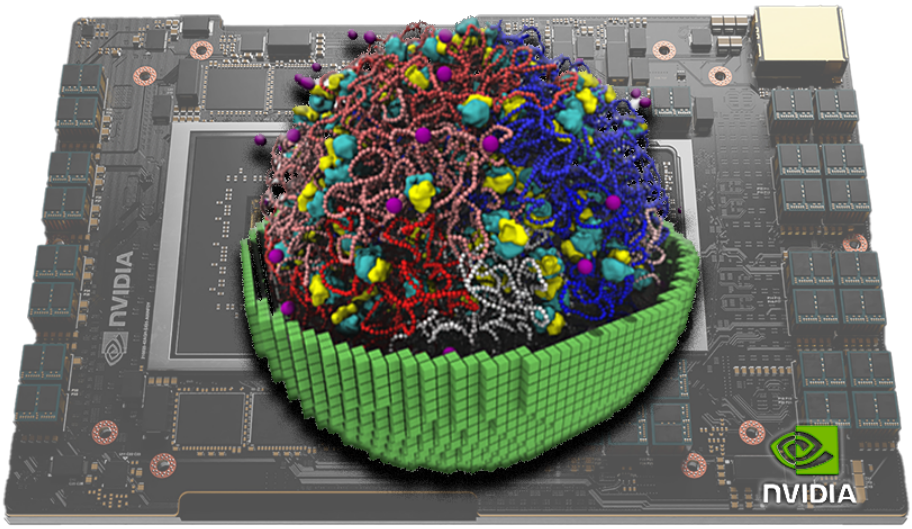
Lattice Microbes Tutorials



Advanced Computational Workshop
Simulations and Visualization of a Minimal Bacterial Cell
May 6-10, 2024

Tianyu Wu, Enguang Fu, Zane R. Thornburg, Troy A. Brier, Benjamin R. Gilbert,
John E. Stone (NVIDIA), and Zaida Luthey-Schulten

University of Illinois at Urbana-Champaign

Description

The Lattice Microbes Tutorials describe how to use the software to perform and analyze hybrid stochastic/deterministic simulations of a minimal bacterium and a spatially modeled yeast cell.
Lattice Microbes development is supported in part by NSF Science and Technology Center for Quantitative Cell Biology (NSF DBI-2243257) and NSF (MCB: 2221237 “Simulating a growing minimal cell: Integrating experiment and theory”).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Lattice Microbe Overview

Lattice microbe (LM) is a GPU accelerated stochastic simulation platform. The latest published version 2.4 is in https://github.com/Luthey-Schulten-Lab/Lattice_Microbes. In this tutorial, we use an updated version 2.5 under development.
LM is designed to simulate stochastic processes in biological cells using the Chemical Master Equation (CME) and the Reaction Diffusion Master Equation (RDME). With hooking, one can also incorporate metabolic reactions using Ordinary Differential Equation (ODE) solver and chromosome dynamics with Brownian Dynamics (BD) code.
The architecture of LM is shown in Figure 1.1. pyLM and jLM are two python-based Problem Solving Environments (PSE) developed in 2013 [4] and 2018, respectively. The API of pyLM and jLM can be found at https://luthey-schulten.chemistry.illinois.edu/software/LM2.4/API.html.
With pyLM and jLM, one can easily construct the system purely using Python. And the calculation is still performed in C++.

1.2 pyLM Overview

pyLM is a Problem Solving Environment (PSE) for biological simulations [4]. 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.

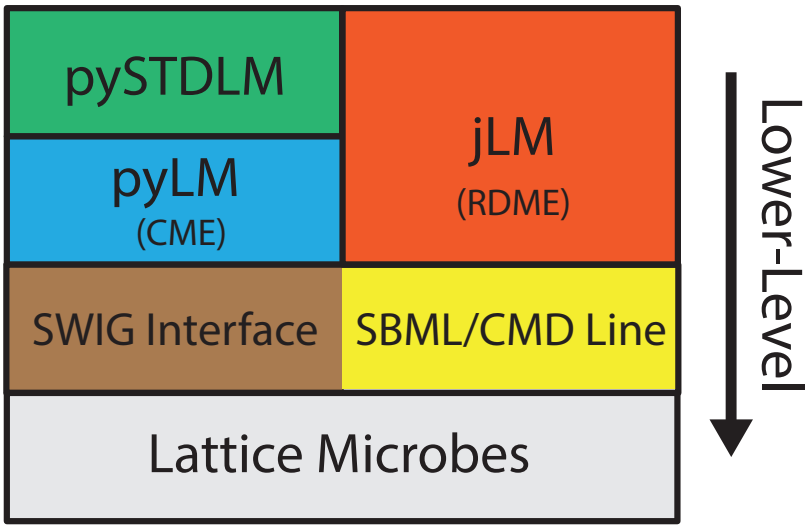


Figure 1.1: A schematic Architecture of the Lattice Microbes software.

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 Reference pdf" in the Luthey-Schulten webpage under Software/Documentations.

Capabilities of pyLM and pySTDLM

pyLM and the included library of standard systems pySTDLM provide a problem solving environment for setting up, running and analyzing stochastic biological simulations [4]. 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

1.3 jLM Overview

jLM is also written in python and designed as an interface for Jupyter Notebook and JupyterLab for RDME simulation. It offers a suite of functions to visualize the RDME simulation states like spatial regions, species and reactions.

Capabilities of jLM

jLM contains a set of different methods and functions to build up a RDME or RDME hybrid simulation. It contains functionality for specifying simulation setup including:

- Named species
- Initial counts and distributions
- Reactions and rates
- Spatial localization and definition
- Diffusion properties
- Visualization of the RDME simulation spatial geometry, reactions and species

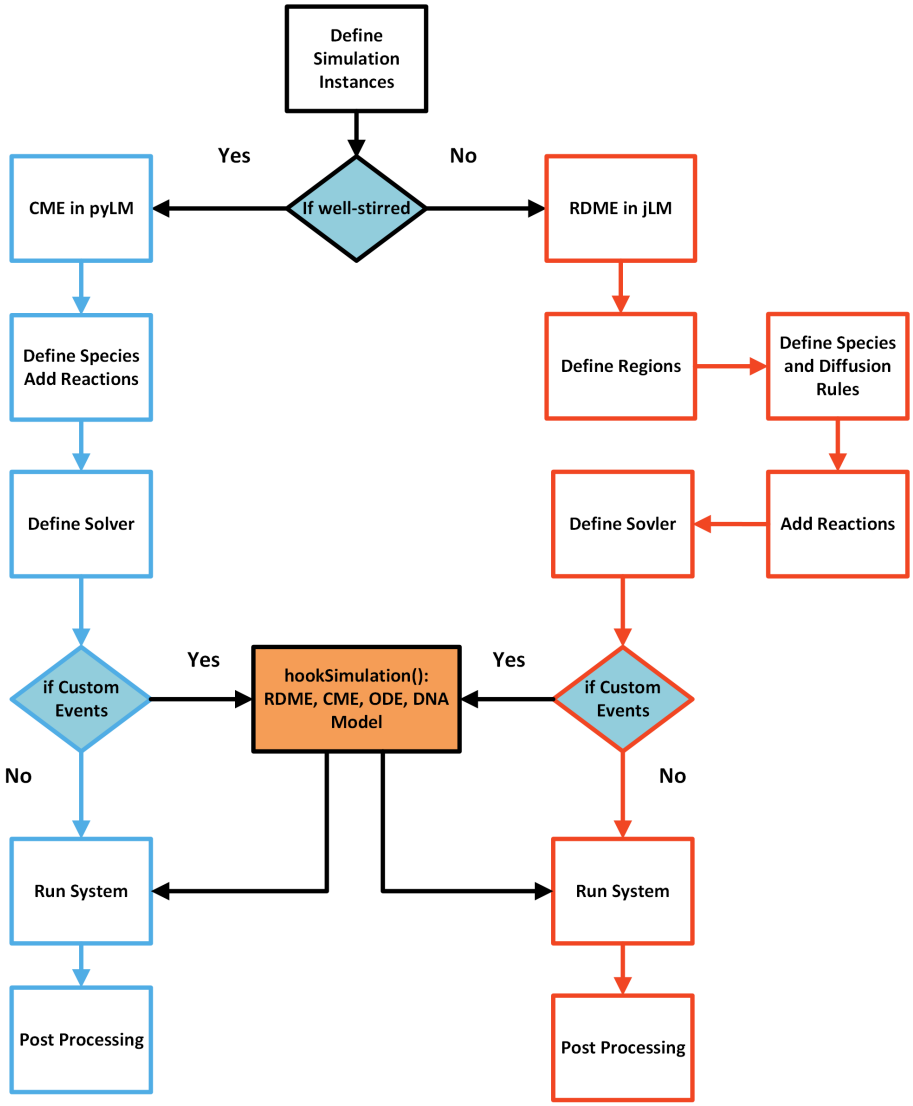


Figure 1.2: Workflow in pyLM and jLM. In jLM, regions and diffusion rules are defined. In the hookSimulation(), various algorithms can be incorporated with CME or RDME. In this tutorial, we will show hybrid CME/ODE and RDME/ODE algorithm implemented by Lattice Microbe

1.4 Stochastic Modeling

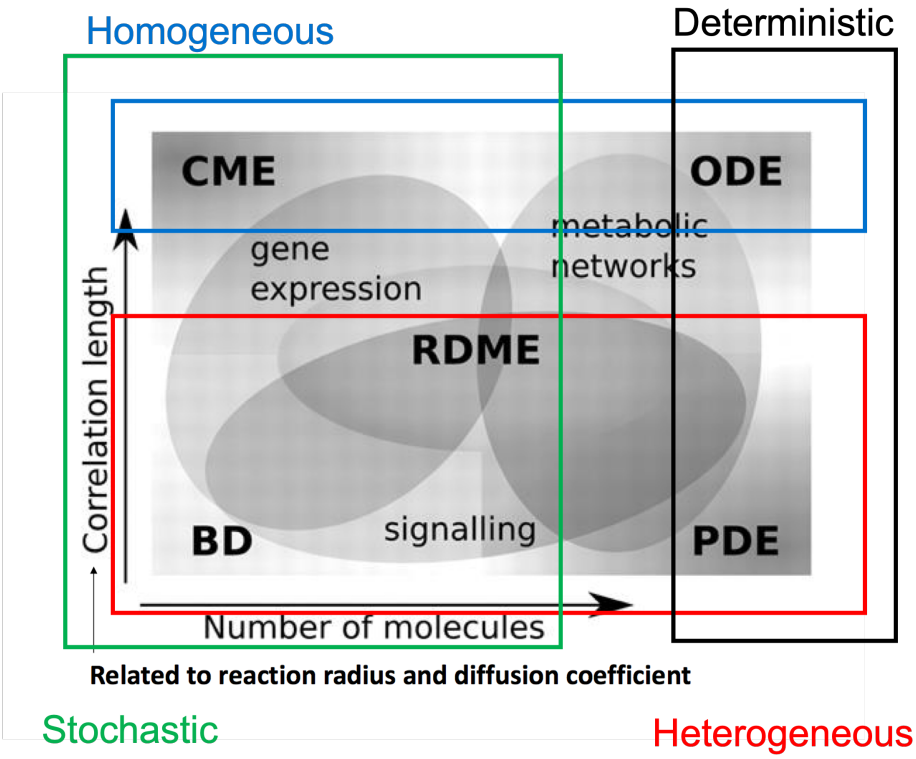


Figure 1.3: Lattice Microbe supports stochastic CME and RDME. With hookSimulation, CME and RDME can communicate with ODE (for metabolism), BD (for chromosome model) and PDE.

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):

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

RDME is used to simulate spatially heterogenous systems. Within each subvolume, the reactions are assumed to be well-stirred and described by CME.

Chapter 2

Tutorials for CME in pyLM

2.1 Tutorial 1: Bimolecular Reaction Solved Stochastically

In the first tutorial, we will motivate stochastic modeling with a simple example of a bimolecular reaction. The counts of reactants are small enought that you will see the fluctuations and variance of the stochastic result compared to the deterministic one.
You will use **Tut1.2-CMEBimol.ipynb** to run the CME of the Bimolecular reaction with pyLM in Lattice Microbe.
We will simulate the association/dissociation reaction of hypothetical molecules:

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

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 $k_r = 0.351/s$.
Under ODE representation using concentrations of speceis, the equations are:

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

Since we are using counts rather than concentrations in CME, the forward rate constant (2nd order) is divided by Avogadro's number and the volume of the cell which takes us to units of molecules per second. Conversion to stochastic rate constant is shown in Table 2.1.

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

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

Now go to Jupyter Notebook file **Tut1.2-CMEBimol.ipynb** (code in List 2.1).

Listing 2.1: `tut1.2-StochBimol` — Code used to solve the bimolecular reaction with the discrete/stochastic CME implementation using Lattice Microbes.

```
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) # convert from 1.07e5 /M/s to /count/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')
```

As mentioned in the introduction part, pyLM Problem Solving Environment is intensively used to construct the CME system. To do so, we need to import `pyLM` module.
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. `addReaction` function adds reactions to the defined species.
In Lattice Microbes, you need to specify 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.
The following lines with `addParticles` 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 must 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. In this tutorial, we use Direct Gillespie Algorithm to sample stochastic bimolecular reaction system.
By running the simulation, you will see a long standard output showing the number of finished replicates. Generally, CME simulation will finish rather quickly.
`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. In the post processing part, we showed two build-in functions in `pySTDLM`: `PostProcessing.plotTrace` for a single replicate and `PostProcessing.plotAvgVarFromFile` for the whole ensemble to visualize the results.
Figure 2.1 shows the deterministic result from ODE and stochastic result from CME.

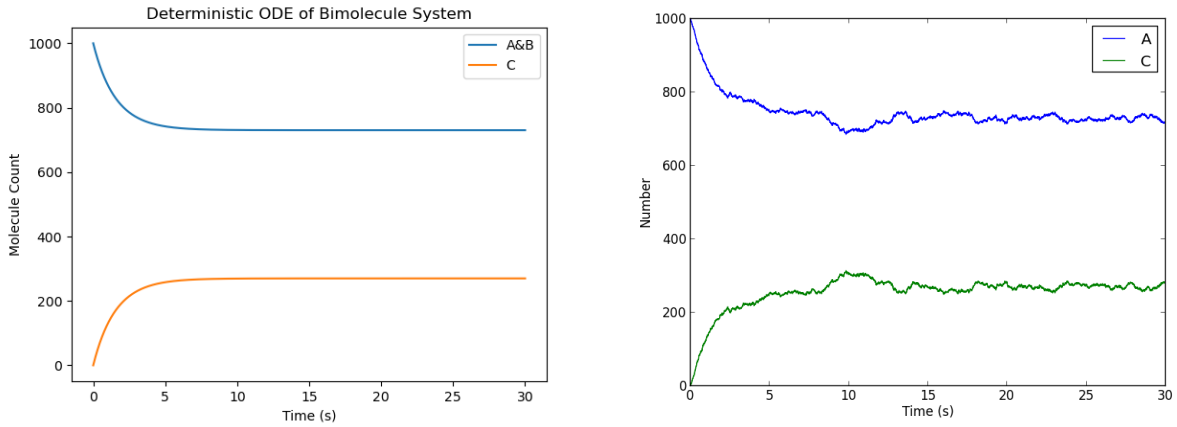


Figure 2.1: Left: Deterministic solution to the bimolecular reaction. Right: One representative stochastic solution of the bimolecular reaction.

In ODE result, what you should note is that the count of each species varies smoothly across the time course. However, the count of each reactant must be in integer numbers due to the discreteness of molecules. Further, the reactions occur via the collision between two molecules, so the change of count also in integer units. Under such reasoning, the ODE result is not accurate anymore for microscopic reactions where the counts of reactants are low. Stochastic modeling was designed to address this point.
You might note that the behavior in stochastic result is qualitatively the same to the deterministic ODE result, 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.
The averaged and variance of species' count across the whole replicates are shown in Figure 2.2. The left one is the result of ten replicates and the right one for 200.

