A Flexible Simulator for Oncolytic Viral Therapy

A Thesis
SUBMITTED TO THE FACULTY OF
UNIVERSITY OF MINNESOTA
BY

David Ryan Berg

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Dr. Zeljko Bajzer

Dr. Claudia Neuhauser

May 2015

Acknowledgements

As I complete my master's degree I'd like to thank Dr. Zeljko Bajzer, for introducing me to the University of Minnesota Rochester's BioMedical Informatics and Computational Biology program. I'm also appreciative for the internships and stipends he obtained for me during the program. Finally, his advice in the writing of this thesis and his willingness, along with Chetan Offord, to repeatedly review and suggest improvements have made this thesis what it is today.

I'm very appreciative for Chetan Offord's assistance with understanding his Fortran code for the forerunner to the program presented in this thesis. Additionally, I'm grateful for his patience while explaining the biological experiments being run in tandem with the computer simulations and for his identification of bugs that worked their way into the program.

Dr. George Paulik's explanations of the Poisson process and exponential races were instrumental in my understanding of why this simulation approach is valid. Additionally, I am most grateful for Dr Paulik's kind words about my abilities as a computer programmer and analyst. They provided much needed encouragement during a time when most feedback was focused on what needed improvement.

Dr. Claudia Neuhauser's explanations of the effect of dimensionality with regard to contact processes was very helpful to my understanding. Additionally, I appreciate all her advice and suggestions as my advisor during the pursuit of my master's degree.

I found Dr. David Dingli's enthusiasm for the latest laboratory techniques to be infectious and his push to expand the scope of this thesis has made me more aware of the roles this software may play in his research.

Finally, I'd like to thank my wife, Tiffany, for her understanding, patience, and financial support while I pursued this degree. If it was not for her picking up the slack around the house, I never could have completed it.

Abstract

Developments in recombinant DNA technology have given researchers the ability to modify viruses so that they are highly selective towards cancer cells. Engineered viruses have successfully treated cancer in human trials. In an effort to better understand viral population dynamics in a temporal context, researchers have turned to mathematical models. Some of these viruses spread only by contact between virus-infected and uninfected tumor cells. Therefore, mathematical models that usually assume populations are well-mixed may not apply. This thesis describes a computational approach to modeling viral population dynamics that takes into account the spatial nature of viral spread by contact.

Table of Contents

Index of Figures

Index of Tables

Motivation & Prior Work

The field of oncolytic virotherapy began to develop around the 1900s with reports of patients experiencing tumor regression that coincided with a naturally obtained virus infection (cf. Kelly, Russell, 2007). Experiments attempting to take advantage of this phenomenon proved unsuccessful, with regressions in the best cases lasting only a few months. As a result the phenomenon was largely ignored until the middle of the twentieth century when improved cell culture techniques led to a resurgence of virotherapy trials. Unfortunately, the results were the same as they were 50-70 years earlier (Alemany, 2013, p. 182). Ultimately, the creation of recombinant DNA technologies allowing the modification of virus genomes brought about a new surge of oncolytic virotherapy research (Alemany, 2013, p. 183). Modern computing capabilities are also playing a significant role in efforts to model and guide in vivo experimentation.

Modeling and Simulating Population Dynamics

To gain a better understanding of how virus and cancer cells interact, researchers have historically turned to mathematical models such as the Lotka-Volterra equations and reaction diffusion equations. However, these models usually assume that populations within the system are well-mixed. This may not be the case in virotherapy where populations of cancer and fusogenic virus cannot move freely throughout the environment. Satō, Matsuda and Sasaki's 1994 paper as well as Durrett and Levin's paper from 1994 both examined this problem, recognizing the importance of the spatial component and local interactions to population growth.

Satō, Matsuda and Sasaki showed in 1994 that Lotka-Volterra models of pathogen invasion into host populations indicate that hosts were never extinguished despite large pathogen transmission or proliferation rates. This was contrasted with computer simulations which relied on repeated random sampling. The simulations were run on a finite, two dimensional, regular network created to simulate growth of parasite and host populations. Parasites were restricted to only allow local transmission, that is they could only infect neighboring cells (a contact process). Here parasites were able to drive the host to extinction with appropriate parameters. Their work suggested that in populations with a spatial structure, pathogens can have a much more lethal impact than in populations where individuals were assumed to be well mixed (p. 261).

Durrett and Levin's 1994 work also showed that spatial, non-spatial, continuous and discrete models of population growth do not always predict the same outcomes. Using ordinary differential equations (ODE), patch models, reaction-diffusion equations (RDE) and interacting particle systems (IPS) they simulated three different cases of species interaction. The patch model was described as one that “recognizes the importance of space at local scales but the collection of patches ... has no spatial structure” (p. 367). In this model, individuals were able to move between randomly chosen patches at a specified rate. However in interacting particle systems, patches were fixed on a network and individuals can only move to neighboring patches. The patch and IPS models are both discrete population, simulated systems as noted in Table 1.

	Continuous	**Discrete**
Non-Spatial	ODE	Patch
Spatial	RDE	IPS

Table 1: Categorization of models.

IPS is spatial while the patch model is non-spatial. In all four modeling approaches only two populations were considered.

These four modeling approaches were applied to the following three cases.

1. The success of one population increases the fitness of a second population; i.e. a symbiotic relationship.
2. Individuals from the populations compete for the same resource.
3. One population is preying on the second. Rates were chosen so that predators always do better than prey but die out if no prey is present.

Durrett and Levin found that all four modeling approaches agreed only in case one where the relationship is symbiotic. In case two where populations compete, the spatial models (RDE & IPS) and non-spatial ones (ODE & Patch) disagreed. Non-spatial models indicated that populations will find a coexisting equilibrium while the spatial models find that one population will win out with the victor spreading from a region where it was able to establish itself (p. 391). In the final case, "the predator population eliminates the prey locally but then dies out itself and the empty space was recolonized by prey" (p. 378). In this case the continuous population systems (ODE & RDE) agreed that both populations die out. However, both discrete population systems indicated that populations can coexist in equilibrium. These results are also presented in Table 2.

	#1	#2	#3
ODE	A	B	D
RDE	A	C	D
IPS	A	C	E
Patch	A	B	E

Table 2: Agreement in modeling approaches. Matching letters indicate matching results.

In 2012, Wodarz et al. built on the work of Satō et al. by applying their computer simulator to the problem of modeling the growth of an adenovirus population on normal

human embryonic kidney epithelial cells growing in a two dimensional layer. Wodarz et al. observed experimental data in which they categorized three types of growth patterns and then matched them with the results of the simulator. These patterns were referred to as “hollow ring”, “disperse” and “filled ring” (p. 2). To obtain agreement with experimental data, the death rate of infected cells was slightly changed while the virus transmission and uninfected cell parameters were kept constant. Simulations were run and parameters were estimated by least squares fitting to observed in vitro data for the time course.

In 2009, Paiva et al. described a relatively complex computer simulation model initially proposed by coauthors Ferreira and Martins. This model considered normal, necrotic, uninfected and infected cancer cells as individual agents with sigmoidal infection rates dependent on the presence of nutrients. Additionally, spread of free virus and nutrients were modeled by reaction-diffusion equations. While the network on which cells grew was two dimensional, cancer cells (infected or uninfected) were able to pile up, mimicking a loss of contact inhibition. Paiva et al. summarized the model by stating that “both mesoscopic and macroscopic scales are inexorably interwoven, although described in terms of distinct physical models: partial differential equations (macroscopic level) and probabilistic cellular automata rules (mesoscopic level) coupled in a single model through division, death, infection, lysis, uptake, release and adsorption rates and diffusion coefficients” (p. 1206).

Similar to Wodarz et al., Paiva et al. found that varying parameter values could result in cancer eradication or continued tumor growth after a brief remission.

Additionally, cancer response had “great sensitivity to the local, stochastic fluctuations of uninfected cell population and free virus concentration.” It was also shown that given the right parameters, the system would oscillate aperiodically while trending towards higher populations of both uninfected cancer cells and free viruses (Paiva et al., 2009).

The immune system's response to the presence of virus negatively affects the success rates of oncolytic virotherapy. Paiva et al. determined that specifying a high virus clearance rate, or immune system response, always results in treatment failure. Yet decreasing the virus clearance rate slightly while holding other parameters constant results in treatment success. This finding is in agreement with observations where oncolytic viruses have a lesser effect in immunocompetent models (cf. Paiva et al., 2009). Finally, they found that “the most likely therapeutic responses are determined by the oncolytic activity and spreading properties of the virus” (Paiva et al., 2009). However the time for lysis, which determines the offset between growth of virus and cancer populations, must also be between 8 and 64 hours.

The simulator described by Paiva et al. in 2009 was further extended in a 2011 paper to treat the oncolytic viruses as discrete agents in order to provide “a more realistic description of virus entry and replication” (Paiva, Martins, & Ferreira, 2011). The updated simulator also makes allowances for an uninfected cancer cell to migrate to a neighboring location, switching positions with any normal or necrotic cells. A third change to the simulator's reaction-diffusion equations distinguished two types of nutrients: those required for survival and those which were required only for cell replication. Parameters were chosen based on the results of other works. Results from this

model were reported as being consistent with those found in Paiva et al, 2009 (Paiva, Martins, & Ferreira, 2011).

In 2010, Reis, Pacheco, Ennis and Dingli created a three dimensional model for measles virus built around a face centered cubic network with twelve neighbors. Two stochastic growth patterns were defined. In one, new tumor cells pushed existing tumor cells away from the new cell leading to a dense tumor. The other has daughter cells occupying a location near the parent cell after taking a random walk. This created a branching effect resulting in fractal-like tumor growth. Simulations found that small, dense tumors were significantly more likely to be eradicated than large, fractal tumors. The authors state that according to “models based on differential equations … 'larger' tumors may be better targets for virotherapy” and point out that the findings from this simulator were contrary to results from these mathematical models. To treat fractal-like tumors, careful selection of time to cell death is critical. Time from infection to death must be great enough that virus infection has time to spread but low enough that the immune response doesn't adversely affect the virus population. There was no matching to in vivo or in vitro data with this simulator (Reis, Pacheco, Ennis & Dingli 2010).

All of the simulators mentioned previously, with the exception of Reis, Pacheco, Ennis and Dingli, operate on a regular two dimensional lattice. However, in vivo tumors grow in three dimensional space. This difference in dimensionality has significant effects in the way populations in a contact process move through space. For example, the work of Ben-Naim, Frachebourg and Krapivsky looks at the effect of dimensionality in the voter model, a type of contact process. This work indicates that the rate of change in

voter model populations depends strongly on dimensionality (1996). If one considers a symmetric, nearest neighbor random walk starting at the origin of an unbounded, two dimensional, integer coordinate system, the walk will almost surely return to the origin at some point in time. However, in three dimensions the probability of returning to the origin drops to approximately 34% (Weisstein). Thus, three dimensional simulators may provide qualitatively different results from two dimensional simulators. Additionally, cell shapes are amorphous, not square or cuboid, which means that a regular lattice may not be an accurate representation of the space in which they grow. With these points in mind we set out to create a simulator that would have the following features as enumerated by Dr Zeljko Bajzer (personal communication, January 20, 2015):

- Be able to run on a two or three dimensional network.
- Be able to run on irregular and lattice based networks.
- Have the capacity to run simulated experiments allowing evaluation of the conjecture made by Wodarz et al.: temporal behavior of cell populations can be adequately modeled by ordinary differential equations describing the size of populations (mean field models). However, the conjecture was indicated only for two dimensional simulators.
- Have the capacity to compare simulated spatial distributions and temporal dynamics to corresponding results obtained for two dimensional in vitro, as well as three dimensional in vitro and in vivo experiments.
- Have the capacity to specify what type of cell occupies a node in the network at runtime.

The simulator described in the rest of this thesis accomplishes the above by accepting as input a network described by a list of adjacencies for each node. The implication of this is that the simulator itself does not have any concept of

dimensionality, allowing it to run on two, three or more dimensions. Additionally, the number of neighbors each node has and their location can vary from node to node. This allows a network to be defined directly from microscopic images of cell culture where nuclei express green fluorescent protein as described in the section Initial State Generated with Voronoi Tessellation from Confocal Microscopy.

Included alongside the core of the simulator are a collection of routines which allow the network state to be artificially modified at various points in the simulation. These routines, with the flexible network representation allow many different experiments to be modeled. For example Chetan Offord describes, “two dimensional in vitro experiments conducted involved serially imaging plated HT1080 (human fibrosarcoma) cells infected with a measles virus, both [expressing] a fluorescent protein. Similar experiments were performed by coating plates with Matrigel, inducing the cells to grow in three dimensional spheroids” (personal communication, January 21, 2015).

In vivo mouse experiments began by inserting KAS-6/1 cancer cells in to the flanks of 6-week-old female CB17 mice with severe immunodeficiencies. An intravenous virus injection of MV-NIS was administered through the mouse's tail vein when the tumors reached an average diameter of 5mm. Tumor volume was recorded throughout the experiment (Dingli et al., 2009).

This simulator is capable of reconstructing the conditions, dimensionality, spatial dynamics and temporal dynamics of each of these experiments.

Program Documentation

This simulator was originally developed in Fortran by Chetan Offord (personal communication, March 11, 2014). The logic from the Fortran version was preserved while changing the data structures to allow for arbitrary networks. Additional routines were added to extend the basic functionality. The simulator operates on a network of nodes that may or may not be occupied by a cell. A cell can be one of three types: a normal cell, a cancer cell, or cancer cell that has been infected with virus. A node with no cell is empty. Cells undergoing replication or spreading infection (source cells) are only permitted to proliferate onto a node of a specific type (target node). Normal and cancer cells are only permitted to target empty nodes while infected cells only target cancer

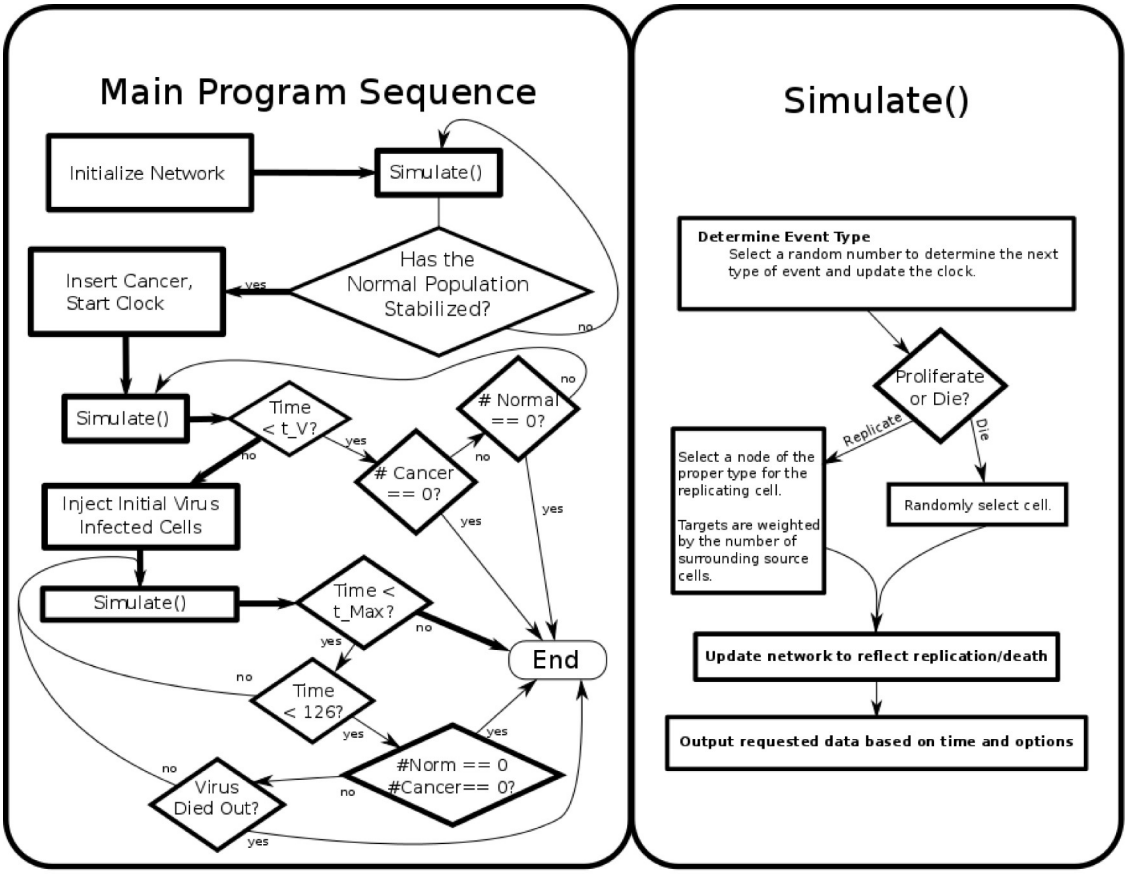


Figure 1: Example of Program Flow

cells. Throughout the text, configurable values are referred to by their respective long format option. For example, `--NormalGrowth` refers to a normal cell's replication rate. Cancer cells are inserted in the network. In the code and program arguments the term “injected” refers to the process of virus-infected cells replacing selected nodes. In the program, these “injection” routines select nodes for viral infection outside of simulator time. If virus-infected cells are to be introduced to the network at time T, then after the first simulation loop that causes simulator time to be greater than or equal to T, the simulation stops. While it is stopped, selected nodes become virus-infected. After the proper number of nodes are infected the simulation, and clock, restart. The simulator, and mathematical model, do not account for the the phase of cell infection with free virus that would be present during in vivo and in vitro experiments.

