StochPy

Stochastic modelling in Python

StochPy User Guide

Release 1.0.3

Timo Maarleveld

November 17, 2011

CONTENTS

Part I

Introduction

StochPy (Stochastic Modelling in Python) is an easy-to-use package, which provides several stochastic simulation algorithms (SSA's). These SSA's can be used to simulate a biochemical system in a stochastic manner. Further, several unique and easy-to-use analysis techniques are provided by StochPy.

The classical approach to simulate a biochemical system is by a set of coupled ordinary differential equations (ODEs). Biological systems are often highly complex, thus there are regularly no analytical solutions available. As a result, a set of coupled ODEs is often numerically solved.

This classical approach has a deterministic nature. Therefore, a given set of ODEs will always produce the same results. These results describe the macroscopic behaviour of the biochemical system, while cell populations have regularly a heterogeneous nature. Further, the change in species amount in a deterministic model is a continuous process, while this is in reality a discrete process.

This deterministic nature becomes really problematic for systems with molecules that have a low copy number. As a result, deterministic models are often inaccurate.

SSA's try to describe the time evolution of a reacting system, such that it takes into account discreteness and stochasticity. Discreteness and stochasticity play often important roles in biochemical systems. Therefore, SSA's are widely used to simulate biochemical systems. Especially for biochemical systems that contain low copy numbers. For such systems, deterministic models often fail to capture the stochasticity of the system, while SSAs are capable of capturing this stochastic behaviour.

StochPy is an extension of PySCeS. PySCeS - the Python Simulator for Cellular Systems - is an extendible tool kit for the analysis and investigation of cellular systems. StochPy operates independently of PySCeS, but it uses several parts of the PySCeS package, such as the text based Model Description Language of PySCeS. This MDL is further explained in *the PySCeS Model Description Language section*. PySCeS provides integrators for time simulation of deterministic models. Further, structural analysis can be done. For example, information about the stoichiometric matrix (N), the kernel matrix (K), and the link matrix (L) can be obtained. Also, metabolic control analysis (MCA) can be performed. In the latest PySCeS releases, a prototype is build-in to use StochPy.

Part II

Start using StochPy

Here, we assume that StochPy is already installed. If not, check README.txt or *installation section* of this user guide.

To start modelling, it is necessary to work in an interactive Python shell. You can use iPython (recommended), IDLE (Python GUI), or simply the `python` command in a terminal:

```
$ python
Python 2.7.1+ (r271:86832, Apr 11 2011, 18:13:53)
[GCC 4.5.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Then, use `import stochpy`:

```
>>> import stochpy

##############################################################################
#                                                                            #
#            Welcome to the interactive StochPy environment                  #
#                                                                            #
##############################################################################
#    StochPy: Stochastic modelling in Python                                 #
#    http://stompy.sourceforge.net                                           #
#    Copyright(C) T.R Maarleveld, B.G. Olivier 2010-2011                     #
#    Email: tmd200@users.sourceforge.net                                     #
#    VU University, Amsterdam, Netherlands                                   #
#    StochPy is distributed under the BSD licence.                           #
##############################################################################

Version 1.0
Output Directory: /home/user/Stochpy
Model Directory: /home/user/Stochpy/pscmodels
```

This message means that the StochPy package is imported and it is ready to use. Use the next command to get more information about StochPy:

```
>>> help(stochpy)
```

One can use this help function for every available function in StochPy:

```
>>> mod = stochpy.SSA()
>>> help(mod)
>>> help(mod.DoStochSim)
```

Consequently, it is possible to start one of the four modules, which will be explained in the next sections.

Part III

Stochastic Modelling

CHAPTER

ONE

MODELLING INPUT

The input for performing stochastic modelling are rate equations and initial conditions. For this, the PySCeS MDL is used as default format, which is further explained in *the PySCeS Model Description Language section*. Also, StochPy supports to use of models that are written in SBML. To use models written in SBML, several libraries are necessary, which is explained in *the installation section*. These SBML input files are converted into the PySCeS MDL format, which are used by the stochastic simulators.

StochPy supports assignments and SBML event facilities which resets species populations during the simulation since the StochPy 1.0.0 release.

1.1 Stochastic versus Deterministic Rate Equations

Deterministic rate equations have normally been used to describe a system of biochemical reactions, whereas these equations are often not valid for stochastic modelling. Therefore, stochastic rate equations are necessary. Therefore, it is important to understand the differences between deterministic and stochastic rate equations and why deterministic rate equations can be invalid. For this reason, the next part of this section explains the differences between deterministic and stochastic rate equations.

First, deterministic models represent species often by concentration, while stochastic models represent species often by amounts. The species amounts are identical to:

```
Na*[X]*Volume(L)
```

Here, Na is the number of Avegadro (6.02*10e23). This is the first step of the conversion of deterministic rate constants (k) to stochastic rate constants (c).

1.2 Zeroth order

Assume the following reaction:

```
--> X
```

The deterministic of rate equation of this reaction is:

```
k (Ms^-1)
```

The stochastic rate equation of this reaction is:

```
c (s^-1)
```

1.3 First order

Consider the following reaction:

```
X --> Y
```

The deterministic rate equation of this reaction is:

```
k*[X]
```

The stochastic rate equation of this reaction is:

```
c*x (s^-1)
```

Here, x corresponds to Na*[X]*Volume(L) particles. This means that x changes k*Na*[X]*Volume(L) = k*x molecules per second, which means that k=c. Thus, first order stochastic and deterministic rate equations are identical.

1.4 Second order

There exist several types of second order reactions. First, consider the next reaction:

```
X+Y --> Z
```

The deterministic rate equation of this reaction is:

```
k*[X]*[Y] (Ms^-1)
```

The stochastic rate equation of this reaction is:: c*x*y (s^-1)

Again, [X] corresponds to Na*[X]*Volume(L) particles and the same is true for [Y]. Therefore, the following relationship holds:

```
k*[X]*[Y] (Ms^-1) = (k*x*y)/(Na*V)^2, hence c = k/(Na*V)
```

Secondly, consider a dimerisation reaction:

```
2X --> Y
```

The deterministic rate equation of this reaction is:

```
k*X^2
```

The stochastic rate equation of this reaction is:

```
0.5*c*x*(x-1)
```

The first important concept to understand is that a dimerisation reaction can only occur if there are at least two molecules of species X available. For this reason, the stochastic rate equation must be zero if there is only one X molecule available.

1.5 Third order

This type of reactions does usually not occur in chemical reactions, but they are described just for the sake of completeness.

There exist three types of third order reactions:

```
1) X+Y+Z --> A
```

The deterministic rate equation of this reaction is:

```
k*[X]*[Y]*[Z]
```

The stochastic rate equation of this reaction is:: c*x*y*z

2. X+2Y –> Z

The deterministic rate equation of this reaction is:

```
k*[X]*([Y]^2)
```

The stochastic rate equation of this reaction is:

```
0.5*c*x*y*(y-1)
```

```
3) 3X --> Y
```

The deterministic rate equation of this reaction is:

```
k*[X]^3
```

The stochastic rate equation of this reaction is:

```
(1/6)*x*(x-1)*(x-2)
```

For the last reaction, three x molecules are necessary, thus this rate equation can not fire if there are less than three molecules of x available.

Further explanations are given in the *Stochastic Simulations Algorithms section*.

CHAPTER

TWO

STOCHASTIC SIMULATION ALGORITHMS

The most important module of this package is the stochastic simulation algorithm module.

2.1 Algorithms

StochPy contains four Stochastic Simulation Algorithms (SSA's):

- Direct Method
- First Reaction Method
- Next Reaction Method
- Tau-Leaping Method

Use the following code to start using the SSA module:

```
>>> mod = stochpy.SSA()
Info: The Direct method is selected to perform the simulations
Parsing file: /home/user/Stochpy/pscmodels/ImmigrationDeath.psc
```

The default algorithm is the Direct Method and the default stochastic model is the Immigration-Death model. Next, a stochastic simulation can be done with the default settings. Of course, another method or model can be selected:

```
>>> mod.Method('FirstReactionMethod')
Info: The First Reaction method is selected to perform the simulations
>>> mod.Method('NextReactionMethod')
Info: The Next Reaction method is selected to perform the simulations
>>> mod.Method('TauLeaping')
Info: The Explicit Tau-Leaping method is selected to perform the simulations
>>> mod.Method('Direct')
Info: The Direct method is selected to perform the simulations
```

2.2 Model Selection

The following comment can be used to select another model in the PySCeS MDL:

```
>>> mod.Model('BurstModel.psc')
Parsing file: /home/user/Stochpy/pscmodels/BurstModel.psc
>>> mod.Model(File='BurstModel.psc',dir='/home/user/Stochpy/pscmodels')
Parsing file: /home/user/Stochpy/pscmodels/BurstModel.psc
```

Alternatively, one can use a model written in the systems biology markup language (SBML):

```
>>> mod.Model('dsmts-001-01.xml','/home/user/')
Info: extension is .xml
Info: single compartment model: locating "Death" in default compartment
Info: single compartment model: locating "Birth" in default compartment
Writing file: /home/user/Stochpy/pscmodels/dsmts-001-01.xml.psc

SBML2PSC
in : /home/user/dsmts-001-01.xml
out: /home/user/Stochpy/pscmodels/dsmts-001-01.xml.psc
Info: SBML data is converted into psc data
and is stored at: /home/user/Stochpy/pscmodels
Parsing file: /home/user/Stochpy/pscmodels/dsmts-001-01.xml.psc

>>> mod.model_file
'dsmts-001-01.xml.psc'
>>> mod.model_dir
'/home/user/Stochpy/pscmodels'
```

2.3 Run a SSA

Then, a simulation can be started. By default, one trajectory of 1000 time steps is generated.:

```
>>> mod.DoStochSim()
Info: 1 trajectory is generated
Number of time steps 1000  End time  15.2161060754
Simulation time 0.0805060863495
>>> mod.Trajectories(5)
Info: The number of trajectories is:  5
>>> mod.DoStochSim()
Info: 5 trajectories are generated
Info: Time simulation output of the trajectories is stored at ...
Info: output is written to: /home/user/Stochpy/ssa_smod.log
Simulation time 0.24516955151
```

Also, it is possible to select the end time of a simulation. Be careful, because this is dangerous! It is dangerous, because the reactions in the model determine when a certain reaction fires. So, dT - the time between two reactions to fire - is large if the majority of the reactions is unlikely to fire, but becomes small if the majority of the reactions is very likely to fire. As a result, model

A takes 1000 time steps to reach the end time, but model B could take more than 1,000,000 time steps to reach the end time:

```
>>> mod.Endtime(100)
```

The high-level function DoStochSim() does more than starting a stochastic simulation. It accepts the following arguments:

```
>>> mod.DoStochSim(method = 'Direct', mode = 'time', end = 50,
trajectories = 1)
Info: The Direct method is selected to perform the simulations
Parsing file: /home/user/Stochpy/pscmodels/ImmigrationDeath.psc
Info: 1 trajectory is generated
Number of time steps 3807  End time  50.0049282696
Simulation time 0.142936944962
```

As a result, one can use one high-level function to determine the modelling options. Moreover, StochPy can show the the current settings:

```
>>> mod.ShowSpecies()
['mRNA']
>>> mod.ShowOverview()
Current Model:        ImmigrationDeath.psc
Simulation end time:  50
Current Algorithm:    <class 'stochpy.DirectMethod.DirectMethod'>
Number of trajectories:       1
Propensities are not tracked
```

Here, ShowSpecies() gives all the species in the model and ShowOverview() gives all the current settings.

Finally, the description of the system can be changed (the model), while you are working with StochPy. For example, you want to simulate the immigration-death model with several values of Ksyn and Kdeg. First, you start a simulation with the default values, but you decide to change these default values. Then, simply change one of the parameters in the file ImmigrationDeathmodel.psc (in the directory where the models are stored) and reload the model with the following high-level function:

```
>>> mod.Reload()
Parsing file: /home/user/Stochpy/pscmodels/ImmigrationDeath.psc
```

2.4 Build-in Analysis Techniques

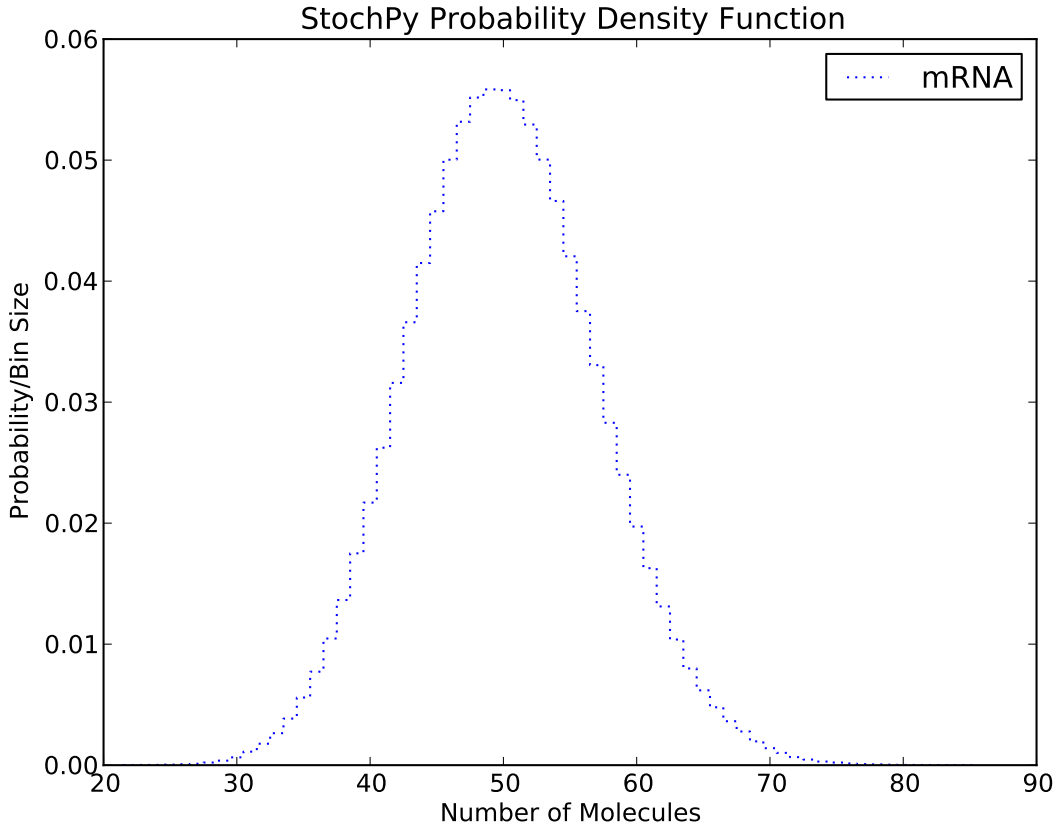
After a simulation, some analysis can be done. Besides information about the time simulation, probability density functions, propensities, and waiting time plots can be created. The default line style is dotted. Furthermore, interpolation can be done if more than 1 trajectory is generated:

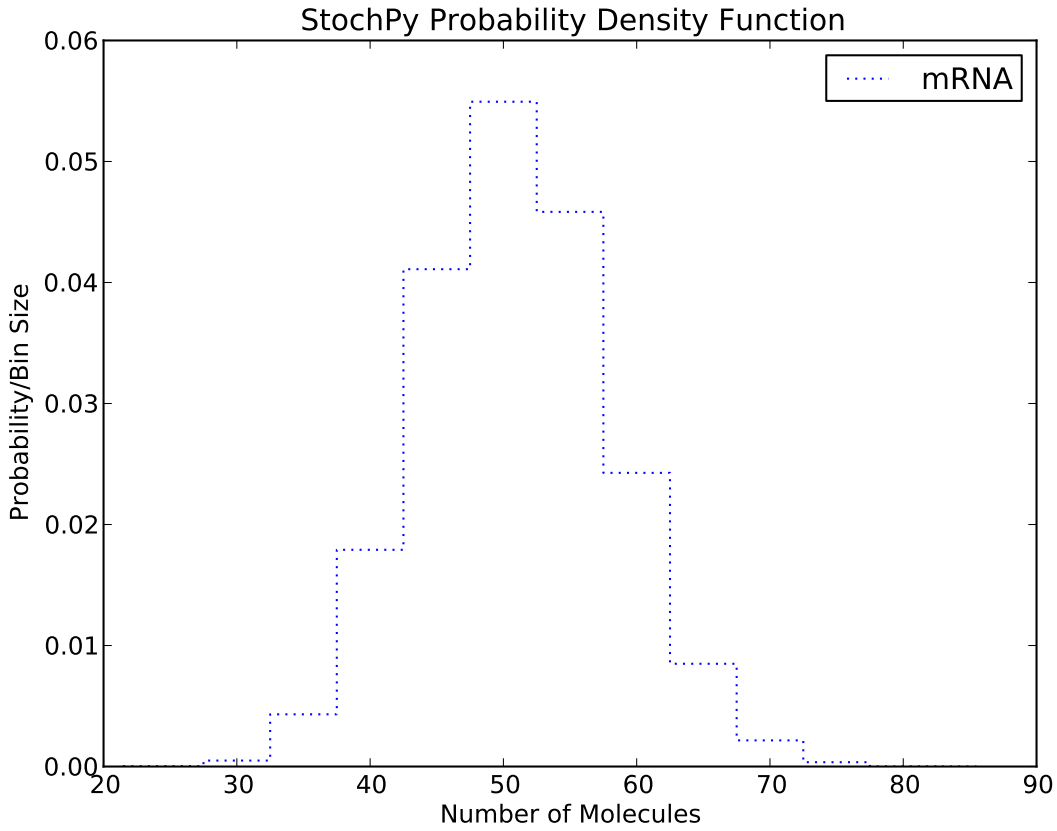
```
>>> mod.PlotTimeSim(linestyle = 'dashed',title = 'StochPy Time Simulation Plot')
>>> mod.PlotPropensities()
```

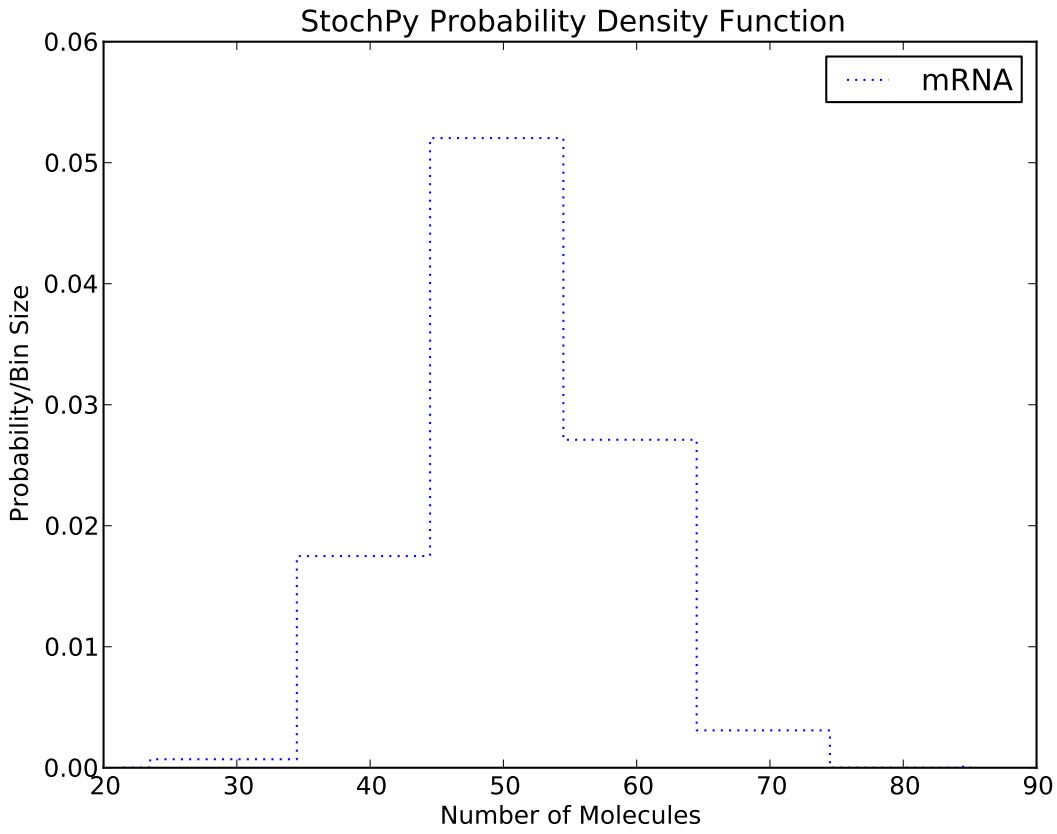
```
>>> mod.PlotDistributions()
>>> mod.PlotWaitingtimes()
>>> mod.PlotInterpolatedData()
```

In addition, one can use binning (default bin size = 1) for plotting of the probability density function:

