

Hybrid Modeling Framework Manual

Rafael Bravo, Mark Robertson-Tessi, Alexander Anderson

January 24, 2018

Abstract

The presented Hybrid Agent Based Modeling Framework provides a platform to get Agent Based Model building started from simple, efficient, generic components, and facilitates easily combining and extending project code in a shared open source repository. The framework components can broadly be classified into: - on and off lattice agent containers, - finite difference diffusion fields, - a gui building system, and - additional tools and utilities for computation and collecting data. These components were designed to operate independently, but are standardized so as to make them easy to interface with one another. Our lab finds this framework a valuable asset for researchers looking to build efficient 2D and 3D models in Java, while not starting entirely from scratch. It is available on github at <https://github.com/torococo/HybridModelingFramework>. The framework requires Java 8 to run, and the java jdk version 1.8 to compile source code.

Contents

1	Setup	3
1.1	Getting Java	3
1.2	Getting the Framework Source Code	3
1.3	Getting IntelliJ IDEA	3
1.4	Setting up the Project	4
2	Introduction	4
2.1	Learning Java	4
2.2	Using IntelliJ IDEA	5
2.3	Understanding the Framework	5
2.4	Source Code Organization	5
2.4.1	Examples	5
2.4.2	LEARN_HERE	5
2.4.3	Framework	6
3	Grids and Agents	6
3.1	Types of Grid	7
3.2	Types of Agent	7
3.3	Grid and Agent Class Definition	7

3.4	Grid Constructors	8
3.5	Agent Constructors	8
3.6	Grid Indexing	9
3.6.1	Single Indexing	9
3.6.2	Square Indexing	9
3.6.3	Point Indexing	9
3.7	Typical Grid Loop	9
3.8	Agent Age and the Tick System	10
3.9	Agent Functions	10
3.10	Universal Grid Functions and Properties	11
3.11	Agent Grid Method Descriptions	12
3.12	PDEGrid Method Descriptions	13
3.13	Griddouble/Gridint/Gridlong Method Descriptions	15
4	Util.java	15
4.1	Util Array Functions	15
4.2	Util Neighborhood Functions	16
4.3	Util Math Functions	17
4.4	Util Misc Functions	17
4.5	Util Color Functions	18
4.6	Util MultiThread Function	19
4.7	Util Save and Load Functions	19
5	Rand.java	20
6	Gui	21
6.1	Types of Gui and Method Descriptions	21
6.1.1	GridWindow	21
6.1.2	GuiWindow	22
6.1.3	Vis2DOpenGL	23
6.1.4	Vis3DOpenGL	23
6.2	Types of GuiComponent and Method Descriptions	25
6.2.1	GridVis	25
6.2.2	GuiLabel	25
6.2.3	GuiButton	25
6.2.4	BoolParam	25
6.2.5	IntParam	25
6.2.6	DoubleParam	25
6.2.7	StringParam	25
6.2.8	ComboBoxParam	25
6.2.9	FileChooserParam	26
7	Tools	26
7.1	Tools/ FileIO	26
7.2	Tools/ SerializableModel	27

8	Example: Competitive Release Model	27
8.1	Competitive Release Introduction	28
8.2	Main Function	31
8.3	Grid Constructor and Properties	32
8.4	InitTumor Function	34
8.5	ModelStep Function	35
8.6	CellStep Function and Cell Properties	36
8.7	DrawModel Function	38
8.8	Imports	38
8.9	Model Results	39

1 Setup

1.1 Getting Java

As pretty as the Framework source code is, your going to need at least the Java8 JDK (Java Development Kit) installed to do anything with it.

To check if the JDK is installed, open a command line/terminal/cmd window and enter

```
1 java -version
2 javac -version
```

If both of these commands print 1.8.anything or later, you're good to go. If not, get the latest Java JDK by googling it or from the oracle website.

<http://www.oracle.com/technetwork/java/javase/downloads/jdk9-downloads-3848520.html>

You don't need to download the demos and samples.

1.2 Getting the Framework Source Code

To download the framework java files, go to

<https://github.com/torococo/AgentFramework>

and click the clone or download button to get a zip file containing the source code along with the included examples.

unzip this folder and put it somewhere easily accessible.

1.3 Getting IntelliJ IDEA

If you're a perfect robot, or a superintelligent alien from the future, then an ide won't improve your code writing at all, so skip this section.

IntelliJ idea is my favorite ide for programming in Java, and it probably should be yours too (<https://dzone.com/articles/why-idea-better-eclipse>). It can be downloaded here:

<https://www.jetbrains.com/idea/download/>

make sure you download the community edition, unless you really don't know what to do with your grant money.

1.4 Setting up the Project

1. Open IntelliJ Idea and click “create project from existing sources” (“[file/new/ project from existing sources](#)” from the main gui) and direct it to the unzipped AgentFramework Source code directory.
2. Continue through the rest of the setup, you can basically click next until it asks for the Java sdk:
 - “[/Library/ Java/ JavaVirtualMachines/](#)” on mac.
 - “[C:\ Program Files\ Java\](#)” on windows.
3. Once the setup is complete we will need to do one more step and add some libraries that allow for 2D and 3D OpenGL visualization:
2. open the IntelliJ IDEA main gui
3. go to “[file/ project structure](#)”
4. click the “[libraries](#)” tab
5. use the minus button to remove any pre-existing library setup
6. click the plus button, and direct the file browser to the “[Framework/ lib](#)” folder.
7. click apply or ok

This will setup the classes, sources, and native library locations that OpenGL needs. Try running the “[Examples/Example3D](#)” program in the examples folder to see that everything is working properly

If you think that the colorscheme that comes with IntelliJ leaves something to be desired, you’re not alone. try “[File/ import settings](#)” and give it the NinjaTurtleScheme.jar file in the top level folder.

2 Introduction

2.1 Learning Java

If your already pretty familiar with programming, and want a quick, shallow description of the language syntax, the first video on this list may be enough to get up to speed and jump into the framework:

https://www.youtube.com/results?search_query=learn+java

The resources are listed in order by popularity, just try some until you find one that jives well. For a more textual perspective,

<https://www.codecademy.com/learn/learn-java>

<http://www.learnjavaonline.org/>

are some potential places to start.

2.2 Using IntelliJ IDEA

Some of the ways in which IntelliJ may make your life easier include:

- automatically importing classes as they are first mentioned in the code (right click on the class name and IntelliJ should offer to import it)
- debugging with the debugger (shocker) when necessary rather than relying on print statements alone (<https://www.youtube.com/watch?v=1bCgzjatcr4>), HINT: while paused in the debugger right click on anything and click “evaluate expression”
- using refactoring to rename variables, change function signatures, etc. without having to go hunting for every place that the variable/function/class is mentioned. (right click on some code and check out the “refactor” submenu)
- using “find usages” and “Go To Declaration” to move fluidly around your code base and to see how its pieces connect together. (right clicking on things will get you there, see “find usages” and the “go to” submenu)
- using Shift-F10 to quickly run the currently open file, and learning other hotkeys to speed up your workflow.
- tapping Shift twice allows you to search the entire codebase for anything
- using fori, iter, and sout shortcuts to create a for loop, foreach loop, and print statement respectively

Don’t worry too much about learning all of these up front, but I suggest that you do check them out as you become more comfortable with the platform.

2.3 Understanding the Framework

Once you have a basic grasp of Java, I recommend skimming this manual as well as checking out the **LEARN_HERE** folder. then look at the examples

2.4 Source Code Organization

At the top level, there are 3 folders which hold different framework parts. These are...

2.4.1 Examples

contains models that use the framework as a basis

2.4.2 LEARN_HERE

contains small examples that serve as tests of framework components

2.4.3 Framework

the top level framework source code folder, contains all of the following subfolders that hold various framework components. Descriptions of the 7 subfolders contained within the framework folder are detailed below.

- Grids: this folder contains all of the Agent and Grid types that the framework supports. There are 2D and 3D, stackable and unstackable, on-lattice and off-lattice agents and grids to contain them. The base classes that the grids and agents extend are also in this folder.
- Gui: this folder contains the main Gui window class, as well as many component classes that can be added to the Gui.
- Tools: this folder contains many useful tool classes, such as a FileIO wrapper, a genetic algorithm class, etc. as well as a Util class that is a container of generic static methods. It also contains classes that are used internally by these tools.
- Interfaces: this folder contains interfaces that are used by the framework internally. It can be useful to reference this folder if a framework function takes a function interface argument, or implements an interface.
- Extensions: contains more specialized framework components that are created by extending the generic ones.
- Lib: contains outside libraries that have been integrated with the framework.
- Util: The only file directly on the top level, it's full of goodies that can be broadly categorized into: color functions, array functions, cellular automata neighborhood functions, and math functions.

The manual from here turns into a glossary that looks at the framework components in some detail (to the extent that I bothered to write about them) and after that a simple but complete model example is provided.

Those looking to dive headfirst into modeling may want to first go to the [LEARN_HERE](#) folder for a hands on approach, or check out the last section for an in depth explanation of a model and use the other chapters as a reference. If you like reading dictionaries in your spare time then read this thing from end to end.

3 Grids and Agents

The bread and butter of the framework consists of different types of Grids and Agents. these are presented below.

3.1 Types of Grid

Grid2D Holds all 2D agents

Grid3D Holds all 3D agents

Grid0D Holds nonspatial agents

PDEGrid2D facilitates modeling a single diffusible field in 2D

PDEGrid3D facilitates modeling a single diffusible field in 3D

3.2 Types of Agent

AgentSQ2Dunstackable “Square” agents in 2D. this agent type is bound to the Grid2D lattice, and only one AgentSQ2Dunstackable can occupy a given lattice position at a time.

AgentSQ2D “Square” agents in 2D. this agent type is bound to the Grid2D lattice, however multiple AgentSQ2Ds can occupy the same lattice position

AgentPT2D “Point” agents in 2D. this agent type is not bound to the Grid2D lattice, and is free to move continuously within the Grid, provided it does not try to cross the Grid boundaries.

