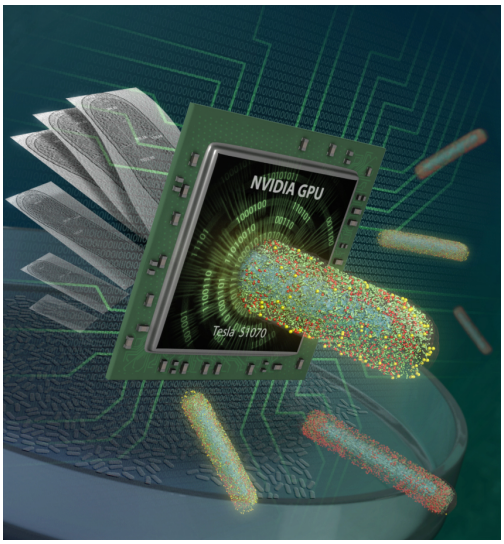
Lattice Microbes Problem Solving Environment Instruction Guide



LM Version 2.3, pyLM Version 1.1
June 2, 2016

Joseph R. Peterson, Mike J. Hallock, Elijah Roberts, John A. Cole, Piyush Labhsetwar, John E. Stone, and Zaida Luthey-Schulten

University of Illinois at Urbana-Champaign
http://www.scs.illinois.edu/schulten/lm

Description

The Lattice Microbes Instruction Guide describes how to use the software to perform and analyze stochastic simulations of spatially modeled microbial cells. Lattice Microbes development is supported in part by the DOE (Office of Science BER) under grant DE-FG02-10ER6510, the NIH (Center for Macromolecular Modeling and Bioinformatics) under grant NIH 9 P41 GM104601-23, and the NSF under grants MCB-08226143 and MCB 12-44570.

Table of Contents

List of Figures . . . . . **iii**

List of Tables . . . . . **iv**

Chapter 1 Introduction **1**
1.1 pyLM Overview 1
1.2 Stochastic Modeling 2
1.3 Capabilities 2

Chapter 2 Tutorials **4**
2.1 A Simple Example: Bimolecular Reaction 4
2.2 Seeing is Believing: *lac* Genetic Switch 9
2.2.1 Well-Stirred 9
2.2.2 Spatially Resolved 15
2.3 Debugging and Command Line Execution 20
2.3.1 Debugging 20
2.3.2 Command Line 23
2.4 Advanced Post-Processing: The Min System 24
2.5 Advanced Uses: Merging RDME with CME 27

Chapter 3 Examples **34**

Chapter 4 License and Copyright **36**

Bibliography **37**

List of Figures

1.1 A schematic of the pyLM and the Lattice Microbes software. 1
1.2 The workflow of the pyLM PSE. 3

2.1 (a) A deterministic solution to the bimolecular reaction. (b) A stochastic solution to the bimolecular simulation. 6
2.2 Average over many independent trajectories sampling the stochastic equation. . . . 9
2.3 A schematic of the *l*ac switch with a simple two-state model [2] (shown in the dotted box) and the full three-state model published subsequently [3]. In the two-state model, the repressor sits on the operator region just downstream from the promoter and prevents the polymerase from binding. However, when an active inducer (in this case an analog of lactose) is available, it binds to the repressor inducing it to unbind and enabling the gene to produce mRNA. The mRNA encodes for a transport protein that actively transports more inducer into the cell, which allows the gene to be on for a longer period of time. Figure from [1]. 10
2.4 A cycle showing five different processes that occur during the induction of the switch. Figure from [1]. 11
2.5 A graph showing the possible interconversions of each species in the system. The nodes are colored by initial count and the edges by the source node. Self edges are shown as loops. 14
2.6 A single time trace from a single replicate showing the mRNA and Protein number over an hour period. 14
2.7 The logarithm of the count of times the state (number of protein and number of messenger) was traversed over the time series averaged over 100 simulations for the first hour of simulation. Two clusters are clear in the map, one close to 0 mRNA and 50 protein and other with about 100 protein and 10 mRNA. 15
2.8 A schematic of a hypothetical RDME simulation. 17
2.9 An example of the open dialog box in VMD. 18
2.10 A Lattice Microbes simulation initially loaded. 19
2.11 a) Graphics Representation window showing what one such particle representation might look like. b) A view showing external inducer (red), internal inducer (gray), *lac* permease protein (blue), *lac* repressor (cyan) and the DNA particle (green). . . 20
2.12 a) An example of a graphic representation of the site types. b) Here the extracellular (also called default) is shown in gray, the membrane, which is about 2 sites thick, is shown in blue and the cytoplasm is shown in red. Both the extracellular and the membrane are cut only show those with x less than 5100 angstroms. 21

2.13 A schematic of the Min protein system model [1]. ATP bound MinD attaches to the membrane and can long fields of bound MinD. MinE binds to membrane bound MinE stimulating its ATPase activity and causing the MinD to leave the membrane where, once back in the cytoplasm, another ATP can replace the bound ADP. 25
2.14 Three times during the simulation showing membrane bound MinD (cyan) oscillating from pole to pole with MinE (red) trailing behind the wavefront. 27
2.15 End to end oscillations of the membrane bound MinD as the system progresses in time. 27
2.16 A schematic of the combined RDME/CME simulation. 28
2.17 A schematic of the combined RDME/CME simulation performed in this section where transcription/translation are spatially resolved and enzyme catalysis is assumed to be a well-stirred process. 33
2.18 Three times during the simulation showing DNA (green), RNA (red), protein (brown) and product molecules added to the simulation (pink). 33

List of Tables

2.1 Reactions available to both CME and RDME. Here, the stochastic rate constant should be computed from the macroscopic rate constant (perhaps from experiment) using the volume of the experiment, V, and Avogadro's number, N_A. 7

Chapter 1

Introduction

This Instruction Guide contains instructions for using the Lattice Microbes Problem Solving Environment, as described in the following publication: [1]. This guide is very much a work in progress and will continue to be expanded. At present, it should contain enough information to get started using the Lattice Microbes software.

1.1 pyLM Overview

pyLM is a Problem Solving Environment (PSE) for biological simulations [1]. Written in Python, it wraps and extends Lattice Microbes. The PSE is comprised of a base set of functionality to set up, monitor and modify simulations, as well as a set of standard post-processing routines that interface to other Python packages, including NumPy, SciPy, H5py, iGraph to name a few.

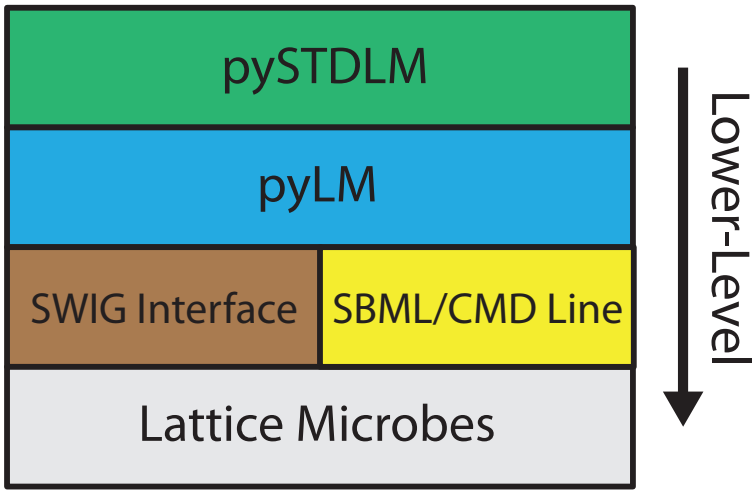


Figure 1.1: A schematic of the pyLM and the Lattice Microbes software.

The PSE is shown schematically in Figure 1.1. It sits on top of a SWIG interface that allows the C++ code to be accessible from the Python terminal. Using pyLM allows the user to set up, run and post-process simulations all within a single script. A general workflow is shown in Figure 1.2.

For tutorials on using pyLM please see the "pyLM Tutorial" and for in-depth description of all pyLM functionality please see the documentation "pyLM Documentation".

1.2 Stochastic Modeling

Lattice Microbes can be used to simulate chemical master equations (CME):

$$\frac{dP(\mathbf{x},t)}{dt} = \sum_{r}^{R}[-a_r(\mathbf{x})P(\mathbf{x},t) + a_r(\mathbf{x}_\nu - \mathbf{S_r})P(\mathbf{x} - \mathbf{S_r},t)]$$

and reaction-diffusion master equations (RDME):

$$\begin{aligned}\frac{dP(\mathbf{x},t)}{dt} &= \sum_{\nu}^{V}\sum_{r}^{R}[-a_r(\mathbf{x}_\nu)P(\mathbf{x}_\nu,t) + a_r(\mathbf{x}_\nu - \mathbf{S_r})P(\mathbf{x}_\nu - \mathbf{S_r},t)] \\ &+ \sum_{\nu}^{V}\sum_{\xi}^{\pm\hat{i},\hat{j},\hat{k}}\sum_{\alpha}^{N}[-d^\alpha x_\nu^\alpha P(\mathbf{x},t) + d^\alpha(x_{\nu+\xi}^\alpha + 1_\nu^\alpha)P(\mathbf{x} + 1_{\nu+\xi}^\alpha - 1_\nu^\alpha,t)]\end{aligned}$$

using a variety of methods.

1.3 Capabilities

pyLM and the included library of standard systems pySTDLM provide a problem solving environment for setting up, running and analyzing stochastic biological simulations [1]. It contains functionality for specifying simulation setup including:

- Named species
- Initial counts and distributions
- Reactions and rates
- Spatial localization and definition
- Diffusion properties
- Obstacles
- Handles to data in the popular Numpy array representation
- Plotting species averages/variances and individual time traces
- Plotting Kymographs of spatial species distributions
- Reaction network generation
- Dynamic reaction network representations

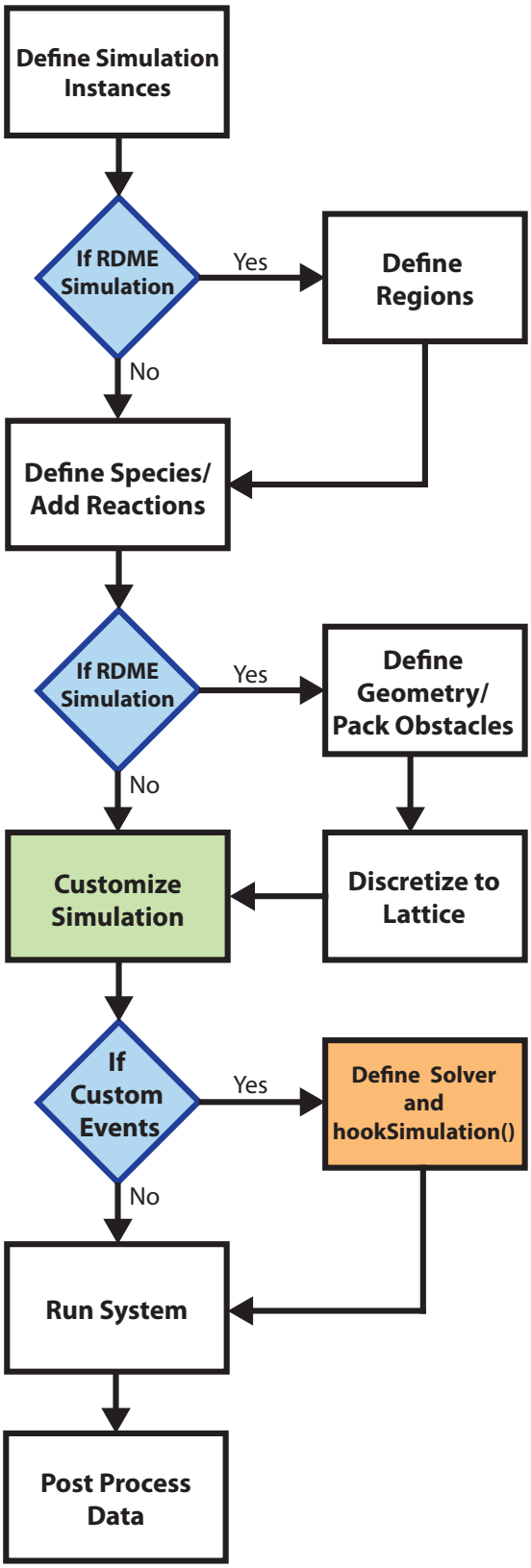


Figure 1.2: The workflow of the pyLM PSE.