```
>>> mod.PlotDistributions(bin_size=5)
>>> mod.PlotDistributions(bin_size=10)
```




StochPy Probability Density Function
mRNA
Probability/Bin Size
0.06
0.05
0.04
0.03
0.02
0.01
0.00
20
30
40
50
60
70
80
90
Number of Molecules


StochPy Probability Density Function
mRNA
Probability/Bin Size
0.06
0.05
0.04
0.03
0.02
0.01
0.00
20
30
40
50
60
70
80
90
Number of Molecules

Each species is plotted by default, but the user can determine which species are plotted:

```
>>> mod.PlotTimeSim(species2plot = ['S1','S2'])
>>> mod.PlotPropensities(rates2plot = 'R1')
>>> mod.PlotWaitingtimes(rates2plot = 'R2')
>>> mod.PlotDistributions('S4')
Error: species S4 is not in the model
```

In addition, stochpy.plt is available to manipulate generated plots or to make your own plots:

```
>>> mod.PlotTimeSim(marker = 'v')
>>> stochpy.plt.title('Your own title')
>>> stochpy.plt.xlabel('Time (s)')
>>> stochpy.plt.xlim([0,10**3])
>>> stochpy.plt.savefig('filename.pdf')    # stores the plot in the cwd
```

Of course, it is also possible to print such information about the simulation, which can be useful if MatPlotLib is not installed:

```
>>> mod.PrintTimeSim()
>>> mod.PrintPropensities()
>>> mod.PrintDistributions()
>>> mod.PrintWaitingtimes()
>>> mod.PrintMeanWaitingtimes()
>>> mod.PrintInterpolatedData()
```

Also, such information can be printed to a text file:

```
>>> mod.Write2File()
>>> mod.Write2File('TimeSim','/home/user/timesmod.txt')
>>> mod.Write2File('Propensities')
>>> mod.Write2File('Distributions')
>>> mod.Write2File('Waitingtimes')
>>> mod.Write2File('Interpol')
```

One can get information about the mean and standard deviation of each species during a simulation:

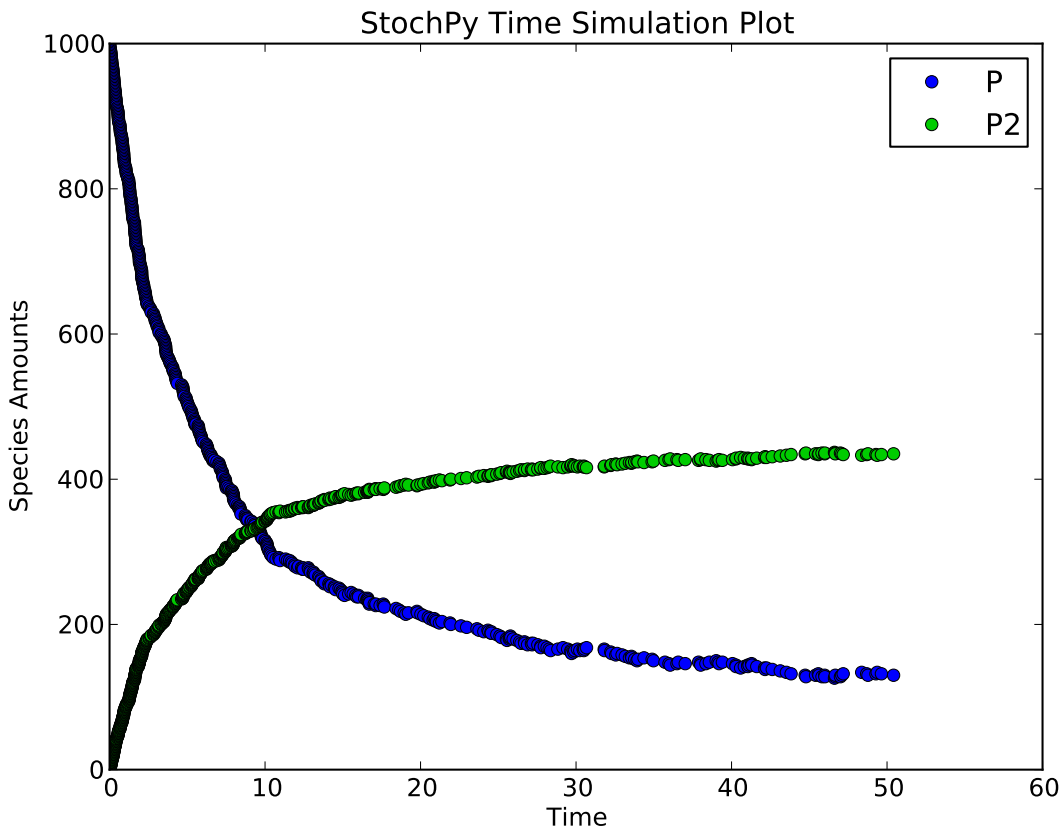
```
>>> mod.ShowMeans()
>>> mod.ShowStandardDeviations()
```

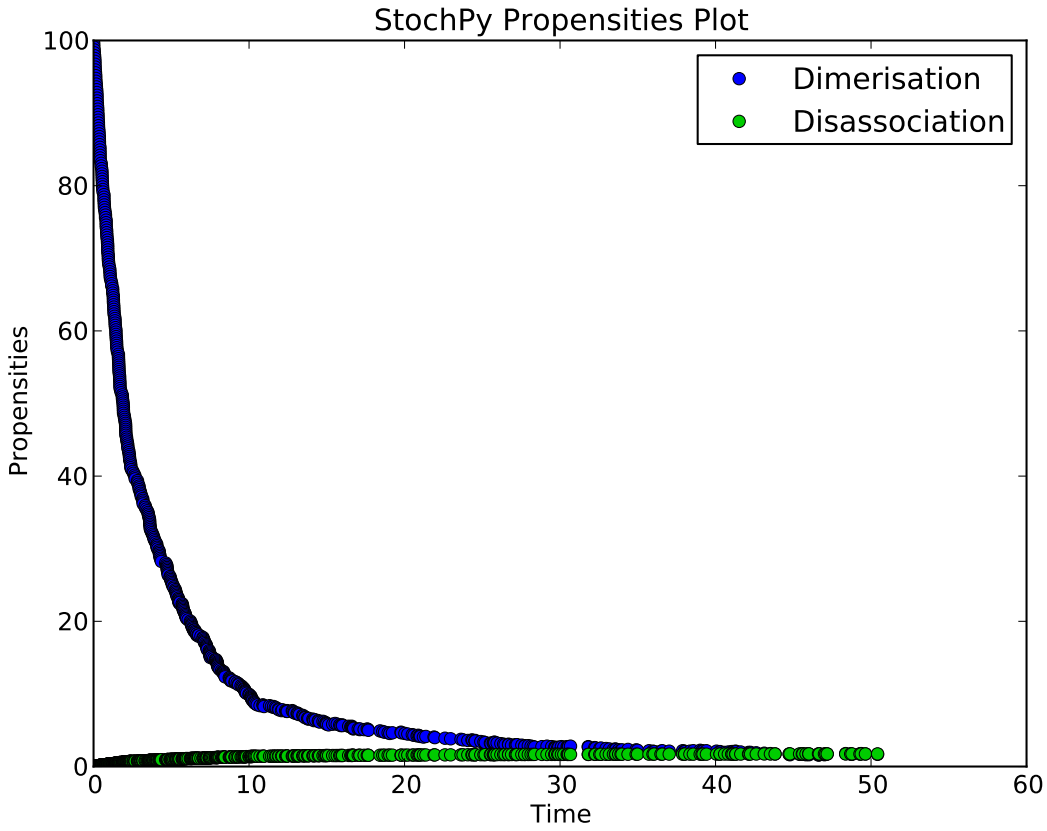
Finally, one can use the data objects that store the simulation data to perform their own type of analysis. See *Using Stochpy as a Library*.

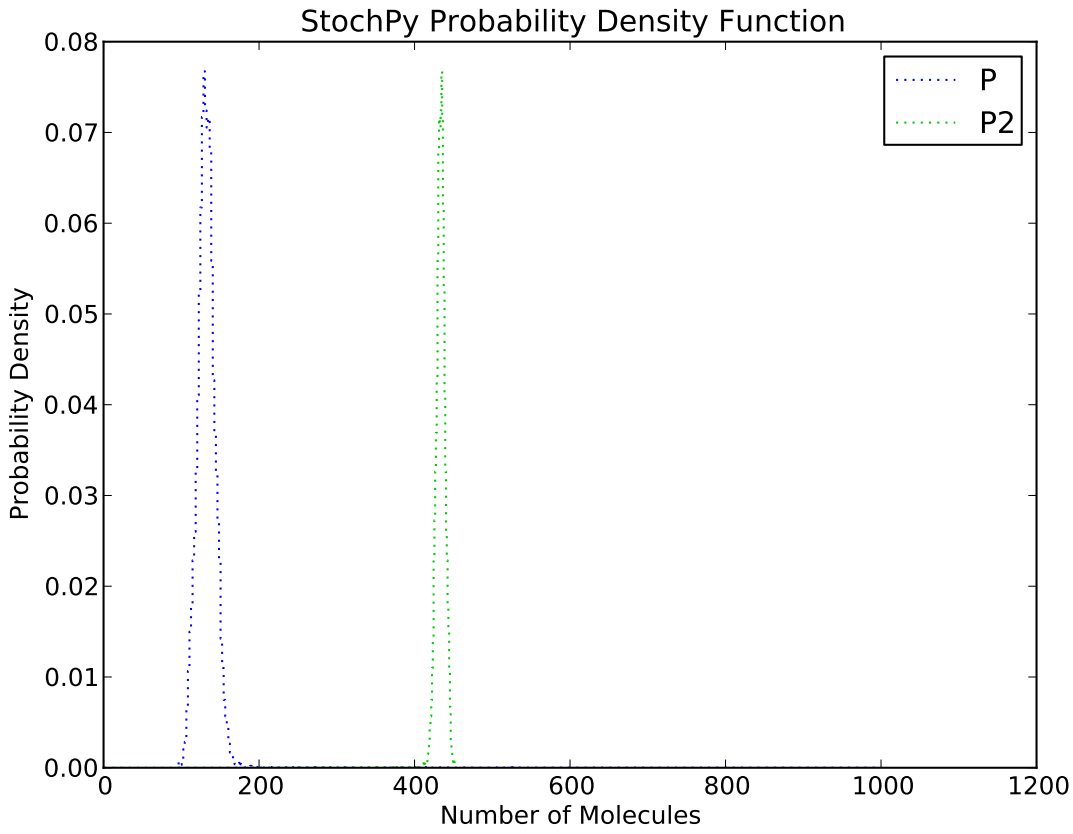
2.5 Examples

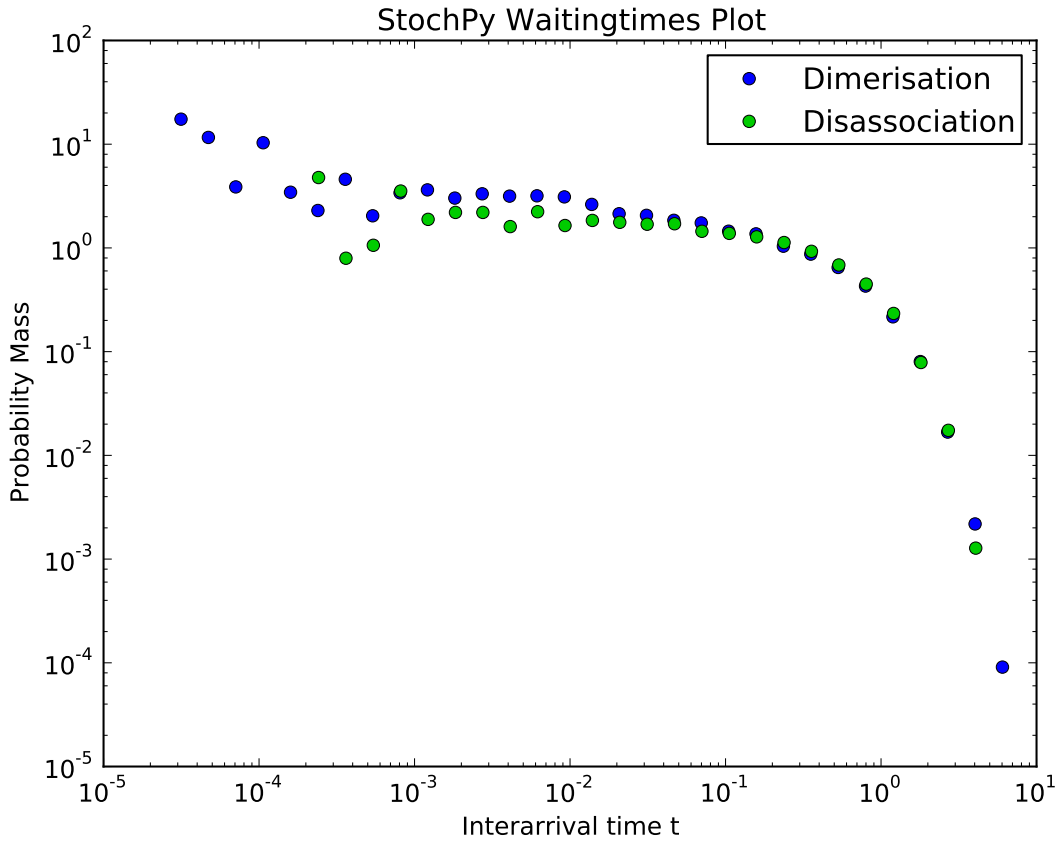
Stochastic simulations are done with a dimerisation model to illustrate how you can use the stochastic simulation algorithm module (we assume that StochPy is already imported). This model contains two species, denoted by P and P2.

