– Turing patterns are common but not robust–

Software Documentation

Natalie S. Scholes[*,1] David Schnoerr[*,1] Mark Isalan,[1,3] Michael Stumpf[†,1,2]

[1]Department of Life Sciences, Imperial College, London SW7 2AZ, UK.
[2]School of BioScience and School of Mathematics and Statistics, University of Melbourne, Melbourne, Australia.
[3]Imperial College Centre for Synthetic Biology, Imperial College London, London, SW7 2AZ, UK.

1 Overview

The here presented software package searches for all stable steady states of a given system numerically using the Matlab ODE solver suite. Subsequently these steady states are analyzed for possible Turing instabilities (see overview of workflow in Figure 1). The generated output files include information regarding the parameter values (intra- and extracellular/diffusion parameters) for which Turing pattern formation is expected (i.e., a Turing I instability is present), as well as each stable steady state that was found per intracellular parameter value combination. The workflow is not restricted to equations such as utilized in the main text (i.e. Hill functions), but can be adapted to a wide range of functions such as mass action kinetics or other non-linear equations.

The code is applicable to arbitrary numbers n of nodes/interacting species and arbitrary ODE functions. However, using the standard algorithm to search for stable steady states and instabilities, as demonstrated in the examples in Sections 4.1 and 4.2, can lead to large running times when analyzing systems with more than three nodes. In Section 4.5 we thus demonstrate how the analysis can be adapted for such cases.

This tutorial describes all code segments that are necessary to run the algorithm. Additionally, more detailed information can be found in the code comments and in the main text "Turing patterns are common but not robust".

2 System requirements

- Installed version of Matlab v2016a or later

3 Installation

- Copy all .m files into a single folder

4 Using the Software

This tutorial is structured into five sections. First we show how chemical reaction networks are defined in Section 4.1. Next, we show how to execute the software to run the Turing analysis in Section 4.2. In 4.3 we subsequently discuss the output data and its handling. Next, some additional ODE examples are provided in Section 4.4. Finally, additional information to apply the code to larger systems is given in Section 4.5.

*Authors N. S. and D. S. contributed equally.
†Corresponding author: m.stumpf@imperial.ac.uk