Chapter 2

Tutorials

It is assumed that the user has at least a working knowledge of the Python programming language including such concepts as 'if' statements, 'while' and 'for' loops, 'tuples', 'lists' and 'function/method calls'. If this is not the case, we recommend the user take the Google Python Class (`https://developers.google.com/edu/python/`) or work through the official Python tutorials (`http://docs.python.org/2/tutorial/`), both of which are available free of charge.

2.1 A Simple Example: Bimolecular Reaction

In the first tutorial, we will motivate stochastic modeling with a simple example of a bimolecular reaction among three low copy number molecular species. First, we will use a traditional continuous ODE approach to solving the equations using the SciPy `odeint` function and then use pyLM and Lattice Microbes to simulate the same system.

We will simulate the association/dissociation reaction of hypothetical molecules:

$$\mathrm{A} + \mathrm{B} \underset{\mathrm{k_r}}{\overset{\mathrm{k_f}}{\rightleftharpoons}} \mathrm{C}$$

Writing out the differential equations will facilitate the ODE definition in SciPy:

$$\begin{aligned}
\frac{d[A]}{dt} &= -k_f[A][B] + k_r[C] \\
\frac{d[B]}{dt} &= -k_f[A][B] + k_r[C] \\
\frac{d[C]}{dt} &= k_f[A][B] - k_r[C]
\end{aligned}$$

Let us start out with a low number of each particles: 1000 of A and B and 0 of C. Let us imagine simulating that problem in a microbe sized volume of 1 fL. Also, let us start off with rates of $k_f = 1.07 \times 10^5 M^{-1}s^{-1}$ and $k_r = 0.351/s$. The Python script in Listing 2.1 will solve this using a deterministic solver.

Listing 2.1: `tut1.1-ODEBimol.py` — Code used to solve the bimolecular reaction with a continuous ODE solver.

```python
import numpy as np
import scipy.integrate as spi
import matplotlib.pyplot as plt

# Constants
V  = 1.0e-15    # L
NA = 6.022e23   # molecules/mole
tstart =  0.0   # s
tend   = 30.0   # s

# Rates
kf=1.07e5/(NA*V) # /M/s
kr=0.351         # /s

# Initial Species Counts
A  = 1000
B  = 1000
C  = 0
S0 = [A, B, C]

# Definition of ODEs
def ds_dt(s, t):
        Ai = s[0]
        Bi = s[1]
        Ci = s[2]
        # Rate equations
        dA_dt = -kf*Ai*Bi + kr*Ci
        dB_dt = -kf*Ai*Bi + kr*Ci
        dC_dt =  kf*Ai*Bi - kr*Ci
        return [dA_dt, dB_dt, dC_dt]

# Solve
t    = np.linspace(tstart, tend, 1000000)
soln = spi.odeint(ds_dt, S0, t)

# Plot
plt.figure()
plt.plot(t, soln[:,0], label="A/B")
plt.plot(t, soln[:,2], label="C")
plt.xlabel('Time (s)')
plt.ylabel('Molecule Count')
plt.legend()
plt.savefig('BimolecularODE.png')
```

Running this code will create the plot shown in Figure 2.1a. Walking through the script, we start out defining some constants, followed by a definition of a function that returns the rates in the previous equations followed by a solve for 1 million different time points. Finally the results are plotted. Of particular note is the division of the forward rate constant by Avogadro's number and the volume of the cell which takes us to units of molecules per second. This is the standard for stochastic simulations and will be described shortly.

What you should note about the results are that the count of each species varies smoothly across

the time course. However, it is reasonable to define molecules at their endpoints, either reactants or products, which necessarily means that the count of each molecule must be a whole number. Furthermore, this implies that the count of the molecules changes in integer units. Stochastic modeling was designed to address this point.

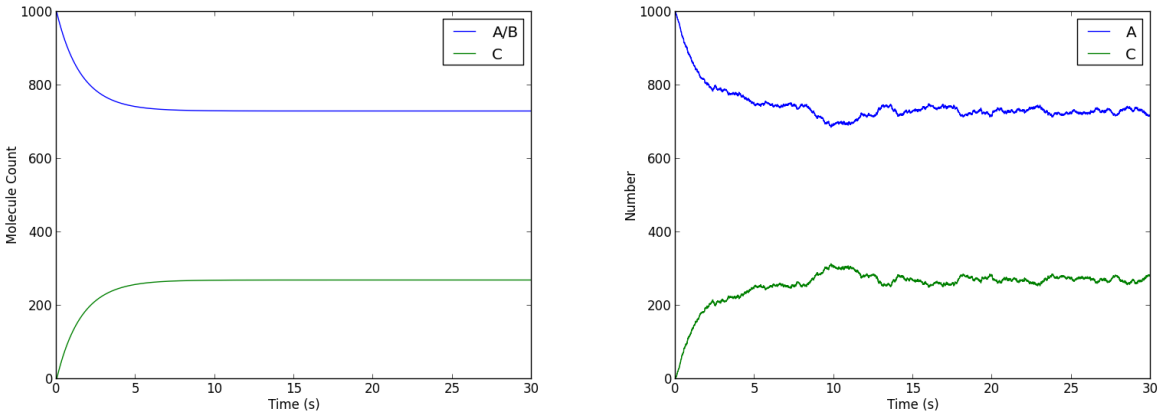


Figure 2.1: (a) A deterministic solution to the bimolecular reaction. (b) A stochastic solution to the bimolecular simulation.

Next, let us try doing the same simulation in Lattice Microbes. The Python script in Listing 2.1 will accomplish this. To run the simulation, you should execute the command:

```
python tut1.2-StochBimol.py
```

The simulation will take from a few seconds to perhaps a minute to complete.

The first thing to note are the new import statements. `pyLM` contains the main modules for setting up reactions. `pyLM.units` contains helper functions for scaling numbers in a legible way. `pySTDLM` is a library of standard functionality such as standard reaction systems, cell systems. In addition it contains a number of pre- and post-processing functionality. The net few lines are merely definitions of some constants. The line `sim=CME.CMESimulation()` creates an empty simulation object. The next lines define the chemical species; in pyLM species are named by python strings and must be registered with the simulation using the `defineSpecies` command.

The next two lines set up the reactions. In Lattice Microbes all reactions are irreversible, necessitating the specification of both the forward and back reactions separately. The first and second argument can be either a tuple of reactants or a string when only one reactant is specified. Lattice Microbes currently supports 0th, 1st and 2nd order reactions, and reaction rates must be specified in the "stochastic" format (see Table 2.1). In the special case of a 0th order reaction, the empty string `''` should be passed as the reactant. In addition, annihilation reactions can be specified by passing the empty string `''` as the product parameter.

Order	Form	Parameters	Macroscopic Units	Stochastic Rate Constant (s^{-1})
0th	$\emptyset \rightarrow$A	k	Ms^{-1}	$k \cdot V \cdot N_A$
1st	A$\rightarrow$B	k	s^{-1}	k
2nd	A+B$\rightarrow$C	k	$M^{-1}s^{-1}$	$\frac{k}{V \cdot N_A}$
2nd (Self)	2A$\rightarrow$B	k	$M^{-1}s^{-1}$	$\frac{k}{V \cdot N_A}$

Table 2.1: Reactions available to both CME and RDME. Here, the stochastic rate constant should be computed from the macroscopic rate constant (perhaps from experiment) using the volume of the experiment, V, and Avogadro's number, N_A.

The following lines merely define the initial species counts. Next the simulation parameters are specified, time steps will be written out every 30 microseconds and the total simulation will run for 30 seconds. The next line is of particular importance; the simulation *m*ust be saved to a file before running the simulation. Finally, we call the `run(...)` command on the simulation object giving it the name of the simulation file, the simulation method and the number of independent trajectories (replicates) to run of that simulation. Finally, we use one of the post processing commands to look plot the time course of the A and C species for replicate number 1 (replicates are indexed starting at 1) into the file 'BimolecularStoch.png'.

Listing 2.2: `tut1.2-StochBimol.py` — Code used to solve the bimolecular reaction with the discrete/stochastic Lattice Microbes CME implementation.

```
import numpy as np
import matplotlib.pyplot as plt
from pyLM import *
from pyLM.units import *
from pySTDLM import *
from pySTDLM.PostProcessing import *

# Constants
V  = 1.0e-15        # L
NA = 6.022e23       # molecules/mole
kf = 1.07e5/(NA*V) # /M/s
kr = 0.351          # /s

# Create our CME simulation object
sim=CME.CMESimulation()

# define our chemical species
species = ['A', 'B', 'C']
sim.defineSpecies(species)

# Add reactions to the simulation
sim.addReaction(reactant=('A','B'), product='C', rate=kf)
sim.addReaction(reactant='C', product=('A','B'), rate=kr)

# Set our initial species counts
sim.addParticles(species='A', count=1000)
```

```
sim.addParticles(species='B', count=1000)
sim.addParticles(species='C', count=0)

# Define simulation parameters: run for 10 seconds, saving data every ms
sim.setWriteInterval(microsecond(30))
sim.setSimulationTime(30)
sim.save('T1.2-bimol.lm')

# Run 1 replicates using the Gillespie solver
sim.run(filename='T1.2-bimol.lm', method="lm::cme::GillespieDSolver",
    replicates=1)

# Plot the solution
plotTraceFromFile(filename='T1.2-bimol.lm', species=['A','C'], replicate=1,
    outfile='BimolecularStoch.png')
```

One example of the output can be seen in Figure 2.1. You might note that the behavior is qualitatively the same, however there appears to be considerable fluctuation, even after the system has come to equilibrium. This is due to the stochastic nature of the process, where the reaction can transiently fluctuate away from the equilibrium value. In addition, you may be able to tell that the changes in particle number from one time to another are in integer increments, though this will become considerably more obvious at lower number of particles.

To convince yourself that the system obeys the the macroscopic limit—or in other words the average over many realizations of the system trajectory will give back the continuous behavior—try modifying the last few lines of the script with the following and rerun:

```
sim.save('T1.3-bimol.lm')

# Run 100 replicates using the Gillespie solver
sim.run(filename='T1.3-bimol.lm', method="lm::cme::GillespieDSolver",
    replicates=100)

# Plot the solution
plotAvgVarFromFile(filename='T1.3-bimol.lm', species=['A','C'], outfile='
    BimolecularAvgVar.png')
```

The last command takes all of the replicates found in the simulation output, performs a time average over their species counts and computes the variance. You should see something similar to the plot in Figure 2.2, Where the average appears to be the continuous limit but the variance is quite large.

So now that you have a working understanding of using the CME solver in pyLM try to answer the following:

1. How does initial particle count affect stochasticity?
2. How do the rates of reaction affect stochasticity?
3. What role does volume play in the reaction system?
4. What is the threshold where the continuous assumption of the ODE breaks down?

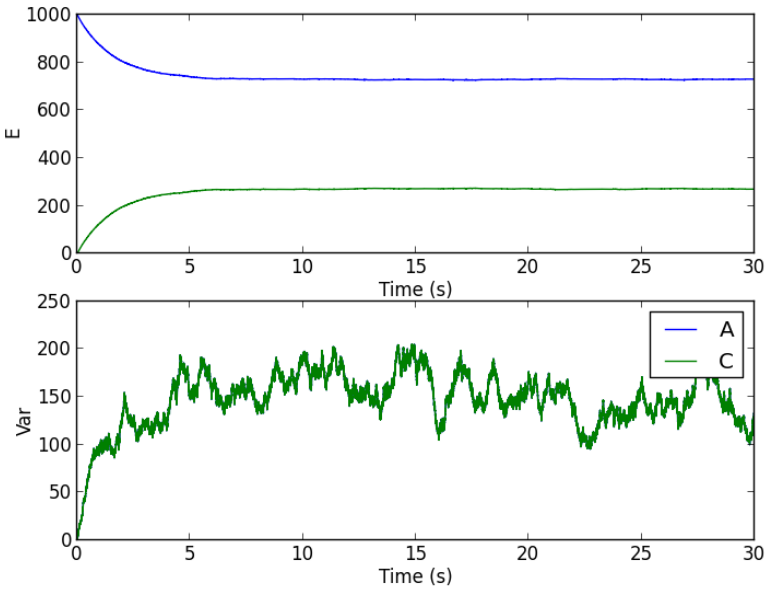


Figure 2.2: Average over many independent trajectories sampling the stochastic equation.

