**Supplemental File 2. Gillespie simulation of LRV1 loss under 2CMA inhibition and Java implementation**

As described in the main text, this simulation models *Leishmania* and LRV1 replication. It utilizes the next-reaction modification (NRM) to the Gillespie algorithm {Gibson, 2000 #77}. The model implements two kinds of *entity*: parasites and virions. Each parasite can contain a number of virions. The model also includes two types of *process*: parasite division and virion replication. The NRM algorithm works by first randomly picking times remaining until each entity completes a process, using a probability distribution defined by that process’ rate. The traditional Gillespie algorithm, which simulates elementary chemical reactions, uses an exponential distribution of waiting times, which is appropriate for Poisson-type processes {Gillespie, 1976 #42}. Here, since cell division and virus replication are complex processes involving many steps, we pick individual event times from Gaussian distributions about the mean parasite doubling and virus replication times {Gibson, 2000 #41}. The selected waiting times are then sorted in a priority queue. At each step, the process with the shortest waiting time is executed and any dependent processes have their waiting times updated. Given only two processes, this model’s behavior is defined by the relationship between those processes’ rates. For simplicity’s sake, we let the parasite division time be exactly 8 hours. Given that the number of virions per parasite remains roughly constant, we assume that the effective virion replication rate is equal to the parasite division time under normal conditions {Kuhlmann, 2017 #74;Weeks, 1992 #69}. If the unit of time in this simulation is converted from hours to multiples of the parasite division time, the simulation’s behavior depends entirely on the ratio between the virion replication time and the parasite doubling time.

The secondary parameters defining this model are the initial number of cells and the initial number of LRV1 particles per cell. We found that modeling an initial population of 1000 cells over time, tracking the viral titer in each cell, gave indistinguishable profiles across multiple runs, so we used this value for each simulation. Initial LRV1 numbers were set at 16 per cell, consistent with previous studies {Kuhlmann, 2017 #11}.

Modeling parasite division required also modeling the distribution of virions between the daughter cells. Previously, we have shown that virions are distributed roughly equally when parasites divide {Kuhlmann, 2017 #74}. In theory, this would be modeled using a binomial distribution. However, such a mechanism would allot all virions to one daughter cell frequently enough (1 in 6.6×105 divisions with a full 16 virions) to make LRV1+ populations unstable and cause spontaneous virus loss far more often than actually observed {Ro, 1997 #34}. Instead, we assumed that virion partitioning is not entirely random, but instead based on location, resulting in daughter cells receiving closer to 50% of the parent’s virions. To model this, the proportion of virions given to new daughter cells was selected from a Gaussian distribution with mean of 50% and standard deviation of 6.25%. This standard deviation was selected to give a vanishingly small chance of stochastically “curing” a daughter of a parasite with 16 virions.

## **Java implementation**

## NRSMain2.java

This class includes the overarching logic that controls simulations. Several arrays of parameters allow users to define and run a series of simulations automatically. This program makes use of the Apache Math Library, version 3.6.1.

package nrs;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.PrintStream;

import java.util.HashMap;

import java.util.LinkedList;

import java.util.Map.Entry;

import java.util.PriorityQueue;

import java.util.TreeMap;

import java.util.concurrent.ForkJoinPool;

import java.util.concurrent.TimeUnit;

// Apache Math 3.6.1 from http://commons.apache.org/proper/commons-math/

import org.apache.commons.math3.random.MersenneTwister;

import org.apache.commons.math3.special.Erf;

public class NRSMain2 {

private static MersenneTwister RAND = new MersenneTwister();

private static double SD\_SCALE = 16;

private static double DEFAULT\_CELL\_T2 = 8;

private static double DEFAULT\_VIRUS\_T2 = 8;

/\*\*

Number of parasites to put into each simulation initially.

\*/

private static int POPULATION\_SIZE = 1000;

/\*\*

How many hours to run the simulation for.

\*/

private static double STOP\_TIME = 48;

/\*\*

How many intermediate population snapshots to record.