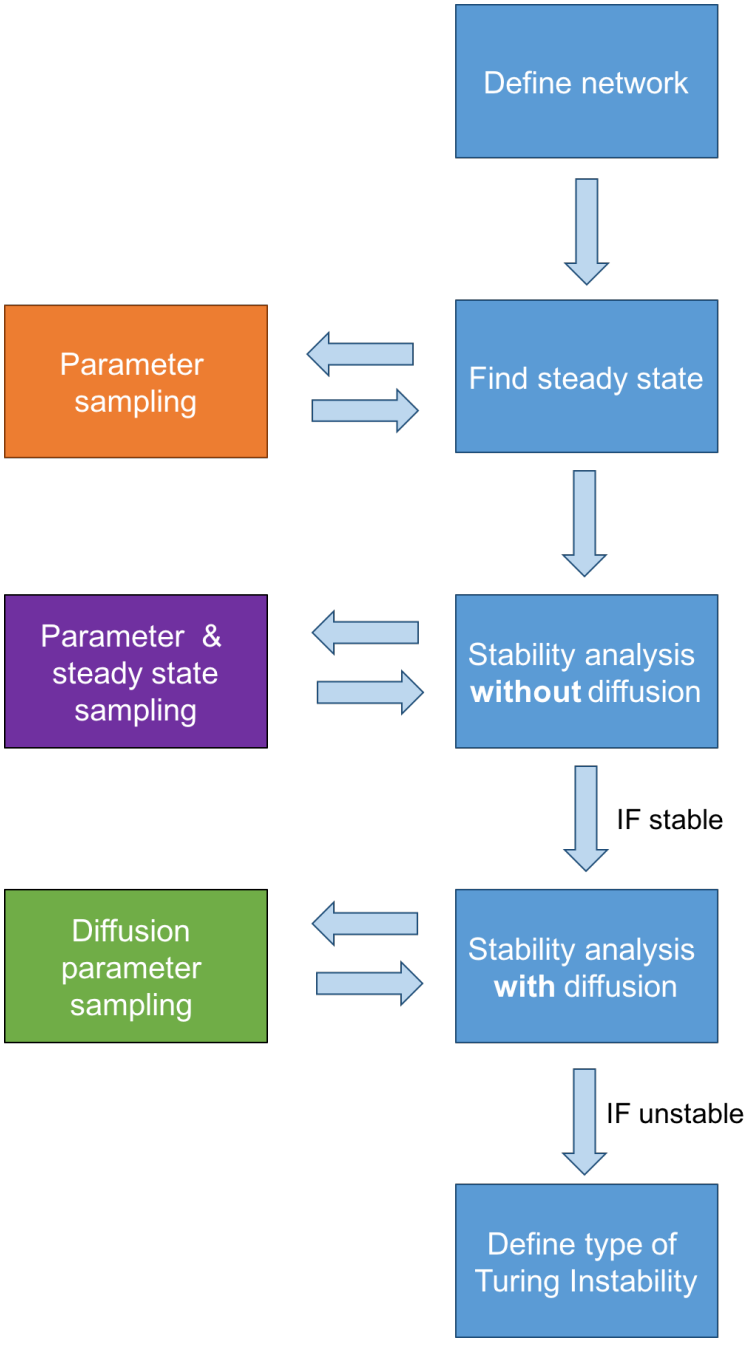


Figure 1: *Visualisation of the different steps of algorithm.*

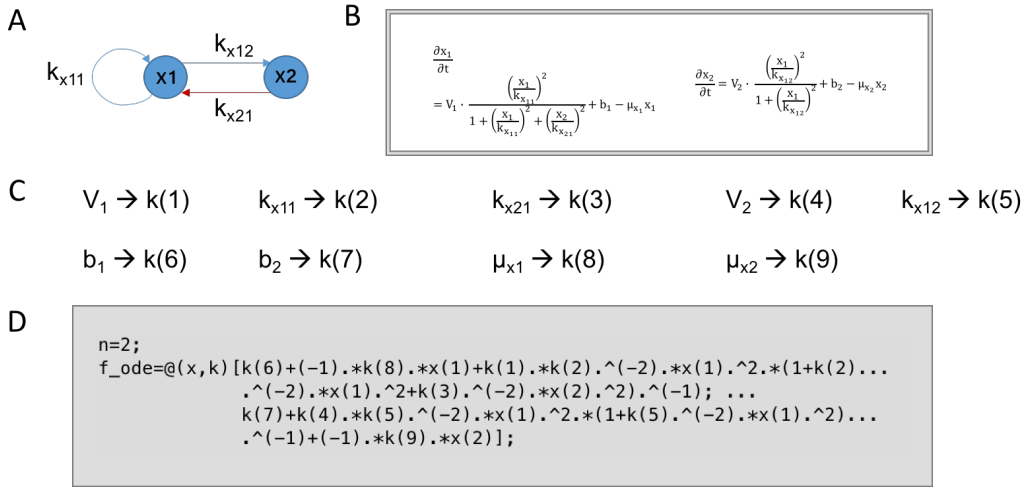


Figure 2: *A: Network representation of 2-node example network. B: ODE equation corresponding to the network shown in A. C: Transformation of parameters into vector representation for formulating the f_ode input function for the algorithm. D: Matlabe code for ODE equation f_ode of the 2-node network shown in A. In addition to the ODE equation of the system, we also need to define the node number n (n = 2;).*

4.1 Defining the system

First we need to define the system. This can be done as shown in the file SystemDefinition_example.m. The first two elements that need to be defined are the number of nodes of the system of interest (variable n) and the ode function *f_ode* of the system. Figure 2A depicts the network example we will exemplify in the following. The network consists of 2 nodes and thus we need to set $n = 2$. For the ODE equation, we chose to define the regulatory interactions as competitive Hill functions. These define the production terms of the species. Additionally, a basal expression (b) and a negative first order mass action term (defining the degradation) are assumed to be present for each node. The corresponding equations are shown in Figure 2B. To use the ODE function shown in Figure 2B as input for the algorithm, we re-write the ODE system, such that the concentrations of the species are specified as vector entries *x(1)* and *x(2)* and all kinetic and intracellular parameters are defined as vector entries *k(1),...,k(p)*. This results in the input function *f_ode*, which is shown in Figure 2D. Further ODE examples are shown in 4.4.