5. What happens to the fluctuations if you have competition for one of the reactants? For example if you had another reaction like: A $\rightleftharpoons$ D

2.2 Seeing is Believing: *lac* Genetic Switch

The *lac* genetic switch in *E. coli* shows considerable stochasticity during the switch from utilizing glucose to lactose [2] (shown in Figure 2.3). In monoclonal colonies growing in exponential phase that use up available glucose, the time for each cell to turn "on" can vary by many minutes or even an hour depending on the availability of lactose and the number of repressor proteins that are present in the cell. To exacerbate this, the DNA for the gene is available in only one copy in the cell which can be "on" or "off" depending on if a repressor is bound, and affects downstream availability and number of mRNA and the protein. The lifecycle of the system, upon addition of lactose (and depletion of the more favored glucose) is depicted in Figure 2.4.
For simplicity, we will consider only the two-state *lac* model. Even in the more limited two-state model, several salient features were discovered including deficiencies in previous models and the extent to which stochasticity played a role [2]. We will initially examine the system under the well stirred assumption and then move on to a spatially resolved model.

2.2.1 Well-Stirred

The *lac* switch model has 23 reactions with 7 different substrates. Setting up of the simulation is very similar to the previous example, so only the differences will be noted. Specifically, this section is designed to show you how custom post-processing can be achieved. The code to set up the simulation can be seen in Listing 2.3.

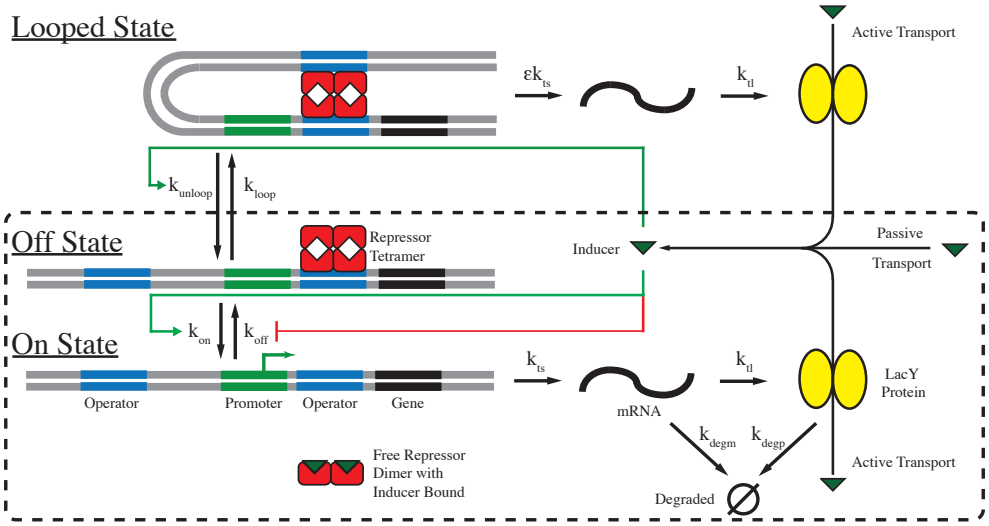


Figure 2.3: A schematic of the *l*ac switch with a simple two-state model [2] (shown in the dotted box) and the full three-state model published subsequently [3]. In the two-state model, the repressor sits on the operator region just downstream from the promoter and prevents the polymerase from binding. However, when an active inducer (in this case an analog of lactose) is available, it binds to the repressor inducing it to unbind and enabling the gene to produce mRNA. The mRNA encodes for a transport protein that actively transports more inducer into the cell, which allows the gene to be on for a longer period of time. Figure from [1].

Listing 2.3: `tut2.1-lac2stateCME.py` — Code to set up and execute the *lac* two-state model with CME.

```
from pyLM import *
from pyLM.units import *
from pySTDLM import *
from pySTDLM.PostProcessing import *
from pySTDLM.NetworkVisualization import *
import math as m
import numpy as np
import matplotlib.pyplot as plt

# Create our simulation object
sim=CME.CMESimulation()

# Add the reactants
species = ['R2','O','R2O','IR2','IR2O','I2R2','I2R2O','mY','Y','I','Iex','YI']
sim.defineSpecies(species)

scalar = 2.076e-9 # Rate conversion from experiments to stochastic

# Add the reactions
# Lac operon regulation
sim.addReaction(reactant=('R2','O'), product='R2O',rate=2.43e6*scalar)
sim.addReaction(('IR2','O'),  'IR2O',       1.21e6*scalar)
sim.addReaction(('I2R2','O'), 'I2R2O',      2.43e4*scalar)
```

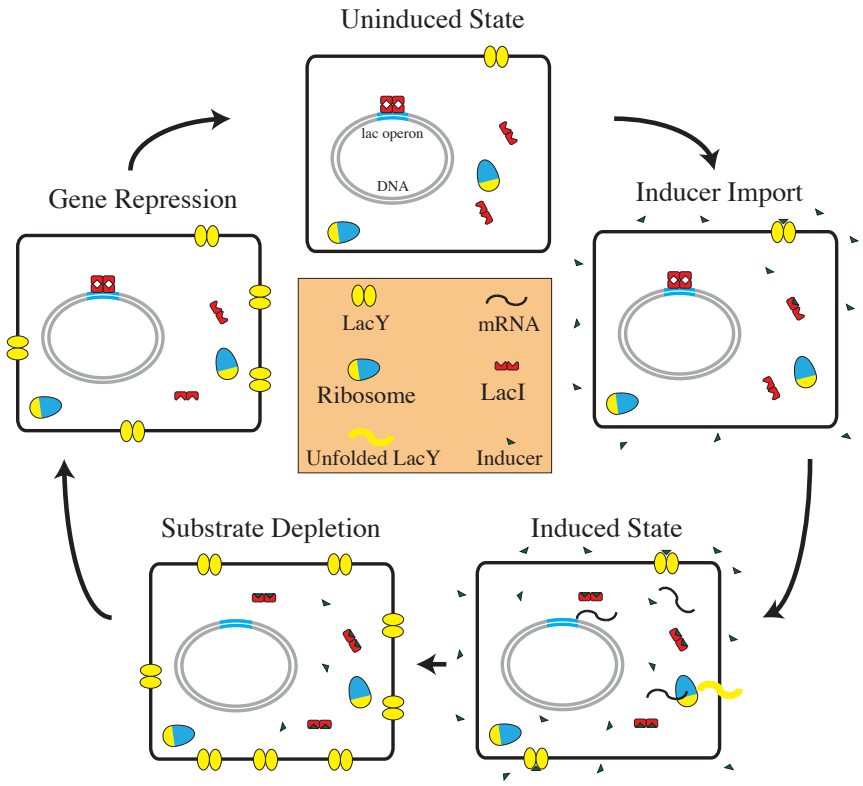


Figure 2.4: A cycle showing five different processes that occur during the induction of the switch. Figure from [1].

```
sim.addReaction('R2O',          ('R2','O'),    6.30e-4)
sim.addReaction('IR2O',         ('IR2','O'),   6.30e-4)
sim.addReaction('I2R2O',        ('I2R2','O'), 3.15e-1)

# Transcription, translation, and degredation
sim.addReaction('O',            ('O','mY'),    1.26e-1)
sim.addReaction('mY',           ('mY','Y'),    4.44e-2)
sim.addReaction('mY',           '',            1.11e-2)
sim.addReaction('Y',            '',            2.10e-4)

# Inducer-repressor interactions
sim.addReaction(('I','R2'),    'IR2',          2.27e4*scalar)
sim.addReaction(('I','IR2'),   'I2R2',         1.14e4*scalar)
sim.addReaction(('I','R2O'),   'IR2O',         6.67e2*scalar)
sim.addReaction(('I','IR2O'), 'I2R2O',        3.33e2*scalar)
sim.addReaction('IR2',          ('I','R2'),    2.00e-1)
sim.addReaction('I2R2',         ('I','IR2'),   4.00e-1)
sim.addReaction('IR2O',         ('I','R2O'),   1.00)
sim.addReaction('I2R2O',        ('I','IR2O'), 2.00)

# Inducer transport
sim.addReaction('Iex',          'I',           2.33e-3)
sim.addReaction('I',            'Iex',         2.33e-3)
sim.addReaction(('Y','Iex'),   ('YI','Iex'), 3.03e4*scalar)
```

```
sim.addReaction('YI',          ('Y','Iex'),  1.20e-1)
sim.addReaction('YI',          ('Y','I'),    1.20e+1)

# Populate the model with particles
sim.addParticles(species='R2',   count=9)
sim.addParticles('O',    1)
sim.addParticles('Y',    30)
sim.addParticles('I',    7224*4)
sim.addParticles('Iex',  7224*4)

# Set up the times
sim.setTimestep(ms(100))
sim.setWriteInterval(1)
sim.setSimulationTime(3600.0)

# Save the simulation state to a file
filename='T2.1-lac2state.lm'
sim.save(filename)

plotCMEReactionNetwork(sim, filename="lacGraph.gml")

# Run the simulation for 10 hours
reps=100
sim.run(filename, "lm::cme::GillespieDSolver", reps)

# Post-processing
for i in range(1,reps):
        plotTraceFromFile(filename, ['mY','Y'], i, "Trace.mY.Y.%d.png"%(i))

##########################
# Custom Post-Processing #
##########################
# Get out the data
fileHandle=openLMFile(filename)
times = getTimesteps(fileHandle)
simmY = reps*[]
simY  = reps*[]
for i in range(1,reps):
        simY.append(getSpecieTrace(fileHandle, 'Y', i))
        simmY.append(getSpecieTrace(fileHandle, 'mY', i))

# Find min and max of mY and Y
minY  = 1e9; maxY  = 0; minmY = 1e9; maxmY = 0
for i in range(reps-1):
        for y in simY[i]:
                minY = min(minY, y)
                maxY = max(maxY, y)
        for my in simmY[i]:
                minmY = min(minmY, my)
                maxmY = max(maxmY, my)

# Histogram the data (the hard way)
hist = np.zeros(shape=(maxmY,maxY))
for i in range(reps-1):
```

```
        for j in range(len(times)):
                hist[simmY[i][j]-1, simY[i][j]-1] += 1.0
# Compute logarithm of histogram value
histLog = np.zeros(shape=(maxmY,maxY))
for i in range(maxmY-1):
        for j in range(maxY-1):
                if hist[i,j] > 0:
                        histLog[i,j] = m.log10(hist[i,j])

# Plot ourselves a histogram
plt.clf()
plt.imshow(histLog.transpose(), interpolation='nearest', origin='lower',
    aspect='auto')
plt.colorbar()
plt.xlabel('mRNA Count')
plt.ylabel('LacY Count')
plt.savefig('LacYmRNAHeatmap.png')

# Clean up after ourselves
closeLMFile(fileHandle)
```

This simulation will take a bit longer than the previous one. It is likely it will take between 5 and 30 minutes.

This time we have used a pre-processing command to plot the network of reactions. Namely, the command `plotCMEReactionNetwork(sim, filename="lacGraph.gml")` will create a graph file called 'lacGraph.gml' which can be opened with a program like Gephi (`https://gephi.org`) or Cytoscape (`http://www.cytoscape.org`), resulting in an image like that seen in Figure 2.5. The use of these softwares is beyond the scope of this guide and we suggest you see the webpages for use.