Normal and cancer cells replicate into neighboring empty nodes while infected cells infect neighboring cancer cells. The rates for replication and infection are collectively referred to as proliferation rates in this text, and growth rates in the code and program arguments. References to time in this text refer to the simulator's perspective of time, not clock time or CPU time. The unit of time is the same as the units used for proliferation and death rates. The program was developed using standard C libraries and has been run under a variety of linux distributions. Uniformly distributed random variables are obtained from Nishimura and Matsumoto's 2002 C implementation of the Mersenne Twister.

A simulation begins with the reading of a set of network files specifying network structure, node locations and optionally, cell types. If cell types are not provided they are

assumed to be normal. The simulator goes through an optional equalization period (`--Equalize`), running until the normal population stabilizes. At that point, cancer cells are inserted, time begins and the simulation runs for `--TimeInfect` days. virus-infected cells then replace selected nodes and the simulation continues until either cancer/virus-infected cells become extinct or `--TimeMax` is reached. This process is graphically shown in Figure 1. How equilibrium is determined, cancer cells are inserted, and nodes are selected for viral infection is explained below.

Running the Simulator

Proliferation rates, death rates, cancer insertion and virus infection parameters can all be specified as command line arguments to the `Cells` program. These arguments and their defaults are detailed by executing `Cells --help`. The output of this command is below.

```
$ ./Cells --help
Usage: Cells [option(s)] datFILE, netFILE, coordFILE
The simulator options are:
 -N --NormalGrowth=NUM   Growth rate of normal cells (default 0)
 -n --NormalDeath=NUM    Death  rate of normal cells (default 0)
 -C --CancerGrowth=NUM   Growth rate of cancer cells (default 0)
 -c --CancerDeath=NUM    Death  rate of cancer cells (default 0)
 -I --ViralIntGrowth=NUM Growth rate of Interior viral cells (default 0)
 -i --ViralIntDeath=NUM  Death  rate of Interior viral cells (default 0)
 -J --ViralExtGrowth=NUM Growth rate of Exterior viral cells (default 0)
 -j --ViralExtDeath=NUM  Death  rate of Exterior viral cells (default 0)
 -V --PercentInfected=NUM  Percentage of cancer nodes that become viral
 -T --InfectionType=TYPE   TYPE=[RANDOM|CENTER|PERIMETER|MULTINODE]
 -P --PercentInterior=NUM  Percent of Normal neighbors for interior
  -r --CancerRadius=NUM     Cancer ball radius in neighbors (default 1)
  -q --CancerCount=NUM      Number of cancer nodes to insert (default 1)
  -R --RandomSeed=NUM       Seed for random number generator(default 11)
  -m --TimeMax=NUM        Maximum time simulator should run (default 75)
  -v --TimeInfect=INTEGER   Time between cancer & virus injection
(default 7)
  -E --Equalize            Don't start until normal population is stable
 The output options are:
  -o --OutputInterval=NUM   Set output frequency (default .5)
  -g --Grid                 Print square 2d grid to stdout.
```

```
  -s --Stats                Print stats to stdout.
  -S --Summary              Print summary to stdout.
  -t --TumorSizes           Print number of tumors and their size
  -W --Watch                Wait for return key after every output
  -F --Files                Output to vpts.dat, vlpt.dat, and vout.dat
  -f --File=NAME            Save network states to NAME.csv
```

Proliferation and Death Rates

Death rates indicate the expected number of cells to experience a death in a unit of time. The simplest way to describe the proliferation rates (referred to as growth rates in the code) is that they indicate the expected number of proliferation attempts made by an individual per unit of time. If the node an individual attempts to proliferate into is not a target of that individual, the replication is suppressed. These suppressions lead to no-ops mentioned below in the section Updating Time. To avoid these no-ops, the simulation was treated as a series of Poisson Processes. This changes the way the rates are handled in the simulator but not their meaning.

--Equalize

If this argument is given then the simulation will be run until the normal cell population has stabilized. Time is then reset to 0 and cancer cells are inserted.

--PercentInfected and --InfectionType

`--PercentInfected` (*P*) indicates the percentage of cancer cells that are changed to virus-infected nodes at `--TimeInfect`. If there are *K* cancer cells then *K***P*/100 = *I* cells will become virus-infected. The `--InfectionType` can be one of four words indicating different techniques for distributing the initial virus-infected cancer cells. The four types are:

RANDOM: Repeatedly select a random cancer cell from a densely packed array to become virus-infected until *I* cells are infected.

CENTER: Average the coordinates of all cancer cells and find the cancer node closest to that average. Infect this cell, its neighbors and neighbors' neighbors until *I* cells are infected. In a network with a wraparound topology, a tumor could be located around the network perimeter. In this case the average position of cancer neighbors would not be at the center of the tumor and the nearest cancer node to the average would be on the tumor's edge. It is important to keep this in mind when using this argument with wraparound networks. Utilizing the ability to specify the node where cancer insertion should be centered as the fourth value in the datFILE would address this issue.

MULTINODE: As with *CENTER*, the average of coordinates of all cancer cells is determined and the cancer node closest to that average is found. A line with random direction is created that passes through this node. The algorithm marks the current node as visited, then moves to the neighbor closest to this line that has not been visited. The algorithm continues moving along the line until a node with neighbors that are not cancerous is found. This last node, its cancerous neighbors and neighbors' neighbors are infected until *I*/3 cells have been infected. This is then repeated two more times so a total of *I* cancer nodes become virus-infected. The *MULTINODE* infection method has the same caveat as *CENTER* with regard to wraparound networks.

PERIMETER: Again, the average of coordinates of all cancer cells is found and the cancer node (C) closest to that average is located. The Cartesian distance from C to all other cancer nodes is calculated. The nodes with the largest distances are infected until *I*

nodes have become infected. Again, wraparound networks can cause problems with this routine.

--PercentInterior

This argument is used to determine if an infected cell is in the interior or exterior of a tumor. The test used is: $\frac{100*(number\ of\ normal\ neighbors)}{(number\ of\ neighbors)} \leq PercentInterior$

If the inequality is true, then an infected node is interior. Otherwise, the node is considered exterior.

--CancerRadius and --CancerCount

Cancer insertion is performed by changing the type of a specified node (C) and all its neighbors, its neighbors' neighbors and so on until there are `--CancerCount` cancer cells or the network distance between C and the next neighbor is greater than `--CancerRadius`, whichever results in more cancer cells. Node C is either specified as the fourth argument in the `datFILE` or assumed to be the first node listed in `netFILE` and `coordFILE`.

--RandomSeed

Allows seeding of the random number generator with a specified value. If not specified, the generator is seeded with 11.

--TimeInfect

This argument allows the user to specify the amount of time that will elapse between cancer insertion and infection with virus.

--TimeMax

Changes the maximum time the simulator should be allowed to run.

--OutputInterval

Specifies how frequently output should be generated.

--Grid

If the number of nodes is a square number, an ASCII art representation of the network will be printed. Normal cells are represented by '.', cancer cells by 'c', interior infected cells by 'i', exterior infected cells by 'I', and empty nodes by '_'. See the example 10x10 grid shown in Figure 2.

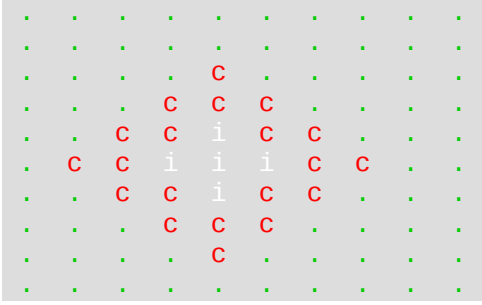


Figure 2: Example of --Grid output.

--Stats

Prints a line to the console indicating the time, normal cell density, cancer cell density and virus-infected cell density at each `--OutputInterval`.

--Summary

Outputs a single line containing the cancer cell density at each `--OutputInterval`.

--TumorSizes

Prints a line indicating the number of disjoint tumors and how many cancer cells are in each. The size list is not sorted but larger tumors are more likely to be indicated first.

--Watch

Requires a key to be pressed after every `–OutputInterval.`

--Files

Generates three files. One containing simulator parameters, the others containing the sum of cancer and virus-infected populations at various time points.

The file vpts.dat contains the *--RandomSeed* followed by the total tumor population immediately after insertion and at every *--OutputInterval* until time 126.

The file vlpt.dat also contains the --*RandomSeed* and total tumor population immediately after insertion. However, the following tumor populations are from every fifth time step until the *--TimeMax*.

File vout.dat contains --*RandomSeed,* all rates, *--PercentInfected,* the time a population went extinct, final population sizes, and what type of equilibrium resulted. The type of equilibrium is indicated with the following values:

1. Cancer died out
2. Virus died out, cancer remains
3. Cancer and virus remain

--File=NAME

If *--File* is specified, then an output file describing the state of the network will be generated at t=0 and at each *--OutputInterval*. The generated file is equivalent to the coordFILE with the last column indicating the node type as follows: 0=empty; 1=normal; 2=cancer; 3=infected. By default the filenames begin with Lattice_State_ followed by the simulator time and finally .csv is appended.

CoordFILE

`coordFILE` is used to describe the position of each node with Cartesian coordinates. The program expects coordinates to be provided beginning with the first node with each subsequent node on its own line. The number of coordinates provided must match the number of dimensions indicated in *datFILE*. Coordinates are expected to be specified as double precision floats. An additional number (0, 1, 2 or 3) can be provided to indicate the node's type (empty, normal, cancer, or infected respectively). These coordinates are used when outputting lattice states and in the non-random routines that replace selected nodes with virus-infected cancer cells.

NetFILE

`netFILE` is used to describe the neighborhood of each node. The description of a node begins with its number of neighbors, then a 1-based line number from `CoordFILE` of each neighboring node. Each of these values should be separated by whitespace. The description of the first node should be followed by whitespace, a description of node two and so on.

DatFILE

`datFILE` should contain, in this order, “total number of nodes in network”, “number of dimensions”, “maximum number of neighbors of any one node” and optionally, “the 1-based line number from `CoordFILE` of a node where cancer insertion should be centered”. Each of these values should be separated by whitespace (space, tab, newline).

Cell Proliferation/Death and the Keeping of Time

The simulator runs serially with only one cell proliferating or dying in each iteration. An iteration is broken into four parts. The first part is updating the simulator time. The second is determining what type of cell to act on, and whether that cell should proliferate or die. Next, the selection of a cell to eliminate or a node to proliferate into takes place. And finally, the network is updated to reflect these changes. These parts constitute the logic that was kept from Chetan Offord's original Fortran version.

The program was left as a serial implementation for simplicity of code after determining that run times were reasonable when executed on a single core. As an example the execution time for the process used to create image F in Table 5 was 38 minutes with a maximum memory usage of 280 megabytes. This simulation was run on an Intel i5-2540M with parameters chosen specifically to give a three population equilibrium. Most simulations do not reach a three population equilibrium and terminate much earlier. Additionally, for most use cases large numbers of simulations are submitted to a processor cluster so further parallelization only adds additional complexity. If the ability to run a single simulation on N cores is needed, the problem can be decomposed by splitting the network into N subnetworks. Each of these subnetworks would be processed in a single thread. The problems of maintaining a single simulation time and communicating changes across subnetworks to the appropriate thread would need to be addressed.

Updating Time

The simulator runs on the assumption that cell proliferation and death are independent events not depending on previous network states. It also assumes that the rates of these events are constant during each iteration. As a consequence, the arrival of events in the simulator is a Poisson process. This approach was selected by Chetan Offord and Drs. Bajzer, Dingli, Neuhauser and Paulik. In a previous version the program iterated through each node evaluating if a cell should proliferate, die, or do nothing (no-op). In this version they noted that computation time was dominated by no-op events (personal communication, April 18, 2014). In a Poisson process, the time to the next event is exponentially distributed. To obtain the parameter for the distribution, the death rates are scaled by the number of relevant cells in the network. Proliferation rates are scaled by a factor obtained through dividing the number of edges connecting a source to a target by the average number of neighbors. While it would be more accurate to divide by the exact number of neighbors for each cell, the average number of neighbors was chosen to simplify bookkeeping.

Time can then be incremented by $T_{inc:}$

$$T_{Inc} = \frac{-\ln(x)}{\lambda_T} \quad \text{where} \quad x \in (0,1) \quad \text{and} \quad \lambda_T = \sum (Scaled\ Rates)$$

This formula is explained further in the Appendix.

Determining Event Type

The scaled rates calculated above are then used as event probabilities. An array is created with each element corresponding to a particular event type (one of proliferation or death of a normal, cancer or infected cell). The array is populated according to the

following formula $E_{-1}=0$; $E_i = E_{i-1}$ + the scaled rate corresponding to the event for that element where *i* is between 0 and one less than the number of event types. A random number *x* is selected between 0 and λ_T and the array element whose value is closest to *x* and less than or equal to it indicates the event type. One could think of the process as creating an unfair dice with each side corresponding to an event type and having a probability of turning up equal to its scaled rate over λ_T.

Determining Event Recipient

A cell is selected for death by random selection from a densely packed array of cells of the appropriate type.

Cells are not actually marked for proliferation. Instead a node is selected to which a cell of the appropriate type could proliferate. A node 'N' with 'i' neighbors that could proliferate into 'N' will be i-times more likely to be selected than a node with only one neighbor that could proliferate into it.

Updating Network

With the time updated, event type determined and a node selected, the last step to take before the iteration repeats is to update the network state. The function `ChangeNodeType()` updates lists of each type of cell, removes and adds the changing node and its neighbors to lists of target nodes as appropriate, and updates the information each node maintains about its neighboring node types.

`ChangeNodeType()` also classifies each virus-infected cell as internal or external to the tumor. `ChangeNodeType()` is instructed to infect a node by being called as either `ChangeNodeType(node, INFECT_INT)` or

ChangeNodeType(node, INFECT_EXT). The two calls are equivalent as the code automatically determines how the cell should be classed. Additionally, every time the neighbor of an infected cell is changed, the routine again checks if the interior/exterior status of the infected cell should change and recursively calls itself to update the neighbors. A counter ensures that the recursion depth is never greater than 1.

Example Runs

Populate 10x10 Grid

The following command takes a 10x10 grid and populates it by centering five infected cells inside a tumor with radius 5. All remaining nodes contain normal cells. This state is printed to the console and saved to file so it can be loaded in future simulations.

```
$ ./Cells -r 3 -v 0 -V 20 -T CENTER -m 0 -f -g networks/10x10.dat
networks/10x10.net networks/10x10.cor
```

“-r 3” set tumor radius to 3 neighbors resulting in 25 cancer cells

“-v 0” inject virus immediately after inserting cancer

“-V 20” 20% of cancer cells (5 = 25*0.2) should become infected

“-T CENTER” place infected cells in the center of the tumor

“-m 0” stop simulation immediately after infection (0 days)

“-f” specifies that the state should be output to file

“-g” causes the network state to be printed to stdout (Figure 2).

Begin Simulation from Specified State

Reaching normal equilibrium can dominate run times when simulation times are short. It can be useful to perform this portion of the simulation a limited number of times, saving the equilibrium state and beginning subsequent simulations from there. So long as normal replication and death rates don't change, the beginning states will only vary

stochastically. The following example goes a step further and also bypasses the cancer insertion and infection routines using the file generated in the previous example.

```
$ ./Cells -r 0 -q 0 -v 0 networks/10x10.dat networks/10x10.net
Lattice_State_0.000000.csv [specify proliferation/death rates]
```

“-r 0” specifies that a cancer insertion radius of 0 should be used

“-q 0” specifies that 0 cancer cells should be inserted

“-v 0” specifies that virus should be injected immediately after cancer insertion

Note that both cancer quantity arguments are 0 and no `--InfectionType` is given. These items combine so that the provided lattice state doesn't change until the main simulation loop begins. Further, time will begin immediately from 0. If this lattice state is actually from a later point in time then be sure to adjust output accordingly during post processing.