Since we are interested in systems in which some nodes diffuse to create Turing patterns, we next define how many and which nodes diffuse. In Figure 3, we show a general example for which $2, ..., n$ nodes diffuse and all possible combinations thereof are tested. *m_min* specifies the minimal amount of diffusing nodes. Here, it is set to 2. *m_max* defines the upper boundary of diffusing entities and is here set to be equal to the node number. This results in a range of $2 - 2$, i.e. both nodes diffusing is the only sampled setting in this scenario.

For three node systems this setting may obviously result in more possible combinations of diffusing nodes to be analyzed. If the settings of *m_min* and *m_max* remain 2 and n respectively, we would consider all possible diffuser combination where two or three nodes diffuse, i.e. *x(1)* and *x(2)*, *x(1)* and *x(3)*, *x(2)* and *x(3)*, or *x(1)*, *x(2)* and *x(3)*.

Which node diffuses is defined by the variable *binary_diffusor*. It can either be a matrix (as depicted in Figure 3) or a simple vector as shown in Figure 8D. For the former, each row vector of *binary_diffusor* defines which nodes diffuse and each column represents one possible combination of diffuser settings. 1's hereby indicate diffusing entities and 0's non-diffusing entities. For 2-node systems this would thus result in a matrix containing following entries: [0 0] (no species diffuse), [1 0] (*x(1)* diffuses), [0 1] (*x(2)* diffuses), [1 1] (both *x(1)* and *x(2)* diffuse). As we have already defined the number range of diffusing nodes by *m_min* and *m_max*, the algorithm will automatically only consider the last setting ([1 1]), to test for Turing instabilities. As an alternative to the matrix, *binary_diffusor* can be defined as a single vector, i.e. for the 2-node example with *x(1)* and *x(2)* diffusing, *binary_diffusor* would be [1 1].

Note: *m_min* and *m_max* always need to be defined.

```
m_min = 2;
m_max = n;
binary_diffusor = permn([0 1],n);
d_range = logspace(-3,3,7);
```

Figure 3: *Variables that need to be defined regarding the diffusion parameter sampling. m_ min represents the minimal number of diffusing nodes; m_ max represents the maximal number of diffusing nodes; binary_ diffusor defines which nodes diffuse: 1's respresent diffusion, 0's no diffusion. Here, binary_ diffusor samples all possible combinations of 0's and 1's and the algorithm automatically samples only those diffusion combinations that lie within the range defined by m_ min and m_ max. d_ range defines the range of the sampled diffusion parameter values.*

```
k_length = 10;
ks = logspace(-1,2,3);
for i = 1:k_length
    k_input{i} = ks;
end
k_grid = combvec(k_input{:});
```

Figure 4: *Variables that need to be defined for the sampling of the kinetic parameters. k_ length defines the number of parameters. ks is a vector that defines all values that should be sampled for each parameter. Here, these values equal 0.1,3.1 and 100. Within these values using the function combvec, we generate a parameter value grid (k_ grid) to sample all possible combinations of the defined parameter values.*

Next, a range of diffusion values needs to be specified using the variable *d_ range* as shown in Figure 3. In all cases the algorithm defines one node to diffuse with a value of 1 and all other diffusing nodes are sampled in the range of *d_ range*: here we sample a diffusion value range from 10^{-3}-10^{3}. This equals a 1000 fold slower or faster diffusion, respectively. The range is sampled on a logarithmic grid.