The next difference is that we use a different solver, namely the Next Reaction solver, which is slightly less efficient than the Gillespie SSA solver used in the previous example [4]. Next, we have a loop that plots the traces of the messenger 'mY' and the count of the protein 'Y' as a function of time. This line produces 100 files and can be commented out if uneccesary.

Finally, we have a set of code that is custom post-processing that does a cluster on the data to get a heat map of the phase-space traversed by the messenger and protein over all the replicates. This demonstrates four other functions that are particularly important. Generally, when your want custom post-processing you will be interacting with the file for a while, so use `openLMFile(filename)` to get a handle to the file. Do not forget to call `closeLMFile(handle)`, or your data file could be corrupted. Next, you will almost undoubtably want the time trace so call `getTimesteps(handle)` to get an array of the simulation timestep times. Finally, you can get different species time traces using the function `getSpecieTrace(handle, specie, replicateNumber)`. The rest is just standard Python, NumPy and matplotlib code. A representative example can be seen in Figure 2.6.

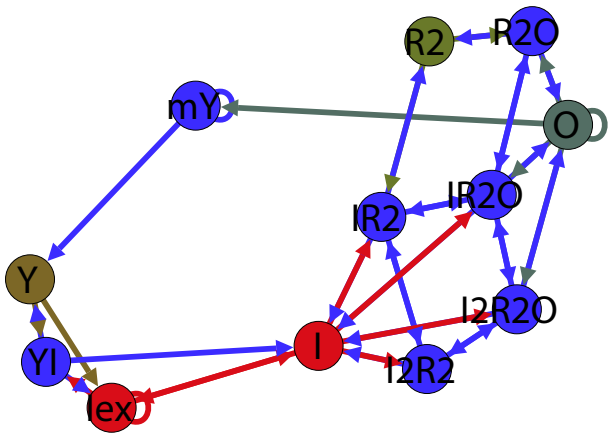


Figure 2.5: A graph showing the possible interconversions of each species in the system. The nodes are colored by initial count and the edges by the source node. Self edges are shown as loops.

When examining a representative time trace one will notice a few interesting features (Figure 2.6). Particularly, mRNA tend to be formed in bursts, which correspond to when a gene is 'on' (no repressor is bound). Correspondingly, there are large increases in the protein count during these times, until the mRNA degrades. One can also map the phase space that is accessed by the simulations. Averaging over 100 simulations over the first hour, and plotting protein (LacY) versus mRNA, one notes that there are two major populations (Figure 2.7). This heat map was generated with the custom post-processing code under the label "Custom Post-Processing" in Listing 2.3.

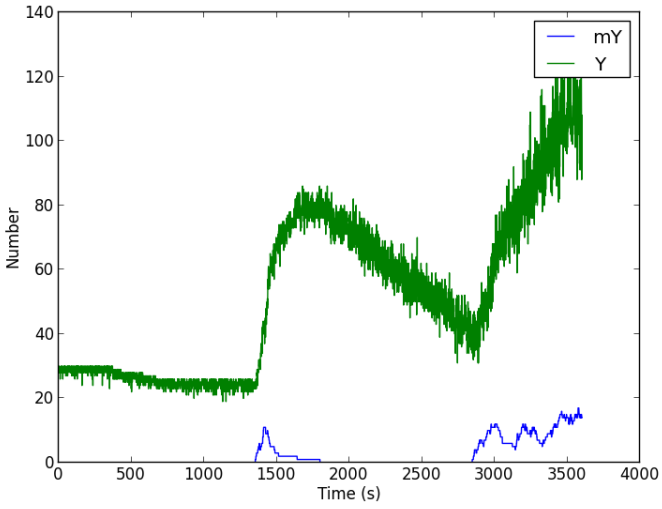


Figure 2.6: A single time trace from a single replicate showing the mRNA and Protein number over an hour period.

So now that you have a working understanding of how to use pyLM to do more complex tasks, try to answer a few of these questions:

1. What effect does the concentration of inducer have on the time to the cell turning 'on'?

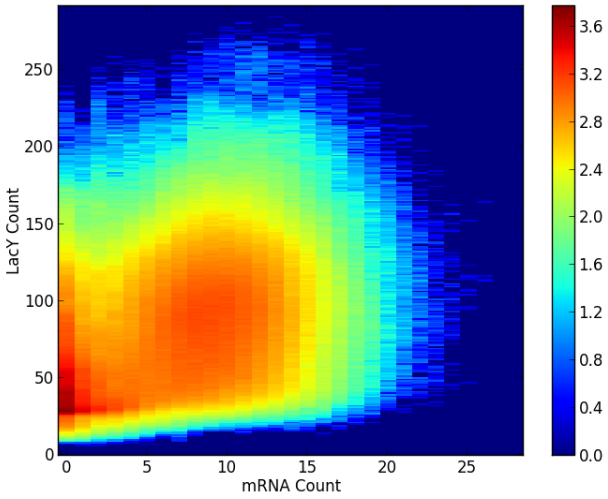


Figure 2.7: The logarithm of the count of times the state (number of protein and number of messenger) was traversed over the time series averaged over 100 simulations for the first hour of simulation. Two clusters are clear in the map, one close to 0 mRNA and 50 protein and other with about 100 protein and 10 mRNA.

2. How long would it take for a cell to become very induced (about 2000 copies of the protein)? Hint: try only a few simulations as they become slower after a very long time as species counts add up.

3. What would happen if the mRNA were longer lived, or if there were much larger bursts?

4. What effect would the extra looped state in the diagram above have on the switching time?

5. What would happen if protein did not degrade (dilute) on the timescale of a cell cycle?

2.2.2 Spatially Resolved

With this introduction to the *lac* switch, we can now move on and see how spatial inhomogeneity can affect stochasticity in the system. This section will take you through setting up your first spatially resolved simulation. Again, this was a simulation from [2]. The code to run this simulation can be seen in Listing 2.4.

Listing 2.4: `tut2.2-lac2stateRDME.py` — Code to set up and execute the *lac* two-state model with RDME.

```
1 from pyLM import *
2 from pyLM.units import *
3 from pySTDLM import *
4 from pySTDLM.PostProcessing import *
5 from pySTDLM.StandardReactionSystems import *
6 from pySTDLM.StandardCells import *
```

```
7
8  # Create our simulation object
9  latticeSpacing=nm(16)
10 sim=RDME.RDMESimulation(dimensions=micron(1.024,1.024,2.048), \
11                         spacing=latticeSpacing)
12
13 # Set up cell geometry
14 sim.buildCapsidCell(length=micron(1.8), diameter=micron(0.5), \
15                     membraneThickness=nm(32))
16
17 # Add the reactants and reactions
18 addLacTwoStateSystem(sim)
19
20 # Get handles to the different regions
21 mem='membrane'
22 cyt='cytoplasm'
23 ext='default'
24
25 # Crowd the cytoplasm
26 packFastGrowingEcoli(sim)
27
28 # Populate the model with particles
29 sim.addParticles(species='R2', region=cyt,  count=9)
30 sim.addParticles('O',   cyt, 1)
31 sim.addParticles('Y',   mem, 30)
32 sim.addParticles('I',   cyt, 7224*2)
33 sim.addParticles('Iex', ext, 7224*2)
34
35 # Set up the times
36 sim.setTimestep(microsecond(50))
37 sim.setWriteInterval(1)
38 sim.setLatticeWriteInterval(1)
39 sim.setSimulationTime(7200)
40
41 # Save the simulation state to a file
42 filename='T2.2-lac2state.lm'
43 sim.save(filename)
44 # Run the simulation for 10 hours
45 reps=10
46 sim.run(filename, "lm::rdme::MpdRdmeSolver", reps)
47
48 # Post-processing
49 for i in range(1,reps+1):
50         print("plotting")
51         plotTraceFromFile(filename, ['mY','Y'], i, "Trace.mY.Y.%d.png"%(i))
```

The Python script will set up a cell with realistic crowding, internal and external inducer. First off, spatial simulations require a lattice spacing to be defined. This spacing is the side length for the volume that is assumed to be "homogeneously" stirred (which allows CME to be applied in each subvolume). While this number can be too large, or two small and the CME assumption breaks down, the discussion of this is beyond the scope of this tutorial. Eight, 16 and 32 nanometers are all fairly common lattice spacings. Se then define the overall volume of the simulation. In this case, we have created a cuboid shaped region with an aspect ratio of 2:1 (Lattice Microbes only

supports cuboid shaped simulations).

At this point, let us take a look at the anatomy of an RDME simulation. A schematic of one is shown in Figure 2.8. As we have already described subvolumes and the overall simulation domain, there are only a few things to note. In Lattice Microbes, particles live in a subvolume, and the actual location within the subvolume is not known, but we can think of a particle as living in the center for simplicity. What happens as time advances in the simulation is that particles within a subvolume can react, either by themselves or with other particles, or they can diffuse to one of the 27 (or fewer) direct neighboring subvolumes. At the current state, Lattice Microbes can support 8 different particles at a given site, with a total of 256 possible species/particle types. Therefore the maximum occupancy of the total simulation domain is $8 \times subvolumes$. In practice, due to the algorithm, you will want to keep this number below 10% of the total allowable occupancy.

Within the overall simulation domain there is a notion of regions. Regions are associated with a particular 'site type'. These could be things like a membrane or the cytoplasm, or something completely not biologically related. All the subvolumes assigned a site type that defines what can happen in that volume. Regions are defined with possible chemical reactions that can occur in them, as well as what type of particles can diffuse into and out of them, as well as the diffusion rate into/out of and between these sites. For instance, one protein might diffuse onto a membrane and have a lower diffusion rate on the membrane than in the cytosol. Or a particular reaction may only occur in the cytosol. Therefore, regions allow us to define what particles can do.

However, regions need not be contiguous space. As shown in the figure, there are two areas in the simulation that are the red region, and one that is the purple. Therefore, all the reactions that can occur in one of the red region can also occur in the other even though they are not connected spatially. This concludes our brief aside on the anatomy of an RDME simulation.

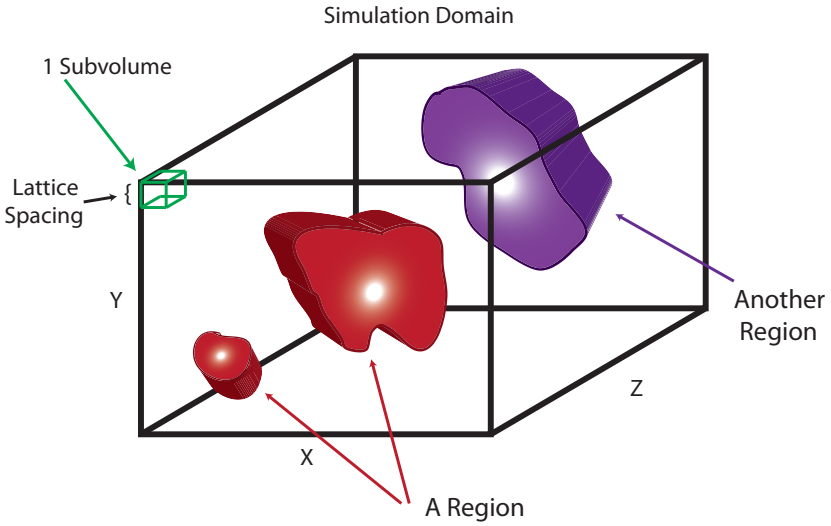


Figure 2.8: A schematic of a hypothetical RDME simulation.

From an analysis of the previous code, it can be seen that one DNA particle 'O' is placed (in this case randomly in the volume) along with a number of lac proteins 'Y' that end up randomly on the membrane, nine repressor dimers 'R2' and quite a few internal and external inducer molecules. Simulation parameters are specified and the simulation then runs for 10 minutes. On a single GPU, this simulation takes about 32 hours to run, so you may want to get a coffee while it runs. Alternatively, we have included a copy of the simulation results in the tutorial files and you can run the script with the `sim.run(filename...)` line commented out, and all the other code will work perfectly. A similar figure to that plotted in the previous section is created with the same plotting command.