\*/

private static double SNAP\_INTERVAL = 8;

/\*\*

\* @param args

\*/

public static void main(String[] args) {

Parasite newCell;

Virion vir;

Simulation sim;

LinkedList<Simulation> sims;

PriorityQueue<Process> equilPop, popClone;

// Set parameters for the simulations we'll be running.

// Each column corresponds to the parameters for one simulation.

// Parasite doubling times

double[] cellT2s = {8,8,8,8,8,8};

// Virus replication times

double[] virusT2s = { 8\*2, 8\*2, 8\*2, 8\*3, 8\*3, 8\*3 };

// How many virions per cell initially

int[] initVirions = { 16, 16, 16, 16, 16, 16 };

// Limit on the number of virions per cell. Keep this >>16.

int[] maxVirions = { 80, 80, 80, 80, 80, 80 };

// create initial population

equilPop = new PriorityQueue<>();

int startPop = POPULATION\_SIZE;

for(int k = 0; k < startPop; k++)

{

// we'll handle the virions belonging to each parasite here too

newCell = new Parasite(NRSMain2.initialTau(DEFAULT\_CELL\_T2,

DEFAULT\_CELL\_T2/SD\_SCALE));

// clone all the virions

for(int j = 0; j < initVirions[0]; j++)

{

// we're messing with virus T2, so we need to recalculate tau

vir = new Virion(newCell,

NRSMain2.initialTau(DEFAULT\_VIRUS\_T2,

DEFAULT\_VIRUS\_T2/SD\_SCALE));

newCell.addVirion(vir);

equilPop.add(vir);

}

equilPop.add(newCell);

}

// set up thread pool to handle all the simulations concurrently

ForkJoinPool threadPool = new ForkJoinPool(Runtime.getRuntime().availableProcessors() - 1);

// now run the actual simulations

sims = new LinkedList<>();

for(int i = 0; i < maxVirions.length; i++)

{

// Clone each parasite and virion

popClone = new PriorityQueue<>();

for(Process p : equilPop)

{

if(p instanceof Parasite)

{

// copy constructor clones the virions too

newCell = new Parasite((Parasite)p);

popClone.add(newCell);

popClone.addAll(newCell.getVirionsView());

// need to update tau values to reflect new rates

newCell.setTau(NRSMain2.initialTau(cellT2s[i], cellT2s[i]/SD\_SCALE));

for(Virion v : newCell.getVirionsView())

{

v.setTau(NRSMain2.initialTau(virusT2s[i], virusT2s[i]/SD\_SCALE));

}

}

}

sim = new Simulation(popClone, cellT2s[i], virusT2s[i], maxVirions[i],

STOP\_TIME, SNAP\_INTERVAL);

sims.add(sim);

threadPool.execute(sim);

}

// wait for all the simulations to end

try {

threadPool.shutdown();

threadPool.awaitTermination(Long.MAX\_VALUE, TimeUnit.DAYS);

} catch (InterruptedException e1) {

e1.printStackTrace();

}

System.out.println("Done!");

System.out.println(sims.getFirst().getSnapshots());

// print out the data to files

for(int i = 0; i < sims.size(); i++)

{

try {

PrintStream ps = new PrintStream(new File("sim-" + i + ".csv"));

printSimulation(sims.get(i), ps);

ps.close();

} catch (FileNotFoundException e) {

System.err.println("Failed to open file: sim-" + i + ".csv");

e.printStackTrace();

}

}

}

/\*\*

\* Pick a random tau value for a process, assuming that instances of that

\* process are at steady state.