Generating Graphics

The `Cells` program itself will only generate the ASCII graphics described above. However, the output files created with the `--File` option can be fed into a program such as Gnuplot to generate images or animations. Some examples are below:

Use the output file from the -f argument of a two dimensional simulation to output the lattice state to file. The following command uses a million node two dimensional network with a radius 300 tumor centered at node (500,500) and infects 20% of the cancer cells:

```
./Cells -r 300 -v 0 -V 20 -T CENTER -m 0 -f networks/1000x1000.dat
networks/1000x1000.n enetworks/1000x1000.cor
```

and generate a color graphic by running the following commands in gnuplot:

```
gnuplot> reset
gnuplot> set size square
gnuplot> set xrange [-100:1100]
```

```
gnuplot> set yrange [-100:1100]
gnuplot> set key outside
gnuplot> set key spacing 2
gnuplot> p "< awk '{if ($3 == 3) print $0}' output.csv" t 'Infect' pt 7
ps 1,\
 "< awk '{if ($3 == 2) print $0}' output.csv" t 'Cancer' pt 7 ps 1,\
 "< awk '{if ($3 == 1) print $0}' output.csv" t 'Normal' pt 7 ps 1,\
 "< awk '{if ($4 == 0) print $0}' output.csv" t 'Empty'  pt 7 ps 1
```

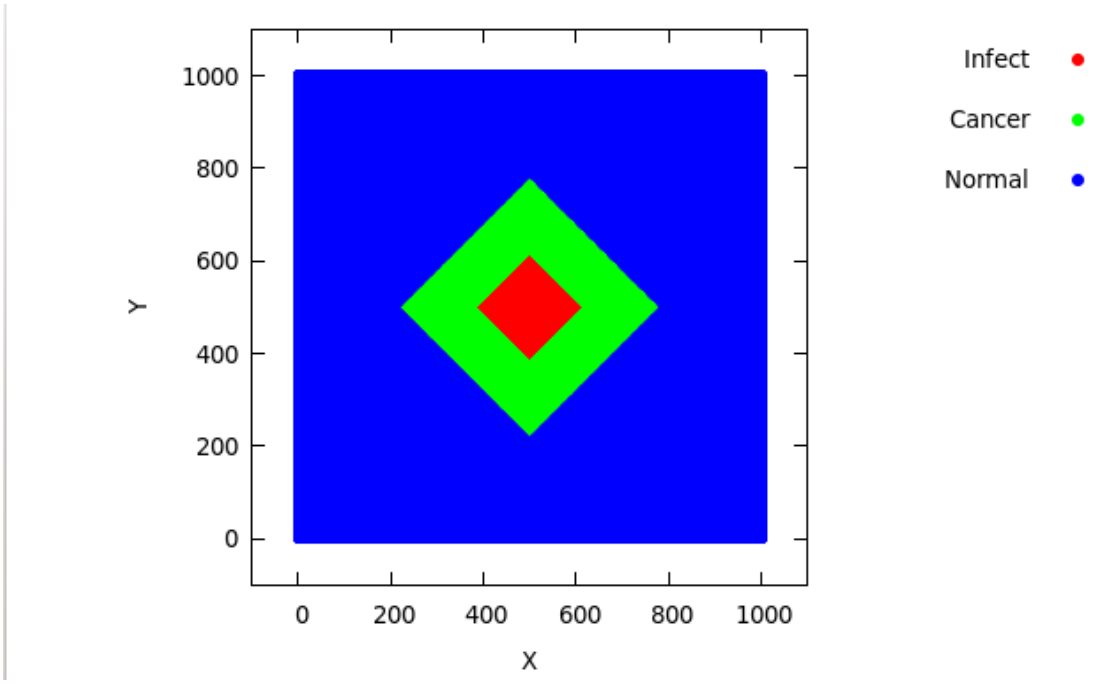


Figure 3: 2D plot of an initial condition.

The resulting graphic is shown in Figure 3.

Plotting Three Dimensions

Gnuplot can also be used to generate three dimensional projections of networks. In the following example, the simulator is run on a 10 by 10 by 10 grid with the center specified as the node at point (5,5,5). The -f

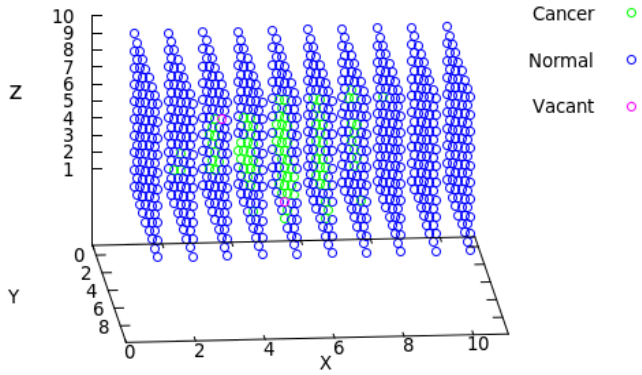


Figure 4: 3D plot of simulation 1 day after insertion of cancer.

switch specifies lattice state will be output to file and the -W switch specifies the program will wait for the return character after every output interval (0.5 by default).

```
./Cells -N 1000.00 -n 1.5 -C 3000.00 -c 1.5 -r 3 -R 74121 -s -f -W -E
networks/10x10x10.dat networks/10x10x10.net networks/10x10x10.cor
```

The program was interrupted at time 1.0 after inserting cancer and the following gnuplot script was called resulting in the plot shown in Figure 4.

```
gnuplot> reset
gnuplot> set size square
gnuplot> set xrange [0:11]
gnuplot> set yrange [0:11]
gnuplot> set zrange [0:11]
gnuplot> set xlabel “X”
gnuplot> set ylabel “Y”
gnuplot> set zlabel “Z”
gnuplot> set key outside
gnuplot> set key spacing 2
gnuplot> set view 113,5
gnuplot> splot "< awk '{if ($4 == 3) print $0;}' Lattice_State_.csv" \
gnuplot>                    title 'Infect' pt 7 ps 1,                 \
gnuplot>       "< awk '{if ($4 == 2) print $0;}' Lattice_State_.csv" \
gnuplot>                    title 'Cancer' pt 7 ps 1,                 \
gnuplot>       "< awk '{if ($4 == 1) print $0;}' Lattice_State_.csv" \
gnuplot>                    title 'Normal' pt 7 ps 1,                 \
gnuplot>       "< awk '{if ($4 == 0) print $0;}' Lattice_State_.csv" \
gnuplot>                    title 'Empty'  pt 7 ps 1
```

Figure 4 can be rotated and scaled in the gnuplot window and the reread command can be used to animate the graphic. To animate, use the --Watch command so that the simulator will stop every interval and wait for user input to continue. Split the above commands in two files. The first contains all the configuration commands. The second contains only the lines which make up the splot command followed by the reread command. At the gnuplot command prompt “load” the first file, then the second. Press [ctrl]+c to stop the reread command and be able to manipulate the image. Call – load “file2.p” – again to restart the animation.

Program Applications

Initial State Generated with Voronoi Tessellation from Confocal Microscopy

The optional 3rd (4th in three dimensions) value in the coordinate files allow the simulator to be started with arbitrary network states generated in external tools. For example, Chetan Offord created the following process that uses the concept of Voronoi tessellations to transform images of tumor and infected cells from a confocal microscope to a network state which the simulator can import. The process used to accomplish this is as follows.

1. Begin with an image from confocal microscopy where uninfected cancer cells express tdTomato, a red fluorescent protein (RFP), infected cancer cells express green fluorescent protein (GFP) and empty medium is seen as black (See Figure 5).
2. Process this image by converting the varying levels of GFP and RFP so that each pixel has a color of red, green or black (See Figure 6).
3. Using image dimensions and a measured average cell size, determine the number of cells (N) in an image of a cell population of equal dimensions.
4. Determine the minimum distance required between two cell centers, then randomly select N centroids

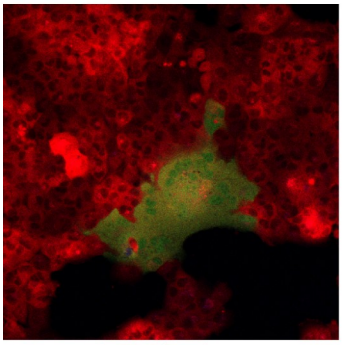


Figure 5: Confocal microscopy capture with cells expressing fluorescent proteins as in step 1.

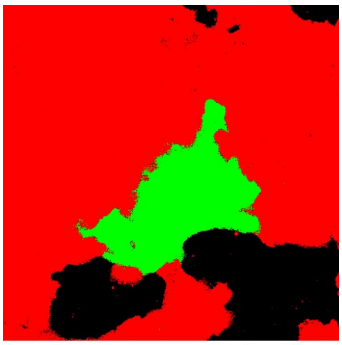


Figure 6: Figure 5 with colors adjusted as in step 2. Virus is concentrated in the center.

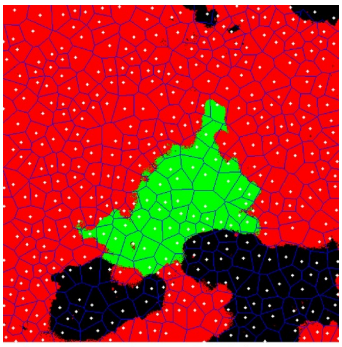


Figure 7: Figure 6 with Voronoi tessellation.

that respect this constraint.

5. Count how many red, green and black pixels are in each node's area. Use the largest count to determine node type. Record this type, along with the coordinates from step 4 in the `coordFILE` file. This last step is shown graphically in Figure 8.

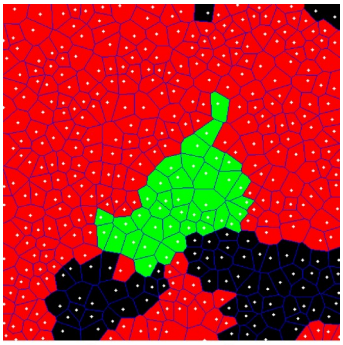


Figure 8: Figure 7 after adjustments described in step 6.

6. Note that this particular set of centroids was selected as the best amongst one thousand random centroid sets when optimizing for the minimum number of pixels changed between Figure 7 and Figure 8. Qualitative analysis between the best set of centroids and the worst showed a large difference in the number of nodes lying on the border of between cell types.
7. Create a Voronoi tessellation with these coordinates as inputs using an external tool such as R-programming's deldir package. So the resulting network better models reality, we removed edges from the network which connected nodes separated by a distance of four cell radii or more. These results are used to create the `netFILE` file. Figure 7 shows this tessellation and the centroids overlaid on the converted microscopy image.

Chetan Offord then used these files as the simulator's initial state for 100,000 simulations to determine which parameter set best fit in vitro population data. Best fit was determined by dividing pixel counts by average cell area. When looking at the images it was found that infected cells spread without cell division, a phenomenon which is not modeled. To adjust for the discrepancy Offord ran the microscopy images through an image processing script that looked for black pixels which turned green in one time step (15 minutes * 73 steps) without transitioning through red. Since virus-infected cells do not replicate to empty space, but infect neighboring tumor cells, this eliminated the spread of virus without affecting real growth.

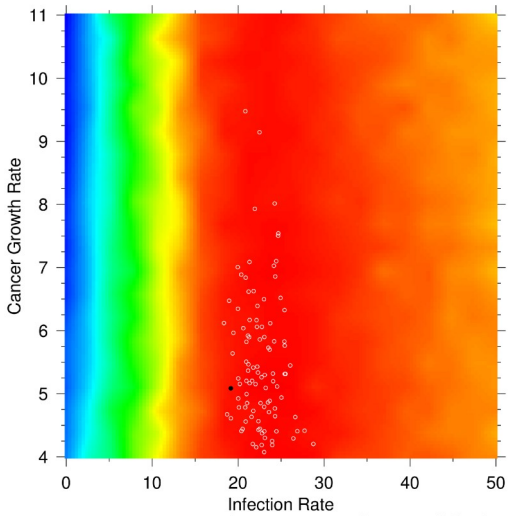


Figure 9: Heat map showing goodness of fit for proliferation parameters (C. Offord personal communication, .February 24, 2015).

In determining the ranges for rate parameters, Offord observed that no cells died in the in vitro experiment, so all death rates were fixed at zero simplifying the fitting process. The range for tumor replication rates was determined by looking at a small cancer

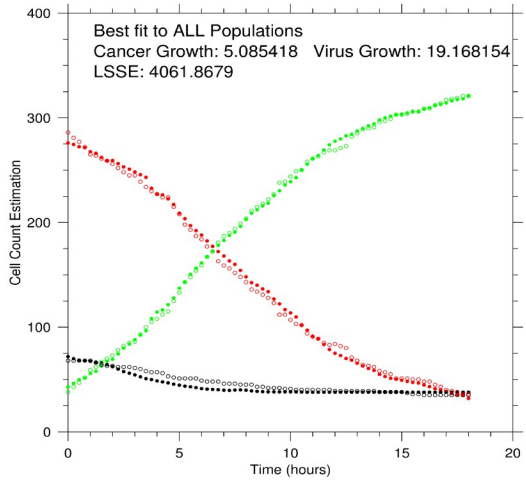


Figure 10: Population vs time for in vitro (solid) and simulation (hollow). The cancer population starts high and ends low; virus population starts low and ends high; empty space is relatively steady at a low level (C. Offord personal communication, .February 24, 2015).

cell colony in absence of virus. The range of virus infection rates was selected by assuming it was the same order of magnitude as the cancer replication rate and setting it to be large enough so fitting results ranged from very poor to excellent then back to poor. In Figure 9 empty white circles show the top 0.1% of parameter sets as determined by best fit to all three populations. The solid black circle in Figure 9 indicates the parameter set used in the simulation which produced the populations plotted in Figure 10. Fitting to just the infected population was also considered, and it was found that the best fit for all three populations was also the best fit to the infected population alone. This was true through the rest of the top 100 with the exception of a few parameter sets exchanging rank (personal communication, February 24, 2015).

Population-Time Comparison to Mean Field Equations

The graphics shown in Tables 5 through 8 show normal, cancer and virus populations as a function of time when simulated using Voronoi lattices and regular grids in both two and three dimensions. All four networks had 1,000,000 nodes. The two dimensional grid network had dimensions of 1,000x1,000 and the three dimensional grid had dimensions of 100x100x100. Voronoi networks were created as described in the section Initial State Generated with Voronoi Tessellation from Confocal Microscopy only with 1,000,000 nodes. None of the four networks had a wrap around topology. Rates for this section were chosen by Offord so that each of the four simulations would end in a three population equilibrium (personal communication, February 4 2015). These rates are

	N	**C**	**V**
Proliferation	0.5	1.0	1.2
Death	0.2	0.1	0.1
Population	90%	9%	1%

Table 3: Simulation parameters.

shown in Table 3. Additionally, simulations were started with a full network where 90% of the nodes were normal cells, 9% were cancer cells and 1% were virus-infected. cancer cells. The virus-infected cells were centered in the network.

If the simulator were allowed to run on an infinitely large and complete (every node neighbors every other node) network, the simulations would be stochastically identical to the mean field equations shown in Figure 11. For comparison to the simulator, the population vs time results from these formulas are also shown in Tables 5 and 7.

Population			***Replication***	***Death***	***Infection***
Normal	$\frac{du_N}{dt}$	$=$	$\lambda_N u_N(1-u_N-u_C-u_V)$	$-\delta_N u_N$	
Cancer	$\frac{du_C}{dt}$	$=$	$\lambda_C u_C(1-u_N-u_C-u_V)$	$-\delta_C u_C$	$-\lambda_V u_C u_V$
Infected	$\frac{du_V}{dt}$	$=$		$-\delta_V u_V$	$+\lambda_V u_C u_V$

Figure 11: Mean Field Formulas. Variables: u_i:percent of population i; δ_i: death rate of population i; λ_i: proliferation rate of population i; i: normal, cancer, virus-infected.

The graphs shown in Table 5 show the population as a function of time of simulations run in two dimensions on a regular grid and on a Voronoi tessellation. Also shown is the population vs time curves given by the mean field equations in Figure 11. As can be seen, all three models result in a three population equilibrium. The plots shown in Table 6 illustrate the spatial output of the two computer simulations. Interesting to note is the difference in viral progression at time 150. Here, in the simulation run on a Voronoi network, the virus has reached the edge of the tumor. However in the simulation run on the regular grid, virus does not reach an edge until time 250. This can also be seen in the

population vs time plots from Table 5 where the virus and cancer populations intersect near time 150 and quickly stabilize in the Voronoi tessellation and at time 250 in the regular grid. The non-edge nodes in Voronoi network have, on average, 6.0 neighbors. On the other hand, non-edge nodes in the regular, two dimensional grid have exactly four neighbors. This greater connectivity allows proliferation to spread more quickly through the network.

The same simulations were repeated in three dimensional regular and Voronoi networks. Table 7 shows population plots for these networks with curves from the mean field equations reproduced for comparison. Table 8 shows three subsets of the networks at various time points. Subsetting was performed by plotting all nodes with a Z value of 0, nodes with Z=25 and nodes with Z=50. In the case of the Voronoi network, centroids were selected pseudo randomly with real values between 0 and 1,000. In the images of the three dimensional Voronoi networks, each slice 0 contains centroids with Z values ranging from 0 to 24, slice 25 contains centroids with Z values from 250 to 274 and slice 50 contains centroids with Z values from 500 to 524.