Perhaps more interesting, however, than this process is the visualization. If you installed Lattice Microbes and the plugin for VMD, you will be able to open spatially resolved simulations. To start, open up VMD, then load the file into the dialog box. The result should look something like that seen in Figure 2.9; if "Lattice Microbes" is not the file type, then the plugin is not installed correctly—please see the Lattice Microbes User Guide. At first, the simulation will look rather boring, like that in Figure 2.10. The spherocylinder was drawn to show the shape of the cell, and several other shapes can be drawn, including cubes, spheres, etc. Delete this by going to the "Tk Console" in the "Extensions" menu and entering the command `graphics top delete all` to reveal just the particles.

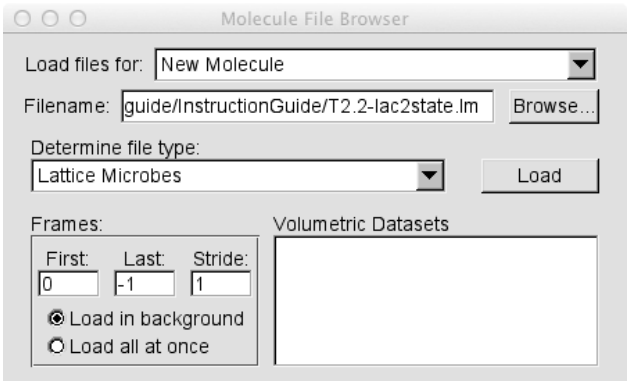


Figure 2.9: An example of the open dialog box in VMD.

Next, we should pretty the simulation up some, as every particle is loaded the same color. Open the "Graphical Representations" tool from the "Grpahics" menu and create representations similar to what you seen in Figure 2.11a. In VMD each particle is known by a 'type' which correspond to the order you registered species with the simulation via the `defineSpecies()` function in pyLM starting with 0 as the first index. You will probably have to play with the "Sphere Scale" to see all the particles. In this case, van Der Waals ("VDW") representation was used, but the "Points" representation is equally useful.

This should result in a view similar to what you seen in Figure 2.11b. From here you can use the slider in the "VMD Main" window to play through the frames of the movie. Play around to see what else you can do.

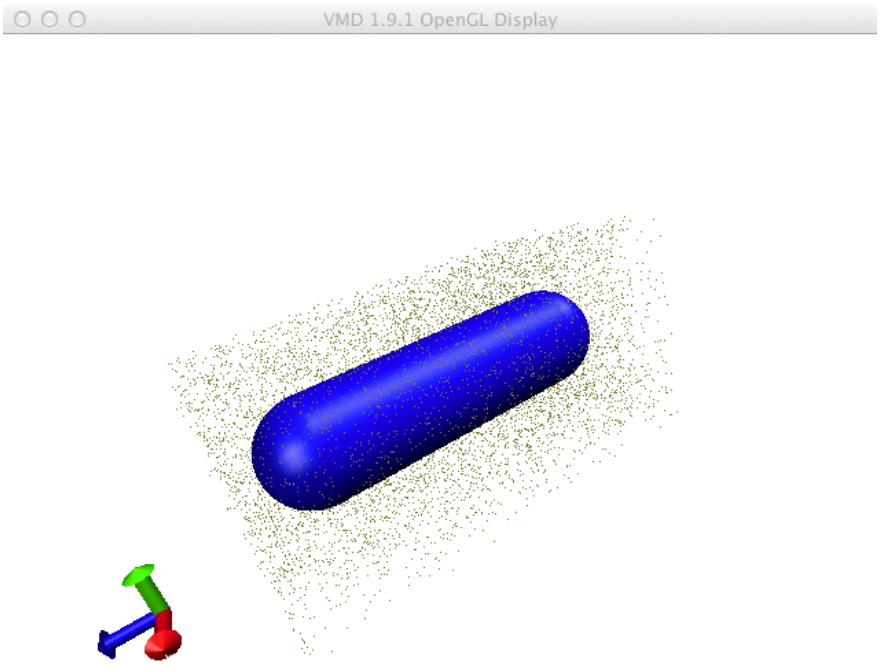


Figure 2.10: A Lattice Microbes simulation initially loaded.

Now, often you will want to make sure the simulation is set up correctly by checking the site locations. VMD does not automatically load these, but it can be configured to load them. Close VMD and reopen the program. Before loading the simulation file, run from the console the following commands:

```
set env(LM_CREATE_OBSTACLE_ATOMS) 1
set env(LM_CREATE_SITE_ATOMS) 1
```

Site types are named "site" in VMD and can be selected similar to the way shown in Figure 2.12. The figure also demonstrates how to cut through the simulation using the selection "x $<$ 51000". In VMD the natural units are in angstroms, so the simulation is about 10240 angstroms in x and y and 20480 in z.

One particular note of interest is that VMD automatically loads the first replicate, if more than one replicates were run. In order to load a different replicate, you must follow the procedure for showing obstacle and site types just mentioned but with the Tk Console command:

```
set env(LM_REPLICATE) #
```

Where the # symbol is the replicate number. Lattice Microbes files are standard HDF5 files, meaning they can be opened with HDFView (which is available from `http://www.hdfgroup.org/HDF5/`) and other features of the simulation can be inspected such as data layout, parameters, etc. However, this is an advanced topic and beyond the scope of these tutorials. The user

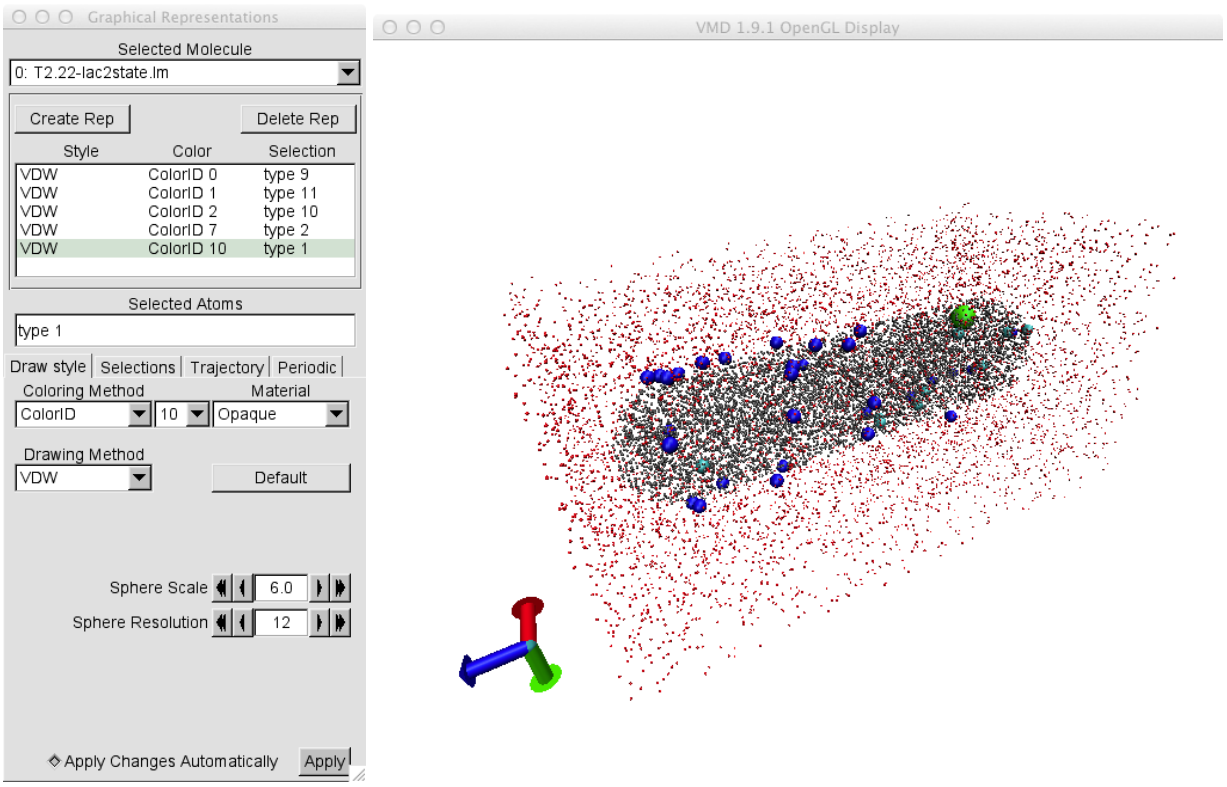


Figure 2.11: a) Graphics Representation window showing what one such particle representation might look like. b) A view showing external inducer (red), internal inducer (gray), *lac* permease protein (blue), *lac* repressor (cyan) and the DNA particle (green).

is, however, encouraged to explore the file if they are curious. **Note: Do not open a Lattice Microbes file while the simulation is running! In rare cases this can cause the data to be corrupted. If you must, we recommend you make a copy of the simulation file and open the copy.**

This concludes our brief introduction to spatially resolved simulations in Lattice Microbes. Much additional functionality for setting up RDME simulations exists and can be found in the Lattice Microbes Reference Guide for pyLM available at the website.

2.3 Debugging and Command Line Execution

At this point, it is appropriate to address two topics that will no doubt be required if you use Lattice Microbes to solve anything but a toy problem. This section describes the process for debugging simulations as well as how to invoke Lattice Microbes simulations from the command line, which is useful when running on supercomputers.

2.3.1 Debugging

Sometimes a simulation will not work on the first try. Python sometimes has the downfall of being too kind to errors, and we as programmers are sloppy. As such, it may fail without a word and

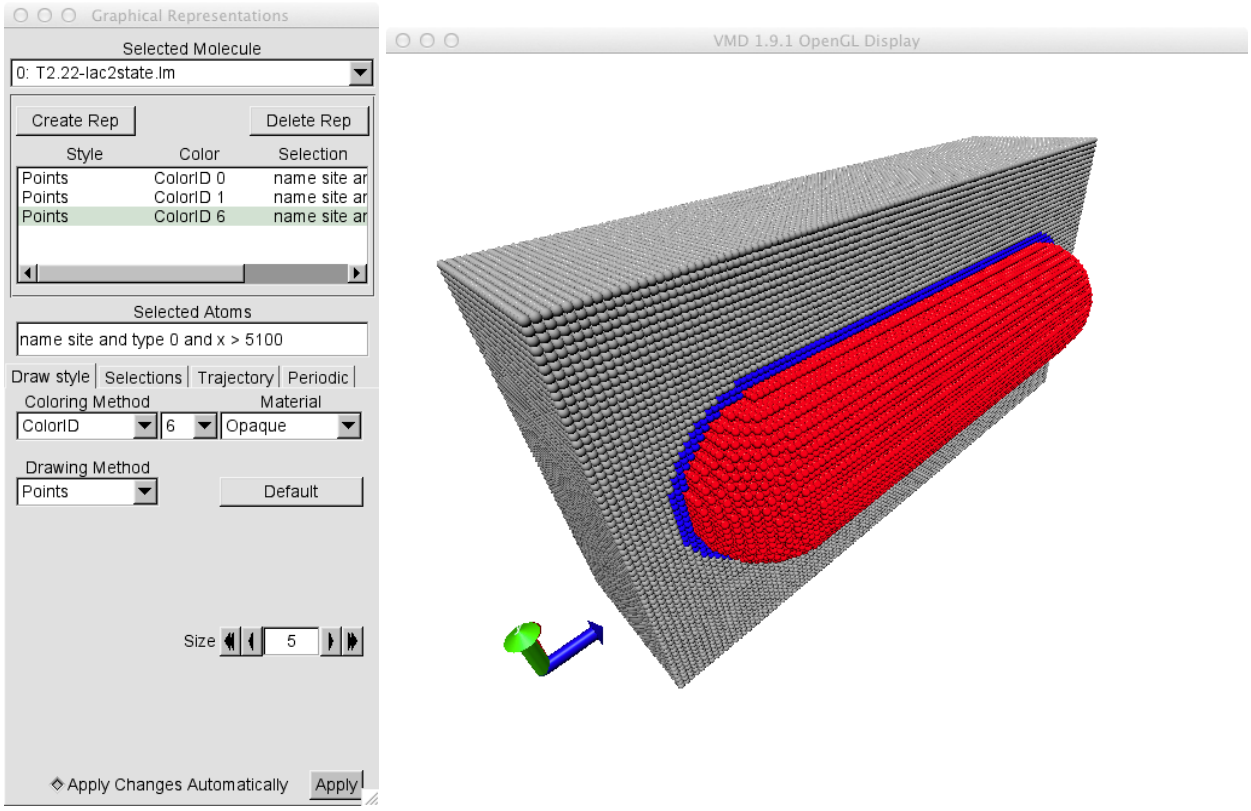


Figure 2.12: a) An example of a graphic representation of the site types. b) Here the extracellular (also called default) is shown in gray, the membrane, which is about 2 sites thick, is shown in blue and the cytoplasm is shown in red. Both the extracellular and the membrane are cut only show those with x less than 5100 angstroms.

not perform the actions you requested. Therefore, you will want to be able to debug what is going on in pyLM as it performs its task. It provides this capability through a Python "logger" which can be activated in the script. We recommend you always activate this functionality when you are designing a simulation and then deactivate it when running production runs. To enable the logging, merely add the following code to your simulation:

Listing 2.5: Code to enable debugging output in pyLM.

```
1 from pyLM import LMLogger
2 LMLogger.setLMLogConsole(logging.WARNING)
```

Five levels of debugging information are available; listed in order of increased output: `logging.DEBUG`, `logging.INFO`, `logging.WARNING`, `logging.ERROR`, `logging.CRITICAL`. Different levels are used throughout pyLM and pySTDLM to inform the user what is going on, as well as any other errors or warning that might have occurred. As an example we have included a test file with an error in Listing 2.6. Try running the code with the various different levels to try to identify the error.