\* @param mu the mean tau value

\* @param sigma the standard deviation of tau values

\* @return the random tau value

\*/

private static double initialTau(double mu, double sigma)

{

double a = mu/(sigma\*Math.sqrt(2));

double b = (1+Erf.erf(a))\*Math.PI\*Math.exp(a\*a);

double slope = b/(mu\*b + sigma\*Math.sqrt(2\*Math.PI));

return RAND.nextDouble()/slope;

}

private static void printSimulation(Simulation sim, PrintStream ps)

{

HashMap<Integer, Long> snap;

long count, sum;

double gSum, time;

StringBuilder hdr, line;

// get maximum number of virions allowed in the simulation

int maxVirions = sim.getMaxVirions();

// get the list of snapshots

TreeMap<Double, HashMap<Integer, Long>> snaps = sim.getSnapshots();

// build header and write it

hdr = new StringBuilder("Time,");

for(int i = 0; i <= maxVirions; i++)

{

hdr.append(i);

hdr.append(',');

}

hdr.append("Total,GeoMean");

ps.println(hdr);

line = new StringBuilder();

// loop over all the snapshots

for(Entry<Double, HashMap<Integer, Long>> entry : snaps.entrySet())

{

time = entry.getKey();

// write out the time stamp

line.append(time);

line.append(',');

// write out the parasite counts

snap = entry.getValue();

sum = 0;

gSum = 0;

for(int i = 0; i <= maxVirions; i++)

{

if(snap.containsKey(i))

{

count = snap.get(i);

}

else

{

count = 0;

}

sum += count;

gSum += count\*Math.log(i+1);

line.append(count);

line.append(',');

}

// write out total and geometric mean (Williams' modification)

line.append(sum);

line.append(',');

gSum = Math.exp(gSum/sum) - 1;

line.append(gSum);

// write the line

ps.println(line);

// reset the line

line.setLength(0);

}

}

}

## Simulation.java

This class implements an individual LRV1 simulation, given an initial parasite population and a set of parameters and options.

package nrs;

import java.util.HashMap;

import java.util.HashSet;

import java.util.LinkedList;

import java.util.Map.Entry;

import java.util.PriorityQueue;

import java.util.TreeMap;

import java.util.concurrent.ForkJoinTask;

import java.util.concurrent.RecursiveAction;

// Apache Math 3.6.1 from http://commons.apache.org/proper/commons-math/

import org.apache.commons.math3.distribution.NormalDistribution;

import org.apache.commons.math3.random.MersenneTwister;

public class Simulation extends RecursiveAction {

private static final long serialVersionUID = -2016479837671294476L;

private static final double SD\_SCALE = 16;

// Narrows the standard deviation of the proportion of virions given to a

// new daughter cell.

private static final double DIVISION\_SD\_SCALE = 2;

private MersenneTwister myRand;

private double myCellT2, myVirusT2;

private int myMaxVirions;

private double myStopTime, mySnapInterval;

private NormalDistribution myCellPDF, myVirusPDF;

private TreeMap<Double, HashMap<Integer, Long>> mySnapshots;

private PriorityQueue<Process> myPopulation;

public Simulation(PriorityQueue<Process> population, double cellT2,

double virusT2, int maxVirions, double stopTime, double snapInterval) {

myPopulation = population;

myCellT2 = cellT2;

myVirusT2 = virusT2;

myMaxVirions = maxVirions;

myStopTime = stopTime;

mySnapInterval = snapInterval;

myRand = new MersenneTwister();

myCellPDF = new NormalDistribution(myRand, myCellT2, myCellT2/25.74);

myVirusPDF = new NormalDistribution(myRand, myVirusT2, myVirusT2/25.74);

mySnapshots = new TreeMap<>();

}

/\*\*

\* Get a map of the snapshots taken by this simulation.

\* @return the snapshots

\*/

public TreeMap<Double, HashMap<Integer, Long>> getSnapshots() {

return mySnapshots;

}

/\*\*

\* @return the maxVirions

\*/

public int getMaxVirions() {

return myMaxVirions;

}

private void run() {

double time, mean, sd;

int numVirions, d1, d2;

LinkedList<Double> snapTimes;

Double nextSnap;

Process rxn;

Parasite cell, daughter;

Virion vir, newVir;

// list of processes with changed tau values

HashSet<Process> needsUpdating = new HashSet<>();

// calculate the list of snapshot times

snapTimes = new LinkedList<Double>();

for(double i = 0; i <= myStopTime; i += mySnapInterval)

{

snapTimes.add(i);

}

nextSnap = snapTimes.pop();

time = 0;