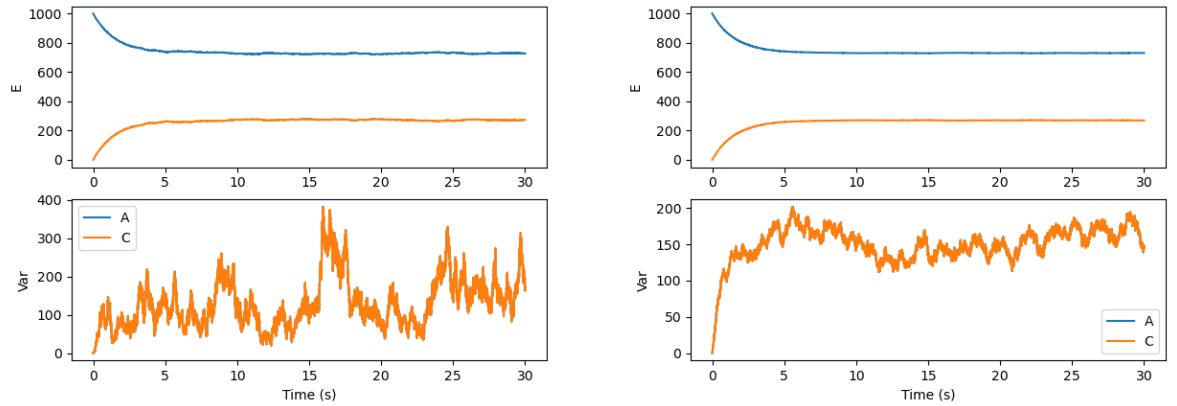


Figure 2.2: Left: Ensemble Average and Variance of 10 replicates in Stochastic Bimolecular Reaction. Right: Ensemble Average and Variance of 200 replicates in Stochastic Bimolecular Reaction.

Based on the preceding discussion, you are encouraged to explore the following questions.
Questions:
1. Try plotting the average and variance of different replicates by changing the number of replicates from 10 to 200 or even more. You need to restart the Jupyter Notebook kernal to start a new CME simulation. See Figure 2.2 for the situation with 10 and 200 replicates.
2. How many replicates are required to get a smooth average? How many for a smooth variance?
3. (Challenge) Can you derive an analytical solution for the system of equations? Try fitting the rate constants using the results of the stochastic simulations. You may find scipy.optimize.curve_fit() useful for this.
4. (Challenge) Does the theoretical average number from CME always be consistent/same with the count from ODE? You may find Page 420 in McQuarrie's classic paper STOCHASTIC APPROACH TO CHEMICAL KINETICS helpful.

2.2 Tutorial 2: Genetic Information Process Model in CME

Now go to Jupyter Notebook file **Tut2.1-GeneticInformationProcess.ipynb**. Here we will simulate a toy genetic information processing (GIP) model.
With 3 species and 4 reactions, the genetic information process starts from the transcription of gene to mRNA. mRNA can be translated to protein or degraded to its monomers. Protein can also be degraded. The reactions and rate constant is shown in Figure 3.2 and Table 2.2.
The rate constants are for DnaA Coding Gene (G_0001) of minimal cell. The first three rate constants are calculate based on the initial concentrations of nucleotides and amino acids charged tRNA in the Cell paper [5]. We also fix the gene copy number to 1 and assume initial count of mRNA to be 1. The initial count of protein is 0. The degradation rate of protein is estimated based half-life 25 hours [6].

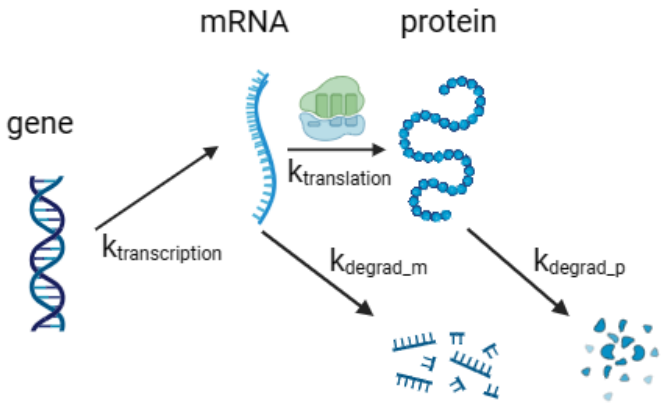


Figure 2.3: Genetic Information Processing Diagram

Table 2.2: Four reactions with their rate constants.

Names	**Reaction**	**Rate Constant (s^{-1})**	**Propensity (s^{-1})**
Transcription	Gene $\rightarrow$ mRNA	$k_{\text{transcription}} = 6.41E-4$	$k_{\text{transcription}}$
Degradation of mRNA	mRNA $\rightarrow \emptyset$	$k_{\text{deg,m}} = 2.59E-3$	$k_{\text{deg,m}} N_{\text{mRNA}}$
Translation	mRNA $\rightarrow$ mRNA + Protein	$k_{\text{translation}} = 7.20E-2$	$k_{\text{translation}} N_{\text{mRNA}}$
Degradation of Protein	Protein $\rightarrow \emptyset$	$k_{\text{deg,p}} = 7.70E-6$	$k_{\text{deg,p}} N_{\text{ptn}}$

In this given Jupyter Notebook, the default simulation length is **600 seconds** with write interval 1 second for **10 replicates**.

Listing 2.2: `Tut.2.1-GeneticInformationProcess` — Code to set up and execute the gene expression model with CME.

```
# Import Standard Python Libraries
import os
import numpy as np
import matplotlib.pyplot as plt

# Import pyLM Libraries
from pyLM import *
from pyLM.units import *
from pySTDLM import *
from pySTDLM.PostProcessing import *


# Constants
v0  = 6.41e-4        # Transcription, s^-1
d0 = 2.59e-3      # degradation of mRNA, s^-1
v1 = 7.2e-2         # translation, s^-1
d1 = 7.70e-6       # degradation of protein, s^-1

# Create our CME simulation object
sim = CME.CMESimulation(name='Gene Expression')

# Define our chemical species
species = ['gene', 'mRNA', 'ptn']
sim.defineSpecies(species)

# Add reactions to the simulation
sim.addReaction(reactant='gene', product=('gene','mRNA'), rate=v0)
sim.addReaction(reactant='mRNA', product='', rate=d0)
sim.addReaction(reactant='mRNA', product=('mRNA','ptn'), rate=v1)
sim.addReaction(reactant='ptn', product='', rate=d1)

# Set our initial species counts
sim.addParticles(species='gene', count=1)
sim.addParticles(species='mRNA', count=1)
sim.addParticles(species='ptn', count=0)

# Simulation time is 6300, entire cell life cycle.
```

```
writeInterval = 1
simtime = 600

sim.setWriteInterval(writeInterval)
sim.setSimulationTime(simtime)

filename = "./T2.1-geneExpression.lm"

os.system("rm -rf %s"%(filename)) # Remove previous LM file

sim.save(filename)

# Run multipled replicates using the Gillespie solver
# Will cost less than 1 minute
reps = 10


sim.run(filename=filename, method="lm::cme::GillespieDSolver", replicates=reps
    )

# Post-processing
fileHandle = PostProcessing.openLMFile(filename) # Create h5py file handle

plotfolder = './plots_gene_expression/'

if not os.path.exists(plotfolder):
    os.mkdir(plotfolder)

# Plot the average and variance of mRNA and protein
for i_specie, specie in enumerate(species):

    if specie != 'gene':
        picturepath = plotfolder + 'Trace.{0}.png'.format(specie)
        PostProcessing.plotAvgVarFromFile(filename = filename, species = [
            specie], outfile = picturepath)
```

By running this Jupyter Notebook, you will see the time traces and distribution of protein and mRNA counts plotted in Figure 2.4. Based on the results of 600 seconds simulation in 10 replicates, you can see the fluctuations of mRNA counts.

Questions:

1. Do mRNA and protein reach steady-state during the 600 seconds' simulation? How can you tell this from the plots?

2. The initial count of protein P_0001/DnaA is 148. Try to change the simulation length to 6300 seconds and increase the replicates to 100. Plot the distribution of protein counts. Does the protein count double during the entire cell cycle? And why this is important?

You can find the answer in Figure 2.5

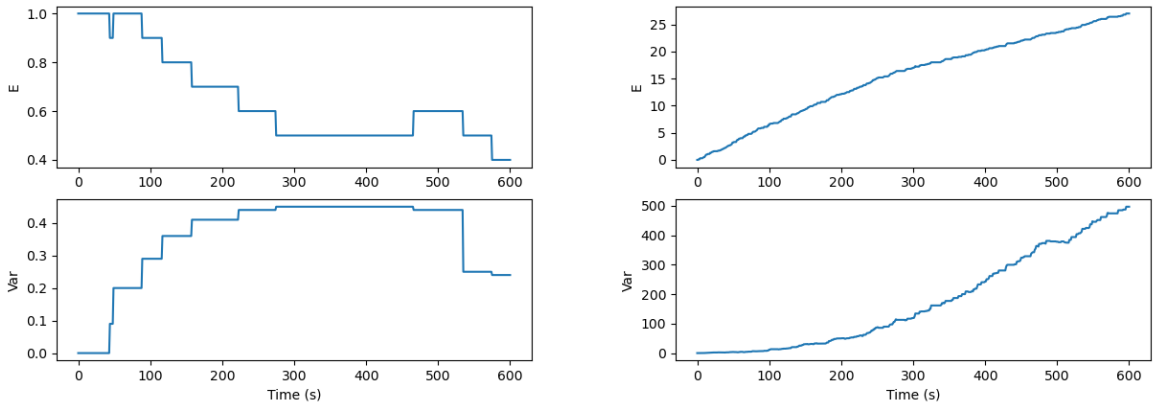


Figure 2.4: Left: Ensemble Averaged mRNA counts and its variance over 600 seconds. Right: Ensemble Averaged protein counts and its variance over 600 seconds. explanation

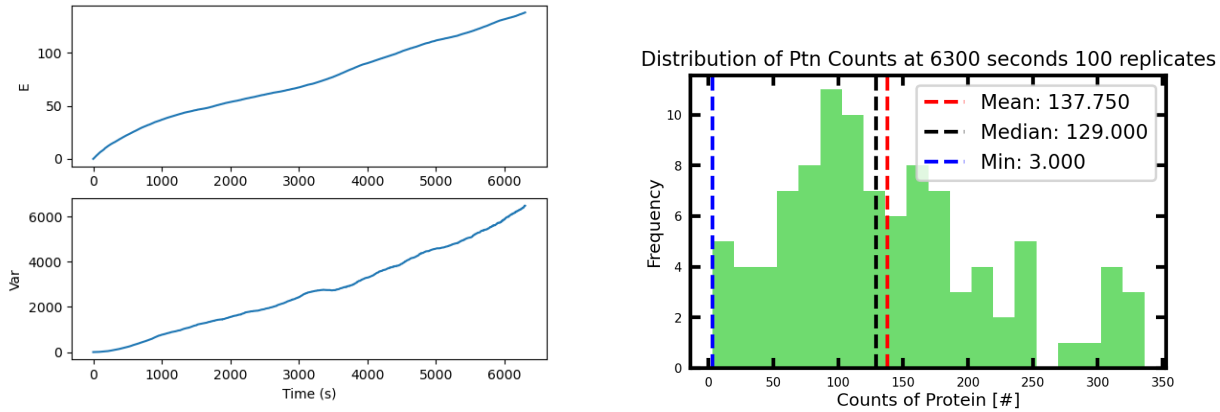


Figure 2.5: Left: Ensemble Averaged protein counts and its variance in 100 replicates over 6300 seconds. Distribution of protein counts in 100 replicates at 6300 second.

3. *Challenge* : What should be the distribution of protein counts among different replicates? You can find an analytical solution under steady state in Swain's 2008 classic paper [7].

2.3 Tutorial 3: CME/ODE Whole Cell Model of Minimal Cell, JCVI-syn3A

In this section, we will show you how to use Lattice Microbe to simulate the Minimal Cell with hybrid CME/ODE algorithm.
Before we moving on, please launch bash file **mpirun.sh** to simulate 2 minutes' cell cycle of 4 replicates in parallel.

2.3.1 Minimal Cell, JCVI-syn3A and its Minimal Genome

Minimal cell, JCVI-syn3A is a synthetic bacterium based on mother organism Mycoplasma mycoides capri published by J. Craig Venter Institute in 2016 [8]. JCVI-syn3A has a doubling time of about 2 hours and consistently forms spherical cells of approximately 200 nm in radius.
JCVI-syn3A's genome is minimal in all living organism, 543 kbp long and containing 452 genes code for proteins and 38 genes for RNAs. The ratio of unclear protein-coding genes is 90 out of 452, which is also smallest compared to other well-studied organism, including yeast and E. coli [9].

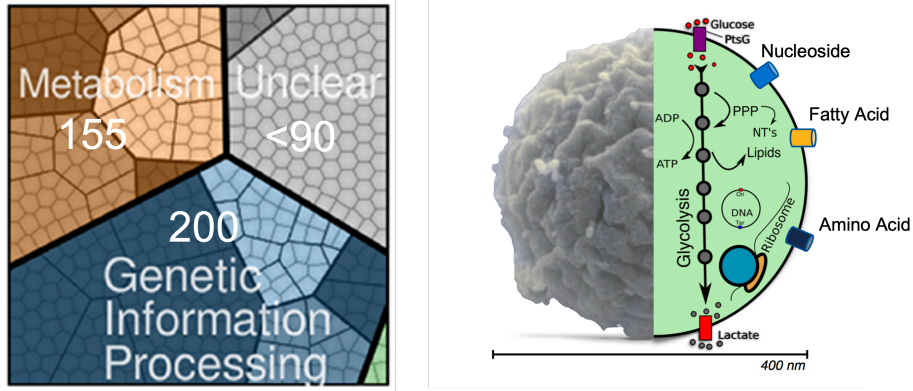


Figure 2.6: Left: Protein Coding Genes of JCVI-syn3A. Only less than 90 genes are unclear. Right: JCVI-syn3A with genetic information processes and metabolism shown

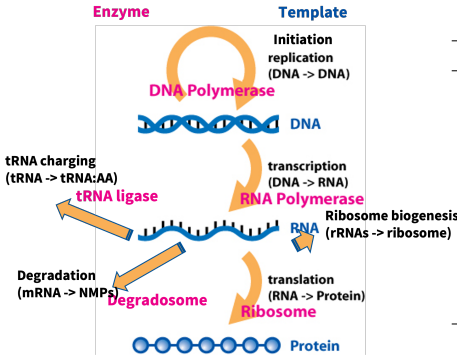
The genome information of JCVI-syn3A is stored as standard Genbank file on NCBI with ACCESSION Number CP016816.2. The Genbank file holds much more information than the pure nucleotide sequence.
Go to Genbank file under input_data folder and open with text editor. Search one example JVCISYN3A_0011. JCVISYN3A_0011 is the LocusTag of this certain gene in organism JCVI-syn3A, serving as a unique idetifier. 0011 is the locusNum that will be heavily used in the modelling to distinguish genes, RNAs, and proteins from each other. The start and end index of JCVISYN3A_0011 is 15153 and 16799, respectively and the nucleotide sequence is at the end of the Genbank file. The function of gene JCVISYN3A_0011 is protein-coding gene, where the protein is "Nucleoside Transporter ABC substrate-binding protein" with amino acid sequence shown.

2.3.2 Genetic Information Processes and Metabolism of JCVI-syn3A

Genetic Information Processes in CME

Genetic information processes connect the blueprint genes to functional proteins. In our current whole cell model of minimal cell, we mainly consider seven types of processes listed in table.
Replication copies the genetic information with the regulation of replication initiation. Transcription copies sequential information from DNA to RNA. There are three types of RNA, including mRNA, rRNA and tRNA that are tightly connected by translation. Translation takes place on ribosomes, where mRNA is read and an amino acid chain is generated according to the sequence of

mRNA. rRNA, together with other ribosome proteins make up ribosomes in ribosomal biogenesis [10]. tRNA is the carriers and identifier of amino acid to the ribosome.