```
>>> mod = stochpy.SSA()
>>> mod.Model('dsmts-003-02.xml.psc')
>>> mod.TrackPropensities()
>>> mod.DoStochSim(end = 50,mode = 'time')
Info: 1 trajectory is generated
Number of time steps 591  End time  50.0230192145
Simulation time 0.0776190757751


>>> mod.PlotTimeSim()
>>> mod.PlotPropensities()
>>> mod.DoStochSim(end=5000,mode = 'time')
Info: 1 trajectory is generated
Number of time steps 17799  End time  5000.10900565
Simulation time 1.39394283295
>>> mod.PlotDistributions()
>>> mod.PlotWaitingtimes()
```




StochPy Propensities Plot
Dimerisation
Disassociation
Propensities
Time
0
20
40
60
80
100
0
10
20
30
40
50
60


StochPy Probability Density Function
P
P2
Probability Density
Number of Molecules
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0
200
400
600
800
1000
1200

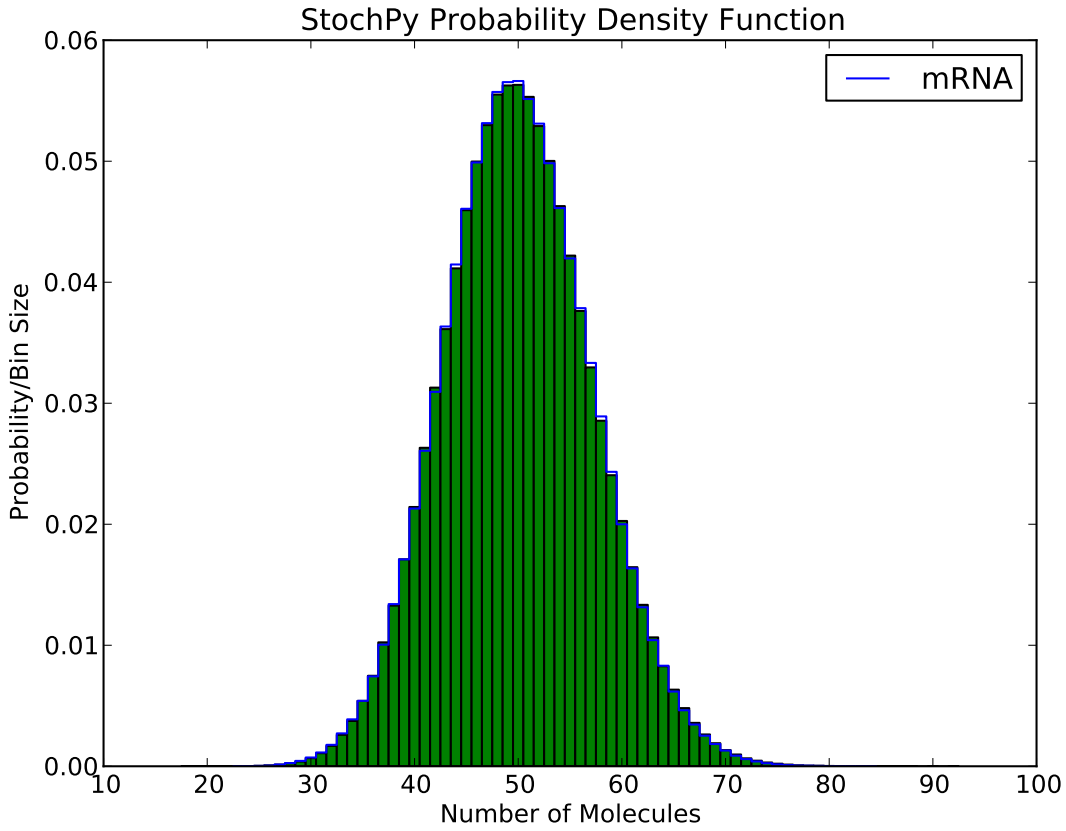


In addition, one can exploit the fact that StochPy is written in Python, which we illustrate with the immigration-death model. This immigration-death model contains one species, which is born born with a constant rate and dies with a first order reaction based on the amount of the species. This immigration-death model is one of the simplest examples of a Poisson process. A Poisson process is a stochastic process where events occur continuously and independently of one another.

Therefore, the mean and variance of this process are both 50. First, N samples are drawn from a Poisson distribution. Secondly, a stochastic simulation is done for N time steps. Then, the probability density function is plotted. Finally, a histogram of the randomly generated Poisson data, which can be plotted into the plot of the probability density function.:

```
>>> import numpy as np
>>> lambda = 50
>>> N = 2500000
>>> data = np.random.poisson(lambda,N)
>>> mod.Model('ImmigrationDeath')  # Ksyn = 10, Kdeg = 0.2, and mRNA(init) = 50
>>> mod.DoStochSim(end=N,mode='steps')
>>> mod.PlotDistributions(linestyle= 'solid')
>>> n, bins, patches = stochpy.plt.hist(data-0.5, max(data)-min(data),
    normed=1, facecolor='green')
>>> mod.ShowMeans()
mRNA  49.9117062522
>>> mod.ShowStandardDeviations()
Species       Standard Deviation
```

```
mRNA   7.0730769793
```

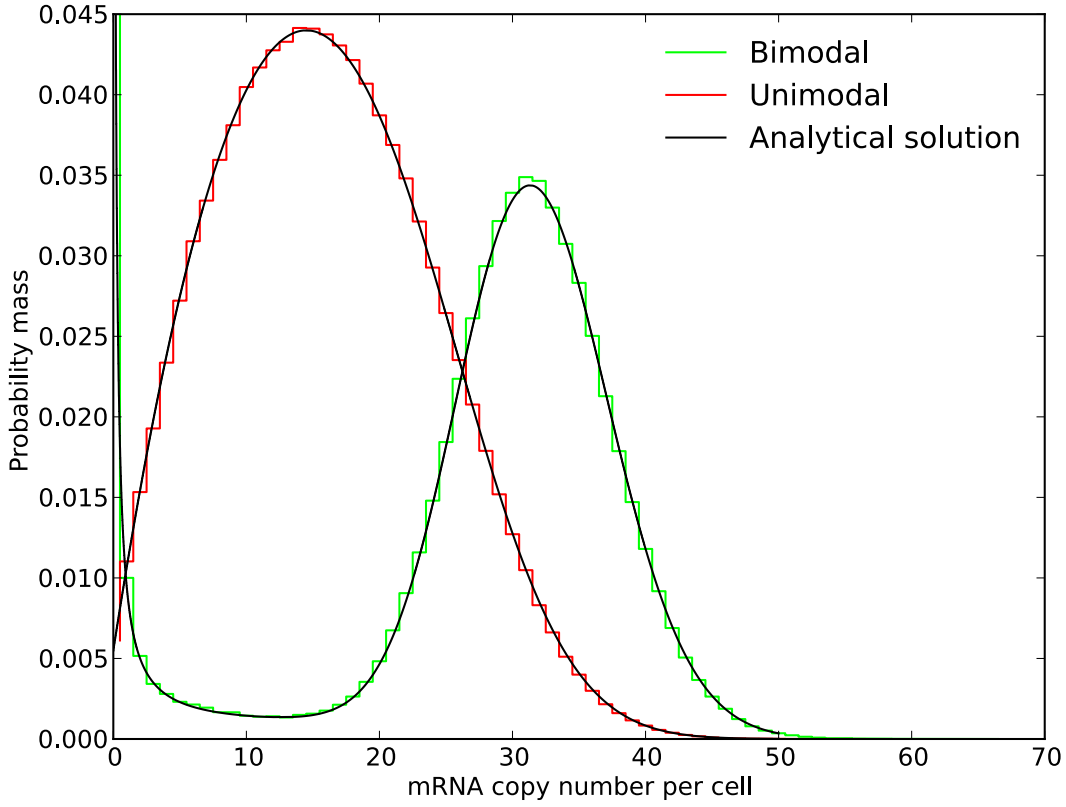


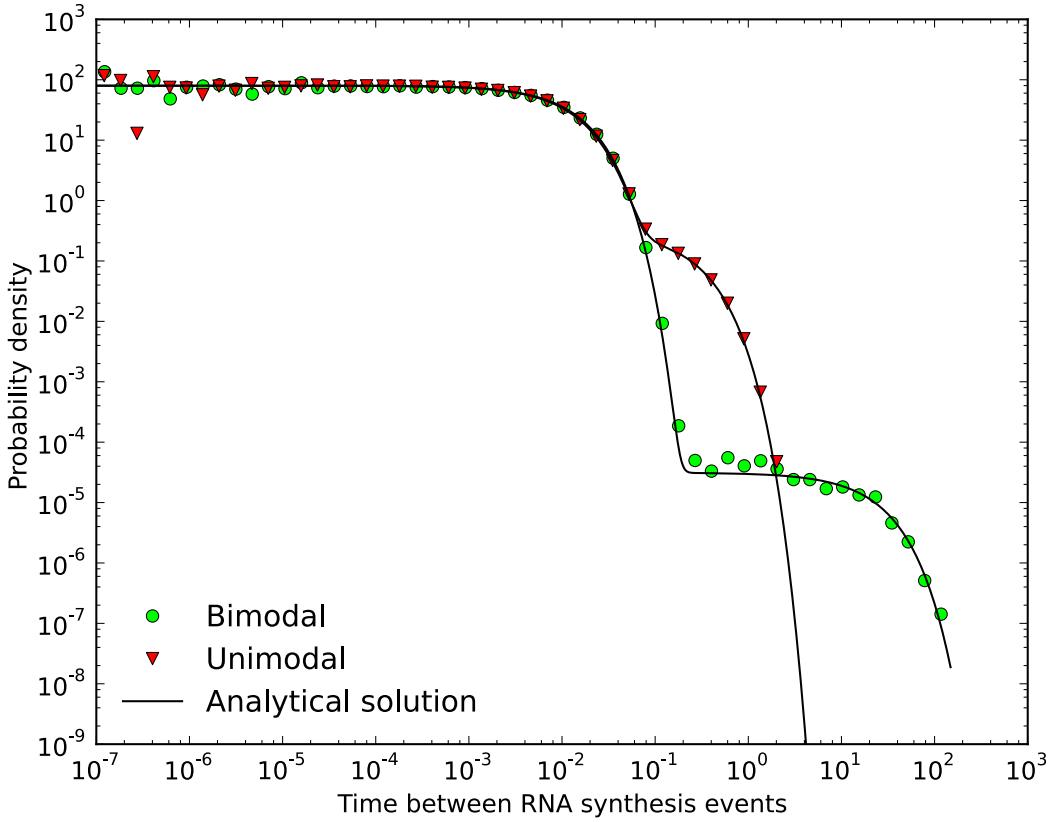
Finally, the capabilities of StochPy is illustrated with 'burstmodel.psc'. Two different parameter sets (kon = 0.05,koff = 0.05, ksyn = 80,kdeg = 2.5 and kon = 5.0,koff = 5.0, ksyn = 80,kdeg = 2.5) are used to demonstrate this model. The mean number of mRNA will be equal, but the distribution is completely different. For the first parameter values set, the number of mRNA molecules is be bimodal distributed. In contrast, the number of mRNA molecules is unimodal distributed for the second parameter values set. Therefore, the standard deviation of the number of mRNA molecules is different. Analytical solutions are plotted in black lines.:

```
>>> import numpy as np
>>> import mpmath
>>> import stochpy
>>> mpmath.mp.pretty = True

>>> def GetAnalyticalPDF(kon,koff,kdeg,ksyn):
  """ Get the analytical probability density function.
  The analytical solution is taken from Sharezaei and Swain 2008
  (Analytical distributions for stochastic gene expression) """
  x_values = np.arange(0,50,0.1)
  y_values = []
  for m in x_values:
      a = ((ksyn/kdeg)**m)*np.exp(-ksyn/kdeg)/mpmath.factorial(m)
      b = mpmath.mp.gamma((kon/kdeg)+m) * mpmath.mp.gamma(kon/kdeg + koff/kdeg)/
       (mpmath.mp.gamma(kon/kdeg + koff/kdeg + m)* mpmath.mp.gamma(kon/kdeg))
      c = mpmath.mp.hyp1f1(koff/kdeg,kon/kdeg + koff/kdeg + m,ksyn/kdeg)
```

```
        y_values.append(a*b*c)
    return x_values,y_values

>>> def GetAnalyticalWaitingtimes(kon,koff,ksyn):
    """ Get analytical waiting times """
    A = mpmath.sqrt(-4*ksyn*kon+(koff + kon + ksyn)**2)
    x = []
    for i in np.arange(-20,5,0.25):
        x.append(mpmath.exp(i))
    y = []
    for t in x:
        B = koff + ksyn - (mpmath.exp(t*A)*(koff+ksyn-kon))-kon+A+ mpmath.exp(t*A)
        p01diff = mpmath.exp(-0.5*t*(koff + kon + ksyn+A))*B/(2.0*A)
        y.append(p01diff*ksyn)
    return (x,y)

    >>> mod = stochpy.SSA()
    >>> mod.Model('Burstmodel.psc')  # Parameter values in Burstmodel.psc: kon = k
    >>> ntimesteps = 1000000
    >>> mod.DoStochSim(end=ntimesteps,mode='steps')
    >>> mod.PlotDistributions(species2plot = 'mRNA', colors=['#32CD32'],marker = '
    >>> mod.PlotWaitingtimes('R3', colors=['#32CD32'],linestyle = 'None',marker='o

    >>> ### Change Parameter values in Burstmodel.psc: kon = koff = 5.0
    >>> mod.Reload()
    >>> mod.DoStochSim(end=ntimesteps,mode='steps')
    >>> stochpy.plt.figure(1)
    >>> mod.PlotDistributions(species2plot = 'mRNA', colors=['r'],marker ='v')

    >>> kon = 0.05
    >>> koff = 0.05
    >>> kdeg = 2.5
    >>> ksyn = 80.0
    >>> x,y = GetAnalyticalPDF(kon,koff,kdeg,ksyn)
    >>> stochpy.plt.step(x,y,color ='k')

    >>> kon = 5.0
    >>> koff = 5.0
    >>> x,y = GetAnalyticalPDF(kon,koff,kdeg,ksyn)
    >>> stochpy.plt.step(x,y,color ='k')
    >>> stochpy.plt.xlabel('mRNA copy number per cell')
    >>> stochpy.plt.ylabel('Probability mass')
    >>> stochpy.plt.legend(['Bimodal','Unimodal', 'Analytical solution'],numpoints
    >>> stochpy.plt.title('')
    >>> stochpy.plt.ylim([0,0.045])

    >>> ###################### Finish the waiting times plot ######################

    >>> ### Change Parameter values ##
    >>> mod.PlotWaitingtimes('R3', colors=['r'],linestyle = 'None',marker='v')
```

```
>>> kon = 0.05
>>> koff = 0.05
>>> (x,y) = GetAnalyticalWaitingtimes(kon,koff,ksyn)
>>> stochpy.plt.plot(x,y,color ='k')

>>> kon = 5.0
>>> koff = 5.0
>>> (x,y) = GetAnalyticalWaitingtimes(kon,koff,ksyn)
>>> stochpy.plt.plot(x,y,color ='k')
>>> stochpy.plt.xlabel('Time between RNA synthesis events')
>>> stochpy.plt.ylabel('Probability density')
>>> stochpy.plt.legend(['Bimodal','Unimodal', 'Analytical solution'],numpoints
>>> stochpy.plt.title('')
>>> stochpy.plt.xlim([10**-7,10**3])
>>> stochpy.plt.ylim([10**-9,10**3])
```





2.6 Using StochPy as a Library

It is straightforward to use StochPy as a library in your code. A data object, data_stochsim, is created for those that want to use StochPy as a library or for people that want to do their own analysis. Species, distributions, propensities, simulation time, and waiting times are stored in for instance NumPy arrays and lists. In addition, labels are stored for each of these data types in separate lists. Of course, data such as distributions are only available if they are calculated.

Furthermore, determined means and standard deviations are stored in dictionaries. Finally, information about the stochastic simulation such as the number of time steps, the simulation end time, and the trajectory is stored.

This data object (data_stochsim) is written to disk space if multiple trajectories are generated. The high-level function GetTrajectoryData(n) can be used to get access to the simulation data of a specific trajectory. By default, the latest generated trajectory is not written to disk space, thus accessible without using the high-level function GetTrajectoryData(n):

```
>>> import stochpy
>>> mod = stochpy.SSA()
>>> mod.DoStochSim(trajectories = 10,mode = 'steps',end = 1000)
>>> mod.data_stochsim
<stochpy.PyscesMiniModel.IntegrationStochasticDataObj object at 0x32cd750>
>>> mod.data_stochsim.simulation_trajectory
```

```
10
>>> mod.data_stochsim.time          # time array (not shown)
>>> mod.data_stochsim.species       # species array (not shown)
>>> mod.data_stochsim.species_labels
>>> mod.data_stochsim.getSpecies()  # time + species array (not shown)
>>> mod.GetMeans()                  # for each species
>>> mod.data_stochsim.means
>>> mod.GetDistributions()
>>> mod.data_stochsim.distributions # for each species (not shown)
>>> mod.GetWaitingtimes()
>>> mod.data_stochsim.waiting_times # for each reaction (not shown)
>>> mod.GetTrajectoryData(5)
>>> mod.data_stochsim.simulation_trajectory
5
```

Alternatively, one can be interested in interpolated data from one simulation with multiple trajectories. A second data object (data_stochsim_interpolated) is becomes available if the user uses one of the functions to get interpolated data:

```
>>> mod.data_stochsim_interpolated
AttributeError: SSA instance has no attribute 'data_stochsim_interpolated'
>>> mod.GetInterpolatedData()
>>> mod.data_stochsim_interpolated
<stochpy.PyscesMiniModel.InterpolatedDataObj object at 0x32cd3d0>
>>> mod.data_stochsim_interpolated.time  # time array (not shown)
>>> mod.data_stochsim_interpolated.means # at every t (not shown)
>>> mod.data_stochsim_interpolated.standard_deviations  # SDs (not shown)
```

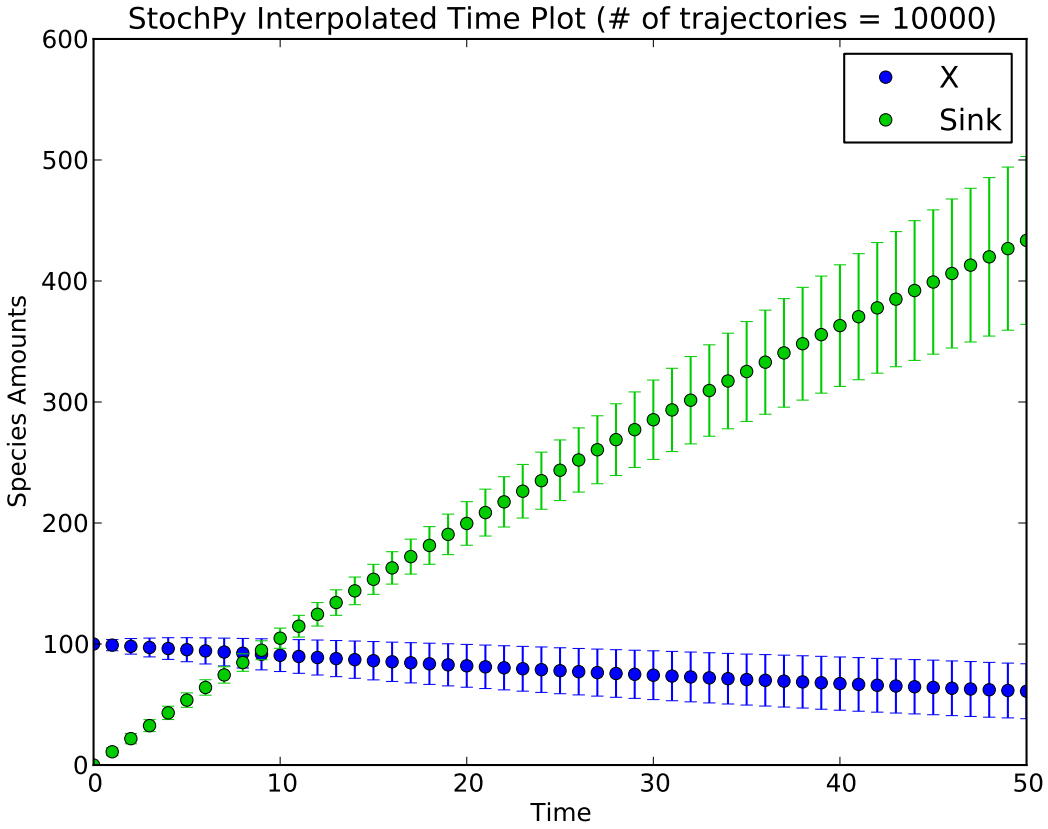
2.7 Stochastic Test suite

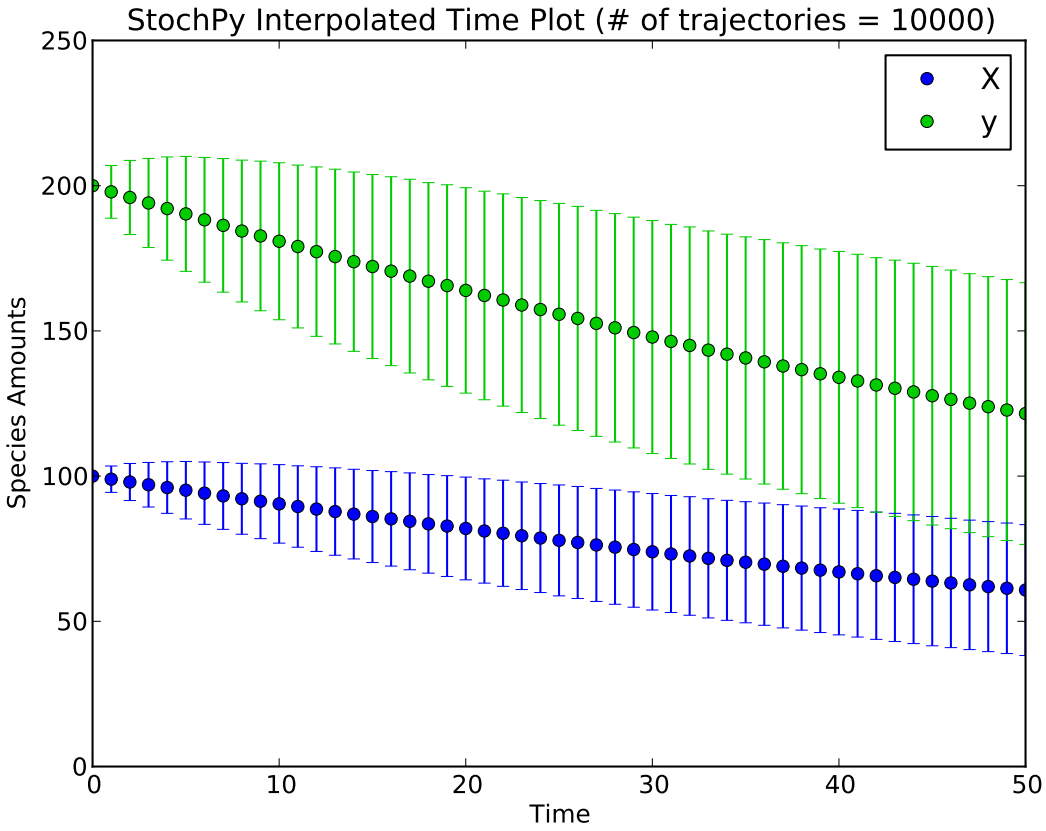
The stochastic test suite from Evans et al. 2008 (The SBML discrete stochastic models test suite) is used to test StochPy. This test suite tests stochastic simulation software on the following points:

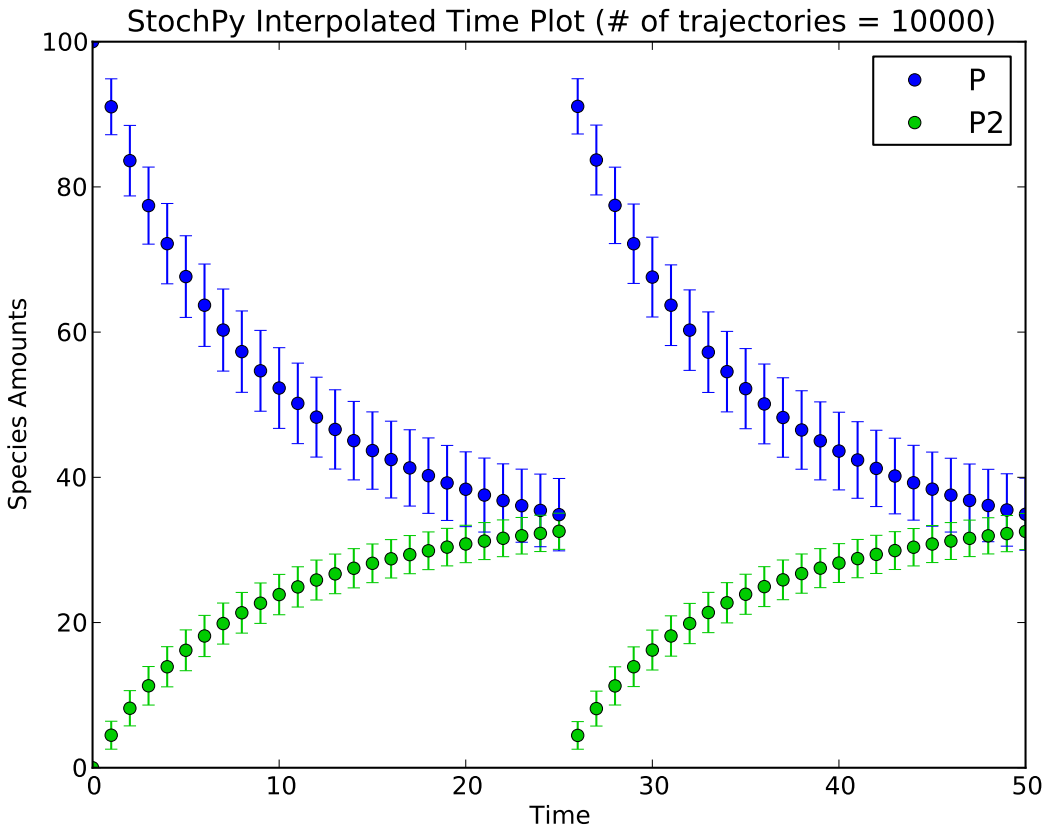
- local and global parameters (parameter overloading)
- boundary conditions
- Cell compartment volume
- hasOnlySubstanceUnits flag
- math expression parsing
- compartment volume explicitly including in the rate laws
- assignment rules
- time events
- species population events

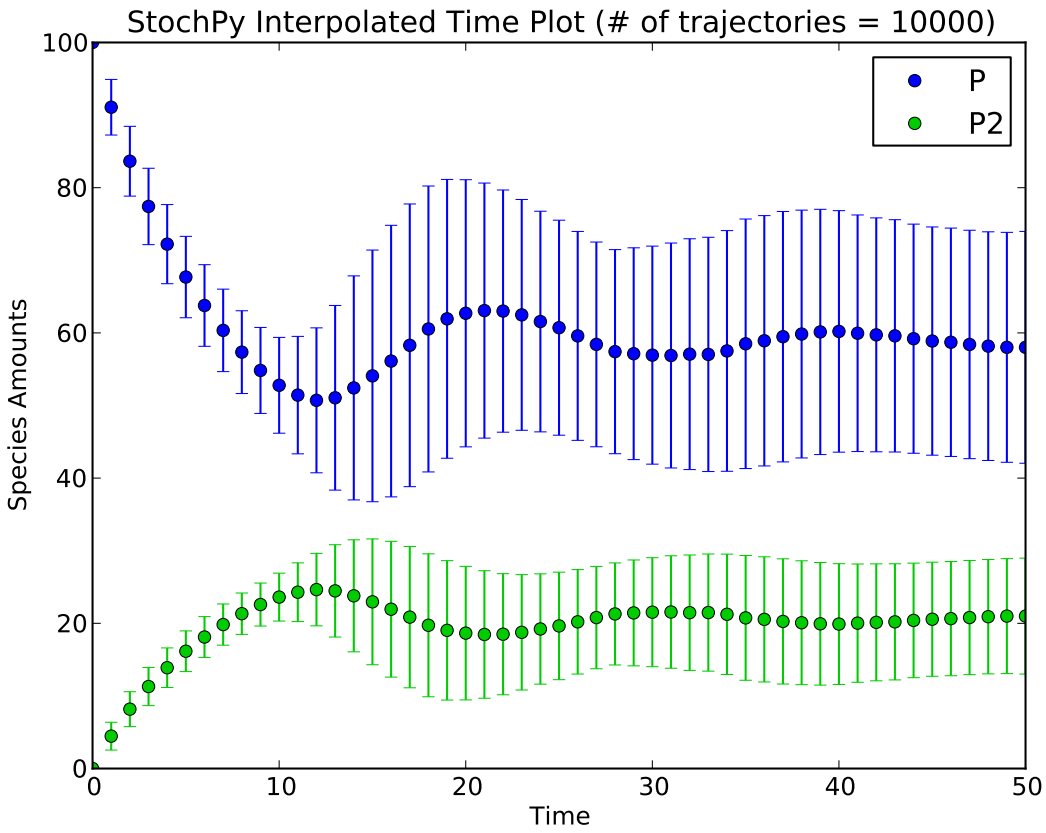
StochPy successfully reproduces the desired results, whereas there is one exception. StochPy converts species concentrations (HasOnlySubstanceUnits = True) to species amounts. Here, the species concentration is multiplied by the volume of the compartment. In the stochastic test suite results, a simulation is done with species concentrations, which gives a different simulation result.

Some examples (dsmts-001-07, dsmts-001-19, dsmts-003-03, and dsmts-003-04) are shown here. These models test on multiple species, assignment rules, time events, and species amount events respectively.




StochPy Interpolated Time Plot (# of trajectories = 10000)
X
y
Species Amounts
Time
0
50
100
150
200
250
0
10
20
30
40
50


StochPy Interpolated Time Plot (# of trajectories = 10000)
P
P2
Species Amounts
Time
0
20
40
60
80
100
0
10
20
30
40
50


StochPy Interpolated Time Plot (# of trajectories = 10000)
Species Amounts
Time
P
P2
0
20
40
60
80
100
0
10
20
30
40
50

CHAPTER

THREE

NUCLEOSOME MODIFICATION SIMULATIONS

As mentioned in the *Stochastic Simulation Algorithms section* StochPy contains four stochastic simulation algorithms. These algorithms can be used to perform simulations on all sorts of biochemical systems.

Usually, default time plots, distributions, propensities, and waiting times are sufficient analysis techniques. However, this is not always to case. Here, we demonstrate the flexibility of StochPy, because it is relatively easy to add your own modules or analysis techniques.

One example is the simulation of nucleosome modification models. Here, each nucleosome can carry several modifications. As a result, each nucleosome is described through several reactions, where each reaction gives information about a particular modification. Here, we are interested in for example the global pattern of these modifications. Therefore, default analysis techniques are insufficient. For this reasons, a nucleosome simulation module was built, which can be used interactively (make sure that StochPy is imported, which is shown in the *Start using StochPy section*):

```
>>> smod = stochpy.NucSim()
Welcome to the nucleosome modification simulation module
Info: The Direct method is selected to perform the simulations
Parsing file: /home/user/Stochpy/pscmodels/model1.psc
```

Now, it is possible to start a simulation with the default settings, but it is again possible to select for example another model or choose another number of time steps, just as is described in the *Stochastic Simulation Algorithms section*:

```
>>> smod.GetGapMeasure()
Gap Measure   0.993
>>> smod.PlotGlobalTimeSim()
>>> smod.PlotGlobalDistributions()
>>> smod.PlotPattern()
>>> smod.PlotStateTimes('M1')
```

Again, it is possible to plot information from not all species:

```
>>> smod.PlotGlobalTimeSim('M')
>>> smod.PlotGlobalTimeSim(['M','A'])
```

And to print the output or write it to a file:

```
>>> smod.PrintGlobalTimeSim()
>>> smod.PrintGlobalDistributions()
>>> smod.PrintPattern()
>>> smod.PrintStateTimes()
>>> smod.Write2File()
```

3.1 Nucleosome Model Builder

An average gene contains about 40 nucleosomes. Therefore, nucleosome modification models get enormously large. As a result, a nucleosome model builder module is created, which quickly builds the user-defined models:

```
>>> model = stochpy.NucModel()
>>> model.Build()
/home/user/Stochpy/pscmodels
The generated model is stored at /home/user/Stochpy/pscmodels/model1.psc
>>> help(model)
```

By default, a nucleosome model with 20 nucleosomes was built which can be considered as a small gene. A small gene was chosen, because the number of reactions explodes for larger number of nucleosomes and 20 nucleosomes was still large enough to get all sorts of biological phenomena. Further, 8 pre-defined models are build-in:

```
>>> model.ModelType(2)
ModelType:    2
```

Of course, the number of nucleosomes can be changed:

```
>>> model.ChangeN(10)
The number of nucleosomes is:         10
>>> model.Build()
/home/user/Stochpy/pscmodels
The generated model is stored at /home/user/Stochpy/pscmodels/model2.psc
```

Notice that the pre-defined models are build at the configuration step of StochPy, thus these are overwritten if new versions is build.

Also, landing zones interactively can be determined interactively:

```
>>> model.BuildLandingZones({'M',[10,11]})
```

Finally, several other high-level functions are developed:

```
>>> model.Recruitment()
Recruitment is activated
>>> model.Neighbours()
Info: Neighbour interactions are activated
>>> model.LongRangeInteractions()
Info: Long range neighbour interactions are activated"
```

```
Info: The default threshold is 7.
      Use Threshold(value) to change the threshold.
>>> model.ChangeThreshold(5)
Info: The Threshold is        5
>>> model.Decay()
Info: (Long range) neighbour interactions are activated
Info: The default threshold is 7.
      Use Threshold(value) to change the threshold.
Info: Decay effect is activated
```

Part IV

Installation and Configuration

CHAPTER

FOUR

INSTALLATION

The following software is required before installing StochPy:

- Python 2.x+ (http://www.python.org/download/releases/2.7.2/)
- NumPy 1.x+ (http://new.scipy.org/download.html)

The Following software is optional but recommended:

- Matplotlib
- libsbml
- libxml2

libsbml is necessary to convert models written in SBML format to the PySCeS MDL. The libsbml software requires a XML parser library which is libxml2 by default.

4.1 Windows

Use the windows installer (StochPy-1.0.0-linux-i686.exe) in directory: /StochPy-1.0.0/

4.2 Linux/MAC OS/Cygwin

In the command line, as root (sudo -s):

```
$ cd /.../StochPy-1.0.0
$ python setup install
```

CHAPTER

FIVE

CONFIGURATION

The StochPy package contains some example models, such as:

- Immigration-Death model
- Burst model
- Decaying-Dimerizing model
- Prokaryotic auto-regulation model

All these files are placed in the directory /home dir/Stochpy/pscmodels/ after the first usage of the package. All these models are written in the PySCeS MDL. This PySCeS MDL is explained in *the PySCeS Model Description Language section*. Of course, SBML models can be used as input. A SBML2PySCeS format converter is available which is used automatically if a SBML file is given as input.

Part V

The PySCeS Model Description Language

> PySCeS: the Python Simulator for Cellular Systems is an extendable toolkit for the analysis and investigation of cellular systems. It is available for download from: http://pysces.sf.net

PySCeS uses an ASCII text based *input file* to describe a cellular system in terms of it's stoichiometry, kinetics, compartments and parameters. Input files may have any filename with the single restriction that, for cross platform compatibility, they must end with the extension *.psc*. In this document we describe the PySCeS Model Description Language (MDL) which has been updated and extended for the PySCeS 0.7.x release.

> PySCeS is distributed under the PySCeS (BSD style) license and is made freely available as Open Source software. See LICENCE.txt for details.

We hope that you will enjoy using our software. If, however, you find any unexpected features (i.e. bugs) or have any suggestions on how we can improve PySCeS and specifically the PySCeS MDL please let us know.

CHAPTER

SIX

DEFINING A PYSCES MODEL

6.1 A kinetic model

The basic description of a kinetic model in the PySCeS MDL contains the following information:

- whether any fixed (boundary) species are present
- the reaction network stoichiometry
- rate equations for each reaction step
- parameter and boundary species initial values
- the initial values of the variable species

Although it is in principle possible to define an ODE based model without reactions or free species, for practical purposes PySCeS requires a minimum of a single reaction. Once this information is obtained it can be organised and written as a PySCeS input file. While this list is the minimum information required for a PySCeS input file the MDL allows the definition of advanced models that contain compartments, global units, functions, rate and assignment rules.

6.2 Model keywords

In PySCeS 0.7.x it is now possible to define keywords that specify model information. Keywords have the general form

```
<keyword>:  <value>
```

The *Modelname* (optional) keyword, containing only alphanumeric characters (or _), describes the model filename (typically used when the model is exported via the PySCeS interface module) while the *Description* keyword is a (short) single line model description.