AgentSQ3Dunstackable “Square” agents in 3D (cube?). this agent type is bound to the Grid3D lattice, and only one AgentSQ3Dunstackable can occupy a given lattice position at a time (gotta love copy paste)

AgentSQ3D “Square” agents in 3D. this agent type is bound to the Grid3D lattice, however multiple AgentSQ3Ds can occupy the same lattice position

AgentPT3D “Point” agents in 3D. this agent type is not bound to the Grid3D lattice, and is free to move continuously within the Grid, provided it does not try to cross the Grid boundaries.

3.3 Grid and Agent Class Definition

for demonstration we will use bits of code from the CompetitiveReleaseModel example. It is a fairly simple yet complete model, so it is a good place to begin learning the syntax of the Framework. Grid classes and Agent classes used in models will usually be created as extensions of the Grid and Agent classes shown above. This extension is done with the following syntax:

Project specific agent definition syntax:

```
1 public class ExModel extends Grid2D<ExCell> {
```

Grid class definition syntax:

```
1 class ExCell extends AgentSQ2Dunstackable<ExModel> {
```

Note the <> after the base class name, in java this is called a generic type argument. This is how we tell the Grid what kinds of Agent it will store, and how we tell the Agents what kind of Grid will store them. It is used by the ExModel and ExlCell to identify each other, so that their constituent functions can return the proper type, and access each other's variables and methods.

3.4 Grid Constructors

In order to create a class that extends any of the Grids, you must provide a constructor. let's look at part of the GOLGrid constructor as an example

```

1 GOLGrid(int x,int y,double livingProb,int runTicks,int
    refreshRateMS,GuiVis vis){
2 super(x,y, GOLAgent.class);

```

the first line declares the constructor and arguments, the second line calls super, which is required since our class extends a class with a constructor. Super is used to call the constructor of the base class. Into super we pass what the Grid2unstackable needs for initialization: an x and y dimension, which define the size of the grid, and GOLAgent.class, which is the class object of the GOLAgent. We pass the class object so that the GOLGrid can use it to create Agents for us, as described in the next section.

3.5 Agent Constructors

a word of warning: **DO NOT DEFINE A CONSTRUCTOR FOR YOUR AGENT CLASSES.**

The Grid that houses the agent will act as a “factory” for agents, and produce them with the NewAgent() function. This will return an agent that is either newly constructed, or an agent that has died and is being recycled. This returning of dead agents for reuse as new ones allows the model to run without tasking the garbage collector with removing all of the dead agents. This will increase the speed and decrease the memory footprint of your model. Instead of a constructor you should define some sort of initialization for your agents, which you do directly after the call to NewAgent. An example from CompRelModel.java:

```

1 CompRelCell seedCell = NewAgentSQ( cellIs [ i ] );
2 seedCell.isRes = rn.nextDouble() < resistantProb;

```

These lines of code come from the InitTumor function that the CompRelModel calls once at the beginning of a simulation. We call the NewAgentSQ() function, and pass in an index (“cellIs” is an array of starting indices to setup the tumor) that marks where to place the new agent. We then set the isRes property using a random number. since this property is the only information individual cells store in this model, setting it is all that is needed to initialize a new cell.

3.6 Grid Indexing

There are 3 different ways to describe or index locations on framework grids:

3.6.1 Single Indexing

since the x dimension, y dimension, and possibly the z dimension values are Grid constants, every square or voxel can be uniquely identified with a single integer index. Functions using this kind of indexing typically end with the SQ (abbreviating Square) phrase at the end of the function name. Single indexing is done with the following mappings:

In 2D: $i(x, y) = x * yDim + y$

In 3D: $i(x, y, z) = x * yDim * zDim + y * zDim + z$

the agents/values in the grids are stored as a single array, so single indexing is actually the most efficient as it requires no conversion.

3.6.2 Square Indexing

Similar to single indexing, square indexing uses a set of integers to refer to a specific square or voxel, as an (x,y) or an (x,y,z) set. Functions using this kind of indexing typically end with the SQ (abbreviating Square) phrase at the end of the function name.

3.6.3 Point Indexing

Uses a set of double values, to define continuous coordinates. Functions using this kind of indexing typically end with the PT (abbreviating Point) phrase at the end of the function name. The integer flooring of a coordinate set corresponds to the Square or Voxel that contains the point.

3.7 Typical Grid Loop

here we look at an example Loop or Run function. This is taken from the GOLGrid class, but all models will usually follow a similar pattern.

```
1 public void Run() {
2     for (int i = 0; i < runTicks; i++) {
3         for (GOLAgent a : this) {
4             a.Step();
5         };
6         IncTick();
7     }
8 }
```

The outer for loop counts the ticks, meaning that we will run for a total of runTicks steps. The inner for loop iterates over all the agents in the grid ("this" here is the grid that calls the Run function), and inside the loop we call that agent's Step function, which is defined in the GOLAgent class. After the inner

loop is finished we call `IncTick()` to increment the tick counter in the `GOLGrid` to the next timestep. If the user wants to shuffle and clean the list of agents, so that next time they will be iterated over in a different order, then they should call `CleanShuffInc()` instead of `IncTick()`.

3.8 Agent Age and the Tick System

Each Grid stores a timestep internally. the `GetTick()` function will return it. There is one essential assumption that the Framework makes about agents and ticks that should not be ignored: when iterating over the agents in the for loop construction as demonstrated in section 3.7, the Grid will automatically skip any agents whose age is 0 (aka those that were just “born” this timestep). This means that the `IncTick` function is essential in order to advance the ages of all agents and make those that were just “born” loop-accessible. Bear in mind also that although just “born” agents are not accessible via for loop, they still do exist on the grid, and can be accessed and manipulated from the grid even before the tick is incremented.

This prevents agents that were just born from acting immediately, which can cause problems such as runaway division. (Fellow nerds: think summoning sickness from Magic The Gathering)

3.9 Agent Functions

here we describe the builtin functions that the Agents expose to the user. (3D adds a `z`)

G(): returns the grid that the agent belongs to

Age(): returns the age of the agent, in ticks

BirthTick(): returns the tick at which the agent was born

Alive(): returns whether or not the agent currently exists on the grid

Isq(): returns the index of the square that the agent is currently on

Xsq(),Ysq(): returns the X or Y indices of the square that the agent is currently on.

Xpt(),Ypt(): returns the X or Y coordinates of the agent. If the Agent is on-lattice, these functions will return the coordinates of the middle of the square that the agent is on.

MoveSQ(x,y),MoveSQ(i): moves the middle of the square at the indices/index specified

MovePT(x,y): moves the coordinates specified

MoveSafeSQ(x,y,wrapX,wrapY),MoveSafePT(x,y,wrapX,wrapY): Similar to the move functions, only it will automatically either apply wraparound, or prevent moving along a particular axis if movement would cause the agent to go out of bounds.

Dispose(): removes the agent from the grid

SwapPoosition(otherAgent): swaps the positions of two agents. useful mostly for the AgentSQ2unstackable and AgentSQ3unstackable classes, which don't allow stacking of agents, making this maneuver otherwise difficult.

HoodToIs(int[]coords,int[]ret): This function takes a set of coordinates (passed in as a contiguous set of xy or xyz pairs) centered around the origin, translates the set of coordinates to be centered around the agent, and computes which indices the translated coordinates map to. The function returns the number of valid locations it set, which if wraparound is disabled may be less than the full set of coordinate pairs. this means that the passed in ret list will not be completely filled with valid entries. See the CellStep function in the Complete Model example for more information.

HoodToEmptyIs(int[]coords,int[]ret),HoodToOccupiedIs(int[]coords,int[]ret,centerX,centerY): These functions are the same in function as the HoodToIs function, but in addition to locations that are outside the grid domain, they will locations that are either empty or occupied with at least one agent, depending on which of the two functions is called.

3.10 Universal Grid Functions and Properties

these functions and properties are shared by all of the different types of grids (3D adds a z):

xDim,yDim: the x or y dimension of the Grid

length: the total number of squares in the grid, equivalent to xDim*yDim

I(x,y): converts a set of coordinates to the index of the square at those coordinates.

ItoX(i),ItoY(i): converts an index of a square to that square's X or Y coordinate

WrapI(x,y): returns the index of the square at the provided x,y coordinates, with wraparound.

In(x,y): returns whether the x and y coordinates provided are inside the Grid.

GetTick(): returns the current tick

IncTick(): increments the current tick by 1.

DistSquared(x1,y1,x2,y2,wrapX,wrapY): gets the distance squared between two positions with or without grid wrap around (if wraparound is enabled, the shortest distance taking this into account will be returned)

ChangeGridsSQ(outsideAgent,x,y),ChangeGridsPT(outsideAgent,x,y): removes the agent from whichever grid it is currently living on, and moves it to this grid. The age of the agent is also reset so that the agent appears to be just born this timestep.

HoodToIs(int[]coords,int[]ret,centerX,centerY): This function takes a set of coordinates (passed in as a contiguous set of xy or xyz pairs) centered around the origin, translates the set of coordinates to be centered around a particular central location, and computes which indices the translated coordinates map to. The function returns the number of valid locations it set, which if wraparound is disabled may be less than the full set of coordinate pairs. this means that the passed in ret list will not be completely filled with valid entries. See the InitTumor function in the Complete Model example for more information.

CoordsToIs(int[]coords,int[]ret): similar to HoodToIs, but without the translation of coordinates to a new central location.

3.11 Agent Grid Method Descriptions

here we describe the builtin functions that the agent containing Grids expose to the user. Most functions that take x,y arguments can alternatively take a single index argument. The Grid will use the index to find the appropriate square. (3D adds a z)

NewAgentSQ(x,y),NewAgentSQ(i): returns a new agent, which will be placed at the center of the indicated square. x,y, and i are assumed to be integers

NewAgentPT(x,y): returns a new agent, which will be placed at the coordinates specified. x and y are assumed to be doubles

GetAgent(x,y)GetAgent(i): Gets a single agent at the specified grid square, beware using this function with stackable agents, as it will only return one of the stack of agents.