The effect connectivity plays on population spread can again be seen in three dimensions. The two dimensional Voronoi tessellation with an average of 6 neighbors equilibrates around a time of 6,000 as seen in Table 5. The three dimensional regular grid also has 6

Network Type	Average Number of Neighbors	Time to Equilibrium
2D Grid	4	6,000
2D Voronoi	6	5,000
3D Grid	6	250
3D Voronoi	16	150

Table 4: Time to equilibrium compared to number of neighbors and dimensions.

neighbors per non-edge node but reaches equilibrium around a time of 250, more than an order of magnitude earlier than in two dimensions. The three dimensional Voronoi network, where non-edge nodes have 16 neighbors on average, has an equilibrium time of approximately 150. The collection of this data in Table 4 shows that while the number of neighbors in a network affects the time to equilibrium, its effect is small compared with the order of magnitude change seen when moving between two dimensional and three dimensional networks with the same neighborhood size.

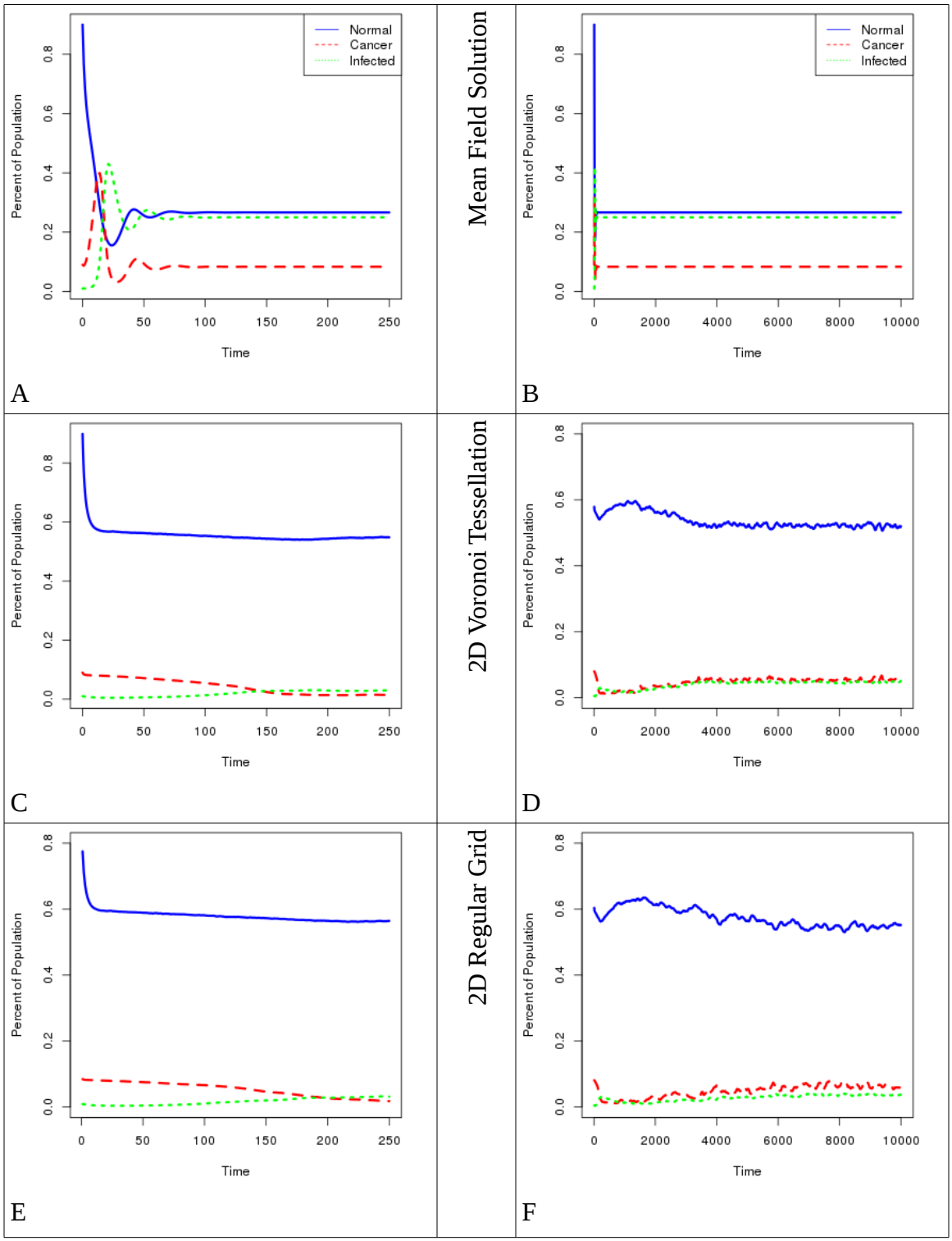
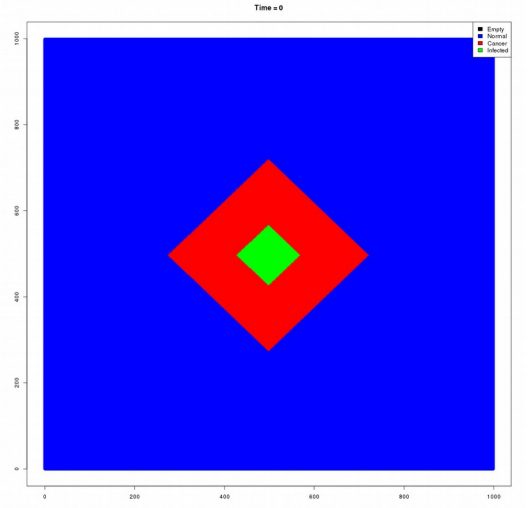
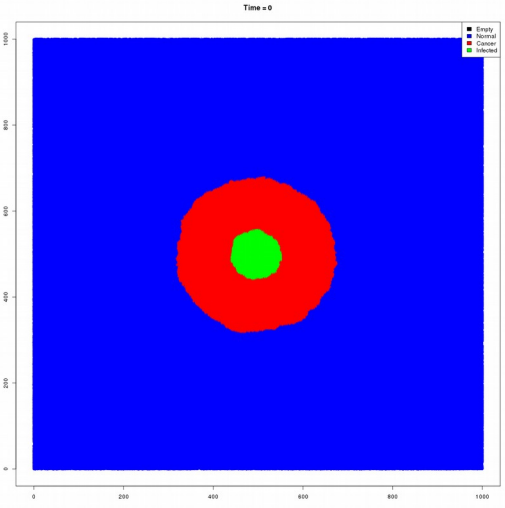
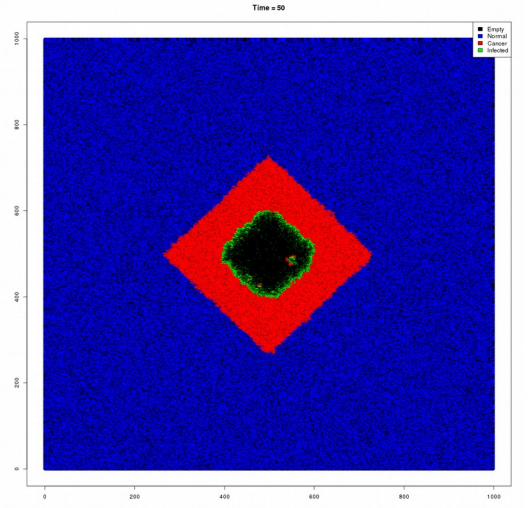
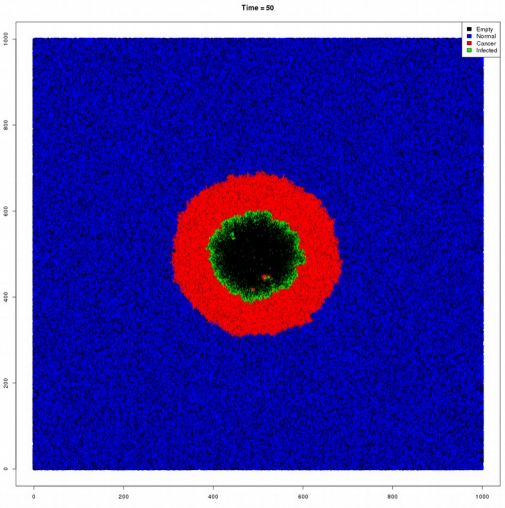
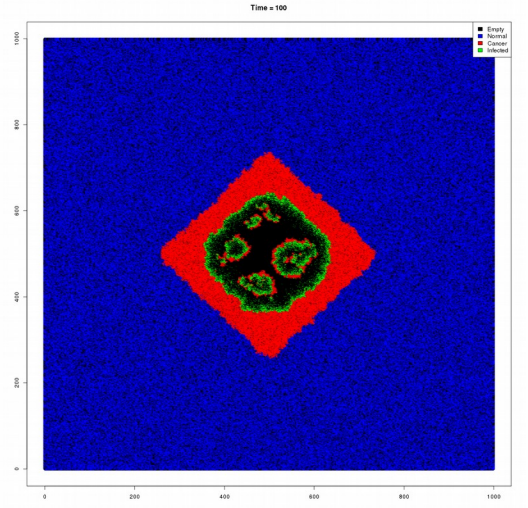
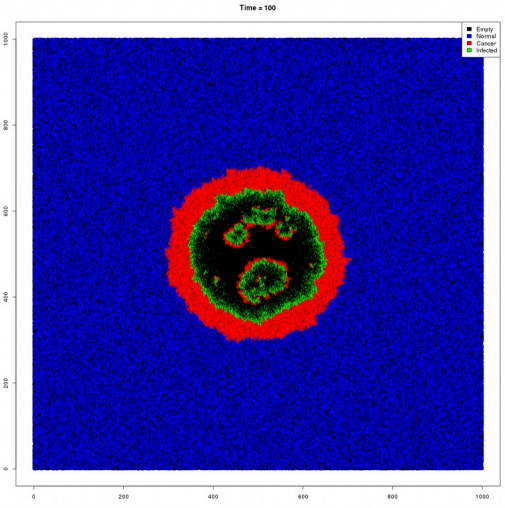


Table 5: Population vs time plots from 2D spatial simulations and the mean field solution.

2D Regular Grid	Time	2D Voronoi Tessellation
	0	
	50	
	100	

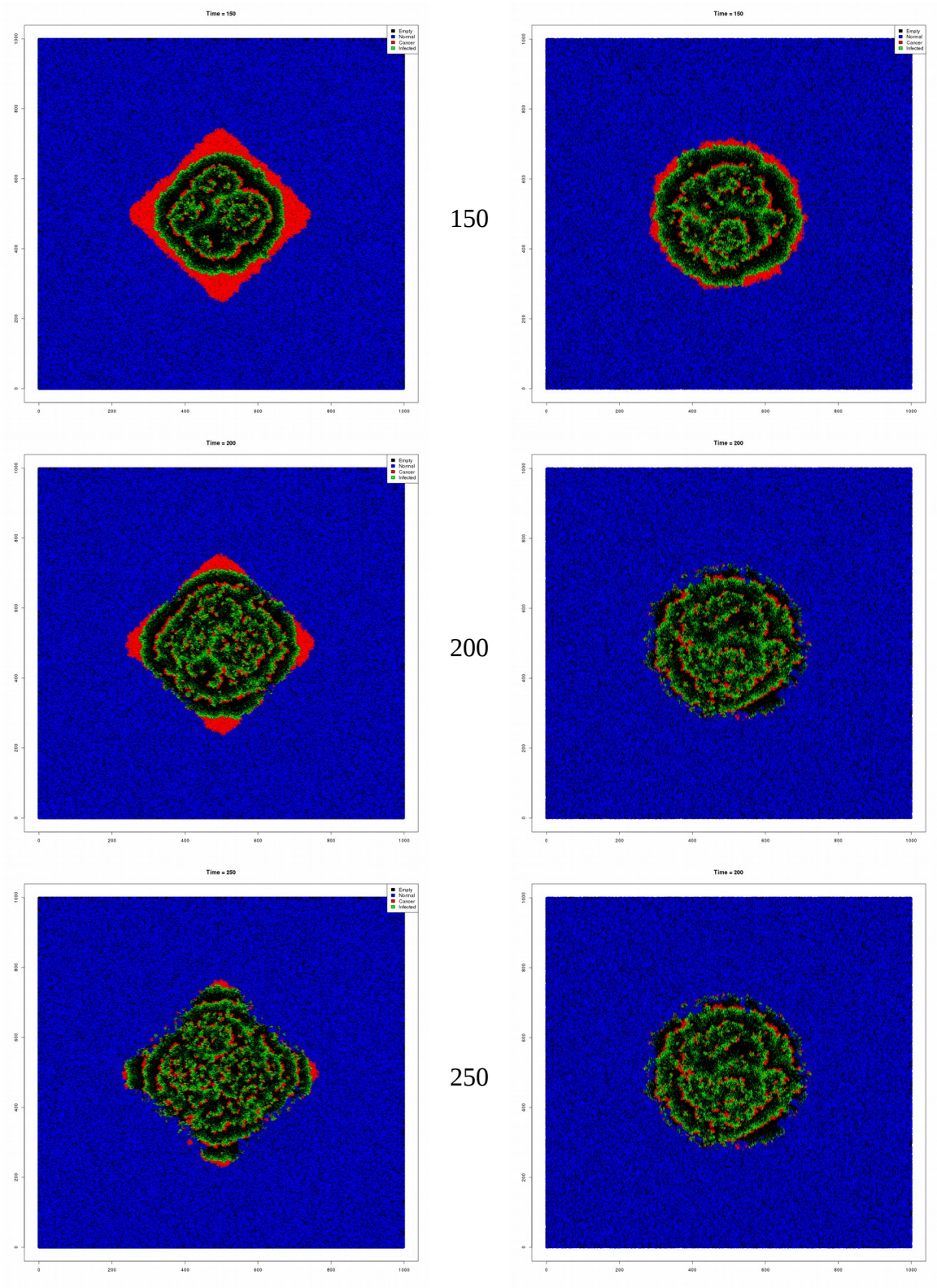


Table 6: Spatial states from 2D simulations.