```
Modelname: rohwer_sucrose1
Description: Sucrose metabolism in sugar cane (Johann M. Rohwer)
```

Two keywords are available for use (optional) with models that have one or more compartments defined. Both take a boolean (True/False) as their value:

- *Species_In_Conc* specifies whether the species symbols used in the rate equations represent a concentration (True, default) or an amount (False).
- *Output_In_Conc* tells PySCeS to output the results of numerical operations in concentrations (True, default) or in amounts (False).

```
Species_In_Conc: True
Output_In_Conc: False
```

More information on the effect these keywords have on the analysis of a model can be found in the PySCeS Reference Manual.

6.3 Global unit definition

PySCeS 0.7 supports the (optional) definition of a set of global units. In doing so we have chosen to follow the general approach used in the Systems Biology Modelling Language (SBML L2V3) specification. The general definition of a PySCeS unit is: `'<UnitType>: <kind>, <multiplier>, <scale>, <exponent>'` where *kind* is a string describing the base unit (for SBML compatibility this should be an SI unit) e.g. mole, litre, second or metre. The base unit is modified by the multiplier, scale and index using the following relationship: *<multiplier> * (<kind> * 10**<scale>)**<index>*. The default unit definitions are:

```
UnitSubstance: mole, 1, 0, 1
UnitVolume: litre, 1, 0, 1
UnitTime: second, 1, 0, 1
UnitLength: metre, 1, 0, 1
UnitArea: metre, 1, 0, 2
```

Please note that defining these values does not affect the numerical analysis of the model in any way.

6.4 Symbol names and comments

Symbol names (i.e. reaction, species, compartment, function, rule and parameter names etc.) must start with either an underscore or letter and be followed by any combination of alphanumeric characters or an underscore. Like all other elements of the input file names are case sensitive:

```
R1
_subA
par1b
ext_1
```

Explicit access to the "current" time in a time simulation is provided by the special symbol `_TIME_`. This is useful in the definition of events and rules (see chapter on advanced model construction for more details).

Comments can be placed anywhere in the input file in one of two ways, as single line comment starting with a # or as a multi-line triple quoted comment """<*comment*>""":

```
# everything after this is ignored

"""
This is a comment
spread over a
few lines.
"""
```

6.5 Compartment definition

By default (as is the case in all PySCeS versions < 0.7) PySCeS assumes that the model exists in a single unit volume compartment. In this case it is **not** necessary to define a compartment and the ODE's therefore describe changes in concentration per time. However, if a compartment is defined, PySCeS assumes that the ODE's describe changes in substance amount per time. Doing this affects how the model is defined in the input file (especially with respect to the definitions of rate equations and species) and the user is **strongly** advised to read the Users Guide before building models in this way. The compartment definition is as follows `Compartment: <name>, <size>, <dimensions>`, where <*name*> is the unique compartment id, <*size*> is the size of the compartment (i.e. length, volume or area) defined by the number of <*dimensions*> (e.g. 1,2,3):

```
Compartment: Cell, 2.0, 3
Compartment: Memb, 1.0, 2
```

6.6 Function definitions

A new addition to the PySCeS MDL is the ability to define SBML styled functions. Simply put these are code substitutions that can be used in rate equation definitions to, for example, simplify the kinetic law. The general syntax for a function is `Function: <name>, <args> {<formula>}` where <*name*> is the unique function id, <*arglist*> is one or more comma separated function arguments. The <*formula*> field, enclosed in curly brackets, may only make use of arguments listed in the <*arglist*> and therefore **cannot** reference model attributes directly. If this functionality is required a forcing function (assignment rule) may be what you are looking for.

```
Function: rmm_num, Vf, s, p, Keq {
Vf*(s - p/Keq)
}

Function: rmm_den, s, p, Ks, Kp {
s + Ks*(1.0 + p/Kp)
}
```

The syntax for function definitions has been adapted from Frank Bergmann and Herbert Sauro's "Human Readable Model Definition Language" (Draft 1).

6.7 Defining fixed species

Boundary species, also known as fixed or external species, are a special class of parameter used when modelling biological systems. The PySCeS MDL fixed species are declared on a single line as `FIX: <fixedlist>`. The <*fixedlist*> is a space separated list of symbol names which should be initialised like any other species or parameter:

```
FIX: Fru_ex Glc_ex ATP ADP UDP phos glycolysis Suc_vac
```

If no fixed species are present in the model then this declaration should be omitted entirely.

6.8 Reaction stoichiometry and rate equations

The reaction stoichiometry and rate equation are defined together as a single reaction step. Each step in the system is defined as having a name (identifier), a stoichiometry (substrates are converted to products) and rate equation (the catalytic activity, described in terms of species and parameters). All reaction definitions should be separated by an empty line. The general format of a reaction in a model with no compartments is:

```
<name>:
        <stoichiometry>
        <rate equation>
```

The <*name*> argument follows the syntax as discussed in a previous section, however, when more than one compartment has been defined it is important to locate the reaction in its specific compartment. This is done using the `@` operator:

```
<name>@<compartment>:
                      <stoichiometry>
                      <rate equation>
```

Where <*compartment*> is a valid compartment name. In either case this then followed either directly (or on the next line) by the reaction stoichiometry.

Each <*stoichiometry*> argument is defined in terms of reaction substrates, appearing on the left hand side and products on the right hand side of an identifier which labels the reaction as either reversible (=) or irreversible (>). If required each reagent's stoichiometric coefficient (PySCeS accepts both integer and floating point) should be included in curly braces *{}* immediately preceding the reagent name. If these are omitted a coefficient of one is assumed:

```
{2.0}Hex_P = Suc6P + UDP  # reversible reaction
Fru_ex > Fru              # irreversible reaction
species_5 > $pool         # a reaction to a sink
```

The PySCeS MDL also allows the use of the *$pool* token that represents a placeholder reagent for reactions that have no net substrate or product. Reversibility of a reaction is only used when

exporting the model to other formats (such as SBML) and in the calculation of elementary modes. It does not affect the numerical evaluation of the rate equations in any way.

Central to any reaction definition is the <*rate equation*> (SBML kinetic law). This should be written as valid Python expression and may fall across more than one line. Standard Python operators + − ⋆ / ⋆⋆ are supported (note the Python power e.g. *2^4* is written as *2**4*). There is no shorthand for multiplication with a bracket so *-2(a+b)^h* would be written as *-2*(a+b)**h}* and normal operator precedence applies:

+, -	addition, subtraction
*, /	multiplication, division
+x,-x	positive, negative
**	exponentiation

Operator precedence increase from top to bottom and left to right (adapted from the Python Reference Manual).

The PySCeS MDL parser has been developed to parse and translate different styles of infix into Python/Numpy based expressions, the following functions are supported in any mathematical expression:

- log, log10, ln, abs
- pow, exp, root, sqrt
- sin, cos, tan, sinh, cosh, tanh
- arccos, arccosh, arcsin, arcsinh, arctan, arctanh
- floor, ceil, ceiling, piecewise
- notanumber, pi, infinity, exponentiale

Logical operators are supported in rules, events etc but *not* in rate equation definitions. The PySCeS parser understands Python infix as well as libSBML and NumPy prefix notation.

- and or xor not
- > gt(x,y) greater(x,y)
- < lt(x,y) less(x,y)
- >= ge(x,y) geq(x,y) greater_equal(x,y)
- <= le(x,y) leq(x,y) less_equal(x,y)
- == eq(x,y) equal(x,y)
- != neq(x,y) not_equal(x,y)

Note that currently the MathML *delay and factorial* functions are not supported. Delay is handled by simply removing it from any expression, e.g. *delay(f(x), delay)* would be parsed as *f(x)*. Support for *piecewise* has been recently added to PySCeS and will be discussed in the *advanced features* section.

A reaction definition when no compartments are defined:

```
R5: Fru + ATP = Hex_P + ADP
    Fru/Ki5_Fru)*(Fru/Km5_Fru)*(ATP/Km5_ATP)/(1 +
    Vmax5/(1 + Fru/Ki5_Fru)*(Fru/Km5_Fru)*(ATP/Km5_ATP)/(1 +
    Fru/Km5_Fru + ATP/Km5_ATP + Fru*ATP/(Km5_Fru*Km5_ATP) +
    ADP/Ki5_ADP)
```

and using the previously defined functions:

```
R6:
   A = B
   rmm_num(V2,A,B,Keq2)/rmm_den(A,B,K2A,K2B)
```

When compartments are defined note how now the reaction is now given a location and that because the ODE's formed from these reactions must be in changes in substance per time the rate equation is multiplied by its compartment size. In this particular example the species symbols represent concentrations (*Species_In_Conc: True*):

```
R1@Cell:
    s1 = s2
    Cell*(Vf1*(s1 - s2/Keq1)/(s1 + KS1*(1 + s2/KP1)))
```

If *Species_In_Conc: True* the location of the species is defined when it is initialised and will be explained later in this manual. The following example shows the species symbols explicitly defined as amounts (*Species_In_Conc: False*):

```
R4@Memb: s3 = s4
    Memb*(Vf4*((s3/Memb) - (s4/Cell)/Keq4)/((s3/Memb)
    + KS4*(1 + (s4/Cell)/KP4)))
```

Please note that at this time we are not certain if this form of rate equation is translatable into valid SBML in a way that is interoperable with other software.

6.9 Species and parameter initialisation

The general form of any species (fixed, free) and parameter is simply:

```
property = value
```

Initialisations can be written in any order anywhere in the input file but for human readability purposes these are usually placed after the reaction that uses them or grouped at the end of the input file. Both decimal and scientific notation is allowed with the following provisions that neither floating point *(1.)* nor scientific shorthand *(1.e-3)* syntax should be used, instead use the full form *(1.0e-3)*, *(0.001)* or *(1.0)*.

Variable or free species are initialised differently depending on whether compartments are present in the model. While in essence the variables are set by the system parameters the

Although the variable species concentrations are determined by the parameters of the system, their initial values are used in various places, calculating total moiety concentrations (if present), time simulation initial values (e.g. time=zero) and as initial guesses for the steady-state algorithms. If an empty initial species pool is required it is not recommended to initialise

these values to zero (in order to prevent potential divide-by-zero errors) but rather to a small value (e.g. 10**-8).

For a model with no compartments these initial values assumed to be concentrations:

```
NADH = 0.001
ATP  = 2.3e-3
sucrose = 1
```

In a model with compartments it is expected that the species are located in a compartment (even if *Species_In_Conc: False*) this is done useing the @ symbol:

```
s1@Memb = 0.01
s2@Cell = 2.0e-4
```

A word of warning, the user is responsible for making sure that the units of the initialised species match those of the model. Please keep in mind that **all** species (and anything that depends on them) is defined in terms of the *Species_In_Conc* keyword. For example, if the preceding initialisations were for *R1* (see Reaction section) then they would be concentrations (as *Species_In_Conc: True*). However, in the next example, we are initialising species for *R4* and they are therefore in amounts (*Species_In_Conc: False*):

```
s3@Memb = 1.0
s4@Cell = 2.0
```

Fixed species are defined in a similar way and although technically a parameter, they should be given a location in compartmental models:

```
# InitExt
X0 = 10.0
X4@Cell = 1.0
```

However, fixed species are true parameters in the sense that their associated compartment size does not affect their value when it changes size. If compartment size dependent behaviour is required an assignment or rate rule should be considered.

Finally, the parameters should be initialised. PySCeS checks if a parameter is defined that is not present in the rate equations and if such parameter initialisations are detected a harmless warning is generated. If, on the other hand, an uninitialised parameter is detected a warning is generated and a value of 1.0 assigned:

```
# InitPar
Vf2 = 10.0
Ks4 = 1.0
```

CHAPTER

SEVEN

ADVANCED MODEL CONSTRUCTION

7.1 Assignment rules

Assignment rules or forcing functions are used to set the value of a model attribute before the ODE's are evaluated. This model attribute can either be a parameter used in the rate equations (this is traditionally used to describe an equilibrium block) a compartment or an arbitrary parameter (commonly used to define some sort of tracking function). Assignment rules can access other model attributes directly and have the generic form `!F <par> = <formula>`. Where <*par*> is the parameter assigned the result of <*formula*>. Assignment rules can be defined anywhere in the input file:

```
!F S_V_Ratio = Mem_Area/Vcyt
!F sigma_test = sigma_P*Pmem + sigma_L*Lmem
```

These rules would set the value of <*par*> which whose value can be followed with using the simulation and steady state extra_data functionality.

7.2 Rate rules

PySCeS now includes support for rate rules which are essentially directly encoded ODE's which are evaluated after the ODE's defined by the model stoichiometry and rate equations. Unlike the SBML rate rule, PySCeS allows one to access a reaction symbol in the rate rules (this is automatically expanded when the model is exported to SBML). The general form of a rate rule is `RateRule: <par> {<function>}`. Where <*name*> is the model attribute (e.g. compartment or parameter) whose rate of change is described by the <*formula*>. It may also be defined anywhere in the input file:

```
RateRule: Mem_Area {
(sigma_P)*(Mem_Area*k4*(P)) + (sigma_L)*(Mem_Area*k5*(L))
}

RateRule: Vcyt {(1.0/Co)*(R1()+(1-m1)*R2()+(1-m2)*R3()-R4()-R5())}
```

Remember to initialise any new parameters used in the rate rules.

7.3 Events

Time dependant events may now be defined whose definition follows the event framework described in the SBML L2V1 specification. The general form of an event is *Event: <name>, <trigger>, <delay> { <assignments> }*. As can be seen an event consists of essentially three parts, a conditional *<trigger>*, a set of one or more *<assignments>* and a *<delay>* between when the trigger is fired (and the assignments are evaluated) and the eventual assignment to the model. Assignments have the general form *<par> = <formula>*. Events have access to the "current" simulation time using the *_TIME_* symbol:

```
Event: event1, _TIME_ > 10 and A > 150.0, 0 {
V1 = V1*vfact
V2 = V2*vfact
}
```

The following event illustrates the use of a delay of ten time units as well as the prefix notation (used by libSBML) for the trigger (PySCeS understands both notations):

```
Event: event2, geq(_TIME_, 15.0), 10 {
V3 = V3*vfact2
}
```

Note: in order for PySCeS to handle events it is necessary to have the PySundials installed

7.4 Piecewise

Although technically an operator piecewise functions are sufficiently complicated to warrant their own section. A piecewise operator is essentially an *if, elif, ..., else* logical operator that can be used to conditionally "set" the value of some model attribute. Currently piecewise is supported in rule constructs and has not been tested directly in rate equation definitions. The piecewise function's most basic incarnation is *piecewise(<val1>, <cond>, <val2>)* which is evaluated as:

```
if <cond>:
    return <val1>
else:
    return <val2>
```

alternatively, *piecewise(<val1>, <cond1>, <val2>, <cond2>, <val3>, <cond3>)*:

```
if <cond1>:
    return <val1>
elif <cond2>:
    return <val1>
elif <cond3>:
    return <val3>
```

or *piecewise(<val1>, <cond1>, <val2>, <cond2>, <val3>, <cond3>, <val4>)*:

```
if <cond1>:
    return <val1>
elif <cond2>:
    return <val2>
elif <cond3>:
    return <val3>
else:
    return <val4>
```

can also be used. A "real-life" example of an assignment rule with a piecewise function:

```
!F Ca2plus=piecewise(0.1, lt(_TIME_,60), 0.1, gt(_TIME_,66.0115), 1)
```

In principle there is no limit on the amount of conditional statements present in a piecewise function, the condition can be a compound statements *a or b and c* and may include the *_TIME_* symbol.

7.5 Reagent placeholder

Some models contain reactions which are defined as only have substrates or products:

```
R1: A + B >

R2: > C + D
```

The implication is that the relevant reagents appear or disappear from or into a constant pool. Unfortunately the *PySCeS* parser does not accept such an unbalanced reaction definition and requires these pools to be represented as a `$pool` token:

```
R1: A + B > $pool

R2: $pool > C + D
```

`$pool` is neither counted as a reagent nor does it ever appear in the stoichiometry (think of it as dev/null) and no other $<str> tokens are allowed.

CHAPTER

EIGHT

EXAMPLE STOMPY INPUT FILES

8.1 Basic model definition

PySCeS test model *pysces_test_linear1.psc*:

```
FIX: x0 x3

R1: x0 = s0
    k1*x0 - k2*s0

R2: s0 = s1
    k3*s0 - k4*s1

R3: s1 = s2
    k5*s1 - k6*s2

R4: s2 = x3
    k7*s2 - k8*x3

# InitExt
x0 = 10.0
x3 = 1.0
# InitPar
k1 = 10.0
k2 = 1.0
k3 = 5.0
k4 = 1.0
k5 = 3.0
k6 = 1.0
k7 = 2.0
k8 = 1.0
# InitVar
s0 = 1.0
s1 = 1.0
s2 = 1.0
```

8.2 Advanced example

This model includes the use of *Compartments*, *KeyWords*, *Units* and *Rules*:

```
Modelname: MWC_wholecell2c
Description: Surovtsev whole cell model using J-HS Hofmeyr's framework

Species_In_Conc: True
Output_In_Conc: True

# Global unit definition
UnitVolume: litre, 1.0, -3, 1
UnitSubstance: mole, 1.0, -6, 1
UnitTime: second, 60, 0, 1

# Compartment definition
Compartment: Vcyt, 1.0, 3
Compartment: Vout, 1.0, 3
Compartment: Mem_Area, 5.15898, 2

FIX: N

R1@Mem_Area: N = M
   Mem_Area*k1*(Pmem)*(N/Vout)

R2@Vcyt: {244}M = P # m1
   Vcyt*k2*(M)

R3@Vcyt: {42}M = L # m2
   Vcyt*k3*(M)*(P)**2

R4@Mem_Area: P = Pmem
   Mem_Area*k4*(P)

R5@Mem_Area: L = Lmem
   Mem_Area*k5*(L)

# Rate rule definition
RateRule: Vcyt {(1.0/Co)*(R1()+(1-m1)*R2()+(1-m2)*R3()-R4()-R5())}
RateRule: Mem_Area {(sigma_P)*R4() + (sigma_L)*R5()}

# Rate rule initialisation
Co = 3.07e5 # uM p_env/(R*T)
m1 = 244
m2 = 42
sigma_P = 0.00069714285714285711
sigma_L = 0.00012

# Assignment rule definition
!F S_V_Ratio = Mem_Area/Vcyt
!F Mconc = (M)/M_init
```

```
!F Lconc = (L)/L_init
!F Pconc = (P)/P_init

# Assignment rule initialisations
M_init = 199693.0
L_init = 102004
P_init = 5303
Mconc = 1.0
Lconc = 1.0
Pconc = 1.0

# Species initialisations
N@Vout = 3.07e5
Pmem@Mem_Area = 37.38415
Lmem@Mem_Area = 8291.2350678770199
M@Vcyt = 199693.0
L@Vcyt = 102004
P@Vcyt = 5303