To sample different sets of parameters k we define the variable *k_ grid*. Its rows contain values for the parameters *k(1)...k(p)* and columns correspond to different parameter sets. For example, we can generate a matrix where every parameter is within the range of 0.1-100 on a logarithmically spaced grid using the code segment shown in Figure 4.

Note: Matlab returns an error message if the matrix *k_ grid* is chosen too large.

For estimating stable steady states the algorithm solves the ODE equations numerically and tests for sufficient convergence. To account for the possibility of multiple stable steady states, the algorithm solves the ODE for several different initial conditions defined by the matrix *c_ ini* (see

```
x_max = 20001;
int = 10000;
c_ini = permn(1:int:x_max,n);
```

Figure 5: *Variables defining the initial conditions of the ODE simulations. These vary from a value of* 1 *to a maximum value x_ max on a regular grid with spacing int. All possible combinations for these values for the different species are stored in c_ ini.*

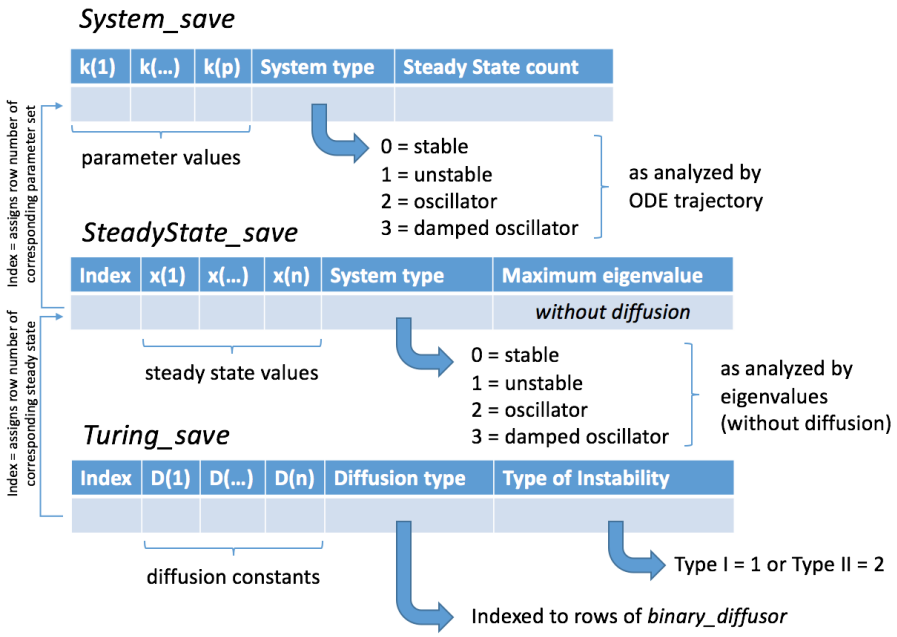


Figure 6: *Structure of output files generated by the code.*

Figure 5). Similar to *k_grid*, rows define initial condition values for nodes *x(1),...,x(n)* and columns denote different sets of initial species values. In Figure 5, we specify three initial conditions (1, 10,001 and 20,001) and simulate the system for all possible combinations with amounts to 3^n (n = node number) initial conditions per parameter set.

4.2 Executing the code

To define the system, the file SystemDefinition_example.m needs to be executed. This stores the definitions to the corresponding variables. Subsequently the numerical analysis can be evaluated, which is done by executing the file NetworkAnalysis.m. The latter compiles all functions that are required to perform the analysis of the system to numerically estimate stable steady states and subsequently test for Turing I or II instabilities. The runtime varies depending on the system size (number of nodes n), number of initial conditions (size of *c_ini*) and the number of parameter sets (size of *k_grid*). More details regarding specific parts of the algorithm can be found in the code's comments.