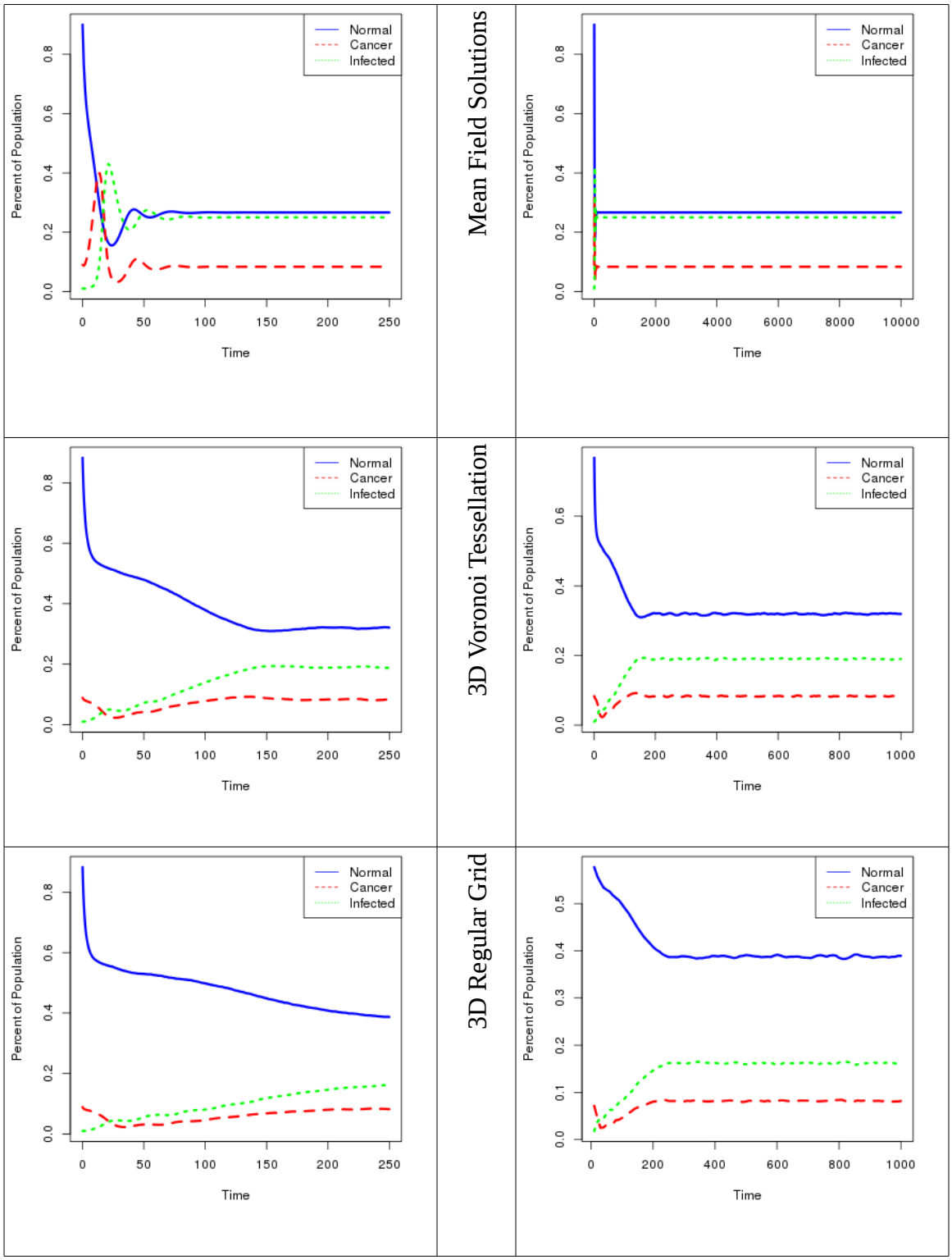
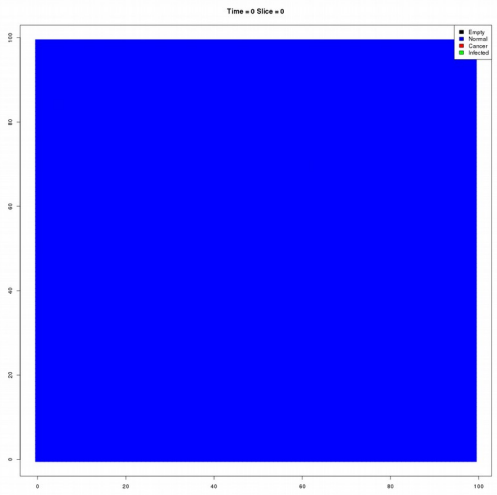
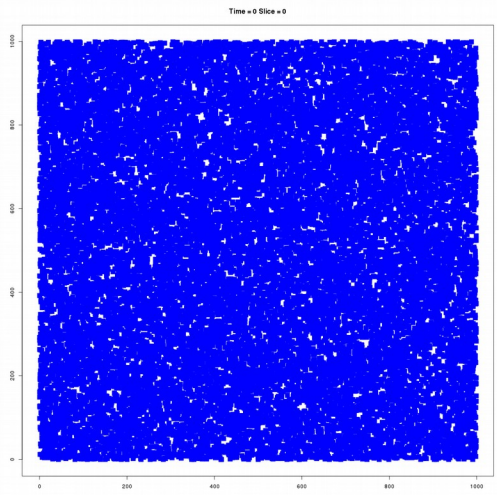
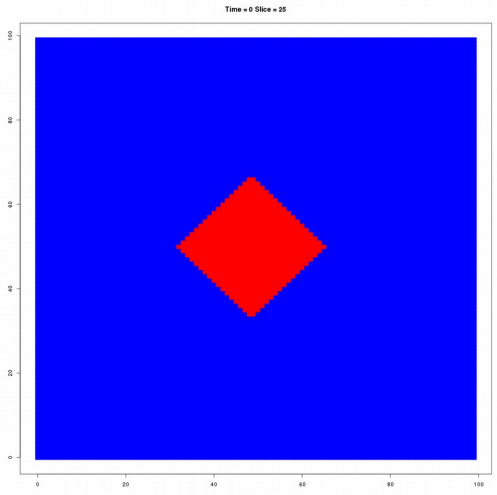
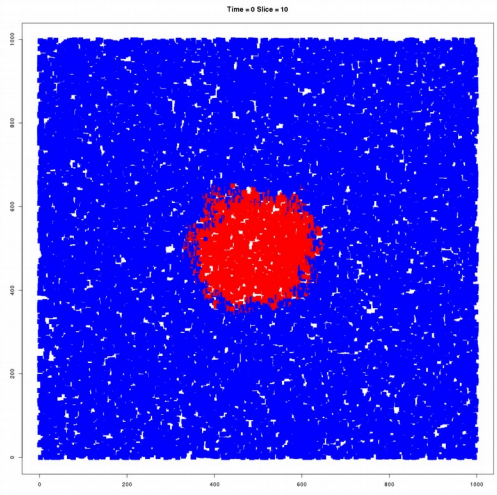
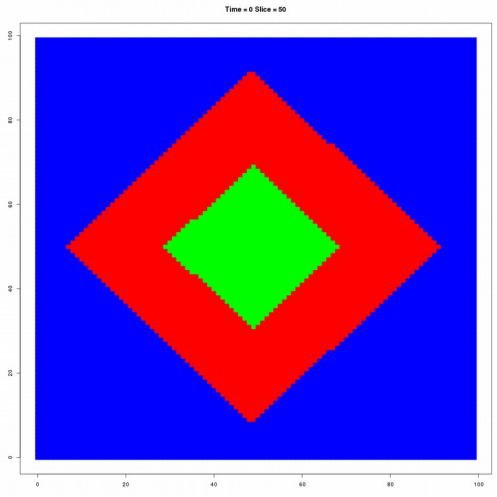
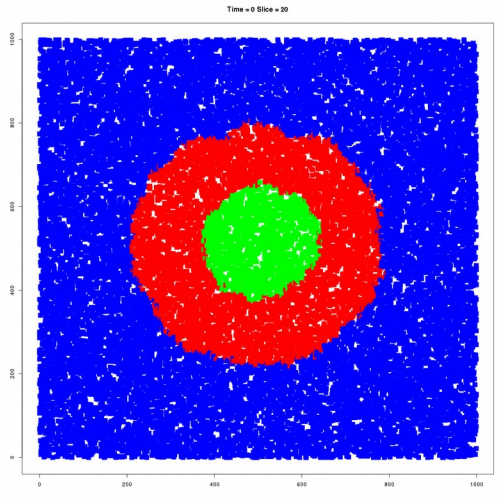
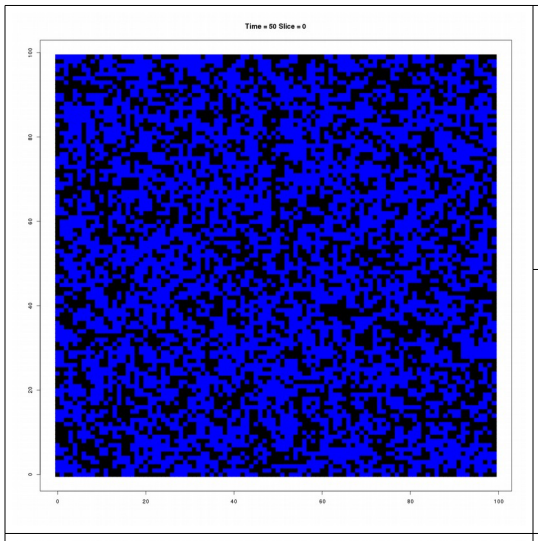


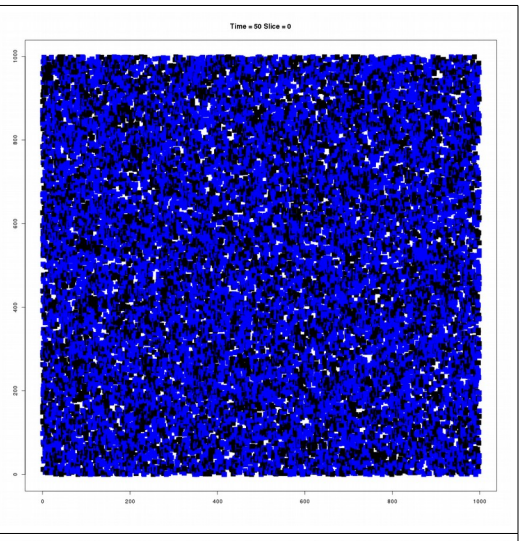
Table 7: 3D population vs time plots and mean field solution. Note the scales of the y-axis are not all equivalent.

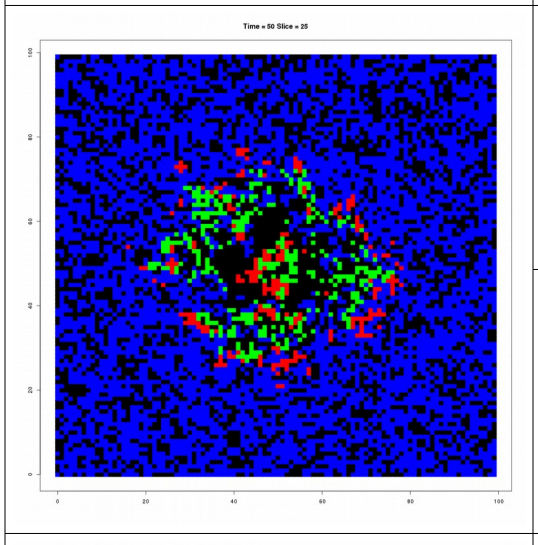
3D Regular Grid		3D Voronoi Tessellation
	**Time:** 0 **Slice:** 50 / 100	
	**Time:** 0 **Slice:** 25 / 100	
	**Time:** 0 **Slice:** 50 / 100	