# Parameter initialisations
k1 = 0.00089709
k2 = 0.000182027
k3 = 1.7539e-010
k4 = 5.0072346e-005
k5 = 0.000574507164

"""
Simulate this model to 200 for maximum happiness and
watch the surface to volume ratio and scaled concentrations.
"""
```

This example illustrates almost all the new features included in the PySCeS MDL. Although it may be slightly more complicated than the basic model described above it is still, by our definition, certainly human readable.

Part VI

StochPy Module documentation

CHAPTER

NINE

STOCHASTIC SIMULATION MODULE

The main module of StochPy

Written by TR Maarleveld, Amsterdam, The Netherlands E-mail: tmd200@users.sourceforge.net Last Change: November 11, 2011

class `stochpy.StochSim.`**`SSA`**(*Method='Direct'*, *File=None*, *dir=None*, *Mode='steps'*, *End=1000*, *Trajectories=1*, *IsTestsuite=False*, *IsInteractive=True*, *IsTrackPropensities=False*, *IsRun=False*)

Input options:

- ***Method*** **[default = 'Direct']** Available methods: 'Direct', 'FirstReactionMethod','TauLeaping','Next Reaction Method'
- *File*: [default = ImmigrationDeath.psc]
- *dir*: [default = /home/user/stochpy/pscmodels/ImmigrationDeath.psc]
- *Mode*: [default = 'steps'] simulation for a total number of 'steps' or until a certain end 'time' (string)
- *End*: [default = 1000] end of the simulation (number of steps or end time) (float)
- *Trajectories*L [default = 1] (integer)
- *TrackPropensities*: [default = False] (Boolean)

Usage (with High-level functions): >>> mod = stochpy.SSA() >>> help(mod) >>> mod.Model(File = 'filename.psc', dir = '/.../') >>> mod.Method('Direct') >>> mod.Reload() >>> mod.Trajectories(5) >>> mod.Timesteps(10000) >>> mod.TrackPropensities() >>> mod.DoStochSim() >>> mod.DoStochSim(end=1000,mode='steps',trajectories=5,method='Direct') >>> mod.PlotTimeSim() >>> mod.PlotPropensities() >>> mod.PlotInterpolatedData() >>> mod.PlotWaitingtimes() >>> mod.PlotDistributions(bin_size = 3) >>> mod.ShowMeans() >>> mod.ShowStandardDeviations() >>> mod.ShowOverview() >>> mod.ShowSpecies() >>> mod.DoTestsuite()

`DeleteTempfiles`()

Deletes all .dat files

DoStochSim(*end=False*, *mode=False*, *method=False*, *trajectories=False*, *epsilon=0.03*)

doStochSim(end=10, mode='steps', method='Direct',trajectories = 1,epsilon = 0.03)

Run a stochastic simulation for until *end* is reached. This can be either time steps or end time (which could be a *HUGE* number of steps).

Input:

- *end* [default=1000] simulation end (steps or time)
- *mode* [default='steps'] simulation mode, can be one of: - *steps* total number of steps to simulate - *time* simulate until time is reached
- *method* [default='Direct'] stochastic algorithm, can be one of: - Direct - FirstReactionMethod - NextReactionMethod - TauLeaping
- *trajectories* [default = 1] number of trajectories
- *epsilon* [default = 0.03] parameter for Tau-Leaping

DoTestsuite(*epsilon_=0.01*, *sim_trajectories=1000*)

Do 10000 simulations until t=50 and print the interpolated result for t = 0,1,2,...,50

Input:

- *epsilon_* [default = 0.01]: useful for Tau-Leaping simulations (float)

Endtime(*t*)

Set the end time of the exact realization of the Markov jump process

Input:

- *t*: end time (float)

GetInterpolatedData()

Perform linear interpolation for each generated trajectory. Linear interpolation is done for all integer time points, between the start time (0) end the endtime.

GetMeanWaitingtimes()

Get the mean waiting times for the selected trajectory

GetMeans()

Get the means of each species for the selected trajectory

GetStandardDeviations()

Get the standard deviations of each species for the selected trajectory

GetTrajectoryData(*n=1*)

Switch to another trajectory, by default, the last trajectory is accesible

Input:

- *n*: [default = 1] get data from a certain trajectory

GetWaitingtimes()

Get for each reaction the waiting times

MeanWaitingtimes()
old version (StochPy 0.9)

Method(*method*)

Input:

- *method*

Select one of the following four methods:

- *Direct*
- *FirstReactionMethod*
- *NextReactionMethod*
- *TauLeaping*

Note: input must be a string –> 'Direct'

Mode(*sim_mode='steps'*)
Run a stochastic simulation for until *end* is reached. This can be either time steps or end time (which could be a *HUGE* number of steps).

Input:

- *sim_mode*: [default = 'steps'] 'time' or 'steps'
- *end*: [default = 1000]

Model(*File*, *dir=None*)
High-level function to determine the model which can be used for stochastic simulations

Input:

- *File*:'filename.psc' (string)
- *dir*: [default = None] the directory where File is located (string)

PlotDistributions(*species2plot=True*, *linestyle='dotted'*, *colors=None*, *title='StochPy Probability Density Function'*, *bin_size=1*)
Plots the PDF for each generated trajectory Default: PlotDistributions() plots PDF for each species

Input:

- *species2plot* [default = True] as a list ['S1','S2']
- *linestyle* [default = 'dotted'] (string)
- *colors* (list)
- *title* [default = 'StochPy Probability Density Function'] (string)
- *bin_size* [default=1] (integer)

PlotInterpolatedData(*species2plot=True*, *linestyle='None'*, *marker='o'*, *colors=None*, *title='StochPy Interpolated Time Plot (# of trajectories =)'*)

Plot the averaged interpolation result. For each time point, the mean and standard deviation are plotted Input:

•*species2plot* [default = True] as a list ['S1','S2']

•*linestyle* [default = 'dotted'] dashed, solid, and dash_dot (string)

•*marker* [default = ','] ('v','o','*')

•*colors*: [default = None] (list)

•*title* [default = StochPy Interpolated Time (# of trajectories = ...)] (string)

PlotPropensities(*rates2plot=True*, *linestyle='solid'*, *marker=''*, *colors=None*, *title='StochPy Propensities Plot'*)

Plot time simulation output for each generated trajectory

Default: PlotPropensities() plots propensities for each species

Input:

- *rates2plot* [default = True]: species as a list ['S1','S2']
- *marker* [default = ''] ('v','o','*',',')
- *linestyle* [default = 'solid']: dashed, dotted, and solid (string)
- *colors* [default = None] (list)
- *title* [default = 'StochPy Propensities Plot'] (string)

PlotTimeSim(*species2plot=True*, *linestyle='solid'*, *marker=''*, *colors=None*, *title='StochPy Time Simulation Plot'*)

Plot time simulation output for each generated trajectory Default: PlotTimeSim() plots time simulation for each species

Input:

- *species2plot*: [default = True] as a list ['S1','S2']
- *linestyle*: [default = 'solid'] dashed, solid, and dash_dot (string)
- *marker* [default = ''] ('v','o','*',',')
- *title*: [default = 'StochPy Time Simulation Plot'] (string)

PlotWaitingtimes(*rates2plot=True*, *linestyle='None'*, *marker='o'*, *colors=None*, *title='StochPy Waitingtimes Plot'*)

Plot obtained waiting times default: PlotWaitingtimes() plots waiting times for all rates

Input:

- *rates2plot*: [default = True] as a list of strings ["R1","R2"]
- *linestyle*: [default = 'None'] dashed, dotted, dash_dot, and solid (string)
- *marker* [default = 'o'] ('v','o','*',',')

- *colors*: [default = None] (list)
- *title* [default = 'StochPy Waitingtimes Plot'] (string)

PrintDistributions()
Print obtained distributions for each generated trajectory

PrintInterpolatedData()
Analyse the interpolated output for each generated trajectory

PrintMeanWaitingtimes()
Print the mean waiting times for the selected trajectory

PrintPropensities()
Print time simulation output for each generated trajectory

PrintTimeSim()
Print time simulation output for each generated trajectory

PrintWaitingtimes()
Print obtained waiting times

Reload()
Reload the entire model again. Useful if the model file has changed

Run(*end=False*, *mode=False*, *method=False*, *trajectories=False*)
Old version

ShowMeans()
Print the means of each species for the selected trajectory

ShowOverview()
Print an overview of the current settings

ShowSpecies()
Print the species of the model

ShowStandardDeviations()
Print the standard deviations of each species for the selected trajectory

Timesteps(*s*)
Set the number of time steps to be generated for each trajectory

Input:

- *s*: number of time steps (integer)

TrackPropensities(*boolean=True*)
Track the propensities through time

Input:

- *boolean*: [default = True]

Trajectories(*traj*)
Set the number of trajectories to be generated

Input:

- *traj*: number of trajectories (integer)

Write2File(*what='TimeSim'*, *directory=None*)
Write output to a file

Input:

- *what*: [default = TimeSim] TimeSim, Propensities, Distributions, Waiting-times, and Interpol (string)
- *directory*: [default = None] (string)

stochpy.StochSim.**usage**()

CHAPTER

TEN

DIRECT METHOD

This program performs the direct Stochastic Simulation Algorithm from Gillespie (1977) [1].This algorithm is used to generate exact realizations of the Markov jump process. Of course, the algorithm is stochastic, so these realizations are different for each run. Only molecule populations are specified. Positions and velocities, such as in Molecular Dynamics (MD) are ignored. This makes the algorithm much faster, because non-reactive molecular collisions can be ignored. Still, this exact SSA is quite slow, because it insists on simulating every individual reaction event, which takes a lot of time if the reactant population is large. Furthermore, even larger problems arise if the model contains distinct processes operating on different time-scales [2].

[1] Gillespie D.T (1977), "Exact stochastic simulation of coupled chemical reactions", J.Phys. Chem. 81:2340-2361 [2] Wilkinson D.J (2009), "Stochastic Modelling for quantitative description of heterogeneous biological systems", Nat Rev Genet; 0(2):122-133

class `stochpy.DirectMethod.`**`DirectMethod`**(*File*, *dir*, *OutputDir*, *TempDir*)

Direct Stochastic Simulation Algorithm from Gillespie (1977) [1].

This algorithm is used to generate exact realizations of the Markov jump process. Of course, the algorithm is stochastic, so these realizations are different for each run.

[1] Gillespie D.T (1977), "Exact stochastic simulation of coupled chemical reactions", J.Phys. Chem. 81:2340-2361

Input:

- *File*: filename.psc
- *dir*: /home/user/Stochpy/pscmodels/filename.psc
- *OutputDir*: /home/user/Stochpy/
- *TempDir*

`AssignmentRules`(*timestep*)

Builds the assignment rules

Input: -*timestep*: (integer)

`DoEvent`()

Do the event of the model

Execute(*Trajectories*, *Endtime*, *Timesteps*, *TrackPropensities*)
Generates T trajectories of the Markov jump process.

Input:

- *Trajectories*: (integer)
- *Endtime* (float)
- *Timesteps* (integer)
- *TrackPropensities*: (boolean)

Endtime or Timesteps is infinite

FillDataStochsim()
Put all simulation data in the data object data_stochsim

GetDistributions()
Get means, standard deviations and the probability at each species amount value

GetEventAtAmount()
Get amount where events happen

GetEventAtTime()
Get times where events happen

Initial_Conditions()
This function initiates the output format with the initial concentrations

MonteCarlo()
Monte Carlo step to determine tau

Parse(*File*, *dir*)
Parses the PySCeS MDL input file, where the model is desribed

Input:

- *File*: filename.psc
- *dir*: /home/user/Stochpy/pscmodels/filename.psc

Propensities()
Determines the propensities to fire for each reaction at the current time point. At t=0, all the rate equations are compiled.

ReactionExecution()
Function that executes the selected reaction that will fire once

ReactionSelection()
Function which selects a reaction that will fire once

Run()
Calculates a time step of the Direct Method

rateFunc(*rate_eval_code*, *r_vec*)
Calculate propensities from the compiled rate equations

Input:

- *rate_eval_code*: compiled rate equations
- *r_vec*: output for the calculated propensities

class stochpy.DirectMethod.**Species**

CHAPTER

ELEVEN

FIRST REACTION METHOD

This module performs the first reaction method Stochastic Simulation Algorithm from Gillespie (1977).

This algorithm is used to generate exact realizations of the Markov jump process. Of course, the algorithm is stochastic, so these realizations are different for each run.

Only molecule populations are specified. Positions and velocities, such as in Molecular Dynamics (MD) are ignored. This makes the algorithm much faster, because non-reactive molecular collisions can be ignored.different Still, this exact SSA is quite slow, because it insists on simulating every individual reaction event, which takes a lot of time if the reactant population is large. Furthermore, even larger problems arise if the model contains distinct processes operating on different time-scales.

Written by TR Maarleveld, Amsterdam, The Netherlands E-mail: tmd200@users.sourceforge.net Last Change: 16 November, 2011

class `stochpy.FirstReactionMethod.`**`FirstReactionMethod`**(*File*, *dir*, *OutputDir*, *TempDir*)

First Reaction Method from Gillespie (1977)

This algorithm is used to generate exact realizations of the Markov jump process. Of course, the algorithm is stochastic, so these realizations are different for each run.

Input:

- *File*: filename.psc
- *dir*: /home/user/Stochpy/pscmodels/filename.psc
- *OutputDir*: /home/user/Stochpy/
- *TempDir*

`AssignmentRules`(*timestep*)

Builds the assignment rules Input:

-*timestep*: integer

`DoEvent`()

Do the event of the model

Execute(*Trajectories*, *Endtime*, *Timesteps*, *TrackPropensities*)
Generates T trajectories of the Markov jump process.

Input:

- *Trajectories*: (integer)
- *Endtime* (float)
- *Timesteps* (integer)
- *TrackPropensities*: (boolean)

Endtime or Timesteps is infinite

FillDataStochsim()
Put all simulation data in the data object data_stochsim

GetDistributions()
Get means, standard deviations and the probability at each species amount value

GetEventAtAmount()
Get amount where events happen

GetEventAtTime()
Get times where events happen

InitialStep()
Monte Carlo step to determine all taus

Initial_Conditions()
This function initiates the output format with the initial concentrations

Parse(*File*, *dir*)
Parses the PySCeS MDL input file, where the model is desribed

Input:

- *File*: filename.psc
- *dir*: /home/user/Stochpy/pscmodels/filename.psc

Propensities()
Determines the propensities to fire for each reaction at the current time point. At t=0, all the rate equations are compiled.

ReactionExecution()
Function that executes the selected reaction that will fire once

ReactionSelection()
Function which selects a reaction that will fire once

Run()
Perform a direct SSA time step and pre-generate M random numbers

rateFunc(*rate_eval_code*, *r_vec*)
Calculate propensities from the compiled rate equations

Input:

- *rate_eval_code*: compiled rate equations
- *r_vec*: output for the calculated propensities

class stochpy.FirstReactionMethod.**Species**

CHAPTER

TWELVE

NEXT REACTION METHOD

This module performs the Next Reaction Method from Gibson and Bruck [1]. Therefore, it is also called the Gibson and Bruck algorithm.

[1] M.A. Gibson and J. "Bruck Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels", J. Phys. Chem., 2000,104,1876-1889

Written by TR Maarleveld, Amsterdam, The Netherlands E-mail: tmd200@users.sourceforge.net Last Change: November 16, 2011

class `stochpy.NextReactionMethod.`**`NextReactionMethod`**(*File*, *dir*, *OutputDir*, *TempDir*)

Next Reaction Methfod from Gibson and Bruck [1].

Input:

- *File*: filename.psc
- *dir*: /home/user/Stompy/pscmodels/filename.psc
- *Outputdir*: /home/user/Stompy/

[1] M.A. Gibson and J. "Bruck Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many

Channels", J. Phys. Chem., 2000,104,1876-1889

`AssignmentRules`(*timestep*)
Builds the assignment rules

Input: *-timestep*: (integer)

`BuildInits`()
Build initials that are necessary to generate a trajectory

`DoEvent`()
Do the event of the model

`Execute`(*Trajectories*, *Endtime*, *Timesteps*, *TrackPropensities*)
Generates T trajectories of the markov jump process.

Input:

- *Trajectories*: (integer)
- *Endtime* (float)
- *Timesteps* (integer)
- *TrackPropensities*: (boolean)

Endtime or Timesteps is infinite

FillDataStochsim()
Put all simulation data in the data object data_stochsim

GetDistributions()
Get means, standard deviations and the probability at each species amount value

GetEventAtAmount()
Get amount where events happen

GetEventAtTime()
Get times where events happen

InitialStep()
Monte Carlo step to determine all taus and to create a binary heap

Initial_Conditions()
This function initiates the output format with the initial concentrations

Parse(*File*, *dir*)
Parses the PySCeS MDL input file, where the model is desribed

Input:

- *File*: filename.psc
- *dir*: /home/user/Stompy/pscmodels/filename.psc

Propensities()
Determines the propensities to fire for each reaction at the current time point. At t=0, all the rate equations are compiled.

ReactionExecution()
Function that executes the selected reaction that will fire once

ReactionSelection()
Function which selects a reaction that will fire once

Run()
Perform a direct SSA time step and pre-generate M random numbers

UpdateHeap()
This function calculates new tau values for reactions that are changed. After this calculations, the binary heap is updated, which is much more efficient then rebuilding the entire heap.

rateFunc(*rate_eval_code*, *r_vec*)
Calculate propensities from the compiled rate equations

Input:

- *rate_eval_code*: compiled rate equations
- *r_vec*: output for the calculated propensities

class stochpy.NextReactionMethod.**Species**

CHAPTER

THIRTEEN

OPTIMIZED TAU-LEAPING

This program performs Optimized Explicit Tau-leaping algorithm, which is an approximate version of the exact Stochastic Simulation Algorithm (SSA). Here, an efficient step size selection procedure for the tau-leaping method [1] is used.

[1] Cao. Y, Gillespie D., Petzold L. (2006), "Efficient step size selection for the tau-leaping method", J.Chem. Phys. 28:124-135

Written by TR Maarleveld, Amsterdam, The Netherlands E-mail: tmd200@users.sourceforge.net Last Change: November 16, 2011

`stochpy.TauLeaping.`**`DetermineOrderHOR`**(*rate_vector*, *reactants*)

Determines once the order of each reaction and the highest order of reaction (HOR) for each species.

Input:

- *Rate_vector*: (list)
- *Reactants*: (nested list)

Output:

- *orders*
- *HORs*
- *HO_info*

`stochpy.TauLeaping.`**`GetSample`**(*probs*)

This function extracts a sample from a list of probabilities. The 'extraction chance' is identical to the probability of the sample.

Input:

- *probs*: (list)

Output:

- *sample*
- *sample index*

`stochpy.TauLeaping.`**`MinPositiveValue`**(*List*)

This function determines the minimum positive value

Input:

- *List*

Output:

- *minimum positive value*

class stochpy.TauLeaping.**OTL**(*File*, *dir*, *OutputDir*, *TempDir*)

Input:

- *File*: filename.psc
- *dir*: /home/user/Stochpy/pscmodels/filename.psc
- *Outputdir*: /home/user/Stochpy/

AssignmentRules(*timestep*)
Builds the assignment rules

Input: *-timestep*: integer

CriticalReactions()
Determines the critical reactions (as a boolean vector)

DetermineMethod()
Determines for each time step what to perform: exact of approximate SSA

DoEvent()
Do the event of the model

Execute(*Trajectories*, *Endtime*, *Timesteps*, *TrackPropensities*, *epsilon=0.03*)
Generates T trajectories of the Markov jump process.

Input:

- *Trajectories* (integer)
- *Endtime* (float)
- *Timesteps* (integer)
- *TrackPropensities* (boolean)
- *epsilon* [default = 0.03] (float)

Endtime or Timesteps is infinite

Execute_K_Reactions()
Perform the determined K reactions

FillDataStochsim()
Put all simulation data in the data object data_stochsim

GetA0c()
Calculate the total propensities for all critical reactions

GetDistributions()
Get means, standard deviations and the probability at each species amount value

GetEventAtAmount()
Get amount where events happen

GetEventAtTime()
Get times where events happen

GetG(*orders*, *hors*, *hor_info*)
Determine the G-vector

Input:

- *orders*
- *hors*: highest order of reaction for each species
- *hor_info*

GetK()
Determines the K-vector, which describes the number of firing reactions for each reaction.

GetMuVar()
Calculate the estimates of mu and var for each species (i)

GetTauPrime()
Calculate tau'

GetTauPrimePrime()
Calculate Tau''

Initial_Conditions()
This function initiates the output format with the initial concentrations

MonteCarlo()
Monte Carlo step to determine tau

Parse(*File*, *dir*)
Parses the PySCeS MDL input file, where the model is described

Input:

- *File*: filename.psc
- *dir*: /home/user/Stochpy/pscmodels/filename.psc

Propensities()
Determines the propensities to fire for each reaction at the current time point. At t=0, all the rate equations are compiled.

ReactionExecution()
Function that executes the selected reaction that will fire once

ReactionSelection()
Function which selects a reaction that will fire once

RunExactSSA()
Perform a direct method SSA time step

rateFunc(*rate_eval_code*, *r_vec*)

Calculate propensities from the compiled rate equations

Input:

- *rate_eval_code*: compiled rate equations
- *r_vec*: output for the calculated propensities

class stochpy.TauLeaping.**Species**

CHAPTER

FOURTEEN

ANALYSIS

This module provides functions for Stochastic Simulation Algorithms Analysis (SSA). Implemented SSAs import this module to perform their analysis. It plots time simulations, distributions and waiting times.

Written by TR Maarleveld, Amsterdam, The Netherlands E-mail: tmd200@users.sourceforge.net Last Change: November 15, 2011

stochpy.modules.Analysis.**Binning**(*x*, *y*, *bin_size*)

Binning of the PDF

Input:

- *x*: list of x-values
- *y*: list of probabilities for each x[i]
- *bin_size*: (integer)

stochpy.modules.Analysis.**Count**(*data_*, *edges_*)

Input:

- *data_*
- *edges_*

class stochpy.modules.Analysis.**DoPlotting**(*species_labels*, *rate_labels*, *plotnum=1*)

This class initiates the plotting options.

Input:

- *species_labels*: [S1,S2, ..., Sn]
- *rate_labels*: [R1, R2, ..., Rm]

AverageTimeSimulation(*means_set*, *sds_set*, *time*, *species2plot*, *linestyle*, *marker_*, *colors*, *title*)

Plots the interpolated time simulation results. Makes use of the ObtainInterpolationResults function, which determines the input for this function out of the SSA output.

Input:

- *means_set*: (nested list)
- *sds_set*: (nested list)
- *time*: (list)
- *linestyle*: (string)
- *title* (string)

Distributions(*distributions*, *species2plot*, *traj_index*, *linestyle*, *colors*, *title*, *bin_size*)

Plots the distributions of the simulated metabolites/molecules.

Input:

- *distributions* (nested list)
- *species2plot* (list)
- *traj_index* (integer)
- *colors* (list)
- *title* (string)

Propensities(*data*, *rates2plot*, *traj_index*, *linestyle*, *marker*, *colors*, *title*)

Tracks the propensities through time

Input:

- *data*: (array)
- *rates2plot* (list)
- *traj_index* (integer)
- *linestyle* (string)
- *title* (string)

ResetPlotnum()

Reset figure numbers if trajectories > 1

TimeSimulation(*data*, *species2plot*, *traj_index*, *linestyle*, *marker*, *colors*, *title*)

Time simulation plot

Input:

- *data*: (array)
- *species2plot* (list)
- *traj_index* (integer)
- *linestyle* (string)
- *marker* string)
- *colors* (list)
- *title* (string)

Waitingtimes(*waiting_times*, *rates2plot*, *traj_index*, *linestyle*, *marker*, *colors*, *title*)

Plots the waiting times for each reaction in the model. Makes use of ObtainWaitingtimes to derive the waiting times out of the SSA output.

Input:

- *waiting_times* (dict)
- *rates2plot* (list)
- *traj_index* (integer)
- *linestyle* (string)
- *title* (string)

stochpy.modules.Analysis.**LogBin**(*data*, *factor*)

Function that creates log bins

Input:

- *data*: list
- *factor* : determines the width of the bins (float)

Output:

- *x*: x-values (list)
- *y*: y-values (list)
- *nbins*: (integer)

stochpy.modules.Analysis.**ObtainInterpolationResults**(*interpolated_output*, *points*, *endtime*)

Gets the interpolated output after interpolation

Input:

- *interpolated_output*: (nested list)
- *points*: list of integer time points of interpolation

stochpy.modules.Analysis.**ObtainWaitingtimes**(*data_stochsim*, *num_reactions*)

This function extracts the waiting times for each reaction of the model from the used SSA output.

Input:

- *data_stochsim*: data object that stores all simulation data
- *num_reactions*: number of reactions (integer)

output:

- *waiting times*: nested list

Note: It is impossible to use this function in combination with the Tau-leaping method, because the Tau-Leaping results are not exact!

stochpy.modules.Analysis.**getDataForTimeSimPlot**(*data*)

CHAPTER

FIFTEEN

INDEXED PRIORITY QUEUE (IPQ)

This module builds a index priority queue (IPQ) - Heap - which contains a lot of usefull functions that can manipulate this heap so that it is not necessary to rebuild the whole heap if something changes.

The Next Reaction Method from Gibson&Bruck uses such a heap to optimize the speed of the exact stochastic simulation algorithm, which goes to log2(number of reactions) if the heap is sparse.

Written by TR Maarleveld, Amsterdam, The Netherlands E-mail: tmd200@users.sourceforge.net Last Change: January 22, 2010

class `stochpy.modules.Heap.`**`BinaryHeap`**(*taus*)

This class builds and manipulates a binary tree

`GetChildren`()

Determine the possible children from a certain node. It returns the indices of the children

`GetParent`()

Determine the parents from a certain node. Note that it returns the indices of the parents

`Print`()

Print binary tree

`Swap`(*r1*, *r2*)

Swaps two nodes in the binary tree with indices r1 and r2

`Update_AUX`(*tau_node*, *j*)

Update the binary tree. This function uses GetParent,GetChildren and Swap to change the positions of nodes in the tree, so it does not build a new tree, but updates some of the nodes, which is an advantage if the tree is sparse.

Input:

- *tau*: updated tau value
- *j*: reaction number

Note: This function is far from optimized

stochpy.modules.Heap.**show_tree**(*tree*, *total_width=36*, *fill=' '*)

Prints a tree

Input:

- *tree*
- *total_width* [default = 36]
- *fil* [default = ' ']

CHAPTER

SIXTEEN

DNORM

Module which contains functions that can calculate the density for the normal distribution for given x-values and it has the ability to normalize these densities.

See http://mathworld.wolfram.com/NormalDistribution.html/ for more information about the normal distribution

Written by TR Maarleveld, Amsterdam, The Netherlands E-mail: tmd200@users.sourceforge.net Last Change: Augustus 10, 2010

`stochpy.modules.dnorm.`**`dnorm`**(*X*, *mu=0*, *sigma=1.5*)

Named after the dnorm function from programming language R. Density generation for the normal distribution with mean = mu and standard deviation = sigma. Input:

- *x* : vector or integer of 'x-axis' values
- *mu* [default = 0]
- *sigma* [defaul = 1.5]

Output:

- float or list with floats containing the calculated values

`stochpy.modules.dnorm.`**`normalized_dnorm`**(*X*, *mu=0*, *sigma=1.5*)

Generate normalized values based on the maximum of the bell-shaped normal distribution. It uses the self defined dnorm function to calculate these values. Input:

- *x* : vector or integer of 'x-axis' values
- *mu* [default = 0]
- *sigma* [defaul = 1.5]*

Output:

- float or list with floats containing the calculated values

CHAPTER

SEVENTEEN

PYSCES MDL PARSER

The PySCeS parser is used to import a model written in the MDL of PySCeS. Further, all required input do to stochastic simulations is build.

Written by Timo Maarleveld, Amsterdam, The Netherlands E-mail: tmd200@users.sourceforge.net Last Change: October 21, 2011

class stochpy.PyscesMiniModel.**Function**(*name*, *mod*)
Function class ported from Core2 to enable the use of functions in PySCeS

addFormula(*formula*)

setArg(*var*, *value=None*)

class stochpy.PyscesMiniModel.**IntegrationStochasticDataObj**
This class is specifically designed to store the results of a stochastic time simulation It has methods for setting the Time, Labels, Species and Propensity data and getting Time, Species and Rate (including time) arrays. However, of more use:

- •getOutput(*args) feed this method species/rate labels and it will return an array of [time, sp1, r1, ...]
- •getDataAtTime(time) the data generated at time point "time".
- •getDataInTimeInterval(time, bounds=None) more intelligent version of the above returns an array of all data points where: time-bounds <= time <= time+bounds

getAllSimData(*lbls=False*)
Return an array of time + all available simulation data

Input:

- *lbls* [default=False] return only the data array or (data array, list of labels)

getDataAtTime(*time*)
Return all data generated at "time"

Input:

- *time* the required exact time point

getDataInTimeInterval(*time*, *bounds=None*)
Returns an array of all data in interval: time-bounds <= time <= time+bounds where bound defaults to step size

Input:

- *time* the interval midpoint
- *bounds* [default=None] interval half span defaults to step size

getPropensities(*lbls=False*)
Return time+propensity array

Input:

- *lbls* [default=False] return only the time+propensity array or optionally both the data array and a list of column label

getSimData(**args*, ***kwargs*)
Feed this method species/xdata labels and it will return an array of [time, sp1, ...]

Input:

- 'speces_l', 'xdatal' ...
- *lbls* [default=False] return only the data array or (data array, list of labels)

getSpecies(*lbls=False*)
Return an array fo time+species

Input: - *lbls* [default=False] return only the time+species array or optionally both the data array and a list of column label

getTime(*lbls=False*)
Return the time vector

Input:

- *lbls* [default=False] return only the time array or optionally both the time array and time label

getWaitingtimes(*lbls=False*, *traj=[]*)
Return waiting times, time+waiting_time array

Input:

- *lbls* [default=False] return only the time+waiting_time array or optionally both the data array and a list of column label
- *traj* [default=[0]] return the firs or trajectories defined in this list

getXData(*lbls=False*)
Return time+xdata array

Input: - *lbls* [default=False] return only the time+xdata array or optionally both the data array and a list of column label

setDist(*distributions*, *means*, *sds*)
setDist stuff for the determination of distributions

setFiredReactions(*fired_reactions*)
Set the reactions that fired Input:

•*fired_reactions* a list of fired reactions

setLabels (*species*)
Set the species Input:

•*species* a list of species labels

setMeanWaitingtimes (*waiting_times*)
///

setPropensities (*propensities*)
Sets an array of propensities.

Input:

- *propensities* a list of propensities

setPropensitiesLabels (*labels*)

setSimulationInfo (*timesteps*, *endtime*, *simulation_trajectory*)
setSimulationInfo

setSpecies (*species*, *lbls=None*)
Set the species array Input:

•*species* an array of species vs time data

•*lbls* [default=None] a list of species labels

setTime (*time*, *lbl=None*)
Set the time vector

Input"

- *time* a 1d array of time points
- *lbl* [default=None] is "Time" set as required

setWaitingtimes (*waiting_times*, *lbls=None*)
Set the *waiting_times* this data structure is not an array but a nested list of: waiting time log bins per reaction per trajectory:

waiting_times = [traj_1, ..., traj_n] traj_1 = [wt_J1, ..., wt_Jn] # in order of SSA_REACTIONS wt_J1 = (xval, yval, nbin) xval =[x_1, ..., x_n] yval =[y_1, ..., y_n] nbin = n

Input:

- *waiting_times* a list of waiting times
- *lbls* [default=None] a list of matching reaction names

setXData (*xdata*, *lbls=None*)
Sets an array of extra simulation data

Input: - *xdata* an array of xdata vs time - *lbls* [default=None] a list of xdata labels

class stochpy.PyscesMiniModel.**InterpolatedDataObj**

class stochpy.PyscesMiniModel.**NewCoreBase**
Core2 base class, needed here as we use Core2 derived classes in PySCeS

get(*attr*)
Return an attribute whose name is str(attr)

getName()

setName(*name*)

class stochpy.PyscesMiniModel.**PySCeS_Connector**(*ModelFile*, *ModelDir*, *IsNRM=False*)

BuildDependencyGraph()
Function which builds a dependency graph

BuildReactions()
Extract information out of each reaction, such as what are the reagents/reactants and which parameter is used for that particular reaction.

BuildX()
Builds the initial concentrations of all species (X).

DetermineAffects()
Determine the affects for each reaction

class stochpy.PyscesMiniModel.**PyscesInputFileParser**(*File*, *dir*, *output_dir=None*)
This class contains the PySCeS model loading

InitialiseInputFile()
Parse the input file associated with the PySCeS model instance and assign the basic model attributes

buildN()
Generates the stoichiometric matrix N from the parsed model description. Returns a stoichiometric matrix (N) as a NumPy array

PySCeS - Python Simulator for Cellular Systems (http://pysces.sourceforge.net)

Copyright (C) 2004-2009 B.G. Olivier, J.M. Rohwer, J.-H.S Hofmeyr all rights reserved,

Brett G. Olivier (bgoli@users.sourceforge.net) Triple-J Group for Molecular Cell Physiology Stellenbosch University, South Africa.

Permission to use, modify, and distribute this software is given under the terms of the PySceS (BSD style) license. See LICENSE.txt that came with this distribution for specifics. pys NO WARRANTY IS EXPRESSED OR IMPLIED. USE AT YOUR OWN RISK. Brett G. Olivier

CHAPTER

EIGHTEEN

PYSCESINTERFACES

Interfaces converting to and from PySCeS models - makes use of Brett's Core2

class stochpy.PyscesInterfaces.**Core2interfaces**

Defines interfaces for translating PySCeS model objects into and from other formats.

convertSBML2PSC(*sbmlfile*, *sbmldir=None*, *pscfile=None*, *pscdir=None*)

Convert an SBML file to a PySCeS MDL input file.

- *sbmlfile*: the SBML file name
- *sbmldir*: the directory of SBML files (if None current working directory is assumed)
- *pscfile*: the output PSC file name (if None *sbmlfile*.psc is used)
- *pscdir*: the PSC output directory (if None the stochpy.model_dir is used)

readMod2Core(*mod*, *iValues=True*)

Convert a PySCeS model object to core2

- *iValues*: if True then the models initial values are used (or the current values if False).

readSBMLToCore(*filename*, *directory=None*)

Reads the SBML file specified with filename and converts it into a core2 object stochpy.interface.core

- *filename*: the SBML file
- *directory*: (optional) the SBML file directory None means try the current working directory

writeCore2PSC(*filename=None*, *directory=None*, *getstrbuf=False*)

Writes a Core2 object to a PSC file.

- *filename*: writes <filename>.xml or <model_name>.xml if None
- *directory*: (optional) an output directory
- *getstrbuf*: if True a StringIO buffer is returned instead of writing to disk

writeCore2SBML(*filename=None*, *directory=None*, *getdocument=False*)

Writes Core2 object to an SBML file.

- *filename*: writes <filename>.xml or <model_name>.xml if None
- *directory*: (optional) an output directory
- *getdocument*: if True an SBML document object is returned instead of writing to disk or

writeMod2PSC(*mod*, *filename=None*, *directory=None*, *iValues=True*, *getstrbuf=False*)

Writes a PySCeS model object to a PSC file.

- *filename*: writes <filename>.psc or <model_name>.psc if None
- *directory*: (optional) an output directory
- *iValues*: if True then the models initial values are used (or the current values if False).
- *getstrbuf*: if True a StringIO buffer is returned instead of writing to disk

writeMod2SBML(*mod*, *filename=None*, *directory=None*, *iValues=True*, *getdocument=False*, *getstrbuf=False*)

Writes a PySCeS model object to an SBML file.

- *filename*: writes <filename>.xml or <model_name>.xml if None
- *directory*: (optional) an output directory
- *iValues*: if True then the models initial values are used (or the current values if False).
- *getdocument*: if True an SBML document object is returned instead of writing to disk or
- *getstrbuf*: if True a StringIO buffer is returned instead of writing to disk

CHAPTER

NINETEEN

STOCHASTIC NUCLEOSOME MODIFICATION SIMULATIONS

Performs nucleosome modification simulations based on stochastic simulation algorithms (SSA).

Hard coded for one trajectory

Written by TR Maarleveld, Amsterdam, The Netherlands E-mail: tmd200@users.sourceforge.net Last Change: September 19, 2011

class stochpy.NucleosomeSimulations.**NucSim**(*Method='Direct'*, *File=None*, *dir=None*, *Mode='steps'*, *End=1000*, *Trajectories=1*, *IsInteractive=True*)

Performs Stochastic Nucleosome Modification Simulations.