Name	Biological Function
Replication	Copy genetic Information
Replication Initiation	Regulate replication
Transcription	Copy DNA into RNA
Translation	Convert mRNA into functional proteins
Degradation	Regulate translation and recycle NMPs
Ribosome Biogenesis	Assembly ribosomes
tRNA Charging	Carry AAs for translation

Figure 2.7: Left: Genetic Information Flow. Right: Functions of Seven Genetic Information Processes

As the busiest species in genetic information processes, mRNA can also be degraded by binding with degradosome. This competition of mRNA to bind with ribosome or degradosome is a important pathway to regulate genetic information processes that will shown in the following result.
In this tutorial, we will show you the plots from the high degradosome binding rate and you will simulate and plot the result from low degradosome binding rate to do the comparison.
Replication initiation is modeled by the binding of multi-domain protein DnaA (encoded by JCVISYN3A_0001) and replisome with certain region called oriC on the chromosome [6].

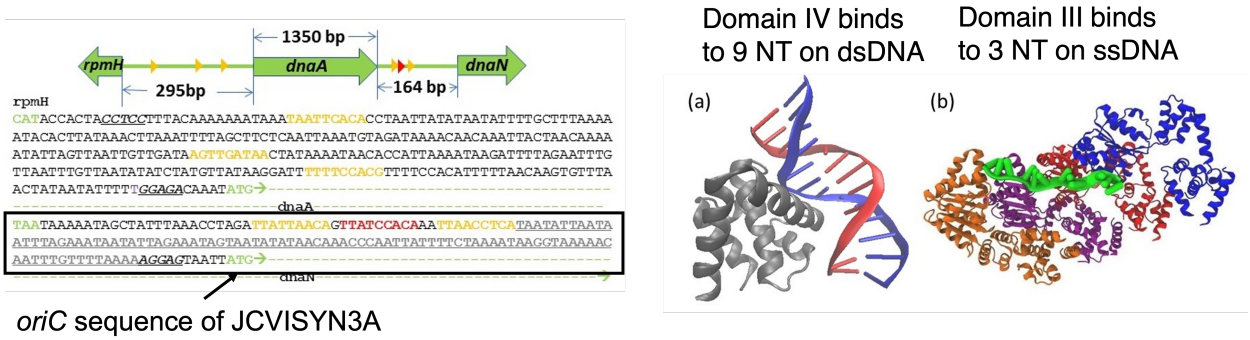


Figure 2.8: Left: oriC region. 9 nucleotide signature binding with DnaA domain IV shown in yellow and red, 3 nuclotide AT rich region binding with DnaA domain III shown in grey Right: PDB structure of DnaA domain IV and domain III binding with chromosome a): [1] b): [2]

First stage of initiation is the binding of DnaA's Domain IV with nine-nucleotide signatures on double-strand DNA(three of them, shown in red and yellow in 2.8). This binding will open a pocket for DnaA's Domain III to bind with AT rich region on single-strand DNA following the nine-nucleotide signatures to build a filament of DnaA on single-strand DNA. The final step is the binding of replisome with the chromosome after the filament grows above 15 DnaA with 30 DnaA maximum due to the length of AT rich region is 90. In current model, the assembly of replisome is not explicitly included.

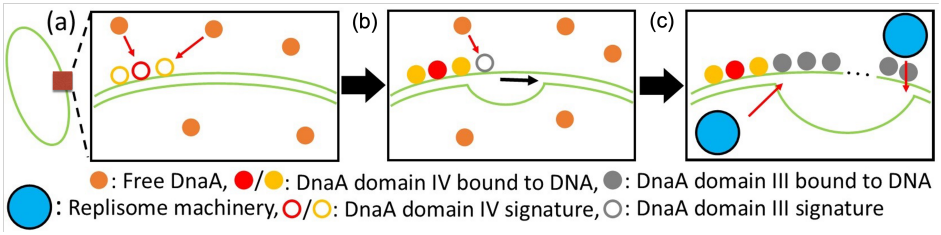


Figure 2.9: Three stage DNA replication initiation

Replication occurs gene by gene when replisomes move along the circular chromosome from oriC to ter in two directions. Since the locations of genes are fixed, the replication happens in order. So our replication model is ordered and bidirectional.

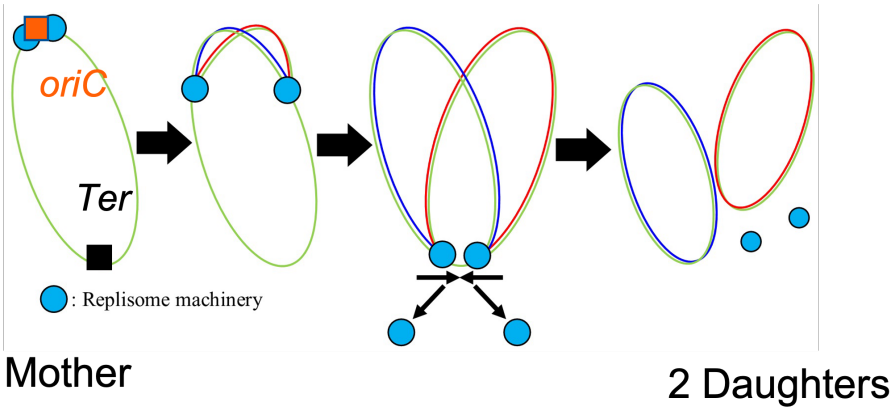


Figure 2.10: Ordered and Bidirectional Replication

The equivalent reaction to duplicate certain gene is in Table 2.3 The rate to replicate one gene in Table 2.4 is determined by a polymerization rate form proposed in [11].

Table 2.3: Reactions and Rates for Replication, Transcription, Translation and Degradation

Processes	**Reactions**	**Kinetic Constant**
Replication	$G_{\text{locusNum}} \rightarrow 2G_{\text{locusNum}}$	$k_{\text{trans}}^{locusNum} = v_{\text{replication}}^{locusNum}$
Transcription	$RNAP + G_{\text{locusNum}} \rightarrow RNAP : G_{\text{locusNum}}$	$k_{\text{trsc}}^{\text{binding}}$
	$RNAP : G_{\text{locusNum}} \rightarrow R_{\text{locusNum}} + RNAP + G_{\text{locusNum}}$	$k_{\text{trsc}}^{locusNum} = s_{\text{trsc}}^{locusNum} v_{\text{trsc}}^{locusNum}$
Translation	$Ribosome + R_{\text{locusNum}} \rightarrow Ribosome : R_{\text{locusNum}}$	$k_{\text{trans}}^{\text{binding}}$
	$Ribosome : R_{\text{locusNum}} \rightarrow P_{\text{locusNum}} + Ribosome + R_{\text{locusNum}}$	$k_{\text{trans}}^{locusNum} = v_{\text{trans}}^{locusNum}$
Degradation	$Degradosome + R_{\text{locusNum}} \rightarrow Degradosome : R_{\text{locusNum}}$	$k_{\text{degra}}^{\text{binding}}$
	$Degradosome : R_{\text{locusNum}} \rightarrow NMPs + Degradosome$	$k_{\text{degra}}^{locusNum} = v_{\text{degra}}^{locusNum}$

Transcription, translation and degradation are all depicted as a two-step binding and (de)polymerizatoin reactions where the polymerization in transcription and translation shares the same rate form as that in replication. The rate for degradation from a mRNA to its monomers is calculate by divide the monomer depletion rate over the length of mRNA.

Table 2.4: Rate Form for Replication, Transcription, Translation and Degradation

Processes	(de)polymerization Kinetic Constant	Formulation
Replication	$k^{\text{replication}}_{\text{locusNum}} = v^{\text{locusNum}}_{\text{replication}}$	$\frac{k^{\text{elongation}}_{\text{replication}}}{\frac{K_{D1}K_{D2}}{[dNTP_1][dNTP_2]}+\sum_i \frac{K_{Di}}{[dNTP_i]}+L_{DNA}-1}$
Transcription	$k^{\text{trsc}}_{\text{locusNum}} = s^{\text{promoter}}_{\text{locusNum}} v^{\text{locusNum}}_{\text{trsc}}$	$\frac{k^{\text{elongation}}_{\text{transcription}}}{\frac{K_{D1}K_{D2}}{[NTP_1][NTP_2]}+\sum_i \frac{K_{Di}}{[NTP_i]}+L_{RNA}-1}$
Translation	$k^{\text{trans}}_{\text{locusNum}} = v^{\text{locusNum}}_{\text{trans}}$	$\frac{k^{\text{elongation}}_{\text{translation}}}{\frac{K_{D1}K_{D2}}{[tRNA{:}aa_1][tRNA{:}aa_2]}+\sum_i \frac{K_{Di}}{[tRNA{:}aa_i]}+L_{protein}-1}$
Degradation	$k^{\text{degra}}_{\text{locusNum}} = v^{\text{locusNum}}_{\text{degra}}$	$\frac{k^{\text{degra}}_{\text{depolym}}}{L_{mRNA}}$

Metabolism in ODE and Rate Law

As mentioned, replication, transcription, and translation all require monomers (deoxyribonucleotide (dNTPs), ribonucleotide (NTPs), and amino acids (AAs)) for the polymerizations reactions.
All the supply of monomers comes from metabolism of JCVI-syn3A. The (d)NTPs are generated in nucleotide metabolism that transport extracellular nucleoside and convert nucleoside to (d)NTPs used in the DNA and RNA synthesis. The other building blocks, such as phosphoribosyl pyrophosphate (prpp), phosphoenolpyruvate (pep), and 1,3-Bisphosphoglyceric acid (1,3dpg) along these pathways are the products in glycolysis in central metabolism 2.11.

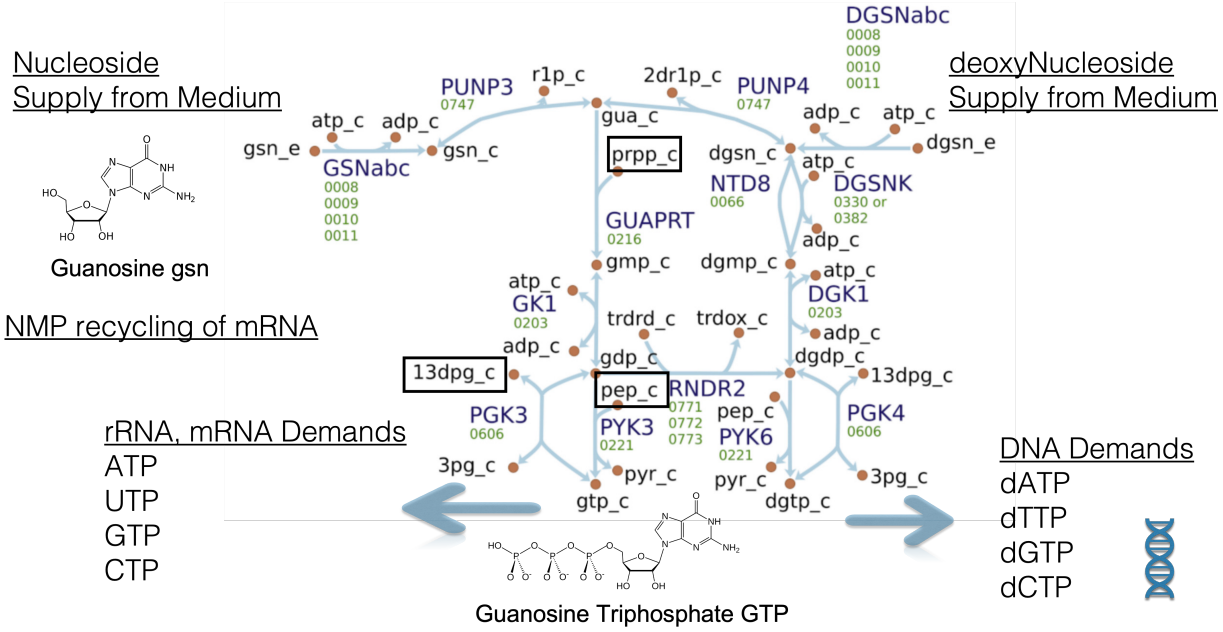


Figure 2.11: Supply of GTP and dGTP for DNA and RNA synthesis in nucleotide metabolism. The metabolites in the bracket come from central metabolism

The whole metabolism network can be separated to five connected sub-networks, including central, nucleotide, lipid, cofactor and amino acid. Glucose 6-phosphate (g6p) the immediate downstream of glucose in glycolysis connects central and lipid metabolism, while Adenosine triphosphate (ATP) that generated in glycolysis provides energy for reactions in all five sub-networks.

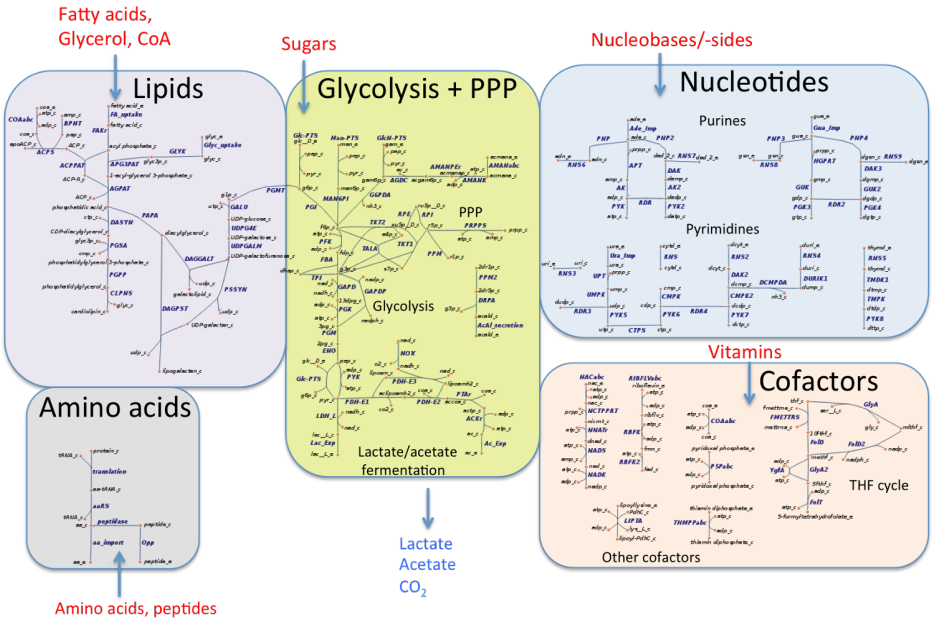


Figure 2.12: Whole Metabolism of JCVI-syn3A with five subsystems: central, nucleotide, lipid, cofactor and amino acid

In our current whole cell modeling, 197 metabolites are included, 148 cytoplasmic and 49 extracellular. 175 reactions connect the metabolites. Shown in the network of central metabolism, the orange nodes are the metabolite IDs. We use suffix '_c' to denote cytoplasmic and '_e' for extracellular. The blue arrows are reactions connecting different metabolites, with names in dark blue and related proteins' locusNums under the names.

ATP as the major energetic molecules energize cellular processes, without which the cell will die. In JCVI-syn3A, glycolysis is the only way to generate ATP. Phosphoenolpyruvate (pep) is an intermediate in both the upstream and downstream in glycolysis.