Listing 2.6: `tut2.3-debugging.py` — Code with an error.

```
import numpy as np
import matplotlib.pyplot as plt
from pyLM import *
from pyLM.units import *
from pySTDLM import *
from pySTDLM.PostProcessing import *

# Turn on logger
from pyLM import LMLogger
LMLogger.setLMLogConsole(logging.WARNING)

# Constants
V  = 1.0e-15       # L
NA = 6.022e23      # molecules/mole
kf = 1.07e5/(NA*V) # /M/s
kr = 0.351         # /s

# Create our CME simulation object
sim=CME.CMESimulation()

# define our chemical species
species = ['A', 'B', 'C']
sim.defineSpecies(species)

# Add reactions to the simulation
sim.addReaction(reactant=('A','B'), product='C', rate=kf)
sim.addReaction(reactant='C', product=('A','B'), rate=kr)

# Set our initial species counts
sim.addParticles(species='a', count=1000)
sim.addParticles(species='B', count=1000)
sim.addParticles(species='C', count=0)

# Define simulation parameters: run for 10 seconds, saving data every ms
sim.setWriteInterval(microsecond(30))
sim.setSimulationTime(30)
sim.save('T2.3-bimol.lm')

# Run 1 replicates using the Gillespie solver
sim.run(filename='T2.3-bimol.lm', method="lm::cme::GillespieDSolver",
    replicates=1)

# Plot the solution
plotTraceFromFile(filename='T2.3-bimol.lm', species=['A','C'], replicate=1,
    outfile='BimolecularStoch.png')
```

In addition, you may want to add your own debugging to the output. As such you can simply add lines such as:

Listing 2.7: Custom debugging output.

```
LMLogger.error("This is an error: %d"%var)
LMLogger.info("General info for me...")
```

```
3 LMLogger.warning("Are you sure you want to do this: %s"%"Something dangerous")
```

2.3.2 Command Line

While launching the simulations from within a Python script is what you will want to do during testing, it may not be the best way to launch the code for a production run. For example, some clusters and supercomputers do not have the correct Python environment, or allow interpreted languages to be run as the primary executable. Therefore, you may want to launch your job from the command line using the `lm` executable. Luckily, you can prototype your simulation in Python as discussed before, then comment out the `sim.save(...)` and `sim.run(...)` commands, run the simulations on a supercomputer, then run the Python script on the already produced output, to leverage the power of scripting for custom post-processing.

Running Lattice Microbes from the command line can be fairly straightforward. For example, running the simulation in Listing 2.4 one would execute the following command:

```
lm  -r 0-2 -sp -sl lm::rdme::MpdRdmeSolver -f T2.2-lac2state.lm
```

To dissect the command, the `-r 0-2` indicates to run three replicates with indices 0, 1 and 2, `-sp` indicates the simulation is spatially resolved (i.e. RDME), `-sl lm::rime::MpdRdmeSolver` is the particular solver to use to sample the stochastic equations and `-f T2.2-lac2state.lm` indicates the file that should be used.

On a supercomputer or cluster where multiple CPU's and/or multiple GPU's are to be used, you may need to specify additional arguments. For example, on a system with 2 GPU's per node, and 8 CPU cores, you could split up 16 replicates as such:

```
mpirun -np 8 lm -r 0-15 -sp -sl lm::rdme::MpdRdmeSolver -gr 1/16 \
       -cr 1/2 -f T2.2-lac2state.lm
```

which will dedicate one sixteenth of a GPU and a half of a CPU core to each replicate. Alternatively, the simulations will be run in two sets if you only run:

```
mpirun -np 8 lm -r 0-15 -sp -sl lm::rdme::MpdRdmeSolver -gr 1/8 \
       -cr 1 -f T2.2-lac2state.lm
```

However, it should be noted that usually on large supercomputers or for complex jobs you may want a dedicated CPU for the data input/output thread. This thread is launched by default per simulation, so you could subtract that number from your total CPU cores.
To get a complete listing of the commands run `lm -h` which should give a prompt similar to:

```
Usage: lm (-h|--help)
Usage: lm (-v|--version)
Usage: lm [OPTIONS] (-l|--list-devices)
Usage: lm [OPTIONS]
Usage: lm [OPTIONS] (-s|--script) script_filename [(-sa|--script-args) script_arguments+]
```

```
Usage: lm [OPTIONS] [SIM_OPTIONS] (-f|--file) simulation_filename

OPTIONS
  -c num_cpus       --cpu=num_cpus
   The number of CPUs on which to execute (default all).
  -cr num           --cpus-per-replicate=num
   The number of CPUs (possibly fractional) to assign per replicate, e.g. "2", "1/4" (default 1).
  -g cuda_devices   --gpu=cuda_devices
   A list of cuda devices on which to execute, e.g. "0-3", "0,2" (default 0).
  -gr num           --gpus-per-replicate=num
   The number of cuda devices (possibly fractional) to assign per replicate, e.g. "2", "1/4" (default 1).
  -nc               --no-capabilities
   Don't print the capabilities of the CUDA devices.
  -nr               --no-reserve-core
   Don't reserve a CPU core for the output thread.

SIM_OPTIONS
  -r replicates     --replicates=replicates
   A list of replicates to run, e.g. "0-9", "0,11,21" (default 0).
  -sp               --spatially-resolved
   The simulations should use the spatially resolved reaction model (default).
  -ws               --well-stirred
   The simulations should use the well-stirred reaction model.
  -sl solver        --solver=solver
   The specific solver class to use for the simulations.
  -ck               --checkpoint=interval
   Enable checkpointing with the given interval as hh:mm:ss (default 00:00:00 -- disabled).
```

2.4 Advanced Post-Processing: The Min System

In this section, we will use another popular stochastic simulation to demonstrate more advanced post-processing using Python. In particular, we will use the Min protein system [5, 6], shown schematically in Figure 2.13, that places the division plane in the center of the cell during division of *E. coli*. As the system evolves from the initial state, an oscillation of the two proteins from one pole of the cell to the other at a frequency close to 1 per minute, with the MinE protein stimulating the detachment of the membrane bound MinD protein activating its ATPase activity. One particular interesting fact is that certain mutants of *E. coli* show anomalous Min system behavior that cannot be captured with deterministic PDE simulations and require the use of stochastic simulation techniques like the RDME.
Again, the simulation setup is fairly easy, and this Min system is available as a standard reaction system in pySTDLM. The code to setup up, run and analyze the simulation is shown in Listing 2.9. Everything before post-processing is similar to that in Section 2.2.2 so will not be discussed.

Listing 2.8: `tut3.1-minsystem.py` — Code to set up and execute the Min protein system with RDME.

```
1 from pyLM import *
2 from pyLM.units import *
3 from pySTDLM.PostProcessing import *
4 from pySTDLM.NetworkVisualization import *
5 import math as m
6 import numpy as np
7 import matplotlib.pyplot as plt
8
9
```

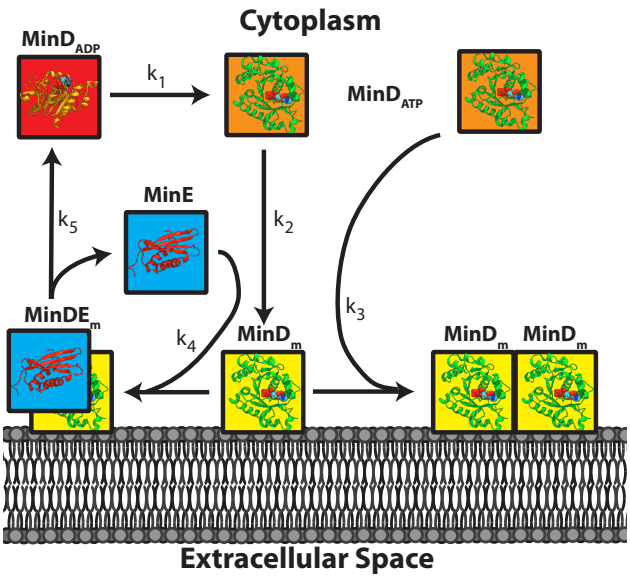


Figure 2.13: A schematic of the Min protein system model [1]. ATP bound MinD attaches to the membrane and can long fields of bound MinD. MinE binds to membrane bound MinE stimulating its ATPase activity and causing the MinD to leave the membrane where, once back in the cytoplasm, another ATP can replace the bound ADP.

```
10 # Create our simulation object
11 sim=RDME.RDMESimulation(dimensions=micron(1.024,1.024,4.096), spacing=nm(32))
12
13 # Build a capsid cell
14 sim.buildCapsidCell(length=micron(4), diameter=micron(1), \
15                     membraneThickness=nm(64))
16
17 # define our chemical species
18 sim.defineSpecies(['minDadp', 'minDatp', 'minDm', 'minE', 'minDEm'])
19
20 # Modify the cytoplasm to add diffusion rates and reactions
21 sim.modifyRegion('cytoplasm') \
22         .setDefaultDiffusionRate(2.5e-12) \
23         .addReaction(reactant='minDadp', product='minDatp', rate=0.5)
24
25 # Modify the membrane to add reactions
26 V=3.14*4.0e-6*0.5e-6*0.5e-6*1000.0 # Liters
27 N_A=6.022e23                  # molecules/mole
28 scalar=1.0/(N_A*V)
29
30 sim.modifyRegion('membrane') \
31         .setDefaultDiffusionRate(2.5e-12) \
32         .setDiffusionRate(species='minDm',  rate=1e-14) \
33         .setDiffusionRate(species='minDEm', rate=1e-14) \
34         .addReaction( 'minDatp', 'minDm',   rate=0.78) \
35         .addReaction(('minDatp','minDm'), ('minDm','minDm'),rate=9e6*scalar) \
36         .addReaction(('minDm','minE'),     'minDEm',     rate=5.56e7*scalar) \
37         .addReaction( 'minDEm',           ('minE','minDadp'),rate=0.7)
```

```
38
39 # Set diffusive properties between regions
40 sim.setTransitionRate(species='minDatp', via='cytoplasm', \
41                       to='membrane', rate=2.5e-12)
42 sim.setTransitionRate(species='minDadp', via='cytoplasm', \
43                       to='membrane', rate=2.5e-12)
44 sim.setTransitionRate(species='minE', via='cytoplasm', \
45                       to='membrane', rate=2.5e-12)
46 sim.setTransitionRate(species='minDatp', to='cytoplasm', \
47                       via='membrane', rate=2.5e-12)
48 sim.setTransitionRate(species='minDadp', to='cytoplasm', \
49                       via='membrane', rate=2.5e-12)
50 sim.setTransitionRate(species='minE', to='cytoplasm', \
51                       via='membrane', rate=2.5e-12)
52
53 # Populate the model with particles
54 sim.addParticles(species='minDatp', region='cytoplasm', count=1758)
55 sim.addParticles(species='minDadp', region='cytoplasm', count=1758)
56 sim.addParticles(species='minE',    region='cytoplasm', count=914)
57
58 # Set simulation Parameters
59 sim.setTimestep(microsecond(50))
60 sim.setWriteInterval(0.5)
61 sim.setLatticeWriteInterval(0.5)
62 sim.setSimulationTime(600)
63
64 # Run simulation
65 filename="T3.1-MinSystem.v3.lm"
66 sim.save(filename)
67 sim.run(filename, method='lm::rdme::MpdRdmeSolver', replicates=1)
68
69
70 ###################
71 # Post-Processing #
72 ###################
73 # Plot Kymogrpahs for MinE and MinD
74 plotOccupancyKymograph(filename, species='minDm', \
75                                     filename='MinDmKymo.png')
76 plotOccupancyKymograph(filename, species='minE', \
77                                     filename='MinEKymo.png')
78
79 # Calculate the Oscillation Frequency
```

The next two commands plot a kymograph of the occupancy of that (x,y) coordinate of the z slice as a function of time. Currently, only slices in z are supported, but this will be expanded in a future release of pyLM. The occupancy is computed as the number of sites that contain a particular particle divided by the total number of sites in that slice. The kymograph for membrane bound MinD can be seen in Figure 2.15.

