular biological systems. Boolean models can be readily constructed from qualitative literature information, but obviously have severe limitations due to the abstraction of activity values to zero and one. We presented an automatic approach to convert Boolean models into systems of ordinary differential equations. Using the Odefy toolbox, we worked through various hands-on examples explaining the creation of Boolean models, the automatic conversion to systems of ODEs and several analysis approaches for the resulting models. In particular, we explained the concepts of steady states (i.e. states that do not change over time), update policies, state spaces, phase planes and systems parameters. Furthermore, we worked with several real biological systems involved in stem cell differentiation, immune system response and embryonal tissue formation. The Odefy toolbox is regularly maintained, open-source and free of charge. Therefore it is a good starting point in the analysis of ODE-converted Boolean models as it can be easily extended and adjusted to specific needs, as well as connected to popular analysis tools like the Systems Biology Toolbox.

9. References

Albert, R. & Othmer, H. G. (2003). The topology of the regulatory interactions predicts the expression pattern of the segment polarity genes in drosophila melanogaster., *J Theor Biol* 223(1): 1–18.

Alon, U. (2006). *An Introduction to Systems Biology: Design Principles of Biological Circuits (Chapman & Hall/Crc Mathematical and Computational Biology Series)*, Chapman & Hall/CRC.

Cantor, A. B. & Orkin, S. H. (2001). Hematopoietic development: a balancing act., *Curr Opin Genet Dev* 11(5): 513–519.
URL: *http://www.ncbi.nlm.nih.gov/pubmed/11532392*

Fauré, A., Naldi, A., Chaouiya, C. & Thieffry, D. (2006). Dynamical analysis of a generic boolean model for the control of the mammalian cell cycle., *Bioinformatics* 22(14): e124–e131.
URL: *http://bioinformatics.oxfordjournals.org/cgi/content/short/22/14/e124*

Glass, L. & Kauffman, S. A. (1973). The logical analysis of continuous, non-linear biochemical control networks., *J Theor Biol* 39(1): 103–129.

Kitano, H. (2002). Systems biology: a brief overview., *Science* 295(5560): 1662–1664.
URL: *http://dx.doi.org/10.1126/science.1069492*

Klamt, S., Saez-Rodriguez, J., Lindquist, J. A., Simeoni, L. & Gilles, E. D. (2006). A methodology for the structural and functional analysis of signaling and regulatory networks., *BMC Bioinformatics* 7: 56.
URL: *http://dx.doi.org/10.1186/1471-2105-7-56*

Klipp, E., Herwig, R., Kowald, A., Wierling, C. & Lehrach, H. (2005). *Systems Biology in Practice: Concepts, Implementation and Application*, 1 edn, Wiley-VCH.
URL: *http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/3527310789*

Krumsiek, J., Pölsterl, S., Wittmann, D. M. & Theis, F. J. (2010). Odefy–from discrete to continuous models., *BMC Bioinformatics* 11: 233.
URL: *http://dx.doi.org/10.1186/1471-2105-11-233*

Lai, X., Nikolov, S., Wolkenhauer, O. & Vera, J. (2009). A multi-level model accounting for the effects of jak2-stat5 signal modulation in erythropoiesis., *Comput Biol Chem* 33(4): 312–324.
URL: *http://dx.doi.org/10.1016/j.compbiolchem.2009.07.003*

Prakash, N. & Wurst, W. (2004). Specification of midbrain territory., *Cell Tissue Res* 318(1): 5–14.
URL: *http://dx.doi.org/10.1007/s00441-004-0955-x*

Samaga, R., Saez-Rodriguez, J., Alexopoulos, L. G., Sorger, P. K. & Klamt, S. (2009). The logic of egfr/erbb signaling: theoretical properties and analysis of high-throughput data., *PLoS Comput Biol* 5(8): e1000438.
URL: *http://dx.doi.org/10.1371/journal.pcbi.1000438*

Schmidt, H. & Jirstrand, M. (2006). Systems biology toolbox for matlab: a computational platform for research in systems biology., *Bioinformatics* 22(4): 514–515.
URL: *http://dx.doi.org/10.1093/bioinformatics/bti799*

Thomas, R. (1991). Regulatory networks seen as asynchronous automata: A logical description, *Journal of Theoretical Biology* 153(1): 1 – 23.

Tyson, J. J., Csikasz-Nagy, A. & Novak, B. (2002). The dynamics of cell cycle regulation., *Bioessays* 24(12): 1095–1109.
URL: *http://dx.doi.org/10.1002/bies.10191*

Vries, G. d., Hillen, T., Lewis, M. & Schõnfisch, B. (2006). *A Course in Mathematical Biology: Quantitative Modeling with Mathematical and Computational (Monographs on Mathematical Modeling and Computation)*, SIAM.

Waldherr, S., Eissing, T., Chaves, M. & Allgöwer, F. (2007). Bistability preserving model reduction in apoptosis, *10th IFAC Comp. Appl. in Biotechn*, pp. 327–332.
URL: *http://arxiv.org/abs/q-bio/0702011*

Werner, E. (2007). All systems go, *Nature* 446(7135): 493–494.
URL: *http://www.nature.com/nature/journal/v446/n7135/full/446493a.html*

Wittmann, D. M., Blöchl, F., Trümbach, D., Wurst, W., Prakash, N. & Theis, F. J. (2009). Spatial analysis of expression patterns predicts genetic interactions at the mid-hindbrain boundary., *PLoS Comput Biol* 5(11): e1000569.
URL: *http://dx.doi.org/10.1371/journal.pcbi.1000569*

Wittmann, D. M., Krumsiek, J., Saez-Rodriguez, J., Lauffenburger, D. A., Klamt, S. & Theis, F. J. (2009). Transforming boolean models to continuous models: methodology and application to t-cell receptor signaling., *BMC Syst Biol* 3: 98.
URL: *http://dx.doi.org/10.1186/1752-0509-3-98*

Zhang, T., Wu, M., Chen, Q. & Sun, Z. (2010). Investigation into the regulation mechanisms of trail apoptosis pathway by mathematical modeling, *Acta Biochimica et Biophysica Sinica* 42(2): 98–108.
URL: *http://abbs.oxfordjournals.org/content/42/2/98.abstract*

IntechOpen

IntechOpen



Applications of MATLAB in Science and Engineering

Edited by Prof. Tadeusz Michalowski

ISBN 978-953-307-708-6
Hard cover, 510 pages
Publisher InTech
Published online 09, September, 2011
Published in print edition September, 2011

The book consists of 24 chapters illustrating a wide range of areas where MATLAB tools are applied. These areas include mathematics, physics, chemistry and chemical engineering, mechanical engineering, biological (molecular biology) and medical sciences, communication and control systems, digital signal, image and video processing, system modeling and simulation. Many interesting problems have been included throughout the book, and its contents will be beneficial for students and professionals in wide areas of interest.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Jan Krumsiek, Dominik M. Wittmann and Fabian J. Theis (2011). From Discrete to Continuous Gene Regulation Models – A Tutorial Using the Odefy Toolbox, Applications of MATLAB in Science and Engineering, Prof. Tadeusz Michalowski (Ed.), ISBN: 978-953-307-708-6, InTech, Available from: http://www.intechopen.com/books/applications-of-matlab-in-science-and-engineering/from-discrete-to-continuous-gene-regulation-models-a-tutorial-using-the-odefy-toolbox

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

© 2011 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike-3.0 License, which permits use, distribution and reproduction for non-commercial purposes, provided the original is properly cited and derivative works building on this content are distributed under the same license.

IntechOpen

IntechOpen