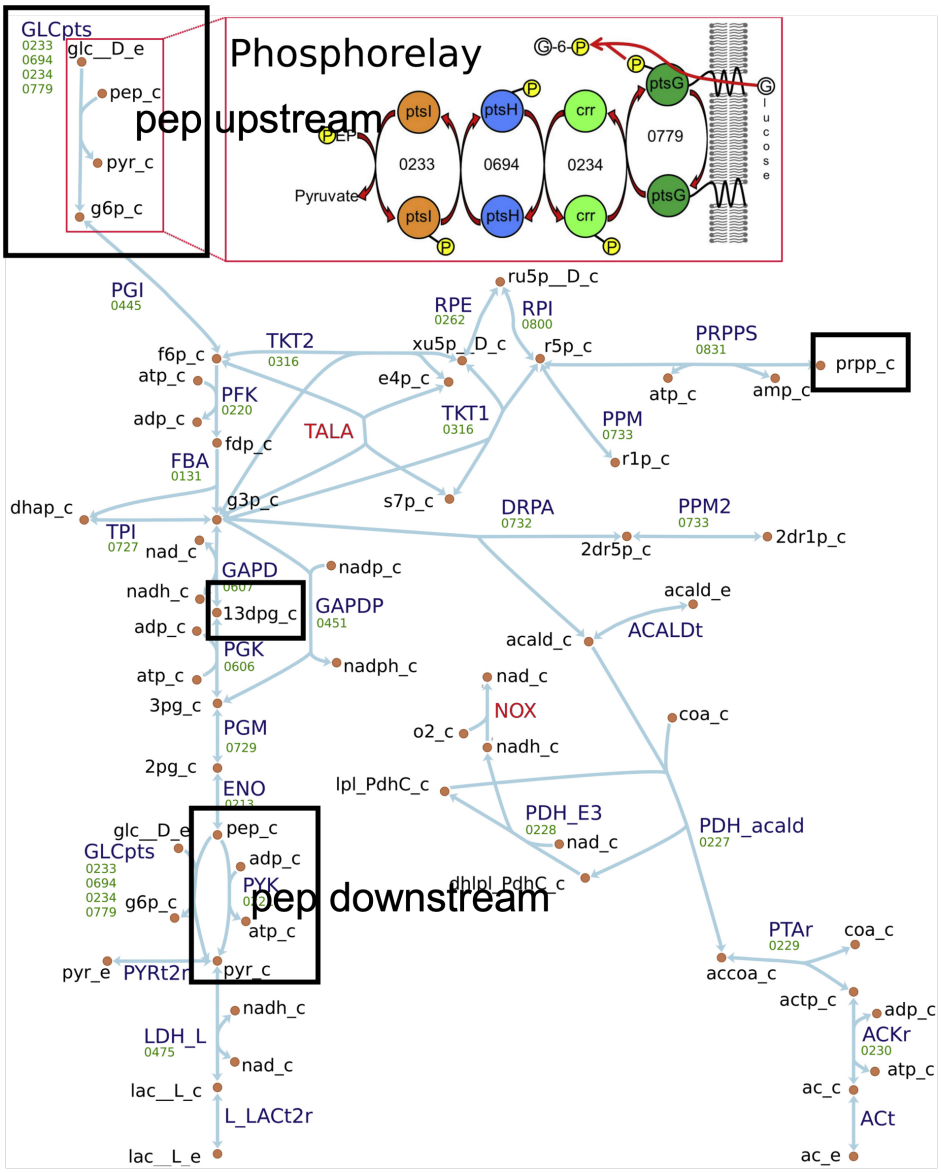


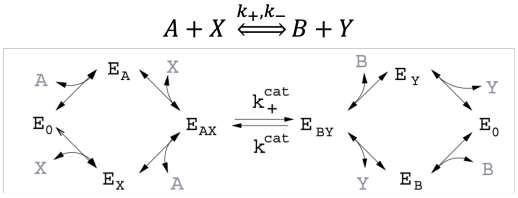
Figure 2.13: Central Metabolism in JCVI-syn3A. Phosphoenolpyruvate (pep) is both the upstream and downstream of glycolysis. 1,3-Bisphosphoglyceric acid (1,3dpg), Phosphoribosyl pyrophosphate (prpp) and pep are in nucleotide metabolism.

Most reactions in the metabolism requires certain proteins produced in genetic information processes as enzymes or transporters. Convenience rate law and random binding model are used to depict the kinetics of these reactions [12].

The random binding model assumes the substrates bind to the enzyme/transporter in arbitrary order and are converted into the products, which then dissociate from the enzyme in arbitrary order. Convenience rate law further assumes the conversion step is the rate-determining step and the binding reactions are in quasi-equilibrium. For reaction

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

, the rate law is



$$flux = \frac{k_+[E]\frac{[A]}{K_{M,A}}\frac{[X]}{K_{M,X}} - k_-[E]\frac{[B]}{K_{M,B}}\frac{[Y]}{K_{M,Y}}}{\left(1+\frac{[A]}{K_{M,A}}\right)\left(1+\frac{[X]}{K_{M,X}}\right)+\left(1+\frac{[B]}{K_{M,B}}\right)\left(1+\frac{[Y]}{K_{M,Y}}\right)-1}$$

K_A dissociation constant, k_+ and k_-forward and backward rate

Figure 2.14: Mechanism of reaction A+X to B+Y in random binding model. [E] is the concentration of enzyme, [A] concentration of molecule A

Similar to genome information, the metabolic network is also stored in standard Systems Biology Markup Language (SBML) file. In the given SBML file of JCVI-syn3A, there are two compartments, cellular and extracellular with metabolites and reactions. However, SBML file itself does not store kinetic parameters and we choose to store them in a Excel file with multiple sheets for the simulation.
We can look at one reaction entry in the SBML file for more insights. Go to the SBML file and search DGSNabc. DGSNabc, the irreversible transport of deoxyguanosine in nuclotide metabolism for example, has three reactants and four products with four proteins involved. The geneProductAssociation is AND, meaning four proteins are all needed to perform this reactions. Biologically, four proteins are four sub-units of nucleoside ABC transporter.

2.3.3 Hybrid CME/ODE Algorithm

As mentioned in the Chemical Master Equation, the discreteness and stochasticity of chemical kinetics play a role when numbers of reactants are now significantly high. For the species in genetic information processes, gene's number approximately is 1 or 2 and for most proteins, their counts are less than 1000. Based on the initial volume of JCVI-syn3A (200 nm radius, 0.035 fL, 1 particle count equals 50 nM), the concentrations of genes are 50 or 100 nM, of proteins less than 50 μM. All of these are dramatically lower than the concentrations of metabolites.

Table 2.5: Concentrations of Species in Genetic Information Processes and Metabolism

	Numbers (Discrete)	**Concentrations (Continuous)**
G_0001	1 or 2	50 or 100 nM
R_0001	< 10	< 500 nM
P_0001	$100 \sim 10^3$	5 μM $\sim$ 50 μM
Ribosome, degradosome	$100 \sim 10^3$	5 μM $\sim$ 50 μM
ATP	$\sim 10^6$	$\sim$ 5 mM

Therefore, it is necessary to simulate the kinetics in genetic information processes with stochastic chemical master equation (CME) and metabolism with deterministic ordinary differential equation (ODE).

To simulate the co-evolution of genetic information processes and metabolism, the communication needs to be performed to describe the interactions between these two subsystems.
Our current implementation of communication was first carried out in Yeast's galactose switch modeling published by our group in 2018 [3]. We first discretize the entire simulation length into piecewise communication time steps (hook intervals, t_H). During each communication time steps, t_H length CME simulation is performed to describe the kinetics in genetic information processes, followed by the communication from CME to ODE by passing the counts of proteins, consumption of monomers (dNTPs, NTPs, and AAs) and recycling of nucleoside monophosphates (NMPs) in mRNA degradation. Then t_H length ODE simulation is performed and the updated cell volume and concentrations of monomers passed to the CME.

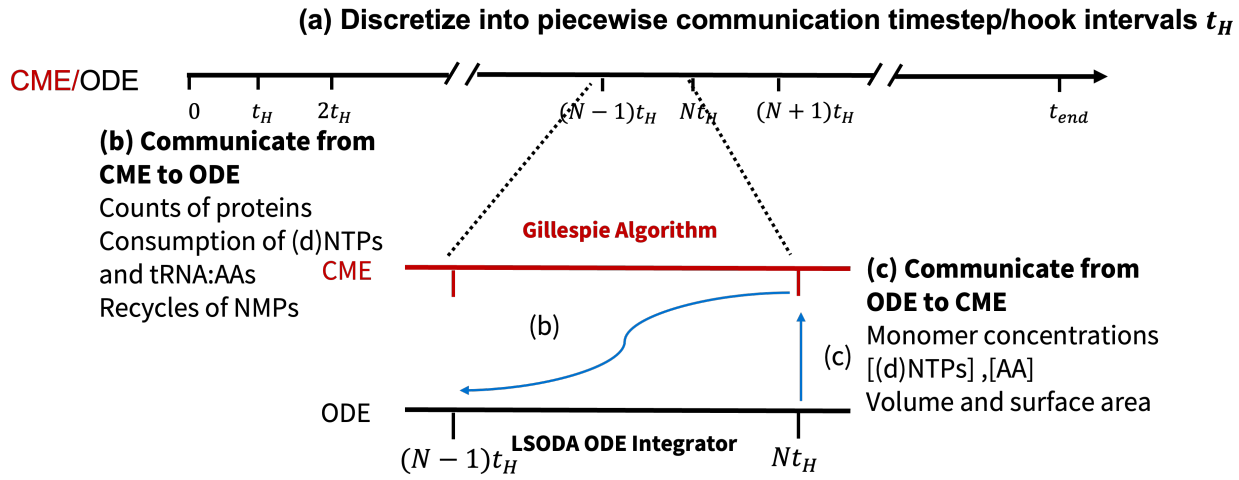


Figure 2.15: Communication Between CME and ODE to simulate the co-evolution of Genetic Information Processes and Metabolism.

In our current model, 6300 seconds' entire cell cycle is discretized with communication step 1 second.

2.3.4 Implementation in Coding

Total four input files are used in whole cell simulation, two Excel files containing initial counts/-concentrations and reactions with kinetic parameters, one Genbank file and one SBML file. The Genbank file contains the sequences and functions of genes, RNAs, and proteins. More details about how these files function in the model are available on github.

Table 2.6: Input Files and Information Read In

File	File Format	Information Read IN
syn3A.gb	Genbank	Sequences and functions of genes, RNA and protein
Syn3A_updated.xml	SBML	Stoichiometric Coefficients
initial_concentrations.xlsx	Excel	Initial counts/concs of proteins, medium and metabolites
kinetic_params.xlsx	Excel	Reactions from SBML and kinetic parameters

We launch independent replicates to obtain the statistically significant cellular dynamics. The main script is **WCM_CMEODE_Hook.py** calling multiple other python scripts to construct and simulate the genetic information processes, metabolism and their interactions over the whole cell cycle.
Essentially, we use pyLM and build-in Gillepsie algorithm to construct and simulate the genetic information processes. The ODEs are written/constructed by odecell, a package developed by former group member and simualted by well-known LSODA algorithm in scipy.
The CME/ODE simulation is solely performed on CPUs, however NVIDIA GPUs and CUDA are needed to run Lattice Microbe. Using Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz, each simulation will finish between 5 to 12 hours with 2 GB RAM usage. The variable time cost mainly comes from the adaptive time step LSODA ODE integrator.
The output files of current model are one log file and three CSV files respectively, recording counts of species in genetic information processes and metabolism, fluxes of metabolic reactions, and cell surface area and volume. General size for counts CSV file are 100 MB with approximate 5000 species and 6301 time points.

2.3.5 Results Analysis

Now run **pkl.ipynb** and **plots.ipynb** to do the analysis.

Emergent Cell Death

As mentioned, pep is an immediate in both upstream and downstream in glycolysis 2.13. If the concentration of pep goes to zero, the phosphorely (transport of glucose inside the cell) will halt, leading to the ATP concentration decrease that will finally affect all cellular processes. Since pep is both the upstream and downstream in glycolysis, the death is irreversible.
What is interesting is that the death is also stochastic, meaning the pep concentration goes to zero only happens in partial replicates. This strongly indicate that the death is related to the stochastic genetic information processes. And we are collaborating to using machine learning to disclose the highly non-linear dynamics of cell death.

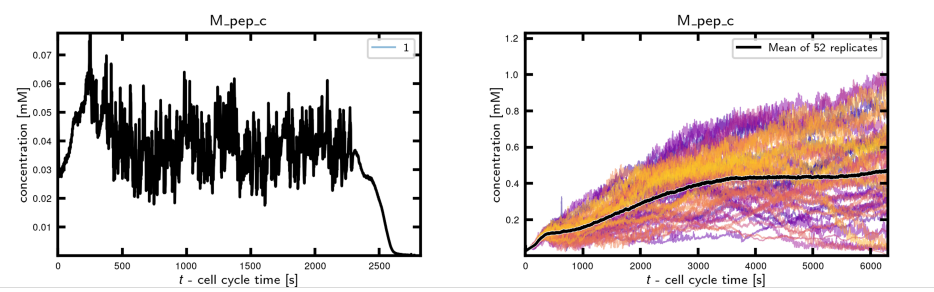


Figure 2.16: Left: Trace of pep of one unhealthy cell. Right: Traces of pep of 52 healthy cells

In the high degradation binding rate, the death happens in 48 replicates out of 100. While for low degradation binding rate, the death is 80 out of 100, much higher.

Cell Growth

Lipid molecules in metabolism and membrane proteins contributes to the membrane surface area. For the volume increase, we assume the cell remains spherical until the volume doubles. Based on the surface area, the radius then the volume is calculated. Most of the cells double the volume between 70 to 80 minutes.

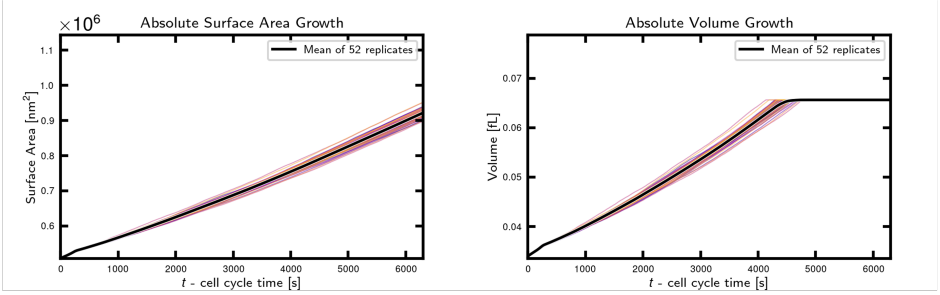


Figure 2.17: Left: Surface Area increase. Right: Cell volume Double

Replication

For the replication, initiation can happen maximum 5 times during the whole cell cycle and the mean is 2.4. In this current model, we only allow replication initiation occurs after the replication finishes.

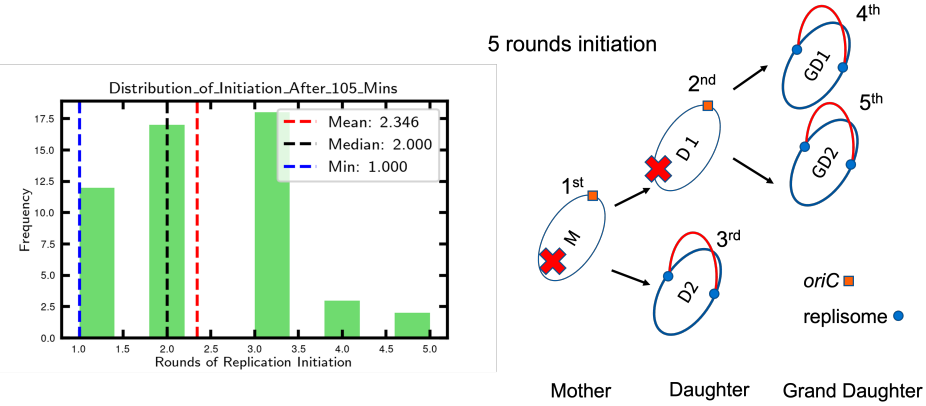


Figure 2.18: Left: Distribution of Rounds of Initiation After the Whole Cell Cycle Right: Illustration of Five Rounds of Initiation

For 5 rounds of replication initiation, the data structure of chromosomes are shown in the right side of Figure 2.18. When the mother chromosome replication initiated and replicated, two daughter chromosomes was generated with no mother chromosome exits. Both daughter chromosomes initiates replication and one of two finishes the replication to generate two grand daughter chromosomes. And both two grand daughters initiates replications. So in this case, we will have three initialized but not finished replication in the cell, one daughter and two grand daughters chromosome.

The first round initiation finishes with 16 minutes with mean 4.4 minutes. And the filament length of DnaA on ssDNA can grow above 20 in Figure 2.19.