// take an initial snapshot

System.out.println("Running simulation: " + this);

mySnapshots.put(nextSnap, this.takeSnapshot());

nextSnap = snapTimes.pop();

// Any time a parasite divides or virion replicates:

// Generate tau for new cells or virions

// Put tau values into the priority queue

// Tau for next event is minimum value in queue

while(time < myStopTime)

{

// (#) Pick reaction from P with minimum tau\_r

rxn = myPopulation.peek();

// Update time

time = rxn.getTau();

// Change numbers of molecules to reflect the reaction

if(rxn instanceof Parasite)

{

// parasite is dividing

cell = (Parasite)rxn;

numVirions = cell.getVirionCount();

// use squashed normal distribution

mean = numVirions/2.0;

sd = Math.sqrt(mean)/2.0/DIVISION\_SD\_SCALE;

d1 = (int) Math.round(sd\*myRand.nextGaussian() + mean);

if(d1 < 0)

{

d1 = 0;

}

else if(d1 >= numVirions)

{

d1 = numVirions;

}

d2 = numVirions - d1;

// if the dividing parasite was at maximum virions, we need to

// restart virus replication

if(numVirions >= myMaxVirions)

{

for(Virion v : cell.getVirionsView())

{

v.setTau(time + this.pickVirusTau());

needsUpdating.add(v);

}

}

// create and populate daughter cell

daughter = new Parasite(time + this.pickParasiteTau());

needsUpdating.add(daughter);

for(int i = 0; i < d2; i++)

{

vir = cell.removeVirion(0);

vir.setHost(daughter);

daughter.addVirion(vir);

}

// pick new tau value for the parent cell

cell.setTau(time + this.pickParasiteTau());

needsUpdating.add(cell);

}

else if(rxn instanceof Virion)

{

// virus is replicating

vir = (Virion)rxn;

cell = vir.getHost();

// create new virion

newVir = new Virion(cell, 0);

cell.addVirion(newVir);

if(cell.getVirionCount() >= myMaxVirions)

{

// maximum number of virions reached, so stop replication

for(Virion v : cell.getVirionsView())

{

v.setTau(Double.POSITIVE\_INFINITY);

needsUpdating.add(v);

}

}

else

{

// set tau for new virion and one that replicated

vir.setTau(time + this.pickVirusTau());

needsUpdating.add(vir);

newVir.setTau(time + this.pickVirusTau());

needsUpdating.add(newVir);

}

}

// Rebuild tau queue to account for changes

myPopulation.removeAll(needsUpdating);

myPopulation.addAll(needsUpdating);

needsUpdating.clear();

// take a snapshot, if applicable

if(time >= nextSnap)

{

mySnapshots.put(nextSnap, this.takeSnapshot());

nextSnap = snapTimes.poll();

}

// Loop back to (#)

}

}

/\*\*

\* Select a random length of time that a parasite will wait before dividing.

\* @return the time

\*/

private double pickParasiteTau()

{

return myCellPDF.sample();

}

/\*\*

\* Select a random length of time that a parasite will wait before dividing.

\* @return the time

\*/

private double pickVirusTau()

{

return myVirusPDF.sample();

}

/\*\*

\* Takes a snapshot of the current parasite population.

\* @return a map with the number of parasites containing a given number

\* of virions

\*/

private HashMap<Integer, Long> takeSnapshot()

{

int numVirions;

Long count;

HashMap<Integer, Long> histo = new HashMap<>();

for(Process p : myPopulation)

{

if(p instanceof Parasite)

{

// increment the bin for parasites with this many virions

numVirions = ((Parasite)p).getVirionCount();

count = histo.get(numVirions);

if(count == null)

histo.put(numVirions, 1L);

else

histo.put(numVirions, count + 1L);

}

}

return histo;

}

@Override

protected void compute() {

int index;

Parasite cell;

Simulation sim1, sim2;

PriorityQueue<Process> pop1, pop2;

if(myPopulation.size() > 200)

{

// this simulation is too big, so split it up by turning it into

// two jobs with half as many parasites

index = 0;

pop1 = new PriorityQueue<>();

pop2 = new PriorityQueue<>();

for(Process p : myPopulation)