Time: 50

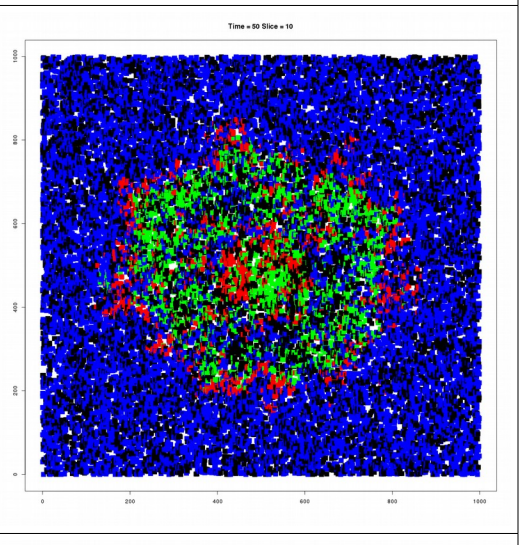
Slice: 0 / 100

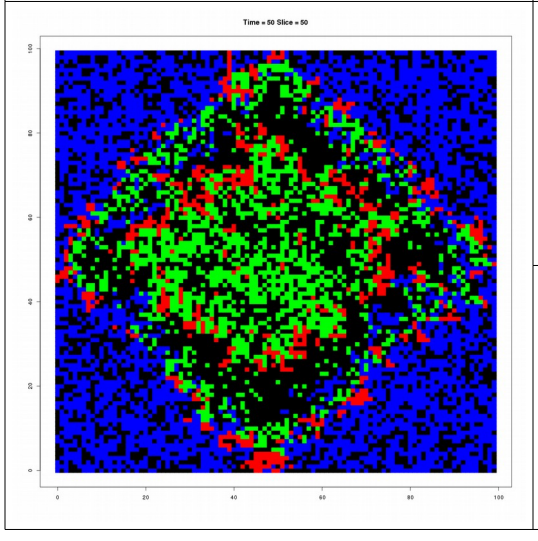




Time: 50

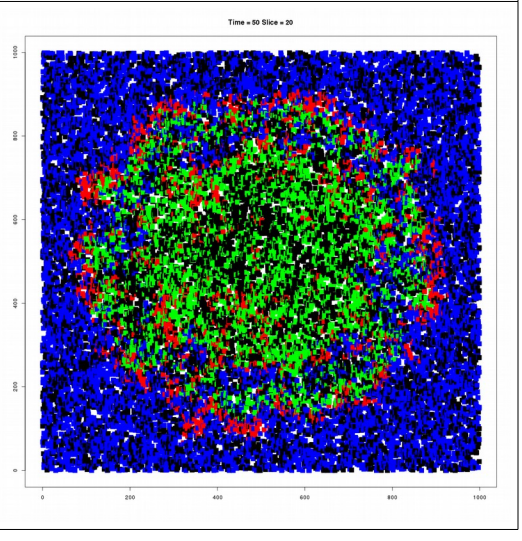
Slice: 25 / 100

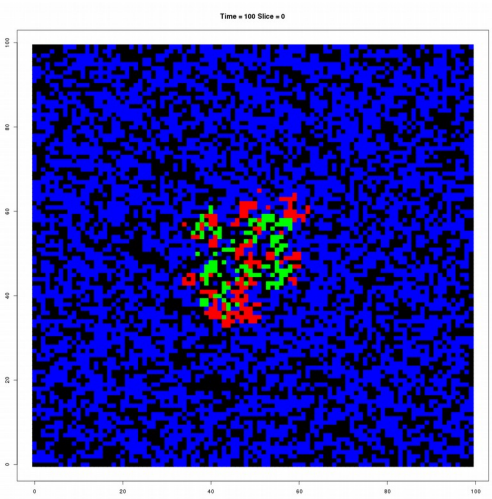




Time: 50

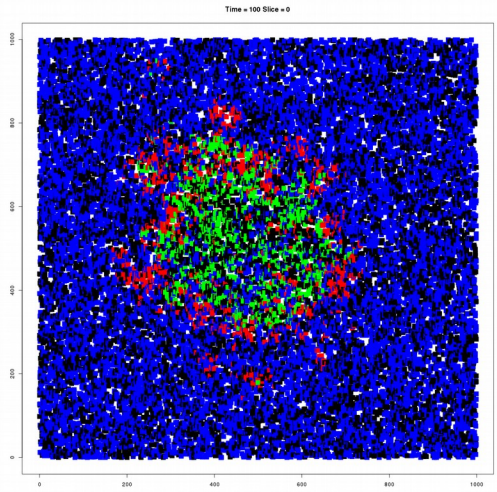
Slice: 50 / 100

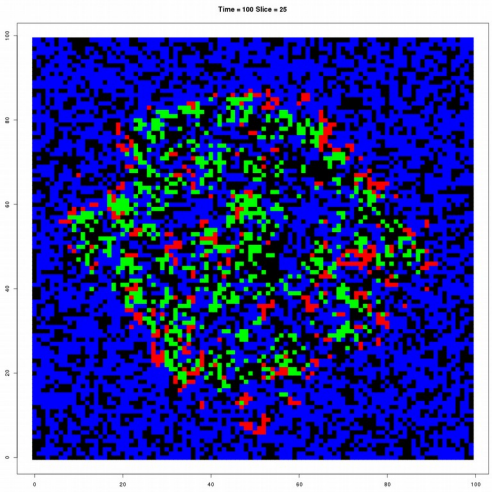




Time: 100

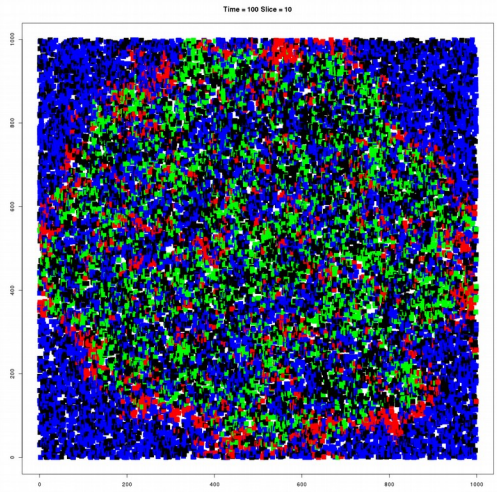
Slice: 0 / 100

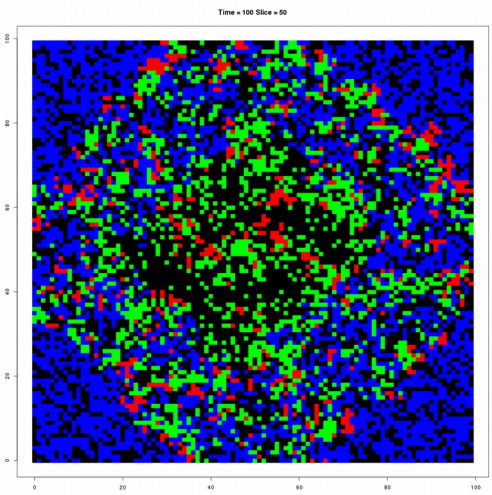




Time: 100

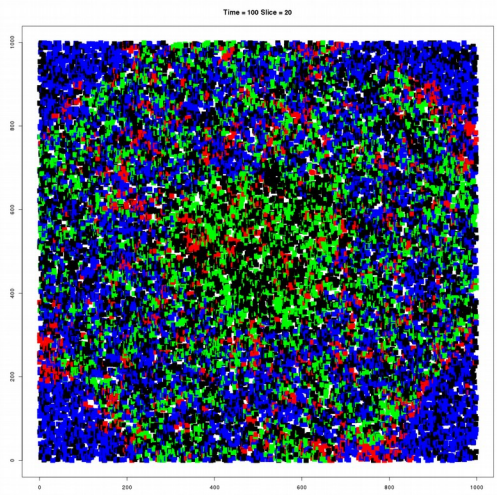
Slice: 25 / 100

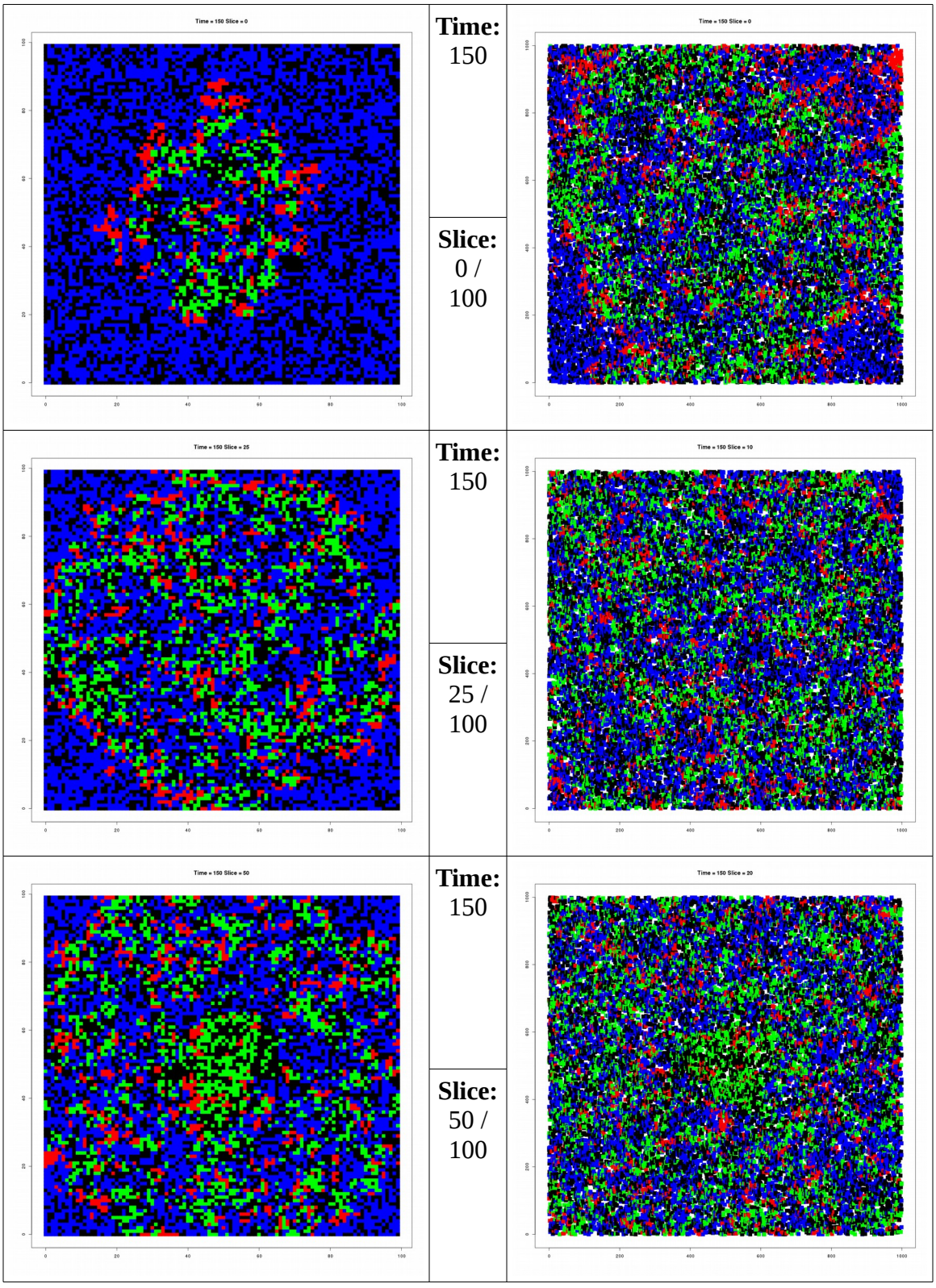


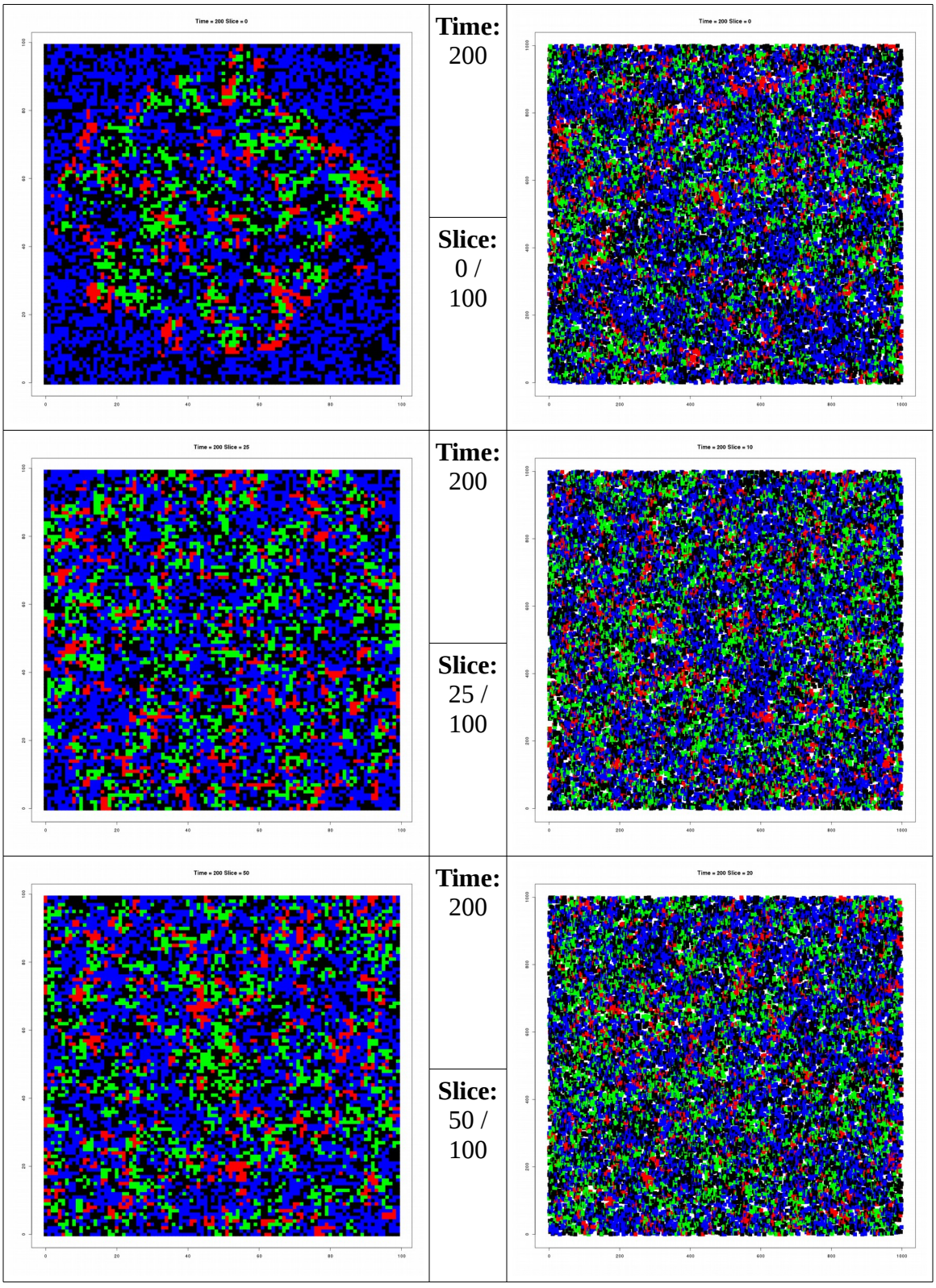


Time: 100

Slice: 50 / 100




Time = 150 Slice = 0
Time: 150
Slice: 0 / 100
Time = 150 Slice = 0
Time = 150 Slice = 25
Time: 150
Slice: 25 / 100
Time = 150 Slice = 10
Time = 150 Slice = 50
Time: 150
Slice: 50 / 100
Time = 150 Slice = 20


Time = 200 Slice = 0
Time: 200
Slice: 0 / 100
Time = 200 Slice = 0
Time = 200 Slice = 25
Time: 200
Slice: 25 / 100
Time = 200 Slice = 10
Time = 200 Slice = 50
Time: 200
Slice: 50 / 100
Time = 200 Slice = 20

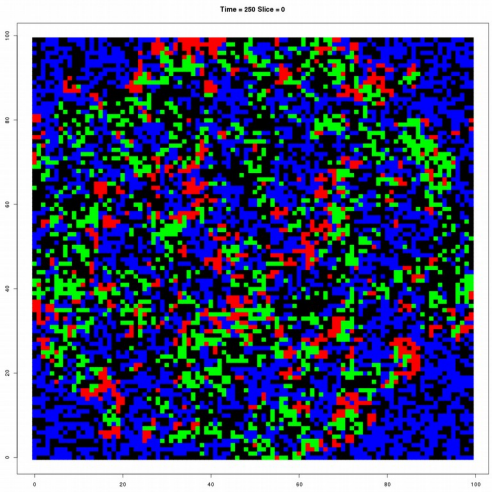
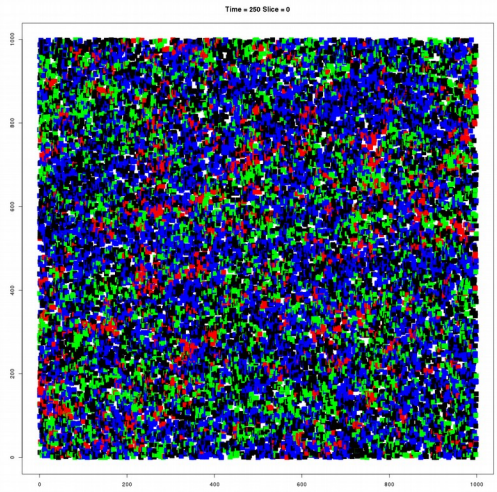
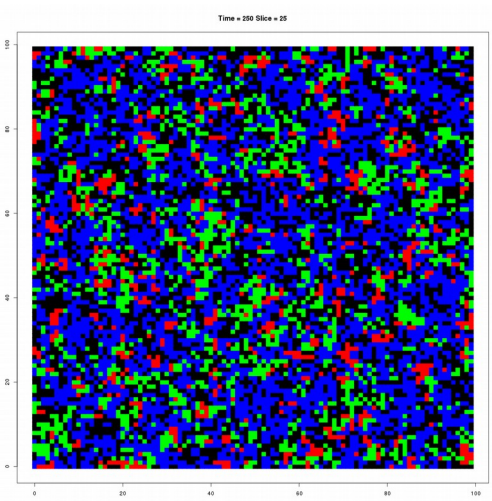
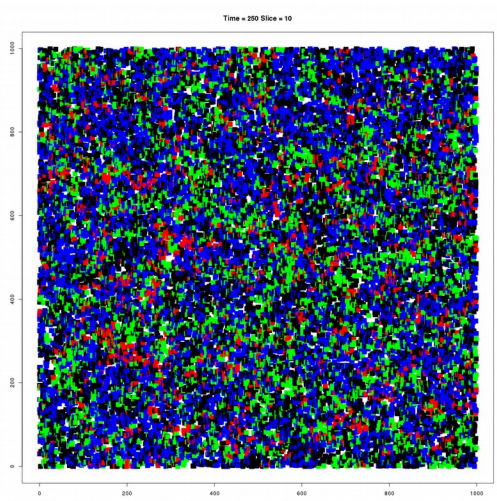
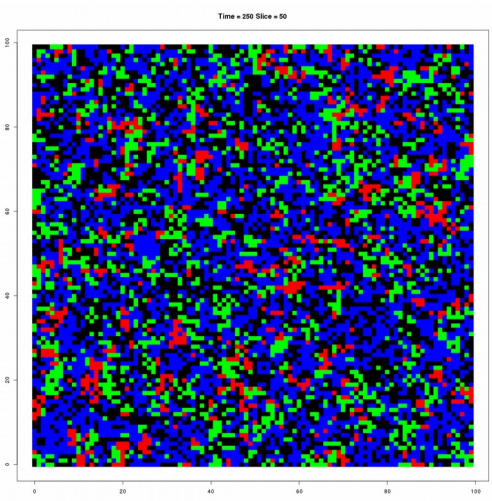
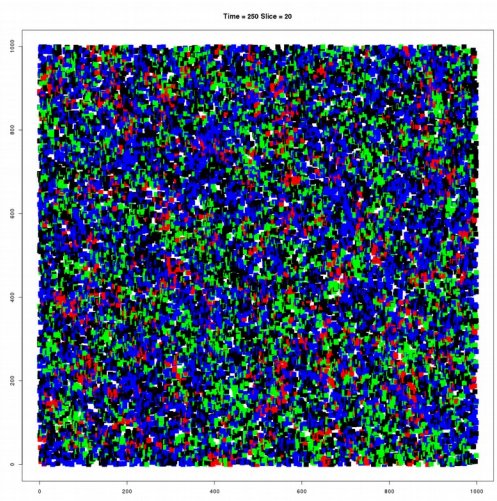
	**Time:** 250 **Slice:** 0 / 100	
	**Time**: 250 **Slice:** 25 / 100	
	**Time:** 250 **Slice:** 50 / 100	

Table 8: Spatial states from 3D simulations.

Equilibrium Analysis

Simulations and mean field can result in one of five equilibriums:

1. Virus Extinction: Virus population has reached zero. In this usage, virus extinction implies that normal cells are also extinct. Since cancer growth is larger than normal growth in these experiments, the only non-empty nodes are cancerous.
2. Cancer Extinction: Once cancer is extinct, virus-infected cells will soon follow if their death rate is greater than zero. This is the only successful result.
3. Three Population Equilibrium: All three cell types are present and have a stable population size.
4. Cancer/Virus Equilibrium: Normal Cells die out leaving cancer cells and virus-infected cells with stable population sizes.
5. Trivial Equilibrium: All populations die out. This type is not considered in this section.

In order to determine what virus parameters are most likely to produce a successful result, Chetan Offord ran 14,000 simulations while varying the infection and death rates of the infected population. Three dimensional simulations were run with parameters he determined by fitting the ratio of replication to death rates for the mean field equations presented in Figure 11 to data from observed tumor growth in live mice (Table 9). The use of ratios explains the exact number for replication rates. Infection rates were varied between 0 and 100 while viral death rates varied between 0 and 15. Offord selected these ranges to cover the range of interesting results. Simulations proceeded as

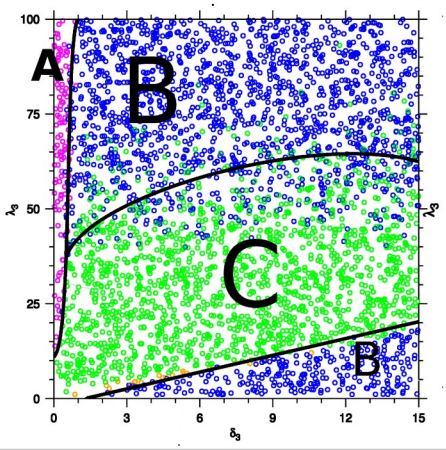
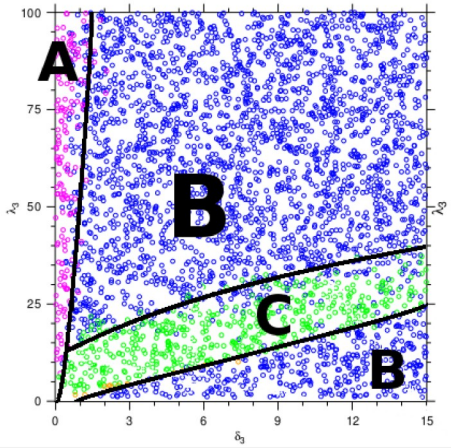
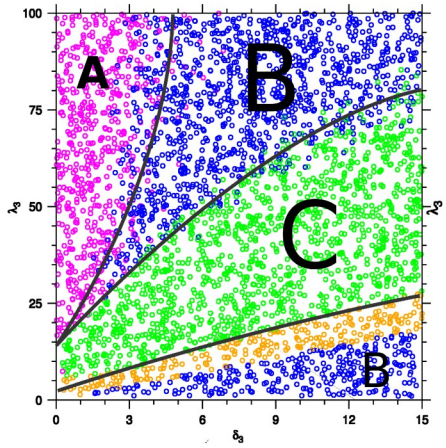
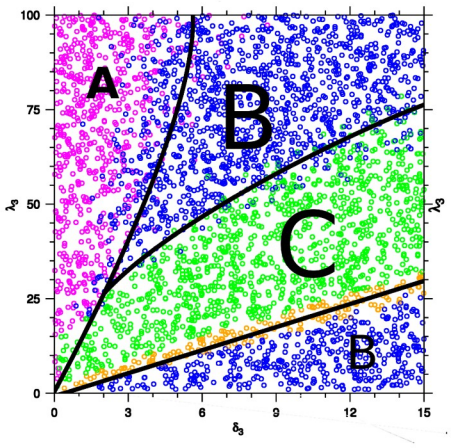
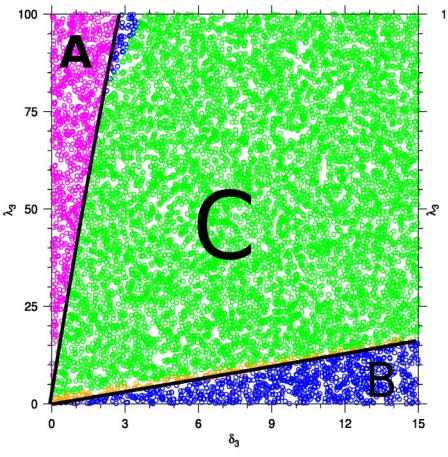
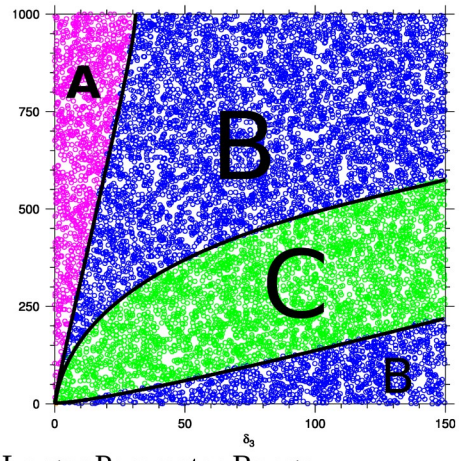
	Normal	**Cancer**
2D Replication	1,000	20,000
3D Replication	1,000	10,000
2D Death	2.000	2.000
3D Death	0.241	0.241

Table 9: Parameters for equilibrium analysis.

described in Figure 1. A completely normal population was simulated until the population stabilized. Five hundred cancer nodes were then inserted into the center of the network. Seven days later, five percent of cancer cells at the center of the tumor were infected. These 14,000 simulations were split evenly among the same four network types described in the section Population-Time Comparison to Mean Field Equations; specifically, two dimensional regular grid, two dimensional Voronoi tessellation, three dimensional regular grid and three dimensional Voronoi tessellation. Simulations were allowed to run until either the tumor populations died out or a time of 1,000 days was reached, which ever occurred first. After running 7,000 three dimensional simulations, the results were plotted and are shown in the first row of Table 10. When running simulations in two dimensions Offord found that the same graphical patterns appeared but were scaled differently. For the two dimensional plots shown in row two of Table 10, Offord chose parameters to provide a similar field of view. Shown in row three are the results of an equilibrium analysis using the three dimensional parameters in the mean field equations and again with the parameter ranges scaled by an order of magnitude. The mean field solutions were modified so that if a population dropped below one millionth of the whole, then that population would die out. This prevented populations that would be less than one cell in a million node network from being able to reestablish themselves (personal communication, February 24, 2015).