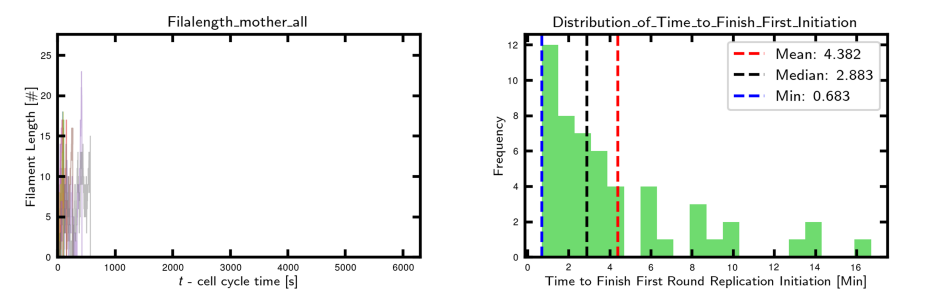


Figure 2.19: Left: Filament Length of DnaA on Mother Chromosome. Right: Distribution of Time to Finish the first Round of Initiation

After the whole cell cycle, average gene copy number is 2.8 with minimal 2.3, meaning all genes are duplicated. It takes 40 to 60 minutes for the whole genome to be replicated.
As mentioned, the supply of nucleotide from metabolism is vital for the function of genetic information processes. In the DNA replication, the deoxythymidine triphosphate (dTTP) may encounter shortage, causing the pause of replication.

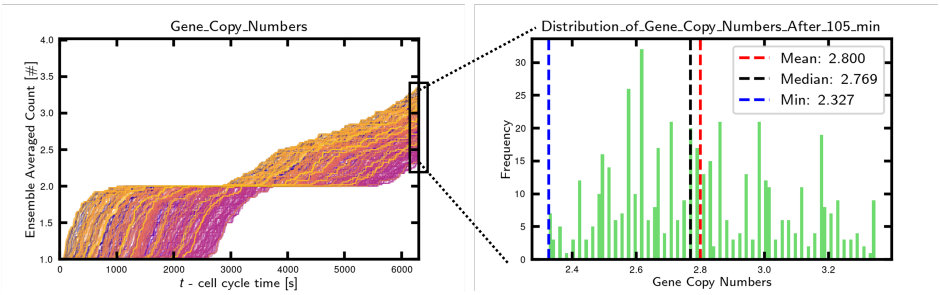


Figure 2.20: Left: Ensemble Average Traces of All genes Over the Entire Cell Cycle. Right: Distribution of Gene Copy Number After the Entire Cell Cycle

Transcriptomics and Proteomics

In the transcriptomincs, we will show the impact of degradation to the counts of mRNA and proteins. mRNA be degraded by first binding with degradosome and then degradation or be translated into proteins by bindign with ribosomes. When we lower the binding rate between mRNA with degradosome, we will see the counts of all mRNAs increases from 200-350 to 350-550 and the active ribosome will increase from 200-250 to 300-400. The increase of mRNA and active ribosome will lead to more proteins generated.

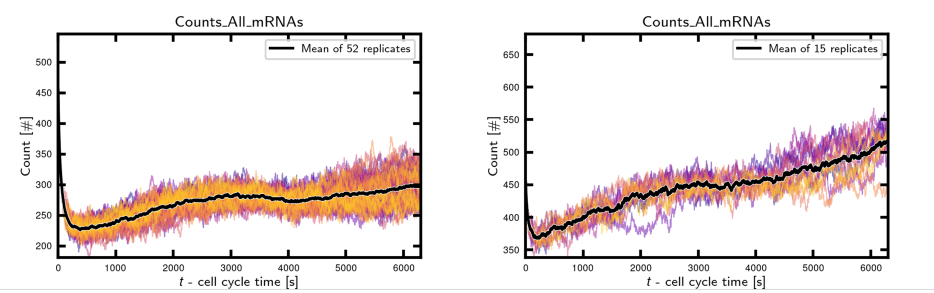


Figure 2.21: Left: Number of total mRNAs inside the Cell with high degradosome binding rate. Right: Number of total mRNAs inside the Cell with low degradosome binding rate

For the low degradosome binding rate, the mean scaled protein abundance is 2.06 while the high degradosome binding rate is 1.66.

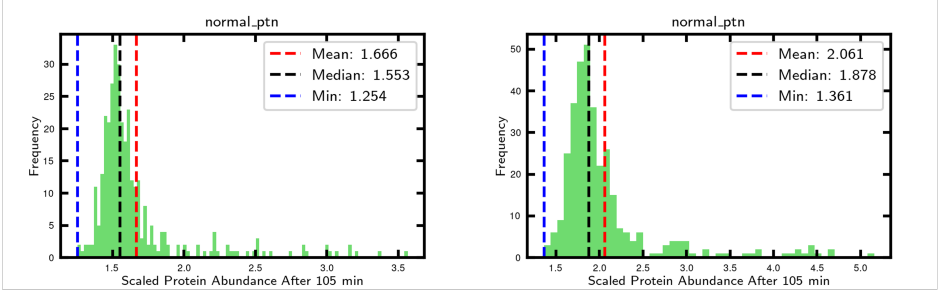


Figure 2.22: Left: Distribution of Scaled Protein Abundance After the Entire Cell Cycle with High Binding Rate with Degradosome. Right: Distribution of Scaled Protein Abundance After the Entire Cell Cycle with Low Binding Rate with Degradosome

Traces of ALL Metabolites

From our current simulation, we can also easily plot all traces of metabolites over the whole cell cycle.

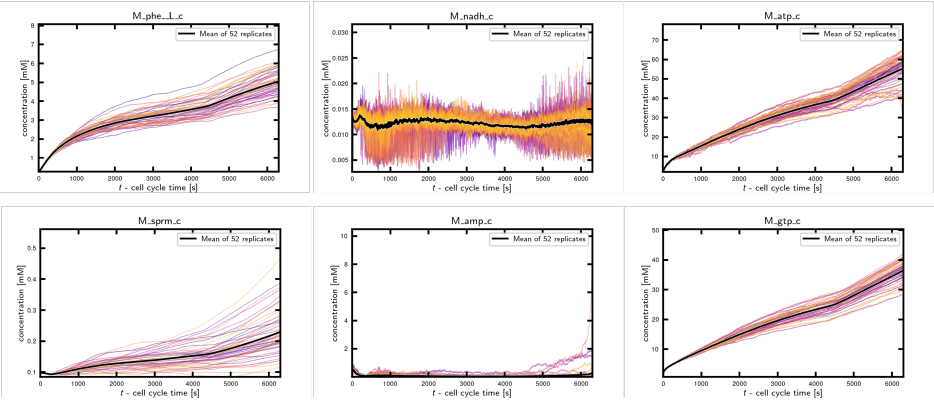


Figure 2.23: Traces of Six Metabolties Over the Entire Cell Cycle

Chapter 3

Tutorials for RDME in jLM

Here, we are going to use lm.jLM for RDME related simulation. All reactions supported and the units are listed in the table 2.1.
Here is spatial geometry display: For the details of the package jLM, we recommend the user read

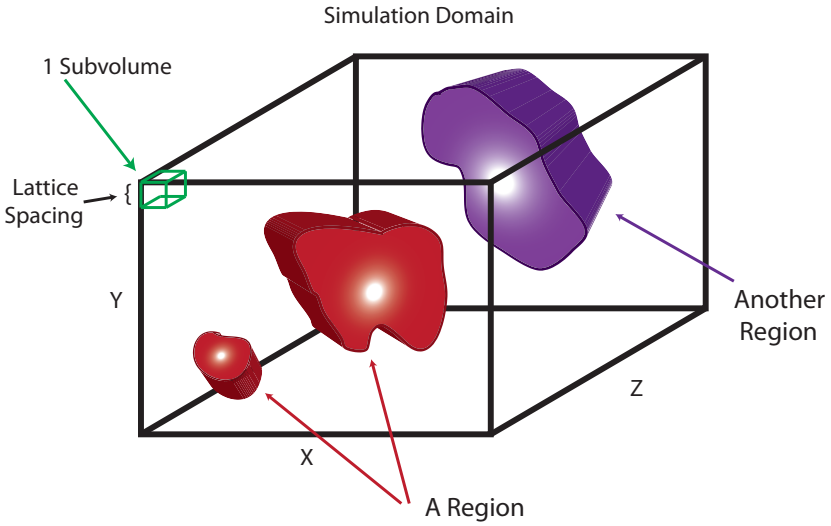


Figure 3.1: RDME schematic

the jLM documentation (`https://luthey-schulten.chemistry.illinois.edu/software/LM2.4/API.html#jlm`).

We also assume 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/3/tutorial/`), both of which are available free of charge.

3.1 Tutorial RDME 1: 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:

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

Since jLM can not do reversible reaction, we will decompose the reaction into two irreversible reactions.

$$\mathrm{A} + \mathrm{B} \xrightarrow{\mathrm{k_f}} \mathrm{C}$$
$$\mathrm{C} \xrightarrow{\mathrm{k_r}} \mathrm{A} + \mathrm{B}$$

We take the same number of each particles as CME: 1000 of A and B and 0 of C all in cytoplasm. We only allow the reactions to take place in the cytoplasm region, and only simulate for 60 seconds.
Please check the first tutorial **TutR.1.1.Bimolecular_unidistribute.ipynb** in the RDME named for source code.

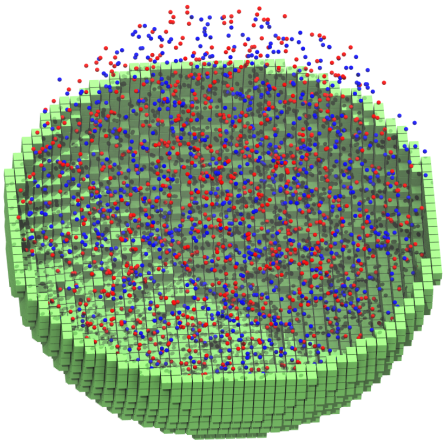
Here is the reactions we use in the simulation:

Reactions Types	Reactions	Reaction Region	Rate Constant
forward	$A + B \xrightarrow{k_f} C$	cytoplasm	$1.07 \times 10^5 \mathrm{M}^{-1}\ \mathrm{S}^{-1}$
reverse	$C \xrightarrow{k_r} A + B$	cytoplasm	$0.351\ \mathrm{s}^{-1}$

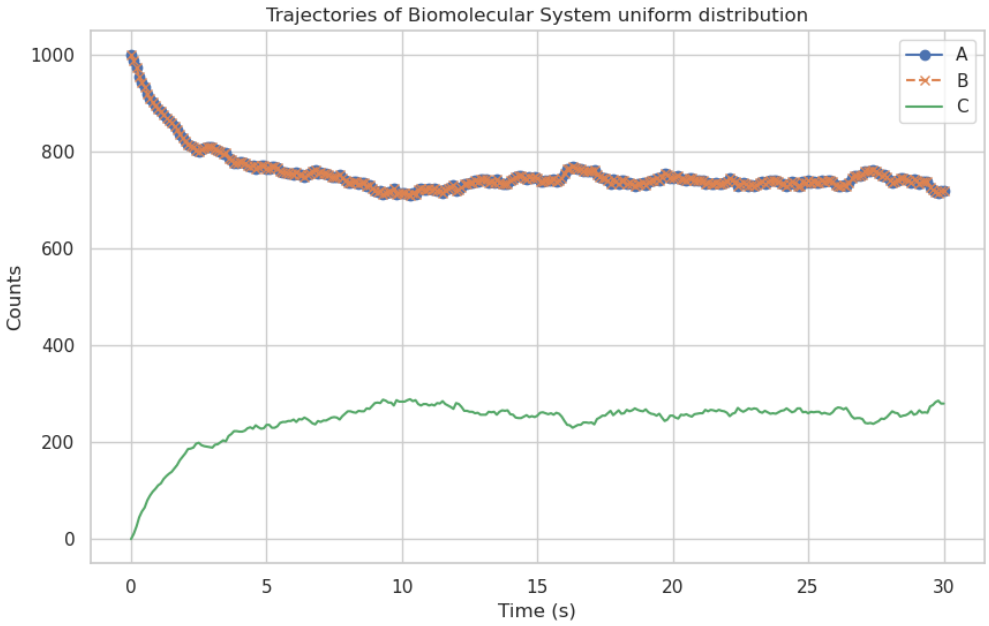
Here is the species and diffusion coefficients we use in the simulation:

Species	Initial Counts	Diffusion Co. $(\boldsymbol{m}^2 \cdot \mathrm{s}^{-1})$	Region
A	1000	1.0×10^{-14}	cytoplasm
B	1000	1.0×10^{-14}	cytoplasm
C	0	1.0×10^{-14}	cytoplasm

Here is the visualization of the initial conditions with A and B uniformly distributed:



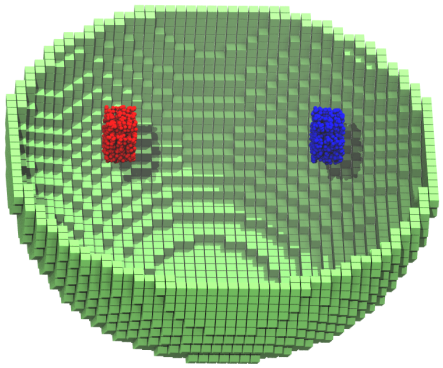
Running this code will show the total counts of species in 30 seconds simulation:



Keep in mind that the results may be different due to the stochastic nature of the simulation.

Next, let us try doing the same simulation, but we initialize the 1000 counts of A and B in two separate clusters as in Figure . The Python script in RDME folder **TutR.1.2.Bimolecular_polardistribute.ipynb** will accomplish this.

To achieve this, I define a custom methods instead of using the built in function jLM.sim.distributenumbers(), since it will only distribute certain number of species in the region uniformly. The polar distribution will be something like this (red is A and blue is B):



For details, please check the source code below:

TutR1.1_bimolecular_uni

May 5, 2024

1 Tutorial 1: RDME- Bimolecular Reaction_Uniform Distribution

1.1 1-1 Uniformly Distribution

The purpose of this tutorial is to get users familiar with the `lm.jLM` functions and methods to build up RDME simulations.

For all the functions used here, you can check the jLM api docs for details.

1.2 0.Environment Check

if you didn't install the environments in docker or any container, we suggest you do it through anaconda/miniconda.

basically, you need to install the dependencies

```
conda env create -f rdme_env.yml
```

Then import jLM package.

[ ]:
```
import jLM                                    # Set up the Jupyter environment
from jLM.RDME import Sim as RDMESim           # Main simulation class
from jLM.RegionBuilder import RegionBuilder # Deal with the spatial geometry
from lm import IntMpdRdmeSolver               # lm::rdme::IntMpdRdmeSolver

import numpy as np
```

2 1. Initialization

2.1 1.1 RDME simulation object creation

First step is to create a object that contains all the essential information to start a simulation.

The class we use here is:

```
sim = jLM.RDME.Sim(simulation_name,
                   filename,
                   dimensions,
                   latticeSpacing,
                   regionName,
                   dt=None)
```

- simulation_name is what we call our system;
- filename: the name of the final output trajectory file.(hdf5 format)
- dimensions: the number of lattices in x,y,z dimension. in list [dimx,dimy,dimz], must be divisible by 32.
- latticeSpacing: the actual physical representation of the length of each subvolume cube, unit: meter.
- the Name of the entire sim region, default="external"
- dt: time steps for the RDME simualtion, unit: second.

[ ]:
```
cellvol = 1.0e-15                # L=dm^3
simname = "Bimolecular_System_uni"
filename = "TutR1.1_bimolecular_uni.lm"
Ldim = [64,64,64]                 # Has to be multiples of 32
lattice_spacing = 32e-9           # in m
regionN = "extracellular"
dt = 25e-6                        # in s
```

[ ]:
```
sim = RDMESim(  simname,
                filename,
                Ldim,
                lattice_spacing,
                regionN,
                dt)
```

in case you forget, or want to redefine some key parameters in the simulation we define above, you can always call methods to adjust them.

for example, we can change the simulation time step by:

[ ]:
```
sim.timestep = 50e-6            # in s
```

next we need to define the total simulation time, the time interval to write results to our trajectory for lattice and species, normally we keep them the same.

Be careful, the unit for Interval is "time steps", and should be an integer.