{

if(p instanceof Parasite)

{

cell = (Parasite)p;

// add parasite and its virions to one population or the other

if(index%2 == 0)

{

pop1.add(cell);

pop1.addAll(cell.getVirionsView());

}

else

{

pop2.add(cell);

pop2.addAll(cell.getVirionsView());

}

index++;

}

}

sim1 = new Simulation(pop1, myCellT2, myVirusT2, myMaxVirions,

myStopTime, mySnapInterval);

sim2 = new Simulation(pop2, myCellT2, myVirusT2, myMaxVirions,

myStopTime, mySnapInterval);

// launch the two simulations and wait for them to complete

ForkJoinTask.invokeAll(sim1, sim2);

this.combine(sim1, sim2);

}

else

{

// we've got a small enough simulation, so just run it

this.run();

}

}

/\*\*

\* Merges two sub-simulations back into this one.

\* @param sim1

\* @param sim2

\*/

private void combine(Simulation sim1, Simulation sim2) {

long count;

Integer numVirions;

Parasite cell;

HashMap<Integer, Long> histo1, histo2;

// clear the out-of-date population

myPopulation.clear();

// add each sub-population

for(Process p : sim1.myPopulation)

{

if(p instanceof Parasite)

{

cell = (Parasite)p;

myPopulation.add(cell);

myPopulation.addAll(cell.getVirionsView());

}

}

for(Process p : sim2.myPopulation)

{

if(p instanceof Parasite)

{

cell = (Parasite)p;

myPopulation.add(cell);

myPopulation.addAll(cell.getVirionsView());

}

}

// combine snapshots

for(Entry<Double, HashMap<Integer, Long>> e : sim1.getSnapshots().entrySet())

{

if(!mySnapshots.containsKey(e.getKey()))

{

// this snapshot is not present in the parent list

mySnapshots.put(e.getKey(), e.getValue());

}

else

{

histo1 = mySnapshots.get(e.getKey());

histo2 = e.getValue();

for(Entry<Integer, Long> eh : histo2.entrySet())

{

numVirions = eh.getKey();

if(histo1.containsKey(numVirions))

{

// add this entry to a preexisting one

count = histo1.get(numVirions);

count += histo2.get(numVirions);

histo1.put(numVirions, count);

}

else

{

// add a new entry

histo1.put(numVirions, eh.getValue());

}

}

}

}

for(Entry<Double, HashMap<Integer, Long>> e : sim2.getSnapshots().entrySet())

{

if(!mySnapshots.containsKey(e.getKey()))

{

// this snapshot is not present in the parent list

mySnapshots.put(e.getKey(), e.getValue());

}

else

{

histo1 = mySnapshots.get(e.getKey());

histo2 = e.getValue();

for(Entry<Integer, Long> eh : histo2.entrySet())

{

numVirions = eh.getKey();

if(histo1.containsKey(numVirions))

{

// add this entry to a preexisting one

count = histo1.get(numVirions);

count += histo2.get(numVirions);

histo1.put(numVirions, count);

}

else

{

// add a new entry

histo1.put(numVirions, eh.getValue());

}

}

}

}

}

/\* (non-Javadoc)

\* @see java.lang.Object#toString()

\*/

@Override

public String toString() {

return "Simulation [cellT2=" + myCellT2 + ", virusT2=" + myVirusT2

+ ", maxVirions=" + myMaxVirions + ", stopTime="

+ myStopTime + ", processes=" + myPopulation.size() + "]";

}

}

## Parasite.java

This class defines the properties of a parasite. It holds the time when this parasite will divide and also keeps track of the virions contained within this parasite.

package nrs;

import java.util.Collection;

import java.util.Collections;

import java.util.LinkedList;

import java.util.List;

public class Parasite extends Process {

private LinkedList<Virion> myVirions;

public Parasite() {

this(Double.POSITIVE\_INFINITY, Collections.<Virion> emptyList());

}

public Parasite(double tau)

{

this(tau, Collections.<Virion> emptyList());

}

/\*\*

Constructs a new parasite with some virions and a given time when

it will divides.