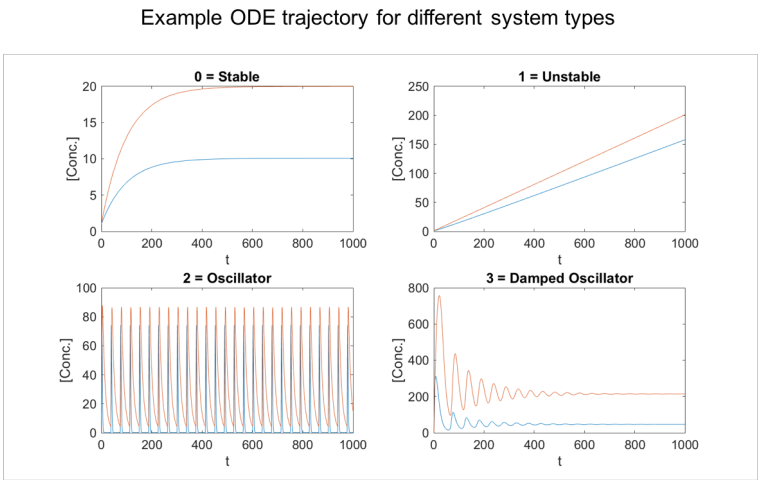
4.3 Output files

After running the algorithm a .mat file will be saved (the filename is specified by the variable *ID*, see file SystemDefinition_example.m). The .mat output file contains three matrices: *System_save*, *SteadyState_save*, *Turing_save*. The data in each of the matrices is structured as shown in Figure 6.

As can be seen in *System_save* each parameter set is assigned a system type. The system type refers to the behavior of the system during the ODE simulation and that of the eigenvalues from the stability analysis. Examples for each system type (stable, unstable, oscillatory and damped oscillatory) are shown in Figure 7. This classification is required to exclude non-converging and oscillatory systems, as these would not form Turing patterns. Additionally, damped oscillators require a refined method for finding the steady state solutions (see main text STAR Methods).

The matrix *Turing_save* specifies for which steady state(s) (and corresponding parameter set) a Turing instability has been found and of which type it is (I or II). Note that only type I instabilities are expected to lead to pattern formation with a finite wavelength and should thus be analyzed. For detailed information on the eigenvalue behavior and how to distinguish a Type I and type II Turing instability, we refer to the main text in "Turing patterns are common but not robust" Figure 1,2 and 4D.

A



B

Eigenvalue properties for the different system types

Stable: all real parts of the eigenvalues < 0

Unstable: at least one real part of the eigenvalue > 0

Oscillator: real part of eigenvalues = 0 & complex conjugated eigenvalues

Damped Oscillator: all real parts of the eigenvalues < 0 & complex conjugated eigenvalues

Figure 7: *Classification of ODE systems into different system types.*

Note: *Turing_save* contains all possible Turing Instabilities of type I and II.

4.4 ODE examples

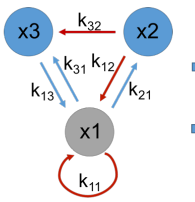
In this section, we illustrate further examples of possible ODE systems and how to analyze them using our software package. Figure 8A shows a 3-node network. Figures 8B and C show the corresponding ODEs for non-competitive or competitive regulations, respectively, using Hill functions as regulatory terms. Similar to the 2-node example above, we include a linear degradation term and a basal production rate for each node. Due to computational cost Vs and bs are here fixed to 100 and 0.1, respectively. For both systems the figures show the ODE equations and how each parameter is translated into the *f_ode* function. *f_ode* needs to be provided as part of the system definition code when running this example ($n = 3$, see Figure 2).

4.5 Running systems with more than three nodes

As pointed out in Section 1, larger number of nodes and/or parameters may lead to unfeasibly large running times. In this section we show how sampling can be reduced to achieve smaller running times, at the sacrifice of some accuracy. We outline three different ways to achieve this in the following. Let us consider a 5-node network ($n = 5$). If we chose the same combinations of initial conditions as for the 2- and 3-node examples studied in previous sections (i.e., three values for each node with all possible combinations), we would have to solve the ODEs for 243 initial conditions (3^5) for each parameter set. Compared to the 2- and 3-node systems where we only had 9 and 27 initial conditions, respectively (scaling 3^n), this increases running times by nearly 10 fold. Consequently, significant reduction can be achieved by using a less precise grid for the initial conditions consisting of two values per node rather than three. This would result in 32 initial conditions for a 5-node system instead of 243.