GetAgents(ArrayList<AgentType>,x,y): adds to the provided arraylist all of the agents on the specified square, only useful with the stackable agent types.

GetPop(): returns the number of agents that are alive on the entire grid.

CleanAgents(): reorders the list of agents so that dead agents will no longer have to be iterated over. don't call this during the middle of iteration!

ShuffleAgents(): shuffles the list of agents, so that they will no longer be iterated over in the same order. don't call this during the middle of iteration!

IncTick(): increments the Grid tick counter by 1, don't call this during the middle of iteration!

CleanShuffInc(): Cleans the agent list, shuffles it, and increments the tick counter. This function is often called at the end of a timestep. don't call this during the middle of iteration! **Reset():** disposes all agents, and resets the tick counter.

HoodToEmptyIs(int[]coords,int[]ret,centerX,centerY) similar to the HoodToIs function, but will only include indices of locations that are empty

HoodToOccupiedIs(int[]coords,int[]ret,centerX,centerY): similar to the HoodToIs function, but will only include indices of locations that are occupied

RandomAgent(rand): returns a random living agent, regardless of whether the agent was born this timestep

3.12 PDEGrid Method Descriptions

The PDEGrid classes do not store agents. They are meant to be used to model Diffusible substances. PDEGrids contain two fields (arrays) of double values: a current field, and a swap field. The intended usage of these fields is that the values from the current timestep are stored in the current field while the swap field is used as a location to write the result of diffusion. The Swap() function exchanges the identities of these fields, so that the current field becomes the swap field and vice versa. Usually this will be called automatically at the end of a diffusion computation. The functions included in the PDEGrid class are the following: (3D adds a z)

GetField(): returns the "current field" double array which is usually used as the main field that agents interact with

GetSwapField(): returns the "swap field" double array which is usually used as scratch to write the result of diffusion.

Get(i),Get(x,y),Set(i),Set(x,y,val),Add(i),Add(x,y,val),Mul(i,val),Mul(x,y,val): gets, sets, adds, or multiplies with a single value in the current field at the specified coordinates

GetSwap(i),GetSwap(x,y),SetSwap(i),SetSwap(x,y,val),MulSwap(i,val),MulSwap(x,y,val): gets, sets, adds, or multiplies a single value in the swap field at the specified coordinates

AddSwap(i),AddSwap(x,y,val),MulSwap(i,val),MulSwap(x,y,val): gets, sets, adds, or multiplies with the swap field value at the specified coordinates

SetAll(val),SetAll(val[]),AddAll(val)MulAll(val): applies the Set/Get/Mul operations to all entries of the current field

SetAllSwap(val),SetAllSwap(val[]),AddAllSwap(val)MulAllSwap(val): applies the Set/Get/Mul operations to all entries of the swap field

MaxDif(): returns the maximum difference in any single lattice position between the current field and the swap field. if the boolean scaled argument is set to true, then the difference will be scaled by the current value.

BoundAll(min,max)BoundAllSwap(min,max): sets all values in the field so that they are between min and max

Swap(): swaps the identities of the current and swap fields

SwapInc(): swaps and also increments the tick

Diffusion(diffRate,wrapX,wrapY): runs diffusion on the current field, putting the result into the swap field, then swaps their identities, so that the now current field stores the result of diffusion. This form of the function assumes either a reflective or wrapping boundary. Note that if the diffusion rate exceeds 0.25, this diffusion method will become numerically unstable.

Diffusion(diffRate,boundaryValue,wrapX,wrapY): has the same effect as the diffusion function without the boundary value argument, except that at the boundaries the function assumes either a constant value (which is the boundary value) or a wrapping boundary.

DiffusionADI(diffRate): runs diffusion on the current field using the ADI (alternating direction implicit) method. ADI is numerically stable at any diffusion rate. However at high rates it may produce artifacts.

DiffusionADI(diffRate,boundaryValue): runs diffusion on the current field using the ADI (alternating direction implicit) method. ADI is numerically stable at any diffusion rate. However at high rates it may produce artifacts. adding a boundary value to the function call will cause boundary conditions to be imposed.

Advection(xVel,yVel): runs advection, which shifts the concentrations using a constant flow with the x and y velocities passed. this signature of the function assumes wrap-around, so there can be no net flux of concentrations.

Advection(xVel,yVel,boundaryValue): runs advection as described above with a boundary value, meaning that the boundary value will advect in from the upwind direction, and the concentration will disappear in the downwind direction.

GradientX(x,y),GradientY(x,y): returns the gradient of the diffusible along the specified axis

3.13 Griddouble/Gridint/Gridlong Method Descriptions

As an alternative to this class, it may be useful to simply employ a double/int/-long array whose length is equal to the length of the other associated grids. The I() function of any associated grids can be used to access values in the double array with x,y or x,y,z coordinates.

GetField(): returns the field double array that the grid stores

Get(i),Get(x,y),Set(i),Set(x,y,val),Add(i),Add(x,y,val),Mul(i,val),Mul(x,y,val): gets, sets, adds, or multiplies with a single value in the field at the specified coordinates

SetAll(val),SetAll(val[]),AddAll(val),MulAll(val): applies the Set/Get/Mul operations to all entries of the field

BoundAll(min,max)BoundAllSwap(min,max): sets all values in the field so that they are between min and max

GradientX(x,y),GradientY(x,y): returns the gradient of the diffusible along the specified axis

4 Util.java

The Util class is one of the most ubiquitous classes in the framework, and contains all of the generic functions that wouldn't make sense to add to any particular object in the framework.

The list of utilities functions in the framework has only grown with time. the list presented here is not exhaustive, so I recommend looking at the file itself if you feel something is missing, on top of that feel free to ask for a new feature or better yet send me an implementation and I'll most likely add it to the repository for everyone to share.

4.1 Util Array Functions

A set of utilities for making array manipulation easier.

ArrToString(arr,delim): useful for collecting data or print statement debugging, returns the contents of an array as a single string. entries are separated by the delimiter argument

IndicesArray(numEntries): generates an array of ascending indices starting with 0, up to nEntries.

Mean(arr): returns the mean of an array

Sum(arr): returns the sum of an array

Norm(arr): returns the euclidean norm of an array

NormSq(arr): returns the squared euclidian norm of an array, somewhat more efficient than Norm

SumTo1(arr): scales all entries in an array so that their sum is 1.

Normalize(arr): scales all entries in an array so that their norm is 1.

4.2 Util Neighborhood Functions

a set of utilities for generating neighborhood arrays. neighborhood arrays are lists of x,y index pairs, of the form $[x1, y1, x2, y2, x3, y3...]$ in the third dimension these are $[x1, y1, z1, x2, y2, z2...]$ these arrays are useful when finding the indices of the locations that make up the neighborhood around an agent when using Grid/Agent functions such as HoodToIs.

These functions should not be called over and over, as this would wastefully create arrays over and over. instead the function should be called once and stored by the grid.

VonNeumannHood(origin?), MooreHood(origin?): returns an array that contains the coordinates of the VonNeumann or Moore Neighborhood in 2D. the boolean argument specifies whether the center of these neighborhood (0,0) should be included as part of the set of coordinates.

VonNeumannHood3D(origin?), MooreHood3D(origin?): returns an array that contains the coordinates of the VonNeumann or Moore Neighborhood in 3D. the boolean argument specifies whether the center of these neighborhood (0,0) should be included as part of the set of coordinates.

HexHoodEvenY(origin?), HexHoodOddY(origin?): to simulate a hex lattice on a Grid2D, the neighborhood used for a given agent changes depending on the position of the agent. The hex hood changes depending on whether the agent is on an even or odd Y position. These functions return an array that contains the proper coordinates for a given case.

TriangleHoodSameParity(origin?), TriangleHoodDifParity(origin?): to simulate a triangle lattice on a Grid2D, the neighborhood used for a given agent changes depending on the position of the agent. The Triangle hood changes depending on whether the parity ("even-ness") of the agents X and Y position match. These functions return an array that contains the proper coordinates for a given case.

CircleHood(origin?, radius): generates a neighborhood of all squares within the radius of a starting position at the middle of the (0,0) origin square. the boolean argument specifies whether (0,0) should be included in the return array.

RectangleHood(origin?, radX, radY): generates a neighborhood of all squares whose x displacement is within radX, and whose y displacement is within radY of a starting position at the middle of the (0,0) origin square. the

boolean argument specifies whether (0,0) should be included in the return array.

AlongLineCoords(x1,y1,x2,y2): returns an array of all squares that touch a line between the two starting positions.

4.3 Util Math Functions

For all of the following functions, and throughout the framework, `rn` refers to a random number generator.

InfiniteLinesIntersection2D(x1,y1,x2,y2,x3,y3,x4,y4,double[]ret): computes the intersection of lines between points 1 and 2, and points 3 and 4. puts the coordinates of the intersection point in `ret`.

Bound(val,min,max): returns the value bounded by the min and max.

Rescale(val,min,max): assumes the starting value is in the 0-1 scale, and rescales it to be in the min-max scale.

ModWrap(val,max): wraps the value provided so that it must be between 0 and max. used to implement wraparound by the framework.

ProtonsToPh(protonConc): converts proton concentration to `ph`

PhToProtons(ph): converts `ph` to proton concentration

ProbScale(prob,duration): converts the probability that an event happens in unit time to the probability that that same event happens in duration time.

Sigmoid(val,stretch,inflectionValue,minCap,maxCap): `val` is the value that the sigmoid function is being applied to, the `stretch` argument stretches or shrinks the sigmoid function along the x axis, the `inflectionValue` governs where the inflection point of the sigmoid is, `minCap` and `maxCap` bound the sigmoid along the y axis.

4.4 Util Misc Functions

TimeStamp(): returns a timestamp string with format "YYYY_MM_DD_HH_MM_SS"

PWD(): returns the current working directory as a string

MemoryUsageStr(): returns a string with information about the current memory usage of the program.