@param tau when this parasite will divide

@param virions the virions inside this parasite

\*/

public Parasite(double tau, Collection<Virion> virions)

{

super(tau);

myVirions = new LinkedList<Virion>(virions);

// make sure all virions are assigned to me

for(Virion v : myVirions)

{

v.setHost(this);

}

}

/\*\*

\* Copy constructor for parasites.

\* @param p the parasite to copy

\*/

public Parasite(Parasite p)

{

super(p.getTau());

myVirions = new LinkedList<Virion>();

Virion copy;

for(Virion v : p.myVirions)

{

copy = new Virion(v);

copy.setHost(this); // set the cloned host

myVirions.add(copy);

}

}

/\*\*

\*

\* @return the virions

\*/

public List<Virion> getVirionsView() {

return Collections.unmodifiableList(myVirions);

}

/\*\*

\* @return

\* @see java.util.LinkedList#size()

\*/

public int getVirionCount() {

return myVirions.size();

}

/\*\*

\* @param e

\* @return

\* @see java.util.LinkedList#add(java.lang.Object)

\*/

public boolean addVirion(Virion e) {

return myVirions.add(e);

}

public Virion removeVirion(int index)

{

return myVirions.remove(index);

}

/\*\*

\* Simply calls the Process equals method, which compares objects based on

\* their tau values.

\* @param obj the other process

\*/

@Override

public boolean equals(Object obj) {

return super.equals(obj);

}

/\* (non-Javadoc)

\* @see nrs.Process#hashCode()

\*/

@Override

public int hashCode() {

return super.hashCode();

}

}

## Virion.java

This class implements the properties of a virion. It keeps track of when this virion will next replicate and what parasite it is infecting.

package nrs;

public class Virion extends Process {

private Parasite myHost;

public Virion(Parasite host, double tau) {

super(tau);

myHost = host;

}

/\*\*

\* Copy constructor for virions.

\* @param v the virion to copy

\*/

public Virion(Virion v)

{

super(v.getTau());

this.myHost = v.myHost;

}

/\*\*

\* @return the host

\*/

public Parasite getHost() {

return myHost;

}

/\*\*

\* @param host the host to set

\*/

public void setHost(Parasite host) {

myHost = host;

}

/\*\*

\* Simply calls the Process equals method, which compares objects based on

\* their tau values.

\* @param obj the other process

\*/

@Override

public boolean equals(Object obj) {

return super.equals(obj);

}

/\* (non-Javadoc)

\* @see nrs.Process#hashCode()

\*/

@Override

public int hashCode() {

return super.hashCode();

}

}

## Process.java

This generic class defines processes in a next-reaction Gillespie simulation. The Parasite and Virion classes represent cell division and virus replication processes, so they inherit from this class.

package nrs;

/\*\*

\* Generic class defining processes in a next-reaction Gillespie simulation.

\*

\*/

public class Process implements Comparable<Process> {

private double myTau;

public Process(double tau)

{

myTau = tau;

}

/\*\*

\* Gets the absolute time when this process will next occur.

\* @return the time

\*/

public double getTau()

{

return myTau;

}

/\*\*

\* Set the absolute time when this process will next occur.

\* @param tau the time

\*/

public void setTau(double tau)

{

myTau = tau;

}

/\* (non-Javadoc)

\* @see java.lang.Comparable#compareTo(java.lang.Object)

\*/

@Override

public int compareTo(Process o) {

return Double.compare(this.myTau, o.myTau);

}

/\* (non-Javadoc)

\* @see java.lang.Object#hashCode()

\*/

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

long temp;

temp = Double.doubleToLongBits(myTau);

result = prime \* result + (int) (temp ^ (temp >>> 32));

return result;

}

/\* (non-Javadoc)

\* @see java.lang.Object#equals(java.lang.Object)

\*/

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (!(obj instanceof Process))

return false;

Process other = (Process) obj;

if (Double.doubleToLongBits(myTau) != Double

.doubleToLongBits(other.myTau))

return false;

return true;

}

}