Note: A reduction of initial conditions increases the possibility of the algorithm not finding all stable steady states.

A



(B) Regulatory function: Hill, Non-Competitive

(C) Regulatory function: Hill, Competitive

(D) System definition

B Non-Competitive

$$\frac{\partial x_1}{\partial t} = 100 \cdot \frac{1}{1+\left(\frac{x_1}{k_{11}}\right)^2} \cdot \frac{1}{1+\left(\frac{x_2}{k_{12}}\right)^2} \cdot \frac{1}{1+\left(\frac{k_{13}}{x_3}\right)^2} + 0.1 - \mu_{x_1} x_1 \qquad \frac{\partial x_2}{\partial t} = 100 \cdot \frac{1}{1+\left(\frac{k_{21}}{x_1}\right)^2} + 0.1 - \mu_{x_2} x_2$$

$$\frac{\partial x_3}{\partial t} = 100 \cdot \frac{1}{1+\left(\frac{k_{31}}{x_1}\right)^2} \cdot \frac{1}{1+\left(\frac{x_1}{k_{32}}\right)^2} + 0.1 - \mu_{x_3} x_3$$

k_{11} → k(1) k_{12} → k(2) k_{13} → k(3) k_{21} → k(4) k_{31} → k(5)
k_{32} → k(6) μ_{x1} → k(7) μ_{x2} → k(8) μ_{x3} → k(9)

```
f_ode=@(x,k)[0.1+(-1).*k(7).*x(1)+100.*(1+k(1).^(-2).*x(1).^2).^(-1).*(1+k(2).^(-2).*x(2).^2).^(-1)
.*(1+k(3).^2.*x(3).^(-2)).^(-1); 0.1+100.*(1+k(4).^2.*x(1).^(-2)).^(-1)+(-1).*k(8).*x(2);
0.1+100.*(1+k(5).^2.*x(1).^(-2)).^(-1).*(1+k(6).^(-2).*x(2).^2).^(-1)+(-1).*k(9).*x(3)];
```

C Competitive

$$\frac{\partial x_1}{\partial t} = 100 \cdot \frac{\left(\frac{x_3}{k_{13}}\right)^2}{1+\left(\frac{x_3}{k_{13}}\right)^2+\left(\frac{x_1}{k_{11}}\right)^2+\left(\frac{x_2}{k_{12}}\right)^2} + 0.1 - \mu_{x_1} x_1$$

$$\frac{\partial x_3}{\partial t} = 100 \cdot \frac{\left(\frac{x_1}{k_{31}}\right)^2}{1+\left(\frac{x_1}{k_{31}}\right)^2+\left(\frac{x_2}{k_{32}}\right)^2} + 0.1 - \mu_{x_3} x_3$$

k_{13} → k(1) k_{11} → k(2) k_{12} → k(3) k_{21} → k(4) k_{31} → k(5)
k_{32} → k(6) μ_{x1} → k(7) μ_{x2} → k(8) μ_{x3} → k(9)

```
f_ode=@(x,k)[0.1+(-1).*k(7).*x(1)+100.*k(1).^(-2).*x(3).^2.*(1+k(2).^(-2).*x(1).^2+k(3).^(-2).*
x(2).^2+k(1).^(-2).*x(3).^2).^(-1);0.1+100.*k(4).^(-2).*x(1).^2.*(1+k(4).^(-2).*x(1).^2).^(-1)+
(-1).*k(8).*x(2);0.1+100.*k(5).^(-2).*x(1).^2.*(1+k(5).^(-2).*x(1).^2+k(6).^(-2).*x(2).^2).^(-1)+
(-1).*k(9).*x(3)];
```

D System definition: Diffusion settings

```
m_min = 2;
m_max = n;
binary_diffusor = [0 1 1];
d_range = logspace(-3,3,7);
```