QuickSort(sortMe,greatestToLeast): requires the passed `sortMe` class to implement the `Sortable` interface. the passed boolean indicates whether the array should be sorted from greatest to least or least to greatest.

4.5 Util Color Functions

These functions generate integers that store RGBA (Red,Green,Blue,Alpha) color channels internally, 8 bits per channel (integer values 0-255). These so-called “ColorInts” are intended to be used as arguments for the GridVis, GridWindow, Vis2DOpenGL, and Vis3DOpenGL to set the color of the pixels or objects being displayed. the RGB components set the color, and the alpha component adds transparency.

All of these functions (except the “Getters”) return a new ColorInt

RGB(r,g,b): sets the rgb color channels using the continuous 0-1 range mapping. the alpha is always set to 1.

RGBA(r,g,b,a): sets the rgb and alpha channels using the continuous 0-1 range mapping.

RGB256(r,g,b): sets the rgb color channels using the discrete 0-255 range mapping. the alpha is always set to 1.

RGBA256(r,g,b,a): sets the rgb and alpha channels using the 0-255 range mapping.

GetRed(color),GetBlue(color),GetGreen(color),GetAlpha(color): returns the value of a single channel using the continuous 0-1 range mapping

GetRed256(color),GetBlue256(color),GetGreen256(color),GetAlpha256(color): returns the value of a single channel using the discrete 0-255 range mapping

SetRed(color),SetGreen(color),SetBlue(color),SetAlpha(color): returns a new ColorInt with the one of its channels changed compared to the argument passed in, uses the continuous 0-1 range mapping

SetRed256(color),SetGreen256(color),SetBlue256(color),SetAlpha256(color): returns a new ColorInt with the one of its channels changed compared to the argument passed in, uses the discrete 0-255 range mapping

CategoricalColor(index): returns a categorical color from a nice mutually distinct set. Valid indices are 0-19. the color order is (blue,red,green,yellow,purple,orange,cyan,pink,brown,light blue,light red,light green,light yellow,light purple, light orange, light cyan,light pink, light brown, light gray, dark gray

HeatMapRGB(val),HeatMap???(val): returns a new colorInt using the heatmap color scale. values are distinguished in the 0-1 range. the heatmap color scale is black at 0, white at 1, and transitions between these by changing one color channel at a time from none to full. the order that the channels are changed is dictated by the order of the 3 letters at the end of the function name. the possible orders are RGB, RBG, GRB, GBR, GRB and GBR

HeatMapRGB(val,min,max),HeatMap???(val): same as the above, but works to distinguish values in the min-max range.

HSBColor(hue,saturation,brightness): returns a new colorInt using the HSB colorspace. values are distinguished in the continuous 0-1 range.

YCbCrColor(y,cb,cr): returns a new colorInt using the YCbCr colorspace. values are distinguished in the continuous 0-1 range.

CbCrPlaneColor(x,y): returns a new colorInt using the CbCr plane at Y=0.5. a very nice colormap for distinguishing position in a 2 dimensional space. x,y are expected to be in the continuous 0-1 range.

4.6 Util MultiThread Function

Multithread(nRuns,nThreads,RunFun): A function so useful that it deserves its own section, the multithread function creates a thread pool and launches a total of nRun threads, with nThreads running simultaneously at a time. the RunFun that is passed in must be a void function that takes an integer argument. when the function is called, this integer will be the index of that particular run in the lineup. This can be used to assign the result of many runs to a single array, for example, if the array is written to once by each RunFun at its run index. If you want to run many simulations simultaneously, this function is for you.

4.7 Util Save and Load Functions

NOTE:SerializableModel Interface In order to save and load models, you must first have the model extend the SerializableModel interface (this interface is defined in the Interfaces folder). this interface has one method, called SetupConstructors. all you have to do in this method is call the AgentGrid function _PassAgentConstructor(AGENT.cass) once for each AgentGrid in your model, where AGENT is the type of agent that the AgentGrid holds. see LEARN_HERE/Agents/SaveLoadModel for an example.

SaveState(model): Saves a model state to a byte array and returns it. The model must implement the SerializableModel interface

SaveState(model,fileName): Saves a model state to a file with the name specified. creates a new file or overwrites one if the file already exists. The model must implement the SerializableModel interface

LoadState(byte[]state): Loads a model form a byte array created with SaveState. The model must implement the SerializableModel interface

LoadState(fileName): Loads a model from a file array created with SaveState. The model must implement the SerializableModel interface

5 Rand.java

Int(bound): generates an integer in the uniformly distributed range 0 up to but not including the bound value.

Double(): generates a double in the range 0 to 1

Double(bound): generates a double in the uniformly distributed range 0 up to the bound value.

Long(bound): generates a long in the uniformly distributed range 0 up to but not including the bound value.

Bool(): generates a random boolean value, with equal probability of true and false.

Binomial(n,p): samples the binomial distribution, returns the number of heads with n weighted coin flips and probability p of heads

Multinomial(double[]probs,n,Binomial,int[]ret,rn): fills the return array with the number of occurrences of each event. n is the total number of occurrences to bin. Binomial is a class in the Tools folder.

Shuffle(arr,sampleSize,numberOfShuffles,rn): shuffles an array, sampleSize is how much of the complete array should be involved in shuffling, and numberOfShuffles is the number of entries of the array that will be shuffled. the shuffling results will always start from the beginning of the array up to numberOfShuffles.

Gaussian(mean,stdDev,rn): samples a gaussian with the mean and standard deviation given.

RandomVariable(double[]probs,rn): samples the distribution of probabilities (which should sum to 1, the SumTo1 function comes in handy here) and returns the index of the probability bin that was randomly chosen.

RandomPointOnSphereEdge(radius,double[]ret,rn): writes into ret the coordinates of a random point on a sphere with given radius centered on (0,0,0)

RandomPointInSphere(radius,double[]ret,rn): writes into ret the coordinates of a random point of inside a sphere with given radius centered on (0,0,0)

RandomPointOnCircleEdge(radius,double[]ret,rn): writes into ret the coordinates of a random point on the edge of a circle with given radius centered on (0,0)

RandomPointInCircle(radius,double[]ret,rn): writes into ret the coordinates of a random point on the edge of a circle with given radius centered on (0,0)

6 Gui

What fun is a model without being able to see and play with it in real time? The Gui classes allow you to easily do this and works on top of the Java Swing gui system. (except the Vis2DOpenGL and Vis3DOpenGL, which are built on lwjgl)

6.1 Types of Gui and Method Descriptions

Here we list the different guis that are provided by the framework and provide a summary of their functions

for a more complex look at how to use the gui system, check out the CSCCA and Polyp3D example models in the ManualModels folder.

6.1.1 GridWindow

The simplest built-in Gui, it is nothing more than a GridVis imbedded in a GuiWindow. Recommended for first-time users.

GridWindow(title,xDim,yDim,scaleFactor,main?,active?): sets up a GridWindow, the pixel dimensions of the created window will be: $Width = xDim * scaleFactor$ and $Height = yDim * scaleFactor$ the main boolean specifies whether the program should exit when the window is closed. the active boolean allows easily toggling the objects on and off.

SetPix(x,y,color),SetPix(i,color): sets an individual pixel on the GridWindow. in the visualization the pixel will take up $scaleFactor * scaleFactor$ screen pixels.

SetPix(x,y,ColorIntGenerator),SetPix(i,ColorIntGenerator): same functionality as SetPix with a color argument, but instead takes a ColorIntGenerator function (a function that takes no arguments and returns an int). the reason to use this method is that when the gui is inactivated the ColorIntGenerator function will not be called, which saves the computation time of generating the color.

GetPix(x,y),GetPix(i): returns the pixel color at that location as a colorInt

Clear(color): sets all pixels to a single color.

Dispose(): disposes of the GridWindow.

TickPause(milliseconds): call this once per step of your model, and the function will ensure that your model runs at the rate provided in milliseconds. the function will take the amount time between calls into account to ensure a consistent tick rate.

PlotSegment(x1,y1,x2,y2,color,scaleX,scaleY): plots a line segment, connecting all pixels between the points defined by $(x1, y1)$ and $(x2, y2)$ with the provided color. If you are using this function on a per-timestep basis, I recommend setting individual pixels with SetPix, as it is more performant. the scaling variables adjust the spatial scale of the points.

PlotLine(double[]xs,double[]ys,color,startPoint,endPoint,scaleX,scaleY): plots a line by drawing segments between consecutive points. point i is defined by $(xs[i], ys[i])$. points are drawn starting at index startPoint, and ending at index endPoint. the scaling variables adjust the spatial scale of the points.

PlotLine(double[]xys,color,startPoint,endPoint,scaleX,scaleY): plots a line by drawing segments between consecutive points. points are expected in the coords format $(x1, y1, x2, y2, x3, y3...)$. point i is defined by $(xys[i*2], xys[i*2+1])$. points are drawn starting at index startPoint, and ending at index endPoint. the scaling variables adjust the spatial scale of the points.

6.1.2 GuiWindow

a container for Gui Components, which will be detailed below. the most supported of the gui types.

GuiWindow(title,main?,CloseAction,active?): the title string is displayed in the top bar, the main boolean specifies whether the program should exit when the window is closed. the active boolean allows easily toggling the objects on and off.

AddCol(column,component): components are added to the gui from top to bottom in columns. when the window is displayed, the rows and columns expand to fit the largest element in each.

RunGui(): once all components have been added, the RunGui function runs the gui and displays it to the screen.

GetBool(label): attempts to pull a boolean from the Param with the corresponding label, works with the BoolParam

GetInt(label): attempts to pull an integer from the Param with the corresponding label, works with the IntParam and ComboBoxParam (returns the index of the chosen option)

GetDouble(label): attempts to pull a double from the Param with the corresponding label, works with the IntParam and DoubleParam

GetString(label): attempts to pull a string from the Param with the corresponding label, works with all Param types

Dispose(): disposes of the GridWindow.