In all images there is a narrow region of cancer extinction along the left (A) with a field of virus extinction to its right (B). Into region B, reaches a branch of three population equilibrium (C). What is hard to see in some of the images, and not marked in

any is a region of cancer/virus equilibrium below region C. Not shown is the fact that there exists a viral death rate large enough that all viral growth rates result in virus extinction.

Voronoi	Dimension	Grid
	3D	
	2D	
	Mean Field	

Larger Parameter Range

Table 10: Equilibrium analysis. The regions A, B and C correspond to all normal, all cancer and three population equilibrium respectively (C. Offord personal communication, .February 24, 2015). Except for the the Mean Field figure on the right, the parameter ranges are the same in all figures.

Concluding Remarks

We set out to create a program that would allow for computer simulation of oncolytic virotherapy. We required that the program be able to run on arbitrary networks and produce both spatial and temporal data. That program has been documented above and it's code follows. In addition to meeting the network and output requirements, the program also provides a number of routines to artificially change the network state at various points of simulation. Examples of these state changes include the insertion of cancer after the stabilization of normal populations and the substitution of virus-infected cells for cancer cells after a specified number of days. Program design choices (i.e. descriptive function and variable names) resulted in a code base that is maintainable and extensible. This will allow other researchers to use this program in their own work.

While built within current understanding of viral therapy, there is still work that needs to be done to determine the predictive value of such simulators and the accuracy of the model. Systematic studies of various initial spatial distributions of cancer cells and infected cancer cells should be performed, looking for insights into spatial population dynamics. If interesting results are revealed they may lead to a better understanding of viral therapy. Also, the predictive value of the simulator needs to be determined. Particularly, is there sufficient likelihood that a successful computer simulation corresponds to a successful in vivo or in vitro experiment? If so, the simulator can be used to focus expensive biology lab resources on experiments that are most likely to produce successful outcomes.

The promise of oncolytic virotherapy providing a cure for cancer that is accessible to the general population is exciting. I hope that this work has a meaningful impact to that goal.