The actual physical time for interval would be:

$$t = Interval * dt$$

since we define $dt = 50 \times 10^{-6} s$, the actual write interval would be:

$$20000 * 50 \times 10^{-6} = 0.1 s$$

[ ]:
```
sim.simulationTime=30.0     # unit is seconds

sim.latticeWriteInterval= int(2000)    # unit is timesteps, dt
sim.speciesWriteInterval= int(2000)    # unit is timesteps, dt
```

2.2 1.2 Spatial Geometry Initialization

now, we need to add more spatial compartments to our system, for simplicity, our first toy system would only have three spatial regions: + extracellular + cytoplasm + plasm membrane

we want each of them mutually excluded, and if we combine them all together, it should be the entire lattice sites we create:

$$\texttt{lattice} = \texttt{extracellular} \cup \texttt{membrane} \cup \texttt{cytoplasm}$$

and naturally

$$\emptyset = \texttt{membrane} \cap \texttt{cytoplasm}$$

$$\emptyset = \texttt{membrane} \cap \texttt{extracellular}$$

$$\emptyset = \texttt{cytoplasm} \cap \texttt{extracellular}$$

In order to design the site lattice geometry, we create a `RegionBuilder` object, which reads the lattice dimensions from our simulation object `sim`.

[ ]:
```
build = RegionBuilder(sim)
```

we create a sphere to represent the cytoplasm with radius=25, center at [32,32,32]

[ ]:
```
radius = int(np.ceil((cellvol*3/4/np.pi)**(1/3)*0.1/lattice_spacing))
print(radius)
cytoplasm = build.ellipsoid(radius = radius, center = [32,32,32])
```

Then we dilate the cytoplasm, and create a surface with the dilation method, and exclude the cytoplasm to represent our membrane

[ ]:
```
cytoplasm_dilation = build.dilate(cytoplasm, se = build.se26)
membrane = cytoplasm_dilation & ~cytoplasm
```

Finally, make sure our extracellular region excludes the cell.

[ ]:
```
# effectively, cytoplasm_dilation = cytoplasm and  membrane
extracellular = ~cytoplasm_dilation
```

we need our simulation system to compose all the regions together.

[ ]:
```
cyt = sim.region('cytoplasm')
mem = sim.region('membrane')
ext = sim.region('extracellular')
```

[ ]:
```
build.compose(
    (sim.region('extracellular'), extracellular),
    (sim.region('cytoplasm'), cytoplasm),
    (sim.region('membrane'), membrane))
```

wonderful, now we have all the geometric infomation, we can visualize it in two different ways: stacks and 3D

[ ]:
```
sim.showRegionStack()
```

[ ]:
```
sim.displayGeometry()
```

2.3 1.2 Species and diffusion Co. Initialization

For simplicity, we only want our bimolecular reaction take place in cytoplasm and only allow each species to diffuse in cytoplasm.

Our Template will be a reversible reaction as:

$$A + B \rightleftharpoons C$$

However, our software only accept one-way reaction, we need to separate it into two irreversible reactions:

$$A + B \xrightarrow{k_f} C$$
$$C \xrightarrow{k_r} A + B$$

initial condition:

#A = 1000

#B = 1000

#C = 0

[ ]:
```
spA = sim.species('A')
spB = sim.species('B')
spC = sim.species('C')
```

- function references:

`sim.species((str)name, [(str,latex)textPepr, (str)annotation])`

Then you can call the following function to check the relevant info about your species:

[ ]:
```
sim.showAllSpecies()
```

[ ]:
```
sim.showSpecies('A')
```

Now, you may notice all Diffusion rates are undefined, so we are going to **define all the diffusion coefficients** for each species.

[ ]:
```
sim.transitionRate(None, None, None, sim.diffusionZero) # initialization of the␣
↪diffusion rates to 0
```

define a custom made diffusion coefficient.

[ ]:
```
sim.diffusionConst('diff1', 1e-14)
```

set diffusion rate:

[ ]:
```
sim.transitionRate(sim.species('A'), sim.region('cytoplasm'), sim.
  ↪region('cytoplasm'), sim.dc.diff1)
# sim.transitionRate(spA, mem, mem, sim.dc.diff1)

sim.transitionRate(spB, cyt, cyt, sim.dc.diff1)    # use defined diffusion rate
sim.transitionRate(spC, cyt, cyt, sim.dc.diff1)
```

check the result:

[ ]:
```
sim.showAllSpecies()
```

2.4 1.3 Reactions Initialization

[ ]:
```
NA = 6.022e23      # molecules/mole
# kf = sim.rateConst('kf', 1.07e5/(NA*cellvol), 2)
kf = sim.rateConst('kf', 1.07e5, 2)
# rate constant for the reaction A + B -> C, second order, /M/s
kr = sim.rateConst('kr', 0.351 , 1)
# rate constant for the reaction C -> A + B, first order, /s
```

Reactions can be added by the function:

`jLM.RDME.Sim.region('region_name').addReaction([reactants],[products], rate_const)`

but remember, **all reactants, products and rate constants are corresponding jLM objects, not a number or string.**

Since we already define the name of the region we want to add reactions, there will be two ways you can add reactions:

[ ]:
```
sim.region('cytoplasm').addReaction([spA,spB], [spC], sim.rc.kf)
cyt.addReaction([spC], [spA,spB], kr)
```

Distribute a concentration of particles uniformly through cytoplasm:

[ ]:
```
sim.distributeNumber(spA, cyt, 1000)
sim.distributeNumber(spB, cyt, 1000)
sim.distributeNumber(spC, cyt, 0)
# sim.sp.A.placeNumberInto(cyt, 1000)
```

[ ]:
```
sim.showAllSpecies()
```

3 2 Check the system state

[ ]:
```
sim
```

now we confirmed it is what we want, we can now proceed to finalize the simulation we create.

[ ]:
```
sim.finalize()
```

4 3. Run the Simulation

[ ]:
```
sim.run(solver=IntMpdRdmeSolver(), cudaDevices=[0])
```

5 4. Analysis

The trajectory will be saved as h5 format, we need the python package `h5py` to access it, or you can download the HDFView to see it in a GUI.

[ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns
from jLM.RDME import File as RDMEFile
```

[ ]:
```
# traj_file = h5py.File('T1_bimolecular.lm ', 'r')
traj = RDMEFile(filename)
ts, As = traj.getNumberTrajectory(species="A")
ts, Bs = traj.getNumberTrajectory(species="B")
ts, Cs = traj.getNumberTrajectory(species="C")
```

[ ]:
```
# now visuzlize the trajectories with seaborn and plt
sns.set(style="whitegrid")
# Create a color palette
palette = sns.color_palette()
plt.figure(figsize=(10,6))
plt.plot(ts, As, label='A',marker='o', linestyle='-',color=palette[0])
plt.plot(ts, Bs, label='B',marker='x', linestyle='--',color=palette[1])
plt.plot(ts, Cs, label='C',color=palette[2])

plt.title('Trajectories of Biomolecular System uniform distribution')
plt.xlabel('Time (s)')
plt.ylabel('Counts')
plt.legend()
plt.savefig('./images/TutR1.1_bimolecular_uni.png')
plt.show()
```

TutR1.2_bimolecular_polardistribute

May 5, 2024

1 Tutorial: RDME- Bimolecular Reaction _ Polar Distribution

The purpose of this tutorial is to get users familiar with the `lm.jLM` functions and show the diffusion phenomenon by comparing the results with the first tutorial TutR1.1_bimolecular_uni.

For comparison, all the parameter are the same, except how we place the particle A and B.

ps: For all the functions used here, you can check the jLM api docs for details.

1.1 0.Environment Check

if you didn't install the environments in docker or any container, we suggest you do it through anaconda/miniconda.

basically, you need to install the dependencies

```
conda env create -f rdme_env.yml
```

Then import jLM package.

[ ]:
```
import jLM                                    # Set up the Jupyter environment
from jLM.RDME import Sim as RDMESim           # Main simulation class
from jLM.RegionBuilder import RegionBuilder # Deal with the spatial geometry
from lm import IntMpdRdmeSolver               # lm::rdme::IntMpdRdmeSolver

import numpy as np
```

2 1. Initialization

2.1 1.1 RDME simulation object creation

First step is to create a object that contains all the essential information to start a simulation.

The class we use here is:

```
sim = jLM.RDME.Sim(simulation_name,
                   filename,
                   dimensions,
                   latticeSpacing,
                   regionName,
                   dt=None)
```

- simulation_name is what we call our system;
- filename: the name of the final output trajectory file.(hdf5 format)
- dimensions: the number of lattices in x,y,z dimension. in list [dimx,dimy,dimz], must be divisible by 32.
- latticeSpacing: the actual physical representation of the length of each subvolume cube, unit: meter.
- the Name of the entire sim region, default="external"
- dt: time steps for the RDME simualtion, unit: second.

[ ]:
```
cellvol = 1.0e-15                # L=dm^3
simname = "Bimolecular_System_polar"
filename = "TutR1.2_bimolecular_polardis.lm"
Ldim = [64,64,64]                 # Has to be multiples of 32
lattice_spacing = 32e-9           # in m
regionN = "extracellular"
dt = 25e-6                        # in s
```

[ ]:
```
sim = RDMESim(  simname,
                filename,
                Ldim,
                lattice_spacing,
                regionN,
                dt)
```

in case you forget, or want to redefine some key parameters in the simulation we define above, you can always call methods to adjust them.

for example, we can change the simulation time step by:

[ ]:
```
sim.timestep = 50e-6              # in s
```

next we need to define the total simulation time, the time interval to write results to our trajectory for lattice and species, normally we keep them the same.

Be careful, the unit for Interval is "time steps", and should be an integer.

The actual physical time for interval would be:

$$t = Interval * dt$$

since we define $dt = 50 \times 10^{-6} s$, the actual write interval would be:

$$2000 * 50 \times 10^{-6} = 0.1s$$

[ ]:
```
sim.simulationTime=30.0     # unit is seconds

sim.latticeWriteInterval= int(2000)    # unit is timesteps, dt
sim.speciesWriteInterval= int(2000)    # unit is timesteps, dt
```

2.2 1.2 Spatial Geometry Initialization

now, we need to add more spatial compartments to our system, for simplicity, our first toy system would only have three spatial regions: + extracellular + cytoplasm + plasm membrane

we want each of them mutually excluded, and if we combine them all together, it should be the entire lattice sites we create:

$$\texttt{lattice} = \texttt{extracellular} \cup \texttt{membrane} \cup \texttt{cytoplasm}$$

and naturally

$$\emptyset = \texttt{membrane} \cap \texttt{cytoplasm}$$

$$\emptyset = \texttt{membrane} \cap \texttt{extracellular}$$

$$\emptyset = \texttt{cytoplasm} \cap \texttt{extracellular}$$

In order to design the site lattice geometry, we create a `RegionBuilder` object, which reads the lattice dimensions from our simulation object `sim`.

[ ]:
```
build = RegionBuilder(sim)
```

we create a sphere to represent the cytoplasm with radius=25, center at [32,32,32]

[ ]:
```
radius = int(np.ceil((cellvol*3/4/np.pi)**(1/3)*0.1/lattice_spacing))
print(radius)
cytoplasm = build.ellipsoid(radius = radius, center = [32,32,32])
```

Then we dilate the cytoplasm, and create a surface with the dilation method, and exclude the cytoplasm to represent our membrane

[ ]:
```
cytoplasm_dilation = build.dilate(cytoplasm, se = build.se26)
membrane = cytoplasm_dilation & ~cytoplasm
```

Finally, make sure our extracellular region excludes the cell.

[ ]:
```
# effectively, cytoplasm_dilation = cytoplasm and  membrane
extracellular = ~cytoplasm_dilation
```

we need our simulation system to compose all the regions together.

[ ]:
```
cyt = sim.region('cytoplasm')
mem = sim.region('membrane')
ext = sim.region('extracellular')
```

[ ]:
```
build.compose(
    (sim.region('extracellular'), extracellular),
    (sim.region('cytoplasm'), cytoplasm),
    (sim.region('membrane'), membrane))
```

wonderful, now we have all the geometric infomation, we can visualize it in two different ways: stacks and 3D

[ ]:
```
sim.showRegionStack()
```

[ ]:
```
sim.displayGeometry()
```

2.3 1.2 Species and diffusion Co. Initialization

For simplicity, we only want our bimolecular reaction take place in cytoplasm and only allow each species to diffuse in cytoplasm.

Our Template will be a reversible reaction as:

$$A + B \rightleftharpoons C$$

However, our software only accept one-way reaction, we need to separate it into two irreversible reactions:

$$A + B \xrightarrow{k_f} C$$
$$C \xrightarrow{k_r} A + B$$

initial condition:

#A = 1000

#B = 1000

#C = 0

[ ]:
```
spA = sim.species('A')
spB = sim.species('B')
spC = sim.species('C')
```

- function references:

`sim.species((str)name, [(str,latex)textPepr, (str)annotation])`

Then you can call the following function to check the relevant info about your species:

[ ]:
```
sim.showAllSpecies()
```

[ ]:
```
sim.showSpecies('A')
```

Now, you may notice all Diffusion rates are undefined, so we are going to **define all the diffusion coefficients** for each species.

[ ]:
```
sim.transitionRate(None, None, None, sim.diffusionZero) # initialization of the␣
↪diffusion rates to 0
```

define a custom made diffusion coefficient.

[ ]:
```
sim.diffusionConst('diff1', 1e-14)
```

set diffusion rate:

```
[ ]: sim.transitionRate(sim.species('A'), sim.region('cytoplasm'), sim.
      ↪region('cytoplasm'), sim.dc.diff1)
     # sim.transitionRate(spA, mem, mem, sim.dc.diff1)

     sim.transitionRate(spB, cyt, cyt, sim.dc.diff1)    # use defined diffusion rate
     sim.transitionRate(spC, cyt, cyt, sim.dc.diff1)
```

check the result:

```
[ ]: sim.showAllSpecies()
```

2.4 1.3 Reactions Initialization

```
[ ]: NA = 6.022e23      # molecules/mole
     # kf = sim.rateConst('kf', 1.07e5/(NA*cellvol), 2)
     kf = sim.rateConst('kf', 1.07e5, 2)
     # rate constant for the reaction A + B -> C, second order, /M/s
     kr = sim.rateConst('kr', 0.351 , 1)
     # rate constant for the reaction C -> A + B, first order, /s
```

Reactions can be added by the function: `jLM.RDME.Sim.region('region_name').addReaction`

Since we already define the name of the region we want to add reactions, there will be two ways you can add reactions:

```
[ ]: sim.region('cytoplasm').addReaction([spA,spB], [spC], sim.rc.kf)
     cyt.addReaction([spC], [spA,spB], kr)
```

2.5 1.4 Add particles

Here, in order to see the effects of diffusion, we put 1000 species A at on side, and 1000 spcecies at the other side. Instead of uniformly distribution.

```
[ ]: # uniformly distribute the species in the cytoplasm
     # sim.distributeNumber(spA, cyt, 1000)
     # sim.distributeNumber(spB, cyt, 1000)
     # sim.distributeNumber(spC, cyt, 0)
     # sim.sp.A.placeNumberInto(cyt, 1000)

     cyto_cent = [32,32,32]
     cyto_left_cent = [32-int(radius/2),32,32]
     cyto_right_cent = [32+int(radius/2),32,32]

     total_A = 1000
     total_B = 1000
     total_C = 0
```

```
# write a BFS function to place the species in the cytoplasm start from␣
 ↪cyto_left_cent
def placeNumbersInCyto(sp, x, y, z, n):
    '''place certain amount of species in the cytoplasm, start from x, y, z
    '''
    neigbhor_26 = [[0,0,1],[0,0,-1],[0,1,0],[0,-1,0],
                [1,0,0],[-1,0,0],[0,1,1],[0,1,-1],
                [0,-1,1],[0,-1,-1],[1,0,1],[1,0,-1],
                [-1,0,1],[-1,0,-1],[1,1,0],[1,-1,0],
                [-1,1,0],[-1,-1,0],[1,1,1],[1,1,-1],
                [1,-1,1],[1,-1,-1],[-1,1,1],[-1,1,-1],
                [-1,-1,1],[-1,-1,-1]]
    q = []
    visited = []
    q.append([x,y,z])
    remaining = n
    while len(q) > 0:
        cur = q.pop(0)

        if remaining <= 0:
            break
        visited.append(cur)
        if remaining / 16 >= 1:

            sim.placeNumber(sp=sp, x=cur[0], y=cur[1], z=cur[2], n=16)
            remaining -= 16
        else:
            sim.placeNumber(sp=sp, x=cur[0], y=cur[1], z=cur[2], n=remaining)
            remaining = 0
        for i in neigbhor_26:
            next = [cur[0]+i[0], cur[1]+i[1], cur[2]+i[2]]
            if next not in q and next not in visited and next[0] >= 0 and␣
 ↪next[0] < 64 and next[1] >= 0 and next[1] < 64 and next[2] >= 0 and next[2]␣
 ↪< 64:
                q.append(next)
    print("successfully placed " + str(n) +" species" + sp.name + " in the␣
 ↪cytoplasm.")
    return visited
```

[ ]:
```
ini_A_reg = placeNumbersInCyto(spA, cyto_left_cent[0], cyto_left_cent[1],␣
 ↪cyto_left_cent[2], total_A)
ini_B_reg = placeNumbersInCyto(spB, cyto_right_cent[0], cyto_right_cent[1],␣
 ↪cyto_right_cent[2], total_B)
```

[ ]:
```
sim.showAllSpecies()
```

3 2 Check the system state

```
[ ]: sim
```

now we confirmed it is what we want, we can now proceed to finalize the simulation we create.