TickPause(millis): call this once per step of your model, and the function will ensure that your model runs at the rate provided in millis. the function will take the amount time between calls into account to ensure a consistent tick rate.

6.1.3 Vis2DOpenGL

A window for visualizing 2D models, especially off lattice ones. I usually recommend using a GridVis instead for 2D models.

Vis2DOpenGL(xPix,yPix,xDim,yDim,title,active?): creates a Vis2DOpenGL window. The dimensions on screen are xPix by yPix. the xDim and yDim dimensions should match the model being drawn. the title string is displayed on the top of the window, the active boolean allows easily disabling the Vis2DOpenGL

Show(): push the OpenGL display to the main gui

CheckClosed(): returns true if the close button has been clicked in the Gui

Dispose(): closes the Vis2DOpenGL. happens automatically when the main function finishes.

Circle(x,y,z,radius,color): Draws a circle. Currently this is the only builtin draw functionality along with the FanShape function from which it is derived.

Line(x1,y1,x2,y2,color): Draws a line between 2 points

LineStrip(double[]xs,double[]ys,color): draws a set of connected line segments, xs and ys are expected to be the same length.

LineStrip(double[]coords,color): draws a set of connected line segments, coords is expected to consist of [x1,y1,x2,y2...] pairs of point coordinates.

TickPause(millis): call this once per step of your model, and the function will ensure that your model runs at the rate provided in millis. the function will take the amount time between calls into account to ensure a consistent tick rate.

6.1.4 Vis3DOpenGL

A window for visualizing 3D models.

Vis3DOpenGL(xPix,yPix,xDim,yDim,title,active?): creates a Vis3DOpenGL window. the dimensions on screen are xPix by yPix. the xDim, yDim, and zDim dimensions should match the model being drawn. the title string is displayed on the top of the window, the active boolean allows easily disabling the Vis3DOpenGL

CONTROLS: to move around inside an active Vis3DOpenGL window, click on the window, after which the following controls apply:

- **Esc Key:** Get the mouse back
- **Mouse Move:** Change look direction
- **W Key:** Move forward
- **S Key:** Move backward
- **D Key:** Move right
- **A Key:** Move left
- **Shift Key:** Move up
- **Space Key:** Move down
- **Q Key:** Temporarily increase move speed
- **E Key:** Temporarily decrease move speed

Clear(color): usually called before anything else: clears the gui

Show(): push the OpenGL display to the main gui

CheckClosed(): returns true if the close button has been clicked in the Gui

Dispose(): closes the Vis3DOpenGL. happens automatically when the main function finishes.

Circle(x,y,z,radius,color): Draws a circle. Currently this is the only builtin draw functionality along with the FanShape function from which it is derived.

CelSphere(x,y,z,radius,color): Draws a cool looking cel-shaded sphere, really several Circle function calls in a row internally.

TickPause(millis): call this once per step of your model, and the function will ensure that your model runs at the rate provided in millis. the function will take the amount time between calls into account to ensure a consistent tick rate.

Line(x1,y1,z1,x2,y2,z2,color): Draws a line between 2 points

LineStrip(double[]xs,double[]ys,double[]zs,color): draws a set of connected line segments, xs and ys are expected to be the same length.

LineStrip(double[]coords,color): draws a set of connected line segments, coords is expected to consist of [x1,y1,z1,x2,y2,z2...] triplets of point coordinates.

6.2 Types of GuiComponent and Method Descriptions

6.2.1 GridVis

a grid of pixels that are each set individually. very fast and useful for displaying the contents of grids

GridVis(gridW,gridH,scaleFactor,active?): sets up a GridVis, the pixel dimensions of the created area will be: $Width = xDim * scaleFactor$ and $Height = yDim * scaleFactor$. The active boolean allows easily toggling the objects on and off. The functions that the GridVis can execute are included above

6.2.2 GuiLabel

a label that displays text on the Gui and can be continuously updated. the GuiLabel's on-screen size will remain fixed at whatever size is needed to render the string first passed to it.

6.2.3 GuiButton

a button that when clicked triggers an interrupting function

6.2.4 BoolParam

a button that can be set and unset, must be labeled. use the GuiWindow Param functions to interact

6.2.5 IntParam

an input line that expects an integer, must be labeled. use the GuiWindow Param functions to interact

6.2.6 DoubleParam

an input line that expects a double, must be labeled. use the GuiWindow Param functions to interact

6.2.7 StringParam

an input line that takes any string, must be labeled. use the GuiWindow Param functions to interact

6.2.8 ComboBoxParam

a dropdown menu of text options, must be labeled. use the GuiWindow Param functions to interact

6.2.9 FileChooserParam

a button that when clicked triggers a gui that facilitates choosing an existing file or creating one, must be labeled. use the GuiWindow Param functions to interact

7 Tools

7.1 Tools/ FileIO

An essential piece of the framework, the FileIO class facilitates easily writing to and reading from files. this is important for collecting data from your models as well as systematically paramaterizing them. the API for the FileIO object is discussed.

FileIO(filename,mode): the FileIO constructor expects a filename or path as a string, and a mode string, of which there are 6 options:

- “**r**” creates a FileIO in read mode, this FileIO is able to read text files
- “**w**” creates a FileIO in write mode, this FileIO is able to write to a new text file
- “**a**” creates a FileIO in append mode, this FileIO is able to append to an existing text file or write to a new file.
- “**rb**” creates a FileIO in read binary mode, this FileIO is able to read binary files
- “**wb**” creates a FileIO in write binary mode, this FileIO is able to write to binary files.
- “**ab**” creates a FileIO in append binary mode, this FileIO is able to append to an existing binary file or write to a new file.

The functions in the next sections are split up based on which mode was used to open the FileIO

Close(): make sure to call this function when finished with the FileIO. the FileIO uses buffers internally to make writing more efficient. without calling close the buffers may never be fully written out.

Read Mode Functions

Read(): returns an arraylist of Strings. each string is one line from the file

ReadLine(): returns the next line from the file as a string

ReadLineDelimit(delimiter),ReadLineIntDelimit(delimiter),etc: returns an array of strings,ints,or doubles, etc. each entry is parsed using the delimiter.

ReadDelimit(delimiter),ReadIntDelimit(delimiter),etc: returns an arraylist of arrays of strings,ints,or doubles, etc. each entry is parsed using the delimiter. each array is a line.

Write Mode Functions

Write(string): writes the string argument to a file

WriteDelimit(arr,delimiter): writes the contents of the provided array to a file, entries are separated using the delimiter.

ReadBinary Mode Functions

ReadBinBool(bool),ReadBinInt(int),ReadBinDouble(double),etc: read the next single value from the binary file.

ReadBinBools(bool[]),ReadBinInts(int[]),ReadBinDoubles(double[]),etc: fills the array argument with values read from the binary file.

WriteBinary Mode Functions

WriteBinBool(bool),WriteBinInt(int),WriteBinDouble(double),etc: writes a single value to the binary file.

WriteBinBools(bool[]),WriteBinInts(int[]),WriteBinDouble(double[]),etc: writes every entry in the array to the binary file.

7.2 Tools/ SerializableModel

This is an incredibly useful interface that allows you to easily save and load entire model states! this is done by having the model class file implement SerializableModel. the only function that must be setup by default is the SetupConstructors function, in which you should go to all of your AgentGrids and call _SetupAgentListConstructor(class) with the class object that that AgentGrid holds.

8 Example: Competitive Release Model

As an example of what the framework is capable of, we now look at a simple but complete example of a hybrid model. We will be designing a model of adaptive therapy based on a publication from our lab [1]. We also showcase the flexibility that the modular component approach brings by displaying 3 different parameterizations of the same model side by side in a “multiwell experiment”.

8.1 Competitive Release Introduction

As in [1], the presented model assumes two competing tumor-cell phenotypes: a rapidly dividing, drug-sensitive phenotype and a slower dividing, drug-resistant phenotype. There is also a drug diffusible that enters into the system through the model edges and is consumed over time by the tumor cells.

Every timestep, each cell has a probability of death and a probability of division. The division probability is affected by phenotype and the availability of space. The death probability is affected by phenotype and the local drug concentration.

An interesting outcome of the model is that pulsed therapy is better at managing the tumor than either constant therapy or no treatment. Under pulsed therapy the sensitive population is kept in check, while still competing spatially with the resistant phenotype and preventing its expansion. The rest of the section goes into detail on how this abstract model is generated.

The modular design of the framework allows us to test 3 different treatment conditions, each with an identical starting tumor (No drug, constant drug, and pulsed drug)

Figure 1 provides a high level look at the structure of the code and should help significantly with understanding how the code pieces fits together. Table 1 provides a quick reference for understanding the functions in the codebase, as well as to which objects these functions belong. Any functions that are used by the example but do not exist in the table are defined within the example itself and explained in detail below the code. Those fluent in Java may be able to understand the example just by reading the code and using the reference table.