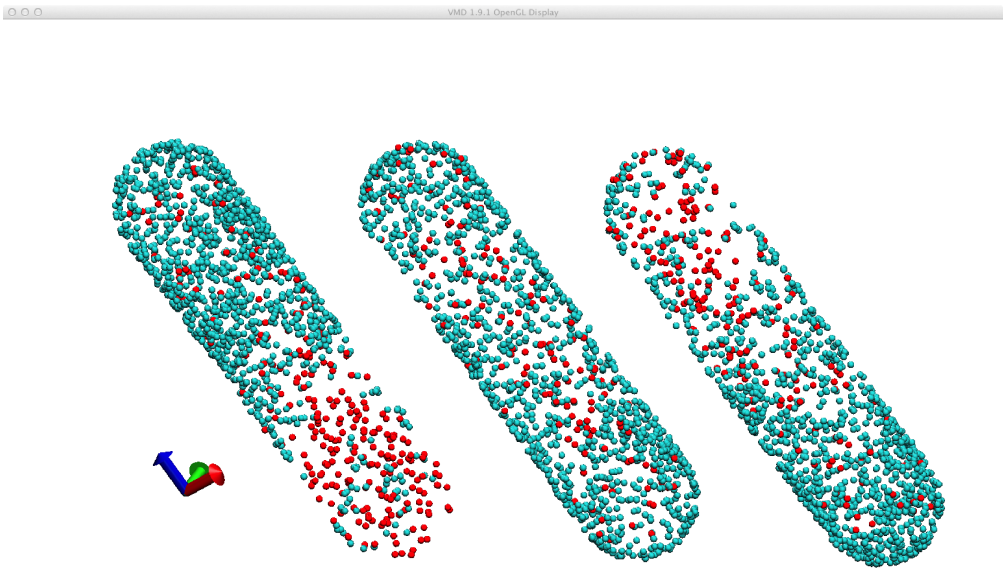


Figure 2.14: Three times during the simulation showing membrane bound MinD (cyan) oscillating from pole to pole with MinE (red) trailing behind the wavefront.

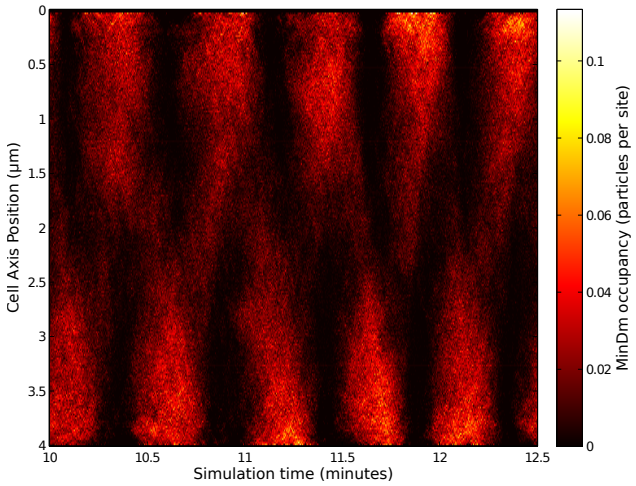


Figure 2.15: End to end oscillations of the membrane bound MinD as the system progresses in time.

2.5 Advanced Uses: Merging RDME with CME

This section will show you how to use a feature of pyLM that allows custom control of the simulation states the simulation progresses. In order to run the examples in this section your HDF5 library needs to have been compiled with the `--enable-threadsafe` option. If the example exhibits random crashing, you will likely have to recompile your HDF5 library. In this section we will create a subclass of the MPD RDME Solver class that will perform custom operations during at a specified time interval (in this case, every time the simulation state is written to file).

This simulation mocks the production of a protein from its gene and the resulting consumption of

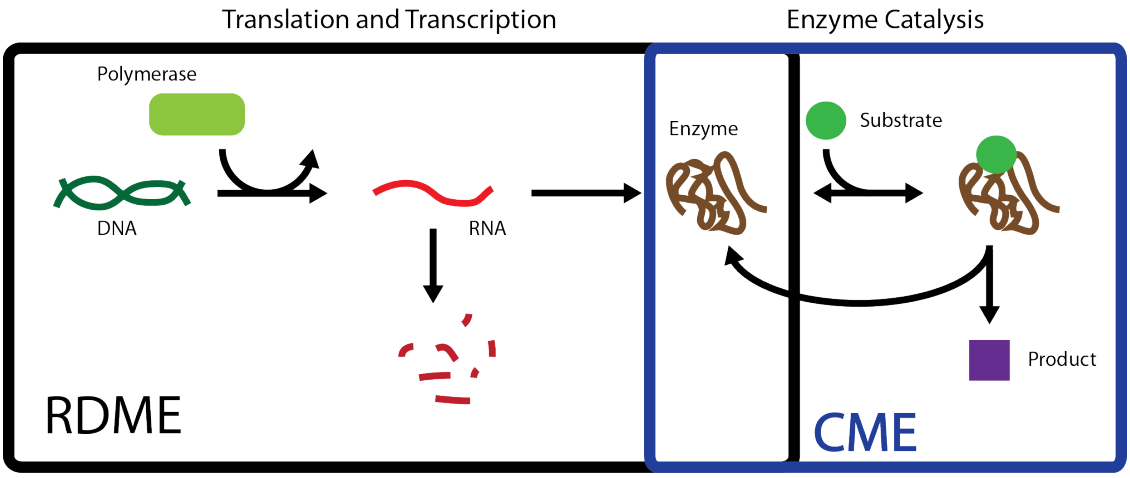


Figure 2.16: A schematic of the combined RDME/CME simulation.

a substrate by that enzyme (see Figure 2.16). To reduce the computational cost of carrying around many substrate and product molecules in the RDME lattice, they are treated as well stirred (i.e. the molecules are much smaller than the enzymes). However, the translational machinery interactions are spatially resolved. Finally, the products from the enzyme reaction are inserted into the RDME simulation domain as they are created, so that they could potentially interact with the DNA to create some sort of feedback (however, this case is not explicitly treated in this tutorial).

The point of the tutorial is to highlight the extendability of the RDME solver in pyLM. Right after turning on the logging, a new class is defined called "MyOwnSolver" that extends the MPD RDME Solver. A few of the internal variables are then created including initial timestep, species and time trace arrays. The meat of the solver is in the function `hookSimulation(...)` which is a time-based interrupt to the Lattice Microbes simulation. Essentially, this function is called every time a lattice is written (the same frequency as that set in `setLatticeWriteInterval(...)`). The RDME simulation pauses, passes the current `time` and `lattice` to this function and pretty much any Python code can be executed.

Listing 2.9: `tut3.2-rdmecme.py` — Code that shows how an RDME and CME simulation can be coupled using a time-based interrupt.

```
1  import os
2  import random
3  import time as tm
4  from pyLM import CME
5  from pyLM import RDME
6  from pyLM import LMLogger
7  from pyLM.units import *
8  import pySTDLM.PostProcessing as pp
9  import numpy as np
10
11 # Turn on Logging
12 import logging
```

```
LMLogger.setLMLogConsole(logging.INFO)
lmlog=LMLogger.LMLogger

### Define our own solver class derived from MpdHybridSolver
class MyOwnSolver(lm.MpdHybridSolver):
        # Timestep Variables
        tsnum=0
        curt=0.0
        delt=ms(250.0)

        # Species Variables
        numS=10000
        numE=0
        numES=0
        numP=0

        # Time Traces
        times=[]
        traceS=[]
        traceE=[]
        traceES=[]
        traceP=[]

        # The hookSimulation method defined here will be called at every frame
        # write time.  The return value is either 0 or 1, which will indicate
        # if we changed the state or not and need the lattice to be copied
           back
        # to the GPU before continuing.  If you do not return 1, your changes
        # will not be reflected.
        def hookSimulation(self, time, lattice):
                print("")
                print("")
                lmlog.info("Hook at time: %f sec"%time)
                lmlog.info("Creating CME simulation...")
                curtime=time
                # Create simulation
                csim=CME.CMESimulation()
                csim.defineSpecies(['E','S','ES','P'])

                # Count enzymes in cell
                parts=lattice.findParticles(4,4)
                Erdme=len(parts)

                # Add reactions and particles
                k1=0.001 # molecules/s
                k2=0.1 # /s
                k3=0.2 # /s
                csim.addReaction(('E','S'),'ES', k1)
                csim.addReaction('ES',('E','S'), k2)
                csim.addReaction('ES',('E','P'), k3)

                csim.addParticles('E',  int(Erdme-self.numES))
                csim.addParticles('ES', int(self.numES))
                csim.addParticles('P',  int(self.numP))
```

```
66                  csim.addParticles('S',  int(self.numS))
67
68                  # Set time data
69                  csim.setWriteInterval(ms(1))
70                  csim.setSimulationTime(self.delt)
71
72                  # Save and run simulation
73                  filename='cmeSim.%d.lm'%self.tsnum
74                  lmlog.info("Saving %s..."%filename)
75                  tm.sleep(1)
76                  csim.save(filename)
77                  lmlog.info("Running CME simulation...")
78                  os.system("lm -r 0-1 -ws -sl lm::cme::GillespieDSolver -f %s"%
                        filename)
79                  tm.sleep(1)
80                  self.tsnum += 1
81
82                  # Read CME state
83                  lmlog.info("Postprocessing...")
84                  fHandle=pp.openLMFile(filename)
85                  S =pp.getSpecieTrace(fHandle, 'S')
86                  endidx=len(S)-1
87                  E =pp.getSpecieTrace(fHandle, 'E')
88                  ES=pp.getSpecieTrace(fHandle, 'ES')
89                  P =pp.getSpecieTrace(fHandle, 'P')
90
91                  ts = pp.getTimesteps(fHandle)
92                  tsShifted=[]
93                  for i in range(len(ts)):
94                          tsShifted.append(ts[i]+self.curt)
95                  self.curt = curtime
96                  self.times.extend(tsShifted)
97                  pp.closeLMFile(fHandle)
98
99                  # Add product to the RDME simulation
100                 for i in range(P[endidx]-self.numP):
101                         while True:
102                                 x=random.randint(0,lattice.getXSize()-1)
103                                 y=random.randint(0,lattice.getYSize()-1)
104                                 z=random.randint(0,lattice.getZSize()-1)
105
106                                 if lattice.getOccupancy(x,y,z) < 7:
107                                         # Add a product particle
108                                         lattice.addParticle(x,y,z,5)
109                                         break
110
111                 # Update Solver internals
112                 lmlog.info("Updating internals...")
113                 self.numS=S[endidx]
114                 self.numE=E[endidx]
115                 self.numES=ES[endidx]
116                 self.numP=P[endidx]
117                 self.traceS.extend(S)
118                 self.traceE.extend(E)
```

```
                self.traceES.extend(ES)
                self.traceP.extend(P)

                lmlog.info("Resuming RDME simulation...")
                return 1

        def saveTraces(self):
                allTraces=[self.times, self.traceE, self.traceS, self.traceES,
                    self.traceP]
                np.savetxt('T3.2-CMETraces.dat', np.transpose(allTraces))

        def plotTraces(self):
                plotStr="gnuplot plotter.gp"
                os.system(plotStr)
# End of MyOwnSolver class.  That's it!



# Create our simulation object
latticeSpacing = 32 #nm
sim=RDME.RDMESimulation(dimensions=micron(1.024,1.024,1.024), spacing=nm(
    latticeSpacing))

# define our chemical species
species = ['DNA', 'Polymerase', 'RNA', 'Enzyme', 'Product']
sim.defineSpecies(species)

# Modify the cytoplasm to add diffusion rates and reactions
reg=sim.modifyRegion('default')
reg.setDefaultDiffusionRate(2.5e-12)
reg.setDiffusionRate('DNA', 0.0)
# 480 nt long with rate of 70 nt/sec
reg.addReaction(reactant=('DNA', 'Polymerase'), product=('DNA', 'Polymerase',
    'RNA'), rate=0.5e-3)
# Protein length ~ 160 aa with rate of 40 aa/sec
reg.addReaction('RNA', ('RNA','Enzyme'), 0.25)
# RNA lifetime of 2 minutes
reg.addReaction('RNA','', 1.0/120.0)
# Enzyme is around for about 1/3 the cell cycle
reg.addReaction('Enzyme','', 1.0/1200.0)

# Add particles
sim.addParticles(species='DNA', region='default', count=1)
sim.addParticles('Polymerase', 'default', 460)

# Set simulation Parameters
sim.setTimestep(microsecond(50))
sim.setWriteInterval(ms(250))
sim.setLatticeWriteInterval(ms(250))
sim.setSimulationTime(60)

rdmeFilename="T3.2-mixedRDMECME.lm"
sim.save(rdmeFilename)