```
>>> sim = stochpy.NucSim()
>>> sim.DoStochSim()
>>> sim.Model(File = 'filename.psc', dir= '/.../.../filename.psc)>>>
sim.PlotGlobalTimeSim()
>>> sim.Timesteps(1000)
>>> sim.Endtime(100)
>>> sim.PlotPattern()
>>> sim.PlotGlobalDistributions()
>>> sim.Write2File()
>>> sim.ShowOverview()
>>> sim.ShowSpecies()
```

DoStochSim()

Perform a SSA run and merge the output to global M, U and A modification results

Endtime(*t*)

Set the end time of exact realization of the Markov jump process

Input:

- *t*: endtime*(float)

GetGapMeasure()

Calculates a 'gap' measure, which is the abs. difference between the number of M and A modifications averaged over a long simulation time. (from Dodd et al. 2007,'Theoritical Analysis of Epigenetic Cell Memory by Nucleosome Modification', Cell 129,813-822).

GetGlobalDistributions()
Determine the distribution patterns of all modification types

GetGlobalModification()
Determine for each modification (M, U or A) the modification frequency at each position. So, it gives for example that position i is over time almost always in a M-modified state.

GetNucleosomeOutput()
Besides nucleosome results, enzyme quantities can be simulated, which are ignored in the output analysis. So, E(type)M[i] and M[i] are both from M[i], both the first one carries an enzyme. Therefore, these results are merged to obtain the output per nucleosome modification position.

GetStateTimes()
Each nucleosome in this model can have 3 different modifications (M,U, or A). This function determines for each nucleosome in the model the distribution of times that it stays in a particular modification.

MergeModifications()
There are multiple modifications possible for each nucleosome, which gives a enormous number of "species". Examples are [A1],[M1], and [U1], while this describes only nucleosome 1. Further, it is interesting to see the modifcation effects for the whole model in stead of for single nucleosomes. Therefore, this function 'merges' the modifications of each nucleosome to the total number of certain modifications at each time point. For instance, imagine a 60 Nucleosome model with 3 possibile modifications (M, U, and A), which contains 180 different 'species'. This model is then reduced to the three modification levels (M, U and A) at each timepoint.

Hardcoded for three possible modifications: M, U, and A (/28/07/10/)

Method(*method*)

Input:

- *method*: (string)

Select one of the four methods:

- *Direct*
- *FirstReactionMethod*
- *NextReactionMethod*
- *TauLeaping*

Model(*File*, *dir=None*)
Give the model, which is used to do stochastic simulations on

Input:

- *File* = 'filename.psc'
- *dir*: [default = None] the directory where File lives"

PlotGlobalDistributions(*species=None*, *linestyle='dotted'*, *title='StochPy Distribution Plot'*)

Plot the distributions patterns of all modification types Default: PlotGlobalDistributions() plots distribution for each species

Input:

- *species*: [default = None] as a list ['S1','S2']
- *linestyle*: [default = 'dotted'] dotted, dashed, and solid
- *title*: [default = StochPy Distribution Plot]

PlotGlobalTimeSim(*species=None*, *linestyle='dotted'*, *title='StochPy Time Simulation Plot'*)

Plot the time simulation of the merged output

Default: PlotGlobalTimeSim() plots time simulation for each species

Input:

- *species*: [default = None] as a list ['S1','S2']
- *linestyle*: [default = 'dotted'] dotted, dashed, and solid

PlotPattern(*species=None*, *linestyle='dotted'*, *title='StochPy Pattern Plot'*)

Plot the average nucleosome modification for each nucleosome position Default: PlotPattern() plots the pattern (position specific distribution) for each species

Input:

- *species*: [default = None] as a list ['S1','S2']
- *linestyle*: [default = 'dotted'] dotted, dashed, and solid
- *title*: [default = 'StochPy Pattern Plot']

PlotStateTimes(*who*)

Plot the state times for a given nucleosome modification.

Input:

- *who*: a certain modification (M5 or A2 for instance)*

PrintGlobalDistributions()

Print the distributions patterns of all modification types

PrintGlobalTimeSim()

Print the time simulation of the merged output

PrintPattern()

Print the average nucleosome modification for each nucleosome pos.

PrintStateTimes()

Print the mean and std of the state times for each nucleosome modification.

Reload()
Reload the entire model again. Usefull if the model file has changed

Run()
old version

ShowOverview()
Print an overview of the current settings

ShowSpecies()
Print the species of the model

Timesteps(*s*)
Set the number of time steps to be generated for each trajectory

Input:

- *s*: Number of time steps (integer)

Write2File(*what='TimeSim'*, *to=None*)
Write output to a file

Input:

- *what*: [default = TimeSim] TimeSim, GlobalDistributions, Pattern
- *to*: Directory/outputname (optional)

Default of the first argument is: TimeSim

CHAPTER

TWENTY

N-NUCLEOSOME MODEL BUILDER

Used as input for Stochastic Simulation Algorithms.

This model builder has several features: - neighbour dependent reactions - neighbour independent reactions - initial modifications are randomly determined - enzyme-landing-locations (23/08/10)

Output is automatically stored in a modelfile at /home/usr/Stochpy/pscmodels/

Written by TR Maarleveld, Amsterdam, The Netherlands E-mail: tmd200@users.sourceforge.net Last Change: September 15, 2011

class stochpy.NucleosomeModel.**NucModel**(*N=20*, *ModelType=1*, *OnRate=1.0*, *OffRate=0.1*, *DiffRate=0.6*, *EnzymeRate=5.0*, *Recruit=0.1*, *LandingZones={'M': [10]}*, *IsRecruit=False*, *IsNeighbours=False*, *IsLongRange=False*, *Threshold=7*, *NeighbourRate=2.0*, *EnzNeighRate=10.0*, *IsDecay=False*)

Builds reactions for a N-nucleosome model, which can be used as input for SSAs. Input:

- *N*: number of nucleosomes (integer)
- *ModelType*: (1-8 or else)
- *OnRate*: (float) U[i] –> EU[i]
- *OffRate*: (float) EM[i] –> M[i]
- *DiffRate*: (float) EM[i] –> EM[i+-1]
- *EnzymeRate*: (float) EU[i] –> EM[i]
- *Recruit*: (float) M[i] –> EM[i]
- *LandingZones*: dictionairy, where the keys are the enzyme types and the values the positions
- *Booleans:* IsRecruit,IsNeighbours,IsLongRange, Threshold,IsDecay

Each input argument has a default value and it is possible to change them in a interactive manner.

Usage (high-level functions): >>> help(model) >>> model.Build() The generated model is stored at /home/user/Stochpy/pscmodels/model1.psc >>> model.ChangeN(15) The number of nucleosomes is: 15 >>> model.Recruitment() Info: Recruitment is activated >>> model.Neighbours() >>> model.LongRangeInteractions() >>> model.ChangeThreshold(5) >>> model.Decay()

Build()
Build one of the pre-defined models or a user-defined one

BuildLandingZones(*landing_zones*)
Builds the landing zones: locations where enzymes can bind to the DNA Input:

•*LandingZones*: (dictionairy) The keys are the enzyme types and the values the positions, such as: {'M':[10]}

ChangeN(*n*)
Change the number of nucleosomes Input:

•*n*: desired number of nucleosomes (integer)

ChangeThreshold(*threshold*)
Determine the threshold for long range neighbour interactions"

Input:

- *threshold*: (integer)

Decay(*IsDecay=True*)
Determine if the long range interactions are at one or multiple locations.

Example: Threshold = 7 U[i] + M[i+7] –> M[i] + M[i+7] k = kneighbour U[i] + M[i+6] –> M[i] + M[i+6] k = kneighbour * 0.80 U[i] + M[i+8] –> M[i] + M[i+8] k = kneighbour * 0.80 U[i] + M[i+5] –> M[i] + M[i+5] k = kneighbour * 0.41 U[i] + M[i+9] –> M[i] + M[i+9] k = kneighbour * 0.41 U[i] + M[i+4] –> M[i] + M[i+4] k = kneighbour * 0.135 U[i] + M[i+10] –> M[i] + M[i+10] k = kneighbour * 0.135

Input:

- *IsDecay* (boolean)

EntireModel()
Uses all the pre-defined functions in this class to build the entire model.

EnzNext(*mod*, *i*, *x*, *n=1*)
Builds in a multi-nucleosome model the interactions between nucleosome[i] and nucleosome[i+x], where enzymes are explicitly simulated. Ofcourse, Nucl[N+5] does not exist, so it does not create interactions with non-existing nucleosomes. Input:

•*mod*: modification type (string)

•*i*: Nucleosome number (integer)

•*x*: Nucleosome Neighbour number (integer)

- *n*: [default=1] Decay value (float)

EnzPrevious (*mod*, *i*, *x*, *n=1*)

Builds in a multi-nucleosome model the interactions between nucleosome[i] and nucleosome[i-x], where enzymes are explicitly simulated. Ofcourse, Nucl[-1] does not exist, so it does not create interactions with non-existing nucleosomes. Input:

- *mod*: modification type (string)
- *i*: Nucleosome number (integer)
- *x*: Nucleosome Neighbour number (integer)
- *n*: [default = 1] Decay value (float)

Enzyme (*mod*, *i*)

Build the reactions that are katalyzed by enzymes that are attached to the DNA chain: E(type)U[i] –> E(type)M[i] etc Input:

- *mod*: modification enzyme (string)
- *i*: Nucleosome number (integer)

EnzymeDropping (*mod*, *Type*, *i*)

Build the dropping of enzymes from nucleosomes: E(type)M[i] –> M[i]

- *mod*: modification (string)
- *Type*: enzyme type
- *i*: Nucleosome number (integer)

EnzymeLanding (*i*, *Type*, *mod*)

Enzyme landing 0 –> E[i] Input:

- *i*: Nucleosome number (integer)
- *Type*: enzyme type
- *mod*: modification (string)

EnzymeMovement (*Type*)

Build the diffusion of enzymes along the DNA-chain: E(type)[i] –> E(type)[i+1], E(type)[i] –> E(type)[i-1] This is done for each modification type, so: E(type)M[i] –> E(type)M[i+1] or to E

(type)[A+1] etc.

Input:

- *Type*: enzyme type

Initials ()

Builds the initial concentrations of the nucleosome modifications. Each nucleosome starts with a certain modification (M, U or A), which is determined randomly.

Kvalues()
Builds to velocity-constants

LongRangeInteractions(*IsLongRange=True*)
Activate or deactivate long range neighbour interactions

Input:

- *IsLongRange*: (boolean)

ModelType(*num*)
Choose a build-in model (1-8) Input:

•*num*: (integer) 1-8

Neighbours(*IsNeighbours=True*)
Activate or deactivate neighbour interactions

Input:

- *IsNeighbours (boolean)*

Next(*Type*, *i*, *x*, *n=1*)
Builds in a multi-nucleosome model the interactions between nucleosome[i] and nucleosome[i-x] Ofcourse, Nucl[-1] does not exist, so it does not create interactions with non-existing nucleosomes Input:

•*Type*: modification enzyme (string)

•*i*: Nucleosome number (integer)

•*x*: Nucleosome Neighbour number (integer)

•*n*: [default = 1] Decay value (float)

Noisy1(*i*)
Builds the noisy conversions from M –> U and from A –> U for each nucleosome in the model. Input:

•*i*: Nucleosome number (integer)

Noisy2(*i*)
Builds the noisy conversions from U–> A and U–> M for each nucleosome in the model. Notice that these conversions are only made if there is no explicit simulation of the enzymes. Input:

•*i*: Nucleosome number (int)

Previous(*Type*, *i*, *x*, *n=1*)
Builds in a multi-nucleosome model the interactions between nucleosome[i] and nucleosome[i-x]. Ofcourse, Nucl[-1] does not exist, so it does not create interactions with non-existing nucleosomes. Input:

•*Type*: modification enzyme (string)

•*i*: Nucleosome number (integer)

•*x*: Nucleosome Neighbour number (integer)

- *n*: [default = 1] Decay value (float)

RecruitEnzymes (*mod*, *i*)
Build reactions, which can recruit enzymes if a nucleosome carries a certain modification. M[i] –> EmM[i], if the model simulates M enzymes explicitly A[i] –> EaA[i], if the model simulates A enzymes explicitly Input:

- *mod*: modification enzyme (string)
- *i*: Nucleosome number (int)

Recruitment (*IsRecruit=True*)
Activate or deactivate recruitment

Input:

- *IsRecruit (boolean)*

WriteLastReaction ()
Print the last created reaction

WriteParms ()
Write the parameters that are used in the model to a file

Written by TR Maarleveld, Amsterdam, The Netherlands E-mail: tmd200@users.sourceforge.net Last Change: September 15, 2011

class stochpy.SBML2PSC.**SBML2PSC**
Module that converts SBML models into PSC models if libxml and libsbml are installed

Usage: >>> converter = stochpy.SBML2PSC() >>> converter.SBML2PSC('file.xml',directory)

SBML2PSC (*sbmlfile*, *sbmldir=None*, *pscfile=None*, *pscdir=None*)
Converts a SBML file to a PySCeS MDL input file.

Input:

- *sbmlfile*: the SBML file name
- *sbmldir*: [default = None] the directory of SBML files (if None current working directory is assumed)
- *pscfile*: [default = None] the output PSC file name (if None *sbmlfile*.psc is used)
- *pscdir*: [default = None] the PSC output directory (if None the pysces.model_dir is used)

Part VII

Indices and tables

- *genindex*
- *search*

PYTHON MODULE INDEX

S