```
[ ]: sim.finalize()
```

4 3. Run the Simulation

```
[ ]: sim.run(solver=IntMpdRdmeSolver(), cudaDevices=[0])
```

5 4. Analysis

The trajectory will be saved as h5 format, we need the python package `h5py` to access it, or you can download the HDFView to see it in a GUI.

```
[ ]: import matplotlib.pyplot as plt
import seaborn as sns
from jLM.RDME import File as RDMEFile
```

```
[ ]: # traj_file = h5py.File('T1_bimolecular.lm ', 'r')
traj = RDMEFile(filename)
ts, As = traj.getNumberTrajectory(species="A")
ts, Bs = traj.getNumberTrajectory(species="B")
ts, Cs = traj.getNumberTrajectory(species="C")
```

```
[ ]: # now visuzlize the trajectories with seaborn and plt
sns.set(style="whitegrid")
# Create a color palette
palette = sns.color_palette()
plt.figure(figsize=(10,6))
plt.plot(ts, As, label='A',marker='o', linestyle='-',color=palette[0])
plt.plot(ts, Bs, label='B',marker='x', linestyle='--',color=palette[1])
plt.plot(ts, Cs, label='C',color=palette[2])

plt.title('Trajectories of Biomolecular System with polar distribution')
plt.xlabel('Time (s)')
plt.ylabel('Counts')
plt.legend()
plt.savefig('./images/TutR1.2_bimolecular_polardis.png')
plt.show()
```

3.2 Tutorial RDME 2: Genetic Information Processing

Here we will simulate a simple genetic information processing(GIP) with one gene, one mRNA and one protein.This tutorial is based on the work from Vahid Shahrezaei and Peter Swain [7]. For comparison, the speciess and reactions will be the same as the one in the CME tutorial.
The schematic is shown here:

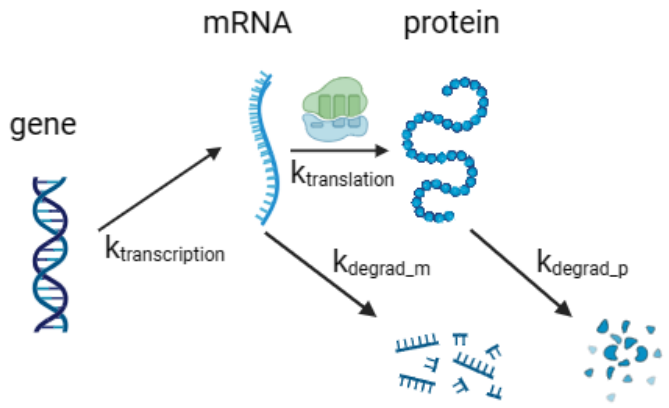


Figure 3.2: Genetic Information Processing Diagram

This section, we introduce the heterogeneity of genetic information processing. Here, we take the similar to deal with gene information processing from our Minimal cell model [5]. Basically, we place the gene as a particle in the start block/cube of the DNA, and randomly place one mRNA in the cytoplasm as our intial condition.
Then we will define the spatial heterogeneity of the reactioins, where transcription only happens in region DNA, translation only happens in region ribosomes and degradation of mRNA and DNA will only happen in shell region. The reason we only allow degradation in the shell region is that the degradosomes only exist in the shell region.

Again, we demonstrate the reactions and species included in the simulation:

Table 3.1: Table of Species with Initial Counts, Diffusion Coefficients, and Regions

Species	Initial Counts	Diffusion Co. $(m^2 \cdot s^{-1})$	Region
gene	1	NA	DNA
mRNA	1	4.3×10^{-14}	DNA, cytoplasm, shell, ribosomes
$\text{mRNA}_{\text{read}}$	0	4.3×10^{-14}	cytoplasm, shell, ribosomes
protein	0	1.0×10^{-12}	Cytoplasm, shell

Table 3.2: Overview of Reaction Types, Reactions, Reaction Regions, and Rate Constants

Reactions Types	Reactions	Reaction Region	Rate Constant (s^{-1})
Transcription	Gene $\rightarrow$ mRNA	DNA	6.41×10^{-4}
Degradation of mRNA	mRNA $\rightarrow \varnothing$	shell	2.59×10^{-2}
Translation	mRNA $\rightarrow$ $\text{mRNA}_{\text{read}}$ + Protein	ribosomes	7.20×10^{-2}
Degradation of Protein	Protein $\rightarrow \varnothing$	shell	7.70×10^{-6}
Pseudo conversion	$\text{mRNA}_{\text{read}} \rightarrow$ mRNA	Cytoplasm, shell	1.0×10^{6}

Questions:

1. Do mRNA and protein reach steady-state during the entire cell cycle? How can you tell this from the plots?

2. $Challenge$: After the workshop, try to increase the simulation time to see the steady state of protein.

For the details, please check the source code below:

TutR2_GeneticInfoProcessing

May 5, 2024

1 TutR2_Genetic Information Process Model

Here we create a simple model includes the simple genetic information processing(GIP):

[ ]:
```
import jLM
from jLM.RegionBuilder import RegionBuilder
from jLM.RDME import Sim as RDMESim
from jLM.RDME import File as RDMEFile

import lm
from lm import IntMpdRdmeSolver


import numpy as np
```

2 1. Create the RDME simulation Object

[ ]:
```
totalTime = 60            # total simulation time, in seconds
timeStep = 50e-6            # time step, in seconds
writeInterval = 20000         # the times steps write the info into␣
 ↪trajectories, units of timeStep, 20000 * 50e-6 = 1s
```

[ ]:
```
from pyLM import LMLogger
LMLogger.setLMLogConsole()
#(logging.DEBUG)
```

[ ]:
```
lattice_spacing = 8e-9 #m
N_edges = [64, 64, 64]
sim_center = [int(N_edges[0]/2),int(N_edges[1]/2),int(N_edges[2]/2)]
N_2_x=int(N_edges[0]/2)
N_2_y=int(N_edges[1]/2)
N_2_z=int(N_edges[2]/2)

filename = 'TutR2.GIP_result.lm'
```

[ ]:
```
sim = RDMESim("T2_GIP",
                  filename,
```

```
                    N_edges,
                    lattice_spacing,
                    "extracellular")

sim.timestep = timeStep
sim.simulationTime=totalTime
sim.latticeWriteInterval=writeInterval
sim.speciesWriteInterval=writeInterval

sim.transitionRate(None, None, None, sim.diffusionZero)
```

3 2. Build up the Minimal Cell Spatial Geometry

We use the minimal bacteria cell JCVI-syn3A as the spatial cell geometry for our genetic information process system.

[ ]:
```
radius_nm = 2.00e-7 #m minimal cell radius
cyto_radius = radius_nm/sim.latticeSpacing  #converted to lattice sites, # of␣
 ↪lattice sites from the center to the edge of the cell
```

[ ]:
```
build = RegionBuilder(sim)

# create a sphere region as the cytoplasm
cytoplasm = build.ellipsoid(radius = cyto_radius, center = sim_center)
# create a spherical surface  as the shell region of the cell
cyto_dilation = build.dilate(cytoplasm, se = build.se26)
shell = cyto_dilation & ~cytoplasm

# create a spherical surface as the membrane
cyto_dilation = build.dilate(cyto_dilation, se = build.se26)
membrane = cyto_dilation & ~shell & ~cytoplasm
```

randomly place 500 ribosomes in cytoplasm:

[ ]:
```
import T2_loading as loader

ribosomes = loader.getRibosomeSites(cytoplasm, N_edges)
```

Load the geometry of DNA from file

[ ]:
```
# Load the DNA file generated from b-Tree Chromo
DNAfile = './supporting_data/x_chain_syn3a_rep00001.bin'# DNA bin file

DNAsites, DNA_pos = loader.getDNAsites(DNAfile, N_edges,lattice_spacing, N_2_x,␣
 ↪N_2_y, N_2_z)

# define cyto plasm and extra cellular region
cytoplasm = cytoplasm & ~DNAsites
```

```
extracellular = ~membrane & ~cytoplasm & ~ribosomes & ~DNAsites
```

[ ]:
```
build.compose(
    (sim.region('extracellular'), extracellular),
    (sim.region('cytoplasm'), cytoplasm),
    (sim.region('DNA'),DNAsites),
    (sim.region('ribosomes'), ribosomes),
    (sim.region('shell'), shell),
    (sim.region('membrane'), membrane))
```

[ ]:
```
ext=sim.region('extracellular')
cyt=sim.region('cytoplasm')
dna=sim.region('DNA')
ribo=sim.region('ribosomes')
she=sim.region('shell')
mem=sim.region('membrane')
```

[ ]:
```
sim.displayGeometry()
```

4 3. Define all species

[ ]:
```
with sim.construct():
    sim.species("gene", texRepr="gene", annotation="gene in gene info process")
    sim.species("mRNA", textRepr="mRNA", annotation="mRNA in gene info process")
    sim.species("mRNAr", textRepr="mRNA_{read}", annotation="mRNA after the␣
 ↪translation")
    sim.species("P", texRepr="Protein", annotation="Protein in gene info␣
 ↪process")
```

[ ]:
```
sp = sim.sp   # species object access
reg = sim.reg # region object access
rc = sim.rc   # rate constant object access
dc = sim.dc   # diffusion constant object access
```

[ ]:
```
sim.transitionRate(None, None, None, sim.diffusionZero)
```

5 4. Define reactions and diffusions

After the mRNA transcribed from the gene represented as a particle in the Gene start site in the 3 dimensional lattice; we only allow mRNA to diffuse out of the `DNA` region to `cytoplasm` and we don't allow the particle to diffuse within the `DNA` region.

We also need to allow the mRNA to diffuse into the ribosome and out-of ribosome for the translation. Here, to prevent the mRNA getting trapped inside the `ribosomes`, and constantly translating, we form a new particle called $mRNA_{read}$.

$mRNA_{read}$ can be instantly convert back to m_{RNA} out side `ribosomes`.

```
[ ]: with sim.construct():
        sim.rateConst("trans", 0.019, order=1, annotation="transcription rate")
        sim.rateConst("transl", 0.0029, order=1, annotation="translation rate")
        sim.rateConst("degrad_m", 0.0023, order=1, annotation="mRNA degradation␣
     ↪rate")
        sim.rateConst("degrad_p", 7.7e-6, order=1, annotation="Protein degradation␣
     ↪rate")
        sim.rateConst("conversion", 1000000, order= 1, annotation="Conversion rate␣
     ↪from mRNA read state to ready state")
        # define all necessary reactions
        # transcription
        sim.reaction([sp.gene], [sp.gene, sp.mRNA], rc.trans, regions=[reg.DNA],␣
     ↪annotation="transcription")
        # mRNA degradation
        sim.reaction([sp.mRNA], [], rc.degrad_m, regions=[reg.shell],␣
     ↪annotation="mRNA degradation")
        sim.reaction([sp.mRNAr],[], rc.degrad_m, regions=[reg.shell],␣
     ↪annotation="mRNA read degradation")
        # translation
        sim.reaction([sp.mRNA], [sp.mRNAr, sp.P], rc.transl, regions=[reg.
     ↪ribosomes], annotation="translation")
        # protein degradation
        sim.reaction([sp.P], [], rc.degrad_p, regions=[reg.shell],␣
     ↪annotation="Protein degradation")

        # conversion from mRNA read state to ready state
        sim.reaction([sp.mRNAr], [sp.mRNA], rc.conversion, regions=[reg.shell, reg.
     ↪cytoplasm], annotation="Conversion from mRNA read state to ready state")
```

now we need all the diffusion coefficients defined:

```
[ ]: with sim.construct():
        sim.transitionRate(None, None, None, sim.diffusionZero)
```

```
[ ]: with sim.construct():
        sim.diffusionConst('mrna',4.13e-14, texRepr=r'D_{mRNA}', annotation="mRNA␣
     ↪diffusion constant for JCVISYN3A_0001")
        sim.diffusionConst('protein', 0.1e-12, texRepr=r'Protein',␣
     ↪annotation="protein diffusion co.")
        # diffusion for mrna
        sim.transitionRate(sp.mRNA, reg.DNA, reg.cytoplasm, dc.mrna)
        sim.transitionRate(sp.mRNA, reg.cytoplasm, reg.ribosomes, dc.mrna)
        sim.transitionRate(sp.mRNA, reg.cytoplasm, reg.cytoplasm, dc.mrna)
        sim.transitionRate(sp.mRNA, reg.ribosomes, reg.ribosomes, dc.mrna)
        # diffusion for mrna_read, it is the read state after the mrna translation,
        # mrna_read formed in ribsomes, and conversion happen in cytoplasm and shell
        # so we must allow it to diffuse in cytoplasm and shell
```

```
    sim.transitionRate(sp.mRNAr, reg.ribosomes, reg.ribosomes, dc.mrna)
    sim.transitionRate(sp.mRNAr, reg.ribosomes, reg.cytoplasm, dc.mrna)
    sim.transitionRate(sp.mRNAr, reg.ribosomes, reg.shell, dc.mrna)
    sim.transitionRate(sp.mRNAr, reg.cytoplasm, reg.cytoplasm, dc.mrna)
    sim.transitionRate(sp.mRNAr, reg.cytoplasm, reg.shell, dc.mrna)
    sim.transitionRate(sp.mRNAr, reg.shell, reg.shell, dc.mrna)
    sim.transitionRate(sp.mRNAr, reg.shell, reg.cytoplasm, dc.mrna)

    # diffusion for protein
    sim.transitionRate(sp.P, reg.ribosomes, reg.ribosomes, dc.protein)
    sim.transitionRate(sp.P, reg.ribosomes, reg.cytoplasm, dc.protein)
    sim.transitionRate(sp.P, reg.ribosomes, reg.shell, dc.protein)
    sim.transitionRate(sp.P, reg.cytoplasm, reg.cytoplasm, dc.protein)
    sim.transitionRate(sp.P, reg.cytoplasm, reg.shell, dc.protein)
    sim.transitionRate(sp.P, reg.shell, reg.shell, dc.protein)
    sim.transitionRate(sp.P, reg.shell, reg.cytoplasm, dc.protein)
```

6 5. Initial counts

We need one DNA particle in the starting site to represent gene. Since our gene is JCVISYN3A_0001, we put the gene particle in the first lattice cube of DNA_region.

Other species inital counts all set to be 0:

#gene = 1

#mRNA = 1

#mRNA_read = 0

#Protein = 0

[ ]:
```
gene_pos = DNA_pos[0]
sim.placeNumber(sp=sp.gene,x=gene_pos[0], y=gene_pos[1], z=gene_pos[2], n=1)

# then we randomly put one mRNA in the cytoplasm
sim.distributeNumber(sp=sp.mRNA, reg=reg.cytoplasm, count=1)
```

[ ]:
```
sim.showAllSpecies()
```

[ ]:
```
sim
```

[ ]:
```
sim.finalize()
```

7 6. Run the simulation

[ ]:
```
sim.run(solver=IntMpdRdmeSolver(), cudaDevices=[0])
```

8 7. Analysis

[ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns
from jLM.RDME import File as RDMEFile
```

[ ]:
```
traj = RDMEFile(filename,replicate=1)
ts, genes = traj.getNumberTrajectory(species="gene")
ts, mRNAs = traj.getNumberTrajectory(species="mRNA")
ts, mRNArs = traj.getNumberTrajectory(species="mRNAr")
ts, proteins = traj.getNumberTrajectory(species="P")
```