```

```
170 # Create an instance of our local solver
171 solver=MyOwnSolver()
172 # Call the 'runSolver' method with the supplied solver
173 # then perform the post-processing
174 sim.runSolver(rdmeFilename, solver=solver)
175
176 # Post-process the data
177 solver.saveTraces()
178 rHandle=pp.openLMFile(rdmeFilename)
179 times=pp.getTimesteps(rHandle)
180 R=pp.getSpecieTrace(rHandle,'RNA')
181 E=pp.getSpecieTrace(rHandle,'Enzyme')
182 P=pp.getSpecieTrace(rHandle,'Product')
183 rdmeTraces=[times,R,E,P]
184 np.savetxt('T3.2-RDMETraces.dat',np.transpose(rdmeTraces))
185 solver.plotTraces()
186 pp.closeLMFile(rHandle)
```

So after simulating for a quarter of a second, the function is called and a new CME simulation for the enzyme catalysis is created. As an input, it uses the current number of enzyme particles in the CME (which it determines by the function call `findParticle(low,high)`, where the low and high are indices into the defined species list). The CME simulation rates and reactions are set and it is run using a system call `os.system(...)` using another instance of Lattice Microbes. Currently, the `runSolver(...)` command cannot be used here as it conflicts with some threads, however this will be rectified in a later version. Finally, when the CME simulation is completed, it reads the number of new product particles, adds them to the RDME simulation in random places using the `addParticle` call. Finally, it copies the CME simulation particle traces to internal variables of the "MyOwnSolver" class for use later.

The rest of the code is fairly boiler plate setting up an RDME simulation and running it. In this case we assume RNA is transcribed at about 70 nucleotides per second and protein is produced at about 40 amino acids per second. We assume a protein of about 160 amino acids length, and pick some numbers for different rates. Finally around line 170 some post processing occurs to plot out several traces using a system call to use Gnuplot.

The results of the simulation can be seen in Figures 2.17 and 2.18. While the results are relatively uninteresting, the code demonstrates how one would merge different types of kinetic methods. For example, the CME could be replaced by a standard set of continuous ordinary-differential equations, or by a steady state method like flux-balance analysis (as was done recently [7, Chapter 13]). Alternatively, one could implement a feedback system where the product particles caused the gene to be turned off with a certain probability. We encourage you to think outside the box and push our software to its limits.

This concludes our basic tutorials. While we only covered a fraction of the possible functionality available in Lattice Microbes/pyLM, we think this is enough to enable you to begin using it. Full documentation is available on the Lattice Microbes website.

If you are interested in giving feedback or suggestions, please email the developers. In addition,

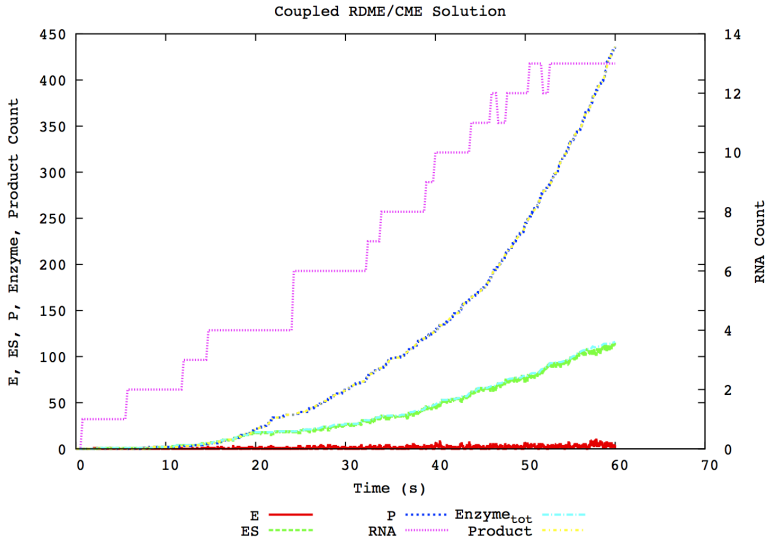


Figure 2.17: A schematic of the combined RDME/CME simulation performed in this section where transcription/translation are spatially resolved and enzyme catalysis is assumed to be a well-stirred process.

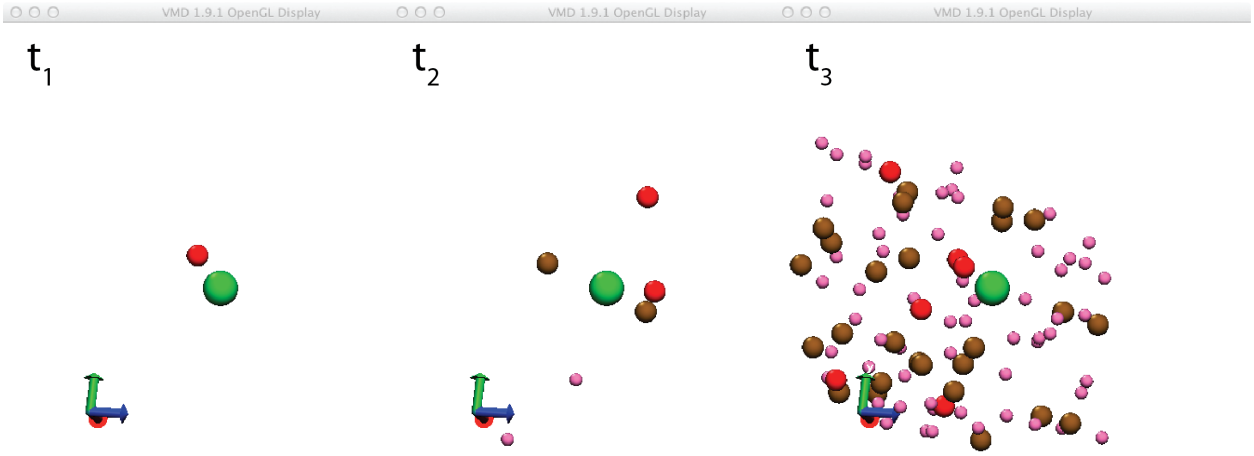


Figure 2.18: Three times during the simulation showing DNA (green), RNA (red), protein (brown) and product molecules added to the simulation (pink).

we look forward to contributions from the community. Furthermore, if you publish any results based on simulations from Lattice Microbes, we ask that you notify us by sending a citation to your paper so that we can follow what work is being done with the code.

Chapter 3

Examples

Many examples demonstrating features of pyLM exist both on the website (`http://www.scs.illinois.edu/schulten/lm/download/lm22/ExampleFiles.tgz`) and in the source code (`src/python/Examples`). Each demonstrates several different features of pyLM, and it is suggested that you read and work through all of the examples before starting to use pyLM for your project. It is recommended that you work through the problems in the order shown, as functionality documented in an earlier file is not described again in later files.
Various functionality is demonstrated in the examples, including:

- CME
    1. example-bimol.py – Demonstrates process of defining molecular species, reactions and initial conditions in a CME simulation
    2. example-bimolConc.py – Same as example-bimol.py using concentrations instead of particle numbers
    3. example-bimol-pp.py – Demonstrates the general trends for writing post-processing code for a CME simulation
    4. example-rnaprotein.py – An example of constitutive gene expression
    5. example-LotkaVolterra.py – An example of the well known Lotka-Volterra problem (predator-prey) simulated with stochasticity
    6. example-stochasticResonator.py – An example of a stochastic resonator problem with CME
    7. example-lac2state.py – This file demonstrates a more complex reaction scheme
- RDME
    1. example-MichaelisMenten.py – This demonstrates how to define a simulation domain in a 3D RDME simulation and defines an enzyme/substrate reaction system to be simulated in the domain
    2. example-minde.py – A demonstration of the popular Min system of *E. coli* showing how to construct default cell shapes, customize diffusion coefficients in regions and diffusion coefficients between regions

3. example-restart.py – This demonstrates how to restart an RDME simulation, however it is a hack as of now

- Advanced Setup

1. example-shapes.py – An example showing how to define complex objects such as boxes, spheres, tori, ellipses and intersections, unions and differences of the various objects
2. example-tightPackedCellArray.py – This example shows a pySTDLM feature that packs a cell shape into a tight regular grid spanning the whole RDME domain
3. example-extendrdme.py – This example shows how to extend the basic RDME solver with a hook that is run on every lattice write, allowing the user to modify the simulation based on the simulation state

Chapter 4

License and Copyright

Bibliography

[1] Peterson, J, Hallock, M, Cole, J, Luthey-Schulten, Z (2013) A Problem Solving Environment for Stochastic Biological Simulations. *Proceedings High Performance Computing Networking, Storage and Analysis Companion (SCC)*.

[2] Roberts, E, Magis, A, Ortiz, JO, Baumeister, W, Luthey-Schulten, Z (2011) Noise contributions in an inducible genetic switch: A whole-cell simulation study. *PLoS Comput. Biol.* 7:e1002010.

[3] Earnest, T, Roberts, E, Assaf, M, Dahmen, K, Luthey-Schulten, Z (2013) DNA looping increases range of bistability in a stochastic model of the *lac* genetic switch. 10.

[4] Roberts, E, Stone, JE, Luthey-Schulten, Z (2013) Lattice microbes: high-performace stochastic simulation method for the reaction-diffusion master equation. 3:245–255.

[5] Loose, M, Kruse, K, Schwille, P (2011) Protein self-organization: Lessons from the min system. *Annual Review of Biophysics* 40:315–336 PMID: 21545286.

[6] Fange, D, Elf, J (2006) Noise-Induced Min Phenotypes in E. coli. *PLoS Comput Biol* 2:e80.

[7] Cole, J, Hallock, M, Labhsetwar, P, Peterson, J, Stone, J, Luthey-Schulten, Z (2014) in *Computational Systems Biology 2nd Edition: From Molecular Mechanisms to Disease*, eds Kriete, A, Eils, R (Elsevier).