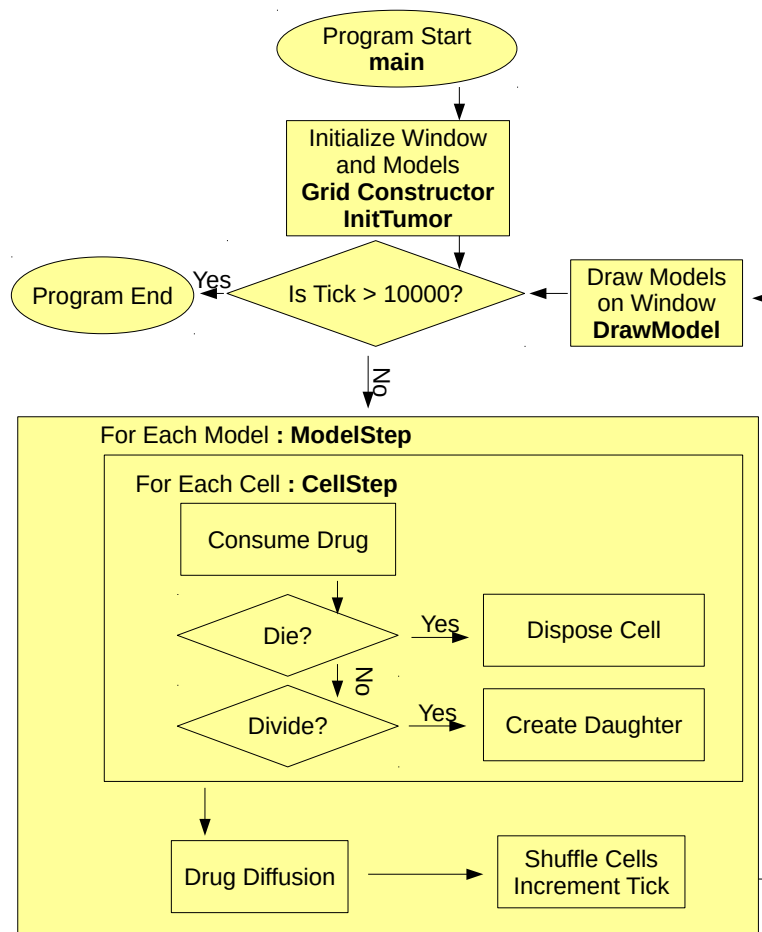


Figure 1: Example program code flowchart. the names of the relevant functions are mentioned in bold where they are first called.

Object	Name	Action
AgentGrid2D	HoodToEmptyIs(NEIGHBORHOOD, INDICES, X, Y)	Finds unoccupied squares in a NEIGHBORHOOD (a set of coordinates of the form [x1,y1,x2,y2,...]) centered on X, Y. Puts these in the INDICES array and returns the number of indices
AgentGrid2D	NewAgentSQ(INDEX)	returns a new agent, placed at the center of the of the square at the provided INDEX.
AgentGrid2D	CleanShuffInc(RANDOM_GENERATOR)	usually called after every timestep to: 1) remove dead agents. 2) shuffle order of agents so next iteration is over a new random order. 3) increment the grid timestep
AgentGrid2D	GetTick()	returns the current grid timestep
AgentGrid2D	ItoX(INDEX), ItoY(INDEX)	these functions convert from a grid position INDEX to the x and y components that point to the same grid position.
AgentSQ2D	G()	Gets the Grid that the agent lives on
AgentSQ2D	Isq()	Gets the index of the grid square that the agent occupies
AgentSQ2D	HoodToEmptyIs(NEIGHBORHOOD, INDICES)	Finds unoccupied squares in a NEIGHBORHOOD centered on the agent. Puts these in the INDICES array and returns the number of indices
AgentSQ2D	Dispose()	removes the agent from the grid and from iteration
PDEGrid2D	Get(INDEX)	Returns the concentration of the PDE field at the given index.
PDEGrid2D	Mul(INDEX, VALUE)	Multiplies the concentration at the given INDEX by VALUE.
PDEGrid2D	DiffusionADI(RATE)	applies diffusion using the ADI method with the rate constant provided. a reflective boundary is assumed.
PDEGrid2D	DiffusionADI(RATE, BOUNDARY_COND)	applies diffusion using the ADI method with the RATE constant provided. the BOUNDARY_COND value diffuses from the grid borders.
GridWindow	SetPix(INDEX, COLOR)	sets the color of an individual pixel on the visualization window
GridWindow	TickPause(MILLISECONDS)	pauses the program between calls to TickPause. The function automatically subtracts the time between calls from the pause time to ensure a consistent framerate.
GridWindow	ToPNG(FILENAME)	writes out the current state of the GuiWindow to a PNG image file
GridWindow	Dispose()	closes the GridWindow
Utils	RGB(RED, GREEN, BLUE)	returns an integer with the requested color in argb format. This value can be used for visualization.
Utils	HeatMapRGB(VALUE)	maps the value argument (which is assumed to be between 0 and 1) to a color in the heat colormap
Utils	CircleHood(INCLUDE_ORIGIN, RADIUS)	returns an integer array of coordinate pairs. These coordinate pairs define a neighborhood of all squares whose centers are within the radius distance of the center (0,0) origin square. The includeOrigin argument specifies that we intend to include the origin in this set of coordinates.
Utils	MooreHood(INCLUDE_ORIGIN)	returns an integer array of coordinate pairs. These coordinate pairs define a moore neighborhood around the (0,0) origin square. The INCLUDE_ORIGIN boolean specifies whether we intend to include the origin in this set of coordinates.
FileIO	Write(STRING)	Writes the STRING to the output file
FileIO	Close()	Closes the output file, also ensures that all written data is recorded.

Table 1: This table contains a summary of the functions available in the code.

8.2 Main Function

We first look at the main function for a bird's-eye view of how the program is structured.

```
1 public static void main(String[] args) {
2     int x = 100, y = 100, visScale = 2, tumorRad = 10, msPause = 0;
3     double resistantProp = 0.5;
4     GridWindow win = new GridWindow("Competitive Release", x*3, y,
5         visScale);
6     ExampleModel[] models = new ExampleModel[3];
7     FileIO popsOut=new FileIO("populations.csv", "w");
8     for (int i = 0; i < models.length; i++) {
9         models[i]=new ExampleModel(x,y,new Rand(1));
10        models[i].InitTumor(tumorRad, resistantProp);
11    }
12    models[0].DRUG_DURATION = 0; //no drug
13    models[1].DRUG_DURATION = models[1].DRUG_PERIOD; //constant drug
14    //Main run loop
15    while (models[0].GetTick() <= 10000) {
16        win.TickPause(msPause);
17        for (int i = 0; i < models.length; i++) {
18            models[i].ModelStep();
19            models[i].DrawModel(win, i);
20        }
21        popsOut.Write(models[0].GetPop()+"", "+models[1].GetPop()+"", "+models[2].GetPop()+"\n");
22        if ((models[0].GetTick()-1)%100==0) {
23            win.ToPNG("ModelsTick" +
24                (models[0].GetTick()-1)+" .png");
25        }
26    }
27    popsOut.Close();
28    win.Dispose();
29 }
```

2&3: we define all of the constants that will be needed to setup the model and display

4: we create a GridWindow which we will use for visualization. The GridWindow is simply a window with a grid of rgb pixels. $x*3, y$ define the dimensions of the pixel grid, and the last argument is a scaling factor which specifies that each pixel on the grid will be viewed as a 5×5 square of pixels on screen. We multiply x by 3 so that we can visualize 3 models side by side in the same window.

5: we create an array with 3 entries to fill in with Models

6: we create a file output object to write to a file called populations.csv

7-10: we fill the model list with models that are initialized identically. Each model will hold and update its own cells and diffusible drug. See the Grid Definition and Constructor section for more details. We also pass the number 0 to each Random number generator object that we create to ensure that the models will behave identically without additional changes.

- 9:** we initialize a circular tumor, composed of Cell agents, as the initial condition of the model. The tumorRad defines the radius of the tumor, and the resistantProp defines the proportion of initial cells that will be resistant. The rest of the initial population will be sensitive. See the InitTumor Function for more details.
- 11-12:** we alter the DRUG_DURATION constant in the 0th Model in our models array to never give drug, in the 1st Model to give drug constantly. by default the 2nd Model will Give Drug periodically. This modification is the only difference between the models.
- 14:** We next execute the main loop for 10000 timesteps, the GetTick function returns the current Model tick. see the ModelStep Function for where the Model tick is incremented.
- 15:** depending on the performance of the machine we are using to model, we may impose that every iteration of the loop should take a minimum number of milliseconds. This slows down the execution and display of the model and makes it easier for the viewer to follow.
- 16:** we loop over all models to update them.
- 17:** we advance the state of the agents and diffusibles in each model by one timestep. See the Model Step Function for more details.
- 18:** we draw the current state of each model to the window. See the Draw Model Function for more details.
- 20:** we write the population sizes of each model every timestep to record how well the different regimens compare.
- 21-22:** every 2500 ticks, we write the state of the model as captured by our GridWindow to a PNG file. we subtract 1 because the first timestep is used for initialization.
- 25-26:** after the main while loop has finished, we close the FileIO object and close the visualization window, and the program ends

8.3 Grid Constructor and Properties

this section covers how the grid is defined and instantiated.

```

1 public class ExampleModel extends AgentGrid2D<ExampleCell> {
2     //model constants
3     public final static int RESISTANT = RGB(0, 1, 0), SENSITIVE =
4         RGB(0, 0, 1);
5     public double DIV_PROB = 0.025, DIV_PROB_RES = 0.01,
6         DEATH_PROB = 0.001, DRUG_START = 400, DRUG_PERIOD = 200,
7         DRUG_DURATION = 40, DRUG_DIFF_RATE = 2, DRUG_UPTAKE = 0.91,
8         DRUG_DEATH = 0.2, DRUG_BOUNDARY_VAL = 1.0;
9     //internal model objects

```



```

7     public PDEGrid2D drug;
8     public Rand rn;
9     public int[] divHood = MooreHood(false);
10    public int[] divIs = new int[divHood.length / 2];
11    public ExampleModel(int xDim, int yDim, Rand rn) {
12        super(xDim, yDim, ExampleCell.class);
13        this.rn = rn;
14        drug = new PDEGrid2D(xDim, yDim);
15    }

```

- 1: the ExampleModel class, which is user defined and specific to this example, is built by extending the generic AgentGrid2D class. The extended grid class requires an agent type parameter, which is the type of agent that will live on the grid. To meet this requirement, we add the <ExampleCell> type parameter to the extension.
- 3: we define RESISTANT and SENSITIVE constants, which are created by the Utils RGB function and are technically colors but are also used as labels for the phenotypes of the cells.
- 4: we define all of the constants that will be needed during the model run. these values can be reassigned after model creation to facilitate testing different parameter settings. In our case, we do this in the main function by modifying the DRUG_DURATION variable for our No-Drug, Constant-Drug, and Pulsed Therapy experiment cases.
- 6: we declare that the model will contain a PDEGrid2D, which will hold the drug concentrations. The PDEGrid2D can only be properly initialized when the x and y dimensions of the model are known, which is why we do not define them until the constructor.
- 7: we declare that the Grid will contain a Random number generator, but take it in as a constructor argument to allow the modeler to seed it if they wish.
- 8: we define an array which stores coordinate pairs in $[x1, y1, x2, y2, \dots]$ format using the builtin MooreHood function. these coordinate pairs define the moore neighborhood, centered around the (0,0) origin. The moore neighborhood is used to look for space when a cell attempts to divide. The boolean argument allows us to either include the origin as a coordinate pair, or leave it out of the set. See the CellStep Function section for more information.
- 9: we define an integer array that is half of the length of the moore neighborhood array. We will use this divIs array to store valid division position indices around a dividing cell. see the Cell Step Function for more information.
- 11: we define the model constructor, taking as arguments the x and y dimensions of the world and a Random number generator.