[ ]:
```
# now visuzlize the trajectories with seaborn and plt
sns.set(style="whitegrid")
# Create a color palette

colors = sns.color_palette("husl", 3)
plt.figure(figsize=(10,6))
plt.plot(ts, genes, label='gene',color=palette[0])
plt.plot(ts, mRNAs + mRNArs, label='mRNA',color=palette[1])
plt.plot(ts, proteins, label='protein',color=palette[2])

plt.title('Trajectories of Genetic information process')
plt.yticks(np.arange(0, 2.1, step=1))
plt.xlabel('Time (s)')
plt.ylabel('Counts')
plt.legend()
plt.savefig('./images/TutR2_GIP.png')
plt.show()
```

3.3 Tutorial RDME 3: Galactose switch Model with RDME-ODE hybrid method

In this tutorial, we will simulate a yeast model with RDME-ODE hybrid method. The model is based on the work from CME-ODE model from David Binachi, et al. [3]. The only differences is that, we put the reactions in RDME model instead of CME model, so we need to define the regions for each reactions.
The original four state cascade regulatory model was proposed by Murat Acar, Attila Becskei and Alexander van Oudenaarden [13] and the model was extended to include the Ga lactose Transporter in 2018. All rate constants were originally taken from Stephen Ramsey, et al. [14].
We briefly introduce the model here, for details, please check the paper [3].
The schematic is shown here:

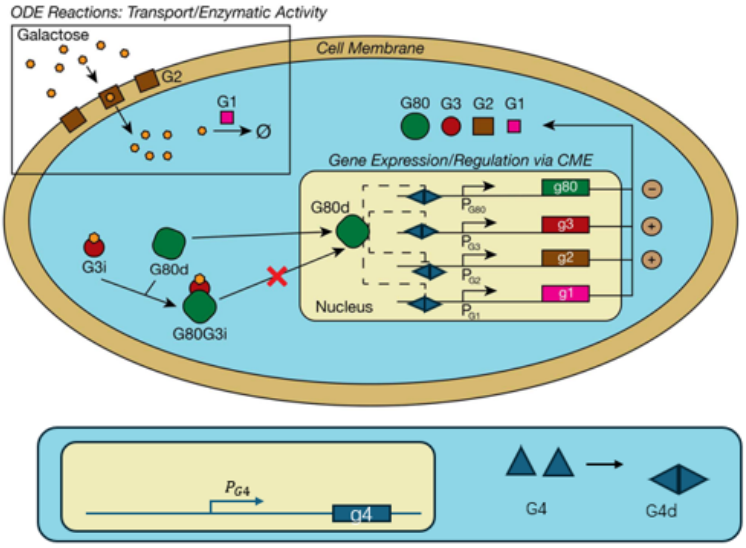


Figure 3.3: Galactose Switch in Baker's Yeast. Figure modified from David Binachi, et al. [3]

We list all species, diffusion coefficients and reaction types below:

Table 3.3: Table of Proteins and Their Functions and Related Complexes

Proteins	Function	Related complexes
Gal1p	Galactose degradation in cytoplasm	G1
Gal2p	Galactose transporter on the cell membrane	G2
Gal3p	Regulate the G80p concentration in cytoplasm	G3, G3i, G80G3i
G80p	Repress the transcription of g1, g2, g3, g80	G80d
Gal4p	Transcriptional activator	G4d

Table 3.4: Diffusion Characteristics of Various Species

Species	Diffusion Direction	Diffusion Coefficient $(m^2 \cdot s^{-1})$
DNA	Fixed in nucleoplasm	0
mRNA	Nucleoplasm $\rightarrow$ cytoplasm	5e-13
Protein out of ribosomes	Ribosomes $\rightarrow$ cytoplasm	10e-13
Transcription factors	Nucleoplasm $\rightarrow$ cytoplasm	10e-13
Cytoplasm Proteins	Cytoplasm $\rightarrow$ nucleoplasm	0
Transporter(G2)	Cytoplasm $\rightarrow$ PlasmaMem	1e-13

Table 3.5: Reactions and Their Characteristics

Reactions Types	Template	Reaction Region
Transcription	$D_{G1} - G_{4d} \rightarrow R_1 + D_{G1} - G_{4d}$	Nucleoplasm
Translation	$R_1 \rightarrow R_1 + G_1$	Ribosomes
DNA Regulation with Gal4d protein	$D_{G1} + G_{4d} \rightarrow D_{G1G4d}$	Nucleoplasm
Dimerization	$G_4 + G_4 \rightarrow G_{4d}$	Nucleoplasm; Cytoplasm
G3 activation	$G_3 + G_{AI} \rightarrow G_{3i}$	Cytoplasm
$G_{3i} + G_{80d_{\text{cyto}}} \rightarrow G_{80G3i}$	Cytoplasm	
Degradation	$G_{4d} \rightarrow \emptyset$	All regions needed

For all the reactions, species abbreviations in the code, please check the supplementary information from the paper [3].
During the class, due to time limit, we will only run simulation for 2 seconds with hook interval 0.1s to verify the model can be executed. In the future, audiences can extend it to 60 minutes to get some interesting results.
Take care that, since G2 is treated separately as three different states in ODE part, the counts of G2 in RDME = G2 + G2GAE + G2GAI in ODE. Similarily, the counts of G1 in RDME = G1 + G1GAI in ODE.

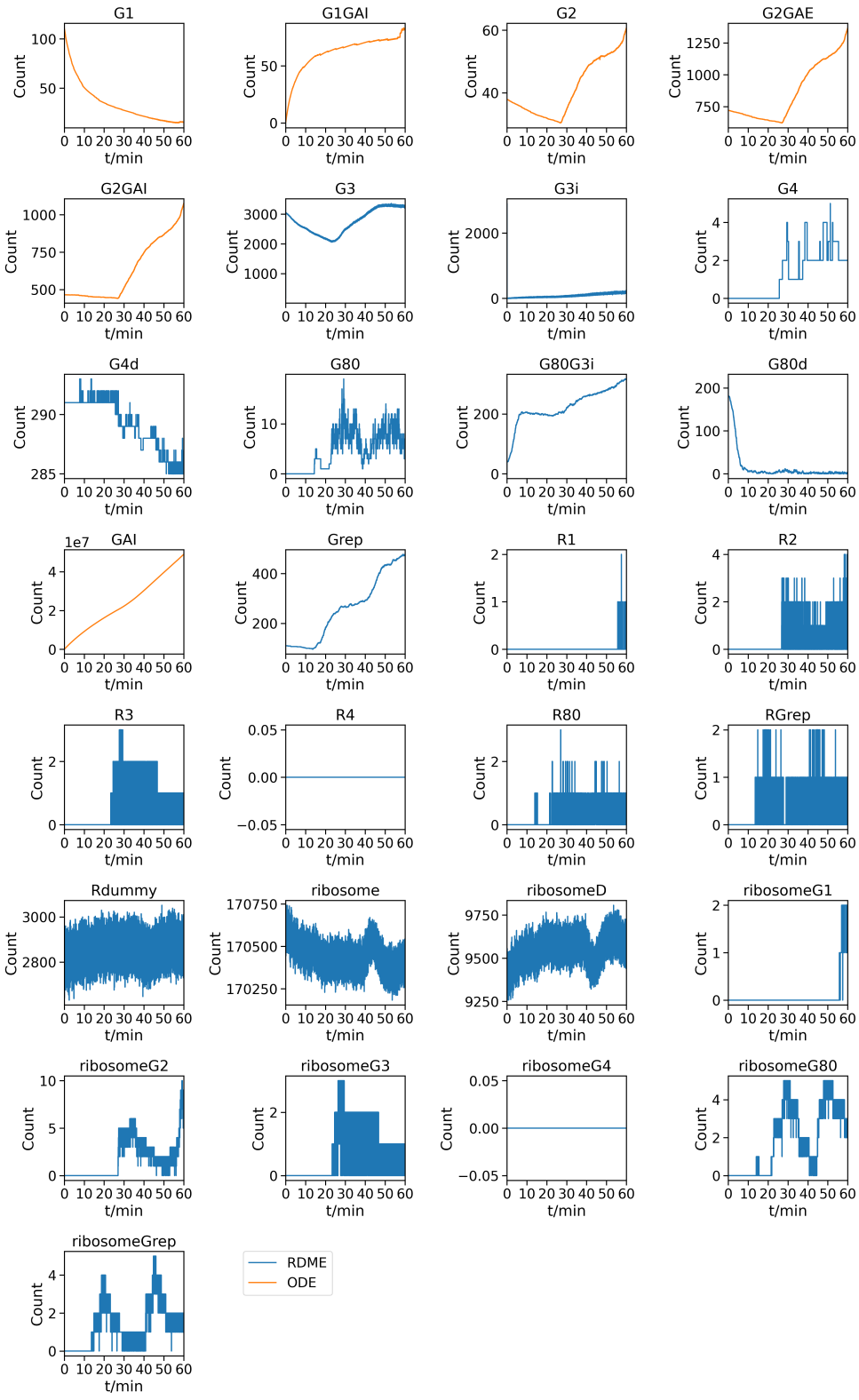


Figure 3.4: results from RDME-ODE hybrid yeast model in 60 mins.

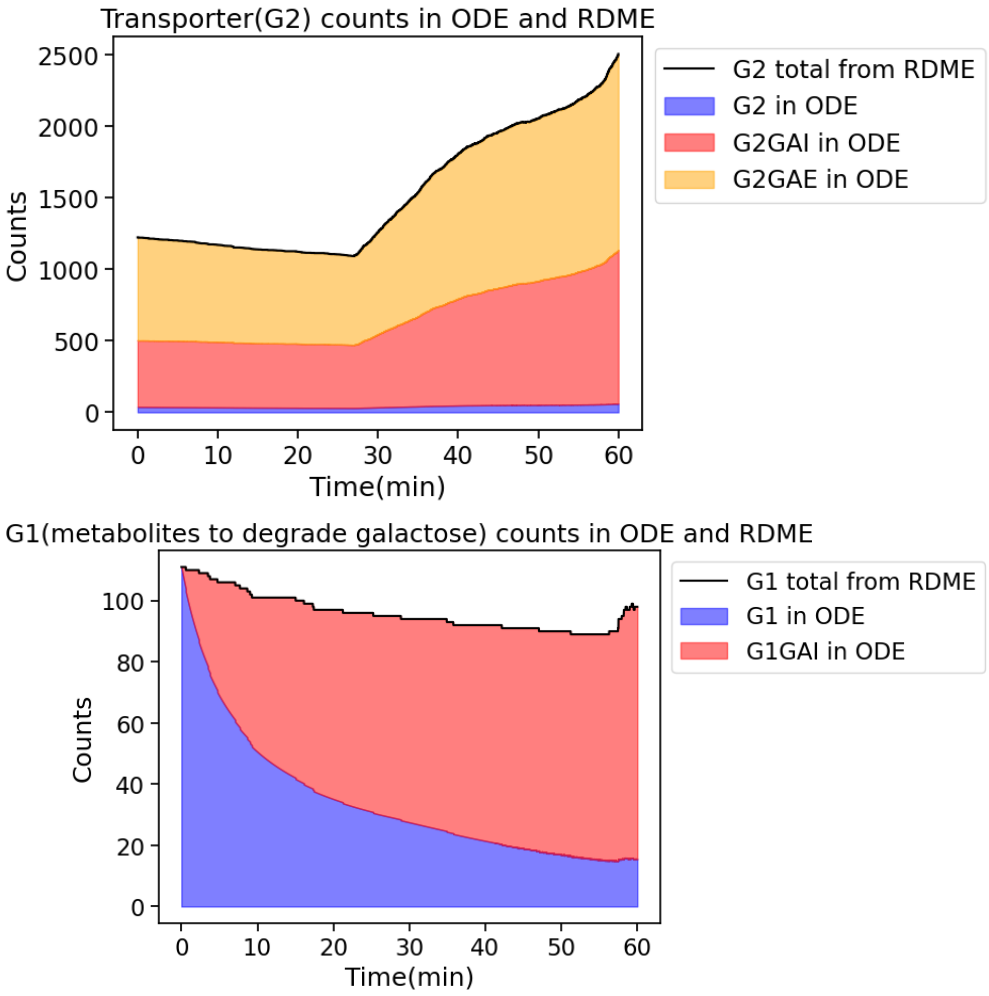


Figure 3.5: G1 and G2 counts comparison from RDME-ODE hybrid yeast model in 60 mins.

Chapter 4

License and Copyright

Bibliography

[1] Fujikawa, N, Kurumizaka, H, Nureki, O, Terada, T, Shirouzu, M, Katayama, T, Yokoyama, S (2003) Structural basis of replication origin recognition by the DnaA protein. *Nucleic Acids Research* 31:2077–2086.

[2] Duderstadt, KE, Chuang, K, Berger, JM (2011) DNA stretching by bacterial initiators promotes replication origin opening. *Nature* 478:209–213 Publisher: Nature Publishing Group.

[3] Bianchi, DM, Peterson, JR, Earnest, TM, Hallock, MJ, Luthey-Schulten, Z (year?) Hybrid CME–ODE method for efficient simulation of the galactose switch in yeast. 12:170–176.

[4] 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)*.

[5] Thornburg, ZR, Bianchi, DM, Brier, TA, Gilbert, BR, Earnest, TM, Melo, MC, Safronova, N, Sáenz, JP, Cook, AT, Wise, KS, Hutchison, CA, Smith, HO, Glass, JI, Luthey-Schulten, Z (year?) Fundamental behaviors emerge from simulations of a living minimal cell. 185:345–360.e28.

[6] Thornburg, ZR, Melo, MCR, Bianchi, D, Brier, TA, Crotty, C, Breuer, M, Smith, HO, Hutchison, CA, Glass, JI, Luthey-Schulten, Z (2019) Kinetic Modeling of the Genetic Information Processes in a Minimal Cell. *Frontiers in Molecular Biosciences* 6.

[7] Shahrezaei, V, Swain, PS (year?) Analytical distributions for stochastic gene expression. 105:17256–17261.

[8] Hutchison, CA, Chuang, RY, Noskov, VN, Assad-Garcia, N, Deerinck, TJ, Ellisman, MH, Gill, J, Kannan, K, Karas, BJ, Ma, L, Pelletier, JF, Qi, ZQ, Richter, RA, Strychalski, EA, Sun, L, Suzuki, Y, Tsvetanova, B, Wise, KS, Smith, HO, Glass, JI, Merryman, C, Gibson, DG, Venter, JC (2016) Design and synthesis of a minimal bacterial genome. *Science* 351:aad6253 Publisher: American Association for the Advancement of Science.

[9] Breuer, M, Earnest, TM, Merryman, C, Wise, KS, Sun, L, Lynott, MR, Hutchison, CA, Smith, HO, Lapek, JD, Gonzalez, DJ, de Crécy-Lagard, V, Haas, D, Hanson, AD, Labhsetwar, P, Glass, JI, Luthey-Schulten, Z (2019) Essential metabolism for a minimal cell. *eLife* 8:e36842 Publisher: eLife Sciences Publications, Ltd.

[10] Earnest, TM, Cole, JA, Peterson, JR, Hallock, MJ, Kuhlman, TE, Luthey-Schulten, Z (2016) Ribosome biogenesis in replicating cells: Integration of experiment and theory. *Biopolymers* 105:735–751 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/bip.22892.

[11] Hofmeyr, JHS, Gqwaka, OP, Rohwer, JM (2013) A generic rate equation for catalysed, template-directed polymerisation. *FEBS Letters* 587:2868–2875 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1016/j.febslet.2013.07.011.

[12] Liebermeister, W, Klipp, E (2006) Bringing metabolic networks to life: convenience rate law and thermodynamic constraints. *Theoretical Biology and Medical Modelling* 3:41.

[13] Acar, M, Becskei, A, Van Oudenaarden, A (year?) Enhancement of cellular memory by reducing stochastic transitions. 435:228–232.

[14] Ramsey, SA, Smith, JJ, Orrell, D, Marelli, M, Petersen, TW, De Atauri, P, Bolouri, H, Aitchison, JD (year?) Dual feedback loops in the GAL regulon suppress cellular heterogeneity in yeast. 38:1082–1087.