Figure 8: *3-node example for different types of regulatory functions. A: Network topology of 3-node network ($n = 3$) to be analysed. B: For non-competitive Hill regulatory interactions the ODE is shown in the top box. To make the system compatible for the algorithm, the parameters are translated into a single vector (see middle) and the resulting f_ode can be used as an input for running the analysis script. C: as B, but with competitive Hill regulatory interactions. D: In this case the variables required for the diffusion sampling (see Figure 4), are defined in the range of $2-n$ $(= 2-3)$ diffusing entities. The binary_diffusor is here contrary to Figure 4, however, set as a single vector input, so that in this case we only sample the case in which x_2 and x_3 diffuse. Here, d_range defines the diffusion values of the second diffusing node, which would here be equal to x_3.*

Alternatively, one can reduce the overall number of analyzed parameter combinations defined in *k_ grid* (see Figure 3) by choosing a smaller or less densely sampled parameter value grid. Another possibility is to use parameter sets sampled randomly instead of using a regular grid. In Figure 9, for example, we limit the total number of computed ODE solutions to $limits = 1.45 \cdot 10^7$. As we sample m initial conditions per parameter set, we thus sample $limits/m$ parameter value sets in total.

Finally, we can also reduce the number of diffusion parameter sets to be sampled. Either *binary_ diffuser* is set to one specific vector (e.g. Figure 9 bottom option 1) or the limits of diffusing nodes can be restrained by setting the maximum number of diffusing nodes, *m_ max*, to a value smaller than $n = 5$ (e.g. Figure 9 bottom option 2).

```
%Reduce initial conditions to sample chosing 2 concentrations per node
n=5;
c_ini = permn([1 20000],n);
```

```
%Option: chose parameters randomly within range: k_lim = [0.1 100];
limits = 14500000; %Suitable limit of number of ODE equation
divi = length(c_ini(:,1)); %Check how many ODE equations are calculated per
%parameter set
l=round(limits/divi); %Number of parameter sets that can be sampled
k_lim = [0.1 100]; %Range of parameters
k_length = 10; %Number of parameters
rng('shuffle') %Set seed according to time
k_grid = zeros(l,k_length);
for i = 1:l
k_grid(i,1:k_length)=(k_lim(2)-k_lim(1)).*rand(1,k_length)+k_lim(1);
end
k_grid=k_grid';
```

```
%Diffusion settings option 1: chose specifically which nodes diffuse
m_min = 2;
m_max = n;
binary_diffusor = [0 1 1 1 0];
d_range = logspace(-3,3,7);
```

```
%Diffusion settings option 2: sample all possible diffusor settings where 2
%nodes are diffusing
m_min = 2;
m_max = 2;
binary_diffusor = permn([0 1],n);
d_range = logspace(-3,3,7);
```

Figure 9: *Example of how to reduce the running time of the code. Top: Reducing the number of initial conditions reduces the number of ODE siumations. Here, 2 initial conditions are sampled per node resulting in* 2^5 *initial conditions, where each node can either have a value of 1 or 20,000. Middle: Rather than sampling the parameters on a regular grid, sampling of random parameter values can be used. To ensure the output files are not too large and the running time remains within feasible running times, we suggest to not solve more than* 1.45^7 *ODE simulations (= limits). As we sample multiple initial conditions per parameter, we need to limit the parameter value sampling by dividing the total number of allowed ODE simulations (= limits) by the number of initial conditions we simulate per parameter combination (= divi). The parameter grid k_grid is then defined within a range of parameter values defined here as k_lim. Please note that the seed of the random number generator (= rng) should be set to 'shuffle'. Bottom: To limit the sampling of the diffusion parameters, two options can be used. Option 1 uses a single vector to define binary_diffusor, option 2 limits the number of possible diffusing molecules to two at a time (i.e.* x_1 *and* x_2 *or* x_2 *and* x_3 *or ...) by changing the range of m_min and m_max.*