12: we call the AgentGrid2D constructor with super, passing it the x and y dimensions of the world, and the Cell Class object. This Class object is used by the Grid to generate a new cell when one is needed.

13&14: the random number generator argument is assigned, and the drug PDEGrid2D is defined with the proper dimensions.

8.4 InitTumor Function

The next segment of code is a function from the Model class that defines how the tumor is first seeded after the Model is created.

```

1 public void InitTumor(int radius, double resistantProb) {
2     //get a list of indices that fill a circle at the center of
      the grid
3     int[] circleCoords = CircleHood(true, radius);
4     int[] cellIs = new int[circleCoords.length / 2];
5     int cellsToPlace = HoodToEmptyIs(circleCoords, cellIs, xDim /
      2, yDim / 2);
6     //place a new tumor cell at each index
7     for (int i = 0; i < cellsToPlace; i++) {
8         NewAgentSQ(cellIs[i]).type = rn.Double() < resistantProb ?
          RESISTANT : SENSITIVE;
9     }
10 }

```

1: the arguments passed to the InitTumor function are the approximate radius of the circular tumor we intend to create and the probability that a given cell created in this way will be of the resistant phenotype.

3: the circleCoords array is set using the builtin CircleHood function, which stores coordinate pairs in $[x_1, y_1, x_2, y_2, \dots]$. These coordinate pairs define a neighborhood of all squares whose centers are within the radius distance of the center $(0, 0)$ origin square. The boolean argument specifies that we intend to include the origin in this set of coordinates, thus making a completely filled circle of coordinates.

4: we define an array that is half of the length of the circleCoords array we just defined. this array will store the Grid indices into which we will place the initial tumor cells.

5: we use the builtin HoodToEmptyIs function which displaces the circle coordinates so that they are centered around the middle of the Grid $(xDim/2, yDim/2)$ and then puts into the cellIs array all of the indices of the displaced circleCoords positions that are unoccupied. The function returns the number of these valid indices found, which is the number of cells that will be placed.

7: we use the builtin NewAgentSQ function to place a new cell at each valid position, and for each new cell placed, we sample a random number in the

range (0–1] and compare to the `resistantProb` argument to set whether the cell should have the resistant phenotype or the sensitive phenotype. The “?” “:” notation may be new to some readers. it is called the conditional operator, and works by evaluating the condition to the left of the question mark, and returns either what is to the left or right of the colon, depending on whether the condition evaluates to true or false, respectively.

8.5 ModelStep Function

this section looks at the main step function which is executed once per tick by the Model.

```

1 public void ModelStep() {
2     for (ExampleCell cell : this) {
3         cell.CellStep();
4     }
5     //check if drug should enter through the boundaries
6     if (GetTick() > DRUG_START && (GetTick() - DRUG_START) %
7         DRUG_PERIOD < DRUG_DURATION) {
8         drug.DiffusionAD1(DRUG_DIFF_RATE, DRUG_BOUNDARY_VAL);
9     } else {
10        drug.DiffusionAD1(DRUG_DIFF_RATE);
11    }
12    CleanShuffInc(rn);
13 }

```

- 2: we iterate over every Cell on the grid. Cells that were created during this timestep are not included until the subsequent timestep.
- 3: we call the CellStep function on every cell. See the CellStep Function for more information.
- 6: the GetTick function is a builtin function that returns the current Grid tick. The If statement logic checks if the tick is past the drug start and if we are in the right portion of the drug period to apply drug. (See the Grid Definition and Constructor section for the values of the constants involved, the DRUG_DURATION variable is set differently for each model in the Main Function)
- 7: if it is time to add drug to the model, we call the builtin DiffusionADI function. the default Diffusion function uses the standard 2D laplacian and is of the form: $\frac{\delta C}{\delta t} = D\nabla^2 C$, where D in this case is the DRUG_DIFF_RATE. ADI diffusion allows us to take larger steps than the standard stable rate limit of 0.25. The additional argument to the diffusion equation specifies the boundary condition value DRUG_BOUNDARY_VAL. this causes drug to diffuse into the PDEGrid2D from the boundary.
- 9: without the second argument and without wrap-around (which is disabled by default) the Diffusion function assumes a reflective boundary, meaning that drug concentration cannot escape through the sides. Therefore the

only way for the drug to exit from the model is via consumption by the Cells. See the CellStep function section, line 6 for more information.

- 11: the CleanShuffInc function actually calls 3 different builtin Grid functions internally: the CleanAgents function, which speeds up iteration over the set of all living agents, the ShuffleAgents function, which randomizes the order of iteration so that the agents are always looped through in random order, and the IncTick function, which increments the model by one timestep. we pass the Random number generator to this function for use during the ShuffleAgents portion.

8.6 CellStep Function and Cell Properties

we next look at how the Agent is defined and at the CellStep function that runs once per Cell per tick.

```

1  class ExampleCell extends AgentSQ2Dunstackable<ExampleModel> {
2      public int type;
3      public void CellStep() {
4          //Consumption of Drug
5          G().drug.Mul(Isq(), G().DRUG_UPTAKE);
6          //Chance of Death, depends on resistance and drug
           concentration
7          if (G().rn.Double() < G().DEATH_PROB + (type == RESISTANT
           ? 0 : G().drug.Get(Isq()) * G().DRUG_DEATH)) {
8              Dispose();
9          }
10         //Chance of Division, depends on resistance
11         else if (G().rn.Double() < (type == RESISTANT ?
           G().DIV_PROB_RES : G().DIV_PROB)) {
12             int nEmptySpaces = HoodToEmptyls(G().divHood,
           G().divls);
13             //If any empty spaces exist, randomly choose one and
           create a daughter cell there
14             if (nEmptySpaces > 0) {
15                 G().NewAgentSQ(G().divls[G().rn.Int(nEmptySpaces)]) . type
           = this.type;
16             }
17         }
18     }
19 }

```

- 1: the ExampleCell class is built by extending the generic AgentSQ2Dunstackable class. the extended Agent class requires the ExampleModel class as a type argument, which is the type of Grid that the Agent will live on. To meet this requirement, we add the <ExampleModel> type parameter to the extension. [SIMPLIFY]
- 2: we define an internal int called type. each Cell holds a value for this field. if the value is RESISTANT, the Cell is of the resistant phenotype, if the value is SENSITIVE, the cell is of the sensitive phenotype. RESISTANT and SENSITIVE constants are defined as Grid Properties

- 5: the `G()` function is used to access the Grid that the Cell lives on, it is used often with Agent functions as the Grid is expected to contain any information that is not local to the Cell itself. Here it is used to get the drug `PDEGrid2` from the Model. we then multiply the value at the Index of the square the agent is currently occupying (`Isq()`) by the drug uptake constant, thus modeling local drug consumption by the Cell.
- 7: We start by generating a random double in the range $(0 - 1]$ and then check if it is less than the death probability. This is effectively the same as sampling weighted coin whose probability of heads is the death probability. To calculate the death probability, we take the default `deathProb` value from the Model. we next use the conditional operator (`? :`) and based on the phenotype we either add the concentration of drug at the cell position times the drug death scaler constant, or add nothing if the Cell is resistant to the drug, meaning that we ignore its effect.
- 8: If our if statement on line 7 evaluates to true, meaning that the Cell has been randomly chosen to die, we call the builtin `Dispose` function on the cell, removing it from the model.
- 11: this line follows a similar logic to line 7, by comparing a random double in the range $(0 - 1]$ to the division probability. The division probability is calculated again via the conditional operator, which based on the phenotype either returns the resistant cell division probability (`divProbRes`), or the sensitive cell division probability (`divProb`).
- 12: If our statement on line 11 evaluates to true, meaning that the Cell has been randomly chosen to divide, we use the builtin `HoodToEmptyIs` function which displaces the `divHood` (the moore neighborhood as defined in the Grid Definition and Constructor section) to be centered around the x and y coordinates of the Cell and then puts into the `divIs` array (as defined in the Grid Definition and Constructor section) all of the indices of the moore neighborhood positions around the Cell that can house a daughter cell (are unoccupied). the function returns the number of these valid indices found.
- 14: If there are no unoccupied locations around the cell, then the cell is contact inhibited and cannot divide.
- 15: if there are one or more empty spaces around the dividing cell, we create a new daughter cell, using the builtin `NewAgentSQ` function and choose the square index that it will start at by randomly sampling the `divIs` array to pull out one if its valid locations. Finally with the `.type=this.type` statement, we set the phenotype of the new daughter cell to the phenotype of the daughter that remains in place, (which is technically the parent Cell) thus maintaining phenotypic inheritance.

8.7 DrawModel Function

The final code snippet looks at the relatively straightforward DrawModel Function, which is used to display a summary of the model state on a GridWindow object. DrawModel is called once for each model per timestep, see the Main Function section for more information.

```
1 public void DrawModel(GuiGrid vis, int iModel) {
2     for (int i = 0; i < length; i++) {
3         ExampleCell drawMe = GetAgent(i);
4         //if the cell does not exist, draw the drug concentration
5         vis.SetPix(ItoX(i)+iModel*xDim, ItoY(i), drawMe == null ?
6             HeatMapRGB(drug.Get(i)) : drawMe.type);
7     }
8 }
```

- 2: we loop over every lattice position of the grid we are drawing, length refers to the number of lattice positions in an individual model.
- 3: we use the builtin GetAgent function to get the Cell that is at the index as the drawMe variable. if there is no cell at the index, then drawMe will be set to null.
- 5: if there is no cell to draw, then we set the pixel color based on the drug concentration at the same index, using the builtin heat colormap, otherwise we use the cell's type to color. we get the pixel to draw to by converting the index to X and Y coordinates using the Grid's ItoX and ItoY functions. we then displace using the passed iModel index along the x dimension to offset where the second and third model are drawn.

8.8 Imports

We look at the imports that are needed. Any modern Java IDE should generate import statements automatically.

```
1 package Examples._6CompetitiveRelease;
2 import Framework.GridsAndAgents.AgentGrid2D;
3 import Framework.GridsAndAgents.PDEGrid2D;
4 import Framework.Gui.GridWindow;
5 import Framework.GridsAndAgents.AgentSQ2DUnstackable;
6 import Framework.Gui.GuiGrid;
7 import Framework.Tools.FileIO;
8 import Framework.Rand;
9 import static Examples._6CompetitiveRelease.ExampleModel.*;
10 import static Framework.Util.*;
```

- 1: the package statement is always needed and specifies where the file exists in the larger project structure
- 2-8: we import all of the classes that we will need for the program

- 9:** we import the static fields of the model so that we can use the type names defined there in the Agent class.
- 10:** we import the static functions of the Util file, which adds all of the Util functions to the current namespace, so we can natively call them. Statically importing Util is recommended for every project.

8.9 Model Results

Table 2 displays the model visualization at tick 0, tick 400, tick 1100, tick 5500, and tick 10,000. The figure caption explores the notable trends visible in each image. Figure 2 displays the population sizes as recorded by the FileIO object at the end of every timestep.

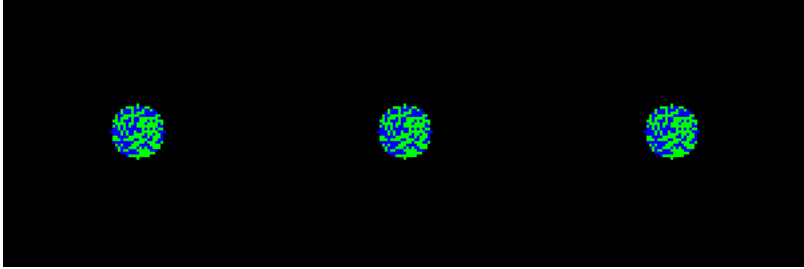
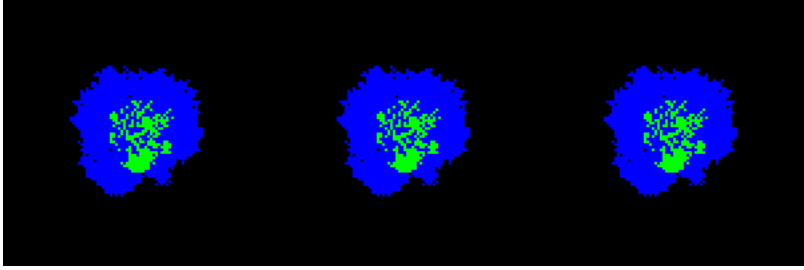
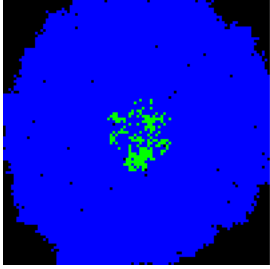
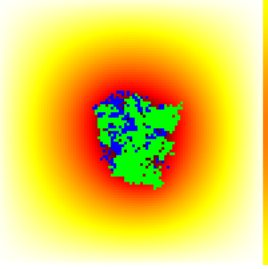
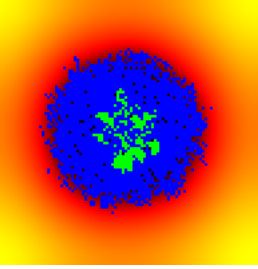
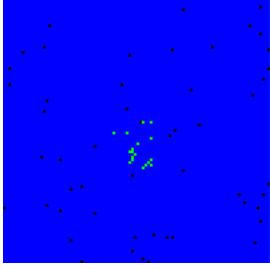
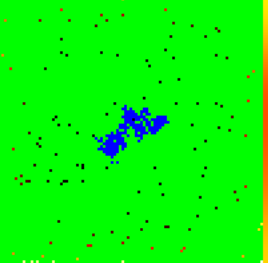
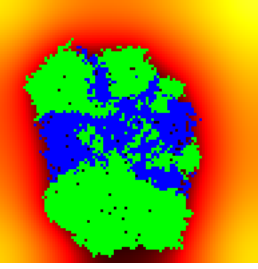
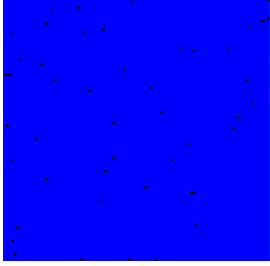
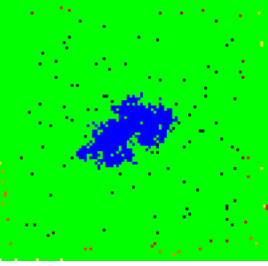
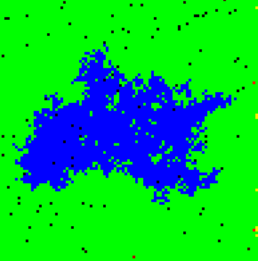
Timestep	No Drug / Constant Drug / Pulsed Drug		
0			
400			
1100			
5500			
10000			

Table 2: Selected model visualization output PNGs. Blue cells are drug sensitive, Green cells are drug resistant, background heatmap colors show drug concentration. At timestep 0 and timestep 400, all 3 models are identical. At tick 1100 the differences in treatment application show different effects: when no drug is applied, the rapidly dividing sensitive cells quickly fill the domain, when drug is applied constantly, the resistant cells overtake the tumor. Pulsed drug kills some sensitive cells, but leaves enough alive to prevent growth of the resistant cells. At tick 5500, the resistant cells have begun to emerge from the center of the pulsed drug model. At tick 10000, all domains are filled. interestingly, the sensitive cells are able to survive in the center of the domain because drug is consumed by cells on the outside, this creates a drug free zone in which

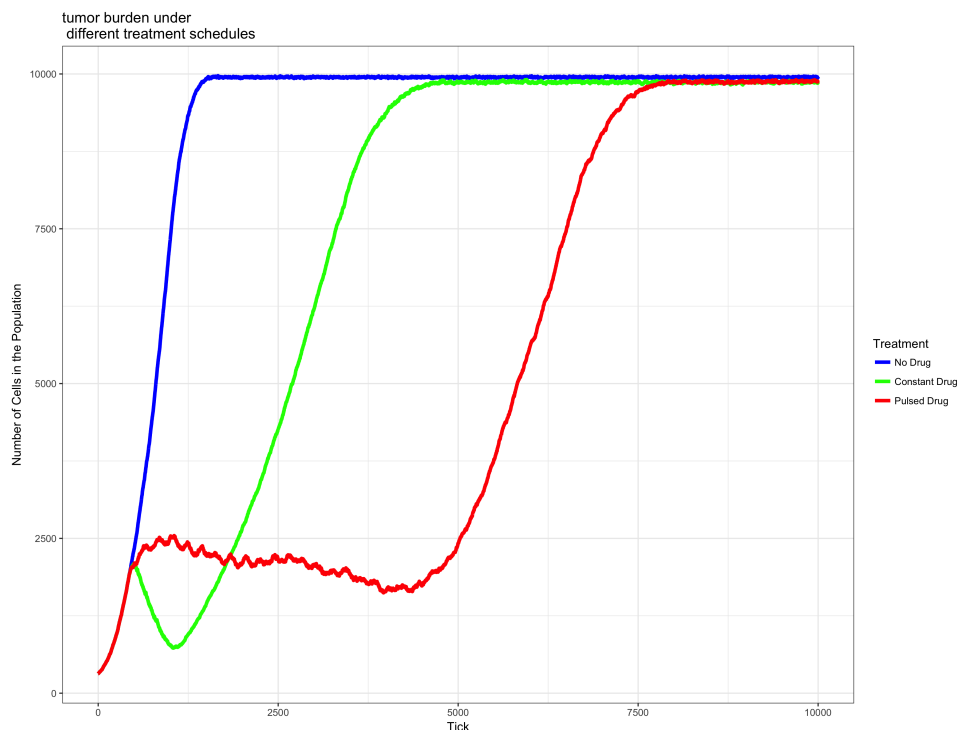


Figure 2: FileIO population output. This plot summarizes the changes in tumor burden over time for each model. This plot was constructed in R using data accumulated in the program output file. Displayed using GGPlot in R Figure arrows

As can be seen in Figure 2 and Table 2, the pulsed therapy is the most effective at preventing tumor growth, however the resistant cells ultimately succeed in breaking out of the tumor center and outcompeting the sensitive cells on the fringes of the tumor. It may be possible to maintain a homeostatic population of sensitive and resistant cells for longer by using a different pulsing schedule, or possibly apative therapy. As the presented model is primarily an example, we do not explore how to improve treatment further. For a more detailed explanation of this problem, see [1].

References

- [1] Jill A Gallaher, Pedro M Enriquez-Navas, Kimberly A Luddy, Robert A Gatenby, and Alexander RA Anderson. Adaptive vs continuous cancer therapy: Exploiting space and trade-offs in drug scheduling.

Acknowledgements

This work was possible through the generous support of NIH funding, Anderson and Tessi acknowledge NCI U54CA193489, Anderson and Bravo acknowledge NCI UH2CA203781.