Simulator Code

Cells.c

```
#include "Cells.h"

struct lattice_t * lttc = NULL;
char *program_name = NULL;
int print_Stats = 0;
int print_Summary = 0;
int print_Grid = 0;
int print_Tumor_Sizes = 0;
int print_Watch = 0;
char *print_File;
int print_Files = 0;
int verify_State = 0;
float percent_Infected = 0.0;
float percent_Interior = 100;
int   do_Equalization = 0;
enum infection_t infect_Type = NONE;
FILE *fpp, *fpl;

int    t_Max_Simulation = 75;
int    t_Infect         =  7;
int    rng_Seed = 11;
float  output_Interval=.5; /* frequency of lattice output */
float  rates[N_CELL_TYPES][N_ACTIONS] = { { 0, 0 },
                                         { 0, 0 },
/*  Virus Interior */                    { 0, 0 },
/*  Virus Exterior */                    { 0, 0 } };

/*
 * CenterOfMass() accepts a pointer to an array of N "node_t nodes".
 * Returns NULL if dimensions don't match or are != 2 || 3
 * finds the average of each node's position (nodes[i]->pos)
coordinates.
 * Stores the coordinates in a Point
 * Returns the address of this point.
 */
Point* CenterOfMass( struct node_t** nodes, const unsigned long N )
{
   unsigned long nodeIt;
   int n_dims = nodes[0]->pos->N;
   int dimIt;
   double *temp = calloc( n_dims, sizeof(double) );

   if( n_dims != 2 && n_dims != 3 )
      return NULL;

   for( nodeIt=0; nodeIt < N; nodeIt++ ) {
      for( dimIt=0; dimIt < n_dims; dimIt++) {
         temp[ dimIt ] += nodes[nodeIt]->pos->data[dimIt];
      }
```

```
   }
   for( dimIt=0; dimIt < n_dims; dimIt++) {
      temp[dimIt] /= N;
   }

   Point* center = NULL;
   switch( n_dims ) {
   case 2:
      center = NewPoint( n_dims, temp[0], temp[1] );
      break;
   case 3:
      center = NewPoint( n_dims, temp[0], temp[1], temp[2] );
      break;
   default:
      break;
   }

   free( temp );

   return center;
} /* End of CenterOfMass() */

struct node_t* FindNodeNear(Point *point,
                            struct node_t ** node_list,
                           unsigned long N )
{
/* Return address of node in node_list with shortest euclidian distance
   to point */
   unsigned long nodeIt;
   unsigned long min_Idx = 0;
   float dx=0;
   float dy=0;
   float dz=0;
   float *distance = malloc( N * sizeof( *distance) );

   for( nodeIt = 0; nodeIt < N; nodeIt++){
      dx = point->data[0] - node_list[nodeIt]->pos->data[0];
      dy = point->data[1] - node_list[nodeIt]->pos->data[1];
      if( point->N == 3 ){
         dz = point->data[2] - node_list[nodeIt]->pos->data[2];
      }
      distance[nodeIt] = (dx*dx + dy*dy + dz*dz);
   }

   for( nodeIt = 1; nodeIt < N; nodeIt++){
      if( distance[nodeIt] < distance[min_Idx] )
         min_Idx = nodeIt;
   }

   free(distance);

   return node_list[min_Idx];
} /* End of FindNodeNear() */
```

```
/* Perform a Breadth First Search of all nodes connected to node_u by
      network_Type neighbors.
   Assume connected nodes have visited==NOTSEEN && distance==INT_MAX
   Sets node->visited to VISITED and node->distance to shortest number
      of links between node_u and node.
   Returns the number of nodes visited */
int BFS (struct node_t* node_u,       /* Node to begin search from */
         enum nodeType_t network_Type,/* Only visit type nbrs */
         int max_Count,               /*Stop after visiting max_Count*/
         enum nodeType_t new_Type )   /* Change nodes to new_Type */ {
   int count=0;
   struct node_t* node_v;
   struct Queue * const Q = NewQueue( 8 );
   node_u->visited = SEEN;
   node_u->distance = 0;
   Enqueue( Q, node_u );

   while( (node_u = Dequeue(Q)) ){
      if( count >= max_Count ){
         continue;
      }
      count++;
      if( node_u->type != new_Type )
         ChangeNodeType( node_u, new_Type );
      node_u->visited = VISITED;
      int nbrIt;
      for( nbrIt = 0; nbrIt < node_u->n_Neighbors; nbrIt++ ) {
         node_v = node_u->neighbors[nbrIt];
         if( node_v->visited == NOTSEEN && node_v->type == network_Type
) {
            node_v->visited = SEEN;
            node_v->distance = node_u->distance+1;
            if( Enqueue( Q, node_v ) == 0 )
               printf(" Problem growing Queue\n" );
         }
      }
   }
   DeleteQueue(Q);
   return count;
}


/* NextNetwork() returns first of "N" nodes in "nodes" that has
     node->visited set to NOTSEEN or NULL if none are found.
   Used along with BFS() to find disconnected portions of a network */
struct node_t* NextNetwork( struct node_t** nodes, unsigned long N ){
   int nodeIt;
   for( nodeIt = 0; nodeIt < N; nodeIt++ ) {
      if( nodes[nodeIt]->visited == NOTSEEN ){
         return nodes[nodeIt];
      }
   }
   return NULL;
}
```

```
/* GetNetworkDetails() ********************************************
   Resets node->visited & node->distance for all cancer nodes in lttc.
   Returns pointer to newly allocated array that needs to be free()d.
   The first element of returned array is the number of tumors N.
   The second to Nth+1 element indicate the sizes of these tumors.
   Elements aren't sorted except by chance
*****************************************************************/
unsigned long* GetNetworkDetails( struct node_t ** nodes, unsigned long
N )
{
   struct node_t *node_u;
   int nodeIt;
   int array_Size = 4;
   unsigned long* sizes = calloc( array_Size, sizeof(*sizes) );
   unsigned long* temp;
   if( sizes == NULL ) {
      fprintf(stderr, "Memory error while calculating tumor sizes.\n");
      return sizes;
   }
   if( N == 0 ){
      return sizes;
   }

   for( nodeIt = 0; nodeIt < N; nodeIt++ ){
      nodes[nodeIt]->visited = NOTSEEN;
      nodes[nodeIt]->distance = INT_MAX;
   }

   node_u = nodes[0];
   while( node_u ) {
      sizes[0] = sizes[0] + 1;
      if( sizes[0] >= array_Size-1 ){
         temp = realloc(sizes, array_Size*2 * sizeof(*temp) );
         if( temp == NULL ){
            fprintf(stderr, "Memory error calculating tumor sizes.\n");
            return sizes;
         }
         sizes = temp;
         array_Size *= 2;
      }
      sizes[ sizes[0] ] = BFS(node_u, nodes[0]->type,
                        INT_MAX, nodes[0]->type);
      node_u = NextNetwork( nodes, N );
   }

   return sizes;
}

float argtof( const char *nptr ) {
/* Checks for invalid values from strtof() */
   float num;
   num = strtof( nptr, NULL );
```

```
   if( !isfinite(num) || num != num ) {
      fprintf( stderr, "Argument is inf or nan. Using default of 0\n" );
      num = 0;
   }

   return num;
}/* End of argtof() */


/** LinReg() ********************************************************/
/*  Calculate slope of the N points (Xi, Yi) using Linear Regression.*/
/*  If intercept isn't NULL, that is calculated as well.            */
/*  Formulas used are those derived in "Mathematical Statistics with */
/*      Applications 6th Edition" by Wackerly on page 538.          */
/********************************************************************/
void LinReg( float const * const X,
             float const * const Y,
         long unsigned int const N,
         float * slope,
         float * intercept ) {
  int It;
  float Sxy=0;
  float Sx =0;
  float Sy =0;
  float Sxx=0;

  for(It = 0; It < N; It++) {
    Sxy += X[It]*Y[It];
    Sx  += X[It];
    Sy  +=       Y[It];
    Sxx += X[It]*X[It];
  }

  *slope = (Sxy - Sx*Sy/N)/(Sxx-Sx*Sx/N);
  if( intercept != NULL )
    *intercept = Sy/N - *slope * Sx/N;

  return;
}
```

```
/** simulate() ************************************************
*  Runs one step of the simulation
*  Expects global struct lattice_t lttc to be in a consistant state
*  Outputs data if indicated by global print_* variables
**********************************************************************/
int Simulate( ) {
   static float output_Timer = 0;
   static int count = 0;
   if( lttc->time == 0 ) {
      output_Timer = output_Interval;
   }

   const struct event_t e_t = DetermineEventType( );
   if( e_t.node_Type == VACANT )
   {
      fprintf(stderr, "Event Type is Vacant?\n");
      return -1;
   }
   struct node_t * const event_Cell = DetermineEventRecipient( e_t );

   if( event_Cell == NULL ) {
      fprintf(stderr, "Failed to select a cell to Grow or Kill\n");
      return -1;
   }

   switch ( e_t.cell_Action ) {
   case KILL:
      ChangeNodeType( event_Cell, VACANT);
      break;
   case GROW:
      ChangeNodeType( event_Cell, e_t.node_Type);
      break;
   default:
      fprintf(stderr, "e_t.cell_Action isn't KILL or GROW?\n");
      fprintf(stderr, "  Continuing w/o taking action or updating
time.\n");
      return -1;
   }
```

```
   /* Output Statistics & Info */
   if( lttc->time > output_Timer ) {
      int newlines=0;
      Point* center = NULL;
      output_Timer += output_Interval;

      if( print_Files ){
/*
Files being printed contain rng_Seed & the number of cancer cells at
injection followed by a list of space separated quantities indicating
the number of Cancer & Infected cells at the intervals specified below.
 vpts.dat (output_Interval beginning at 1 ending at 126) &&
 vlpt.dat (Day 0 + 5th output_Interval beginning at 1)
*/
     unsigned long int n_Tumor = lttc->n_Cells[CANCER]     +
                  lttc->n_Cells[INFECT_INT] +
                  lttc->n_Cells[INFECT_EXT] ;
         if( count == 5 ){
            count = 0;
            fprintf(fpl," %lu ", n_Tumor );
         }
         if( lttc->time < 126 ) {
            fprintf(fpp," %lu ", n_Tumor );
         }
         count++;
      }


      if( print_Tumor_Sizes && lttc->n_Cells[CANCER]){
         if( !newlines ) {
            printf("\n\nTime: %5.2f; ", lttc->time);
            newlines = 1;
         }
         unsigned long* sizes;
         int It;
         sizes = GetNetworkDetails(lttc->cells[CANCER],
                              lttc->n_Cells[CANCER]);
         printf("Number of Tumors: %2lu; Sizes:", sizes[0] );
         for( It = 1; It < sizes[0]+1; It++ ) {
            printf(" %4lu ", sizes[It] );
         }
         center = CenterOfMass( lttc->cells[CANCER],
                           lttc->n_Cells[CANCER] );
         printf("\nCenter of Mass: (");
         for( It = 0; It < lttc->n_Dimensions; It++ )
            printf("%f,", center->data[It]);
         printf(")\n");

         DeletePoint( center );

         free(sizes);
      }
      if( print_Grid ) {
```

```
         if( !newlines ) {
            printf("\n\nTime: %5.2f;\n", lttc->time);
            newlines = 1;
         }
         OutputGrid( lttc );
      }
      if( print_Stats ) {
         OutputStats( lttc );
      }
      if( print_Watch ){
         printf("Press Return.\n");
         char c = getchar();
         if(c == 'q')
            exit(-1);
      }
      if( print_Summary             &&
          lttc->n_Cells[CANCER] > 0 &&
          newlines == 0             &&
          print_Watch == 0  ) {
         OutputSummary( lttc );
      }
      if( print_File ){
         char filename[MAX_LINE_LEN];
         snprintf(filename, MAX_LINE_LEN, "%s%.1f.csv",
                                    print_File, lttc->time);
         OutputLatticeState( filename );
      }

   }
   return 1;
}/* End of Simulate() */


/* Read parameters from data file formatted as: */
/* n_Nodes n_Dimensions n_Nbrs_Max [center_Idx] */
int ReadDataFile( const char* const data,
                  unsigned long * n_Nodes,
                  int* n_Dimensions,
                  unsigned * n_Nbrs_Max,
                  unsigned long * center_Idx ) {
   FILE* fh;
   int return_Val;
   fh = fopen( data, "r" );
   if( fh == NULL ) {
        fprintf(stderr, "Failed to open data file %s.\n", data);
        return( -1 );
   }
   return_Val = fscanf(fh, "%lu %d %u %lu",
                       n_Nodes,
                       n_Dimensions,
                       n_Nbrs_Max,
                       center_Idx);
   switch( return_Val ) {
   case 3:
```

```
      *center_Idx = 0;
      break;
   case 4:
      if ( *center_Idx > *n_Nodes) {
         fprintf(stderr, "Value for center > number of nodes.\n");
         fprintf(stderr, "Using 1 instead. \n");
         *center_Idx = 1;
      }
      /* decrement since file indexes are 1 based & code indices are 0
based */
      (*center_Idx)--;
      break;
   default:
      fprintf(stderr, "Problem reading from datafile %s\n", data);
      return( -1 );
   }
   fclose( fh );/* Finished reading datafile */

   return 0;
} /* End ReadDataFile() */


/* Allocate Space for Network and Lattice members */
struct lattice_t* NewLattice( unsigned long n_Nodes,
                unsigned n_Dimensions,
                unsigned n_Nbrs_Max,
                unsigned long center_Idx,
                enum nodeType_t new_Type ) {

   int nbrIt;
   int It;
   struct lattice_t *L;
   char *error_Msg = "Failed to Allocate memory for lattice\n";
   if( n_Dimensions != 2 && n_Dimensions != 3 ){
      fprintf(stderr, "Invalind number of dimensions\n");
      return(NULL);
   }

   L = calloc (1, sizeof *L);
   if( L == NULL ) {
      fprintf(stderr, "%s", error_Msg);
      return(NULL);
   }
   L->nodes = calloc( n_Nodes , sizeof( struct node_t * ) );
   if( L->nodes == NULL ) {
      fprintf(stderr, "%s", error_Msg);
      return(NULL);
   }
   for( It=0; It < N_CELL_TYPES; It++ ) {
     L->n_Cells[ It ] = 0;
     L->target_Nodes[It] = calloc( n_Nbrs_Max+1, sizeof(struct
node_t**));
     if( L->target_Nodes[It] == NULL ) {
        fprintf(stderr, "%s", error_Msg);
```

```
        return(NULL);
     }
     L->n_Target_Nodes[It]  = calloc( n_Nbrs_Max+1, sizeof(unsigned
long) );
     L->cells[It]        = calloc( n_Nodes, sizeof(struct node_t *) );
     if( L->n_Target_Nodes[It]   == NULL ||
         L->cells[It] == NULL ) {
        fprintf(stderr, "%s", error_Msg);
        return(NULL);
     }
     for( nbrIt=0; nbrIt <= n_Nbrs_Max; nbrIt++ ) {
        L->target_Nodes[It][nbrIt]
                        = calloc( n_Nodes, sizeof(struct node_t*));
        if( L->target_Nodes[It][nbrIt] == NULL ){
           fprintf(stderr, "%s", error_Msg);
           return(NULL);
        }
     }
   }

   /* Initialization of lattice */
   L->time              = 0;
   L->count             = 0;
   L->n_Nbrs_Avg        = 0;
   L->n_Dimensions      = n_Dimensions;
   L->center_Idx        = center_Idx;
   L->n_Nodes           = n_Nodes;
   L->n_Nbrs_Max        = n_Nbrs_Max;
   L->n_Cells[new_Type] = n_Nodes;


   return L;
}


/* Each line of "network" contains two fields
   The first is the number of neighbors that node has
   The second is a comma separated list of indices for each neighbor
   The node's index is it's line number-1
   The neighbor's index is also decremented by 1 to have 0-based
indices */
int ReadNetworkFile( const char* const network, enum nodeType_t new_Type
) {
   FILE* fh;
   unsigned long nodeIt;
   int return_Val;
   unsigned n_Neighbors, nbrIt;

   fh = fopen( network, "r" );
   if( fh == NULL ) {
        fprintf(stderr, "Failed to open network file %s. %s\n",
              network, strerror(errno));
        return( -1 );
   }
```

```
   for( nodeIt = 0; nodeIt < lttc->n_Nodes; nodeIt++ ) {
      struct node_t *node = lttc->nodes[ nodeIt ];
      /* Get number of neighbors for node[ nodeIt ] */
      return_Val = fscanf( fh, "%u", &n_Neighbors );
      if( return_Val != 1 ) {
         fprintf(stderr, "Problem reading from network file %s\n",
network);
         return( -1 );
      }
      lttc->n_Nbrs_Avg += n_Neighbors;
      node->n_Neighbors = n_Neighbors;
      node->n_Nbrs[new_Type] = n_Neighbors;
      node->neighbors = malloc( sizeof(struct node_t*) * n_Neighbors);
      if( node->neighbors == NULL ){
         fprintf(stderr, "Memory error while creating neighbors.\n");
         return(-1);
      }

      /* Populate neighbor lists */
      for( nbrIt = 0; nbrIt < n_Neighbors; nbrIt++ ) {
         unsigned long nbrIdx;
         return_Val = fscanf( fh, "%lu", &nbrIdx);
         if( return_Val != 1 ) {
            fprintf(stderr, "Problem reading network file %s\n",
network);
            return( -1 );
         }
     /* decrement since file indexes are 1 based & code indices are 0
based */
         nbrIdx--;
         node->neighbors[ nbrIt ] = lttc->nodes[ nbrIdx ];
      }
   }
   fclose( fh );
   lttc->n_Nbrs_Avg /= lttc->n_Nodes;

   return 0;
}


/* Each line of file coords gives that node's cartesian coordinates */
/* The optional 3rd (or 4th) field gives that node's type */
int ReadCoordinateFile(  const char* const coords ){
   unsigned long nodeIt;
   struct node_t* cell;
   FILE* coords_fh = fopen( coords, "r" );

   if( coords_fh == NULL ) {
        fprintf(stderr, "Failed to open coordinate file %s. %s\n",
              coords, strerror(errno));
        return( -1 );
   }
```

```
   for( nodeIt = 0; nodeIt < lttc->n_Nodes; nodeIt++ ) {
      char line[MAX_LINE_LEN];
      char type = '0';
      double x, y, z;
      fgets(line, MAX_LINE_LEN, coords_fh);
      int r_val;

      cell = lttc->nodes[nodeIt];
      switch( lttc->n_Dimensions ) {
      case 3:
         r_val = sscanf( line, "%lf %lf %lf %c", &x, &y, &z, &type);
         cell->pos = NewPoint( lttc->n_Dimensions, x, y, z);
         break;
      case 2:
         r_val = sscanf( line, "%lf %lf %c", &x, &y, &type);
         cell->pos = NewPoint( lttc->n_Dimensions, x, y);
         break;
      default:
         return -1;
      }

      if( r_val == lttc->n_Dimensions ){
    type = 1;
      }


      switch( type ){
      case '0':
    ChangeNodeType( cell, VACANT );
    break;
      case '1':
         ChangeNodeType( cell, NORMAL );
         break;
      case '2':
         ChangeNodeType( cell, CANCER );
         break;
      case '3':
         ChangeNodeType( cell, INFECT_EXT );
         break;
      default:
         break;
      }
   }
   fclose( coords_fh );
   return 0;
}


struct node_t * NewNode ( unsigned long nodeIt,
                  enum nodeType_t new_Type ) {
   int typeIt;
   struct node_t * node;
   node = calloc( 1, sizeof(struct node_t));
   if( node == NULL ){
```

```
      fprintf(stderr, "Failed to allocate memory for node %lu\n",
nodeIt);
      return(NULL);
   }
   node->type     = new_Type;
   node->list_Idx = nodeIt;
   node->Idx      = nodeIt;
   node->visited  = NOTSEEN;
   node->distance = INT_MAX;
   for(typeIt = 0; typeIt < N_CELL_TYPES; typeIt++) {
      node->targets_Idx[typeIt] = NO_IDX;
      node->n_Nbrs[typeIt] = 0;
   }

   return node;
}

int ReadLattice(const char* const data,
                const char* const network,
                const char* const coords) {
   unsigned long nodeIt;
   const int new_Type = NORMAL;
   int return_Val;

   unsigned long center_Idx;
   int n_Dimensions;
   unsigned long n_Nodes;
   unsigned n_Nbrs_Max;

   /* Read Data File */
   return_Val = ReadDataFile(
             data, &n_Nodes, &n_Dimensions, &n_Nbrs_Max, &center_Idx);
   if( return_Val != 0 ) {
      fprintf(stderr, "Problem reading from data file %s\n", data);
      return( -1 );
   }

   /* Create Lattice */
   lttc =
      NewLattice( n_Nodes, n_Dimensions, n_Nbrs_Max, center_Idx,
new_Type );
   if( lttc == NULL ) {
      fprintf(stderr, "Problem creating lattice.\n");
      return( -1 );
   }

   /* Allocate Space for & Initialize nodes. */
   for( nodeIt = 0; nodeIt < n_Nodes; nodeIt++ ) {
      lttc->nodes[nodeIt] = NewNode( nodeIt, new_Type );
      if( lttc == NULL ) {
         fprintf(stderr, "Problem creating node %lu.\n", nodeIt);
         return( -1 );
      }
      lttc->cells[new_Type][nodeIt] = lttc->nodes[nodeIt];
```

```
   }

   /* Read Network File */
   return_Val = ReadNetworkFile( network, new_Type );
   if( return_Val != 0 ) {
      fprintf(stderr, "Problem reading from Network file %s\n",
network);
      return( -1 );
   }

   /* Read Coordinate File */
   if( coords != NULL )
   {
      return_Val = ReadCoordinateFile( coords );
      if( return_Val != 0 ) {
     fprintf(stderr, "Problem reading from Coordinate file %s\n",
coords);
     return( -1 );
      }
   }

   return 0;
}/* End of ReadLattice() */


/* AddNeighborsToTargetNodes() ***********************************
   Add node's neighbor's to correct "lttc->target_Nodes" arrays
*****************************************************************/
void AddNeighborsToTargetNodes( struct node_t* const node ) {
   enum nodeType_t type = node->type;
   int nbrIt;
   if( ( type == INFECT_INT       ) &&
       ( node->n_Nbrs[CANCER] > 0 ) ){
      for( nbrIt=0; nbrIt < node->n_Neighbors; nbrIt++){
     struct node_t *nbr = node->neighbors[nbrIt];
     if( nbr->type == CANCER ){
        if( nbr->targets_Idx[INFECT_INT] != NO_IDX ) {
           struct node_t **array =
             lttc->target_Nodes[INFECT_INT][ nbr->n_Nbrs[INFECT_INT]-1 ];
           unsigned long *last_Idx =
          &lttc->n_Target_Nodes[INFECT_INT][ nbr->n_Nbrs[INFECT_INT]-1 ];

           RemoveCell(&nbr->targets_Idx[ INFECT_INT ], array, last_Idx,
             &array[*last_Idx-1]->targets_Idx[INFECT_INT] );
        }

        AddCell(nbr, lttc->target_Nodes[INFECT_INT][nbr-
>n_Nbrs[INFECT_INT]],
          &lttc->n_Target_Nodes[INFECT_INT][nbr->n_Nbrs[INFECT_INT]],
          &nbr->targets_Idx[INFECT_INT] );
     }
      }
   }
   if( ( type == INFECT_EXT       ) &&
```

```
      ( node->n_Nbrs[CANCER] > 0 ) ){
     for( nbrIt=0; nbrIt < node->n_Neighbors; nbrIt++){
    struct node_t *nbr = node->neighbors[nbrIt];
    if( nbr->type == CANCER ){
       if( nbr->targets_Idx[INFECT_EXT] != NO_IDX ) {
          struct node_t **array =
            lttc->target_Nodes[INFECT_EXT][ nbr->n_Nbrs[INFECT_EXT]-1 ];
          unsigned long *last_Idx =
        &lttc->n_Target_Nodes[INFECT_EXT][ nbr->n_Nbrs[INFECT_EXT]-1 ];

          RemoveCell(&nbr->targets_Idx[ INFECT_EXT ], array, last_Idx,
            &array[*last_Idx-1]->targets_Idx[INFECT_EXT] );
       }

       AddCell(nbr, lttc->target_Nodes[INFECT_EXT][nbr-
>n_Nbrs[INFECT_EXT]],
         &lttc->n_Target_Nodes[INFECT_EXT][nbr->n_Nbrs[INFECT_EXT]],
         &nbr->targets_Idx[INFECT_EXT] );
    }
     }
   }
   if( type == NORMAL || type == CANCER ) {
      for( nbrIt=0; nbrIt < node->n_Neighbors; nbrIt++){
         struct node_t *nbr = node->neighbors[nbrIt];
         if( nbr->type == VACANT ){
            if( nbr->targets_Idx[type] != NO_IDX )
            {
               struct node_t **array =
                     lttc->target_Nodes[type][ nbr->n_Nbrs[type]-1 ];
               unsigned long *last_Idx =
                  &lttc->n_Target_Nodes[type][ nbr->n_Nbrs[type]-1 ];

               RemoveCell(&nbr->targets_Idx[ type ], array, last_Idx,
                         &array[*last_Idx-1]->targets_Idx[type] );
            }

            AddCell(nbr, lttc->target_Nodes[type][nbr->n_Nbrs[type]],
                  &lttc->n_Target_Nodes[type][nbr->n_Nbrs[type]],
                  &nbr->targets_Idx[type] );
         }
      }
   }

   return;
}
/* RemoveFromTargetNodes() ***************************************
   Removes node's index from all "lttc->target_Nodes" arrays
   Does not set node->targets_Idx values to NO_IDX
*****************************************************************/
void RemoveFromTargetNodes( struct node_t* const node ) {
   enum nodeType_t typeIt;

   for( typeIt = 0; typeIt < N_CELL_TYPES; typeIt++ ) {
      if( node->targets_Idx[typeIt] != NO_IDX ) {
```

```
         struct node_t **array =
               lttc->target_Nodes[typeIt][ node->n_Nbrs[typeIt] ];
         unsigned long *last_Idx =
            &lttc->n_Target_Nodes[typeIt][ node->n_Nbrs[typeIt] ];

         RemoveCell(&node->targets_Idx[ typeIt ], array, last_Idx,
                    &array[*last_Idx-1]->targets_Idx[typeIt] );
      }
   }
   return;
}


/* AddToTargetNodes() ********************************************
   Add node to correct "lttc->target_Nodes" arrays
   Updates node->targets_Idx
*****************************************************************/
void AddToTargetNodes( struct node_t* const node ) {

   enum nodeType_t type = node->type;

   if( type == CANCER )
   {
      if( node->n_Nbrs[INFECT_INT] > 0 )
         AddCell(node, lttc->target_Nodes[INFECT_INT][node-
>n_Nbrs[INFECT_INT]],
               &lttc->n_Target_Nodes[INFECT_INT][node-
>n_Nbrs[INFECT_INT]],
               &node->targets_Idx[INFECT_INT] );
      if( node->n_Nbrs[INFECT_EXT] > 0 )
         AddCell(node, lttc->target_Nodes[INFECT_EXT][node-
>n_Nbrs[INFECT_EXT]],
               &lttc->n_Target_Nodes[INFECT_EXT][node-
>n_Nbrs[INFECT_EXT]],
               &node->targets_Idx[INFECT_EXT] );
   }
   if( type == VACANT )
   {
      if( node->n_Nbrs[NORMAL] > 0 )
         AddCell(node, lttc->target_Nodes[NORMAL][node->n_Nbrs[NORMAL]],
               &lttc->n_Target_Nodes[NORMAL][node->n_Nbrs[NORMAL]],
               &node->targets_Idx[NORMAL] );
      if( node->n_Nbrs[CANCER] > 0 )
         AddCell(node, lttc->target_Nodes[CANCER][node->n_Nbrs[CANCER]],
               &lttc->n_Target_Nodes[CANCER][node->n_Nbrs[CANCER]],
               &node->targets_Idx[CANCER] );
   }
   return;
}
/* RemoveNeighborsFromTargetNodes() */
void RemoveNeighborsFromTargetNodes( struct node_t* const node ) {

   if (node->type == VACANT)
      return;
```

```
   int nbrIt;
   for (nbrIt = 0; nbrIt < node->n_Neighbors; nbrIt++) {
      struct node_t * nbr = node->neighbors[nbrIt];
      /* Adjust n_Nbrs of each neighbor */
      int n_Nbrs = nbr->n_Nbrs[node->type]--;
      /* If neighbor is a target, move to proper list */
      if( nbr->targets_Idx[node->type] != NO_IDX ) {
     unsigned long *last_Node_Idx =
        &lttc->n_Target_Nodes[node->type][n_Nbrs];
     struct node_t* last_Node =
        lttc->target_Nodes[node->type][n_Nbrs][(*last_Node_Idx)-1];

     RemoveCell( &nbr->targets_Idx[node->type],
              lttc->target_Nodes[node->type][n_Nbrs],
              last_Node_Idx,
             &last_Node->targets_Idx[node->type] );
     if( n_Nbrs-1 > 0 ) {
        AddCell( nbr,
             lttc->target_Nodes[node->type][n_Nbrs-1],
            &lttc->n_Target_Nodes[node->type][n_Nbrs-1],
            &nbr->targets_Idx[node->type] );
     }
       }
   }

   return;
}


/** ChangeNodeType( node, type ) ***********************************
  This routine will change "node"'s type to "type" (CANCER or INFECT)
*****************************************************************/
void ChangeNodeType( struct node_t* node, enum nodeType_t type )
{
   int nbrIt;
   static int recurse = 0;

   if( type == INFECT_EXT || type == INFECT_INT ){
      if( 100.0*node->n_Nbrs[NORMAL]/node->n_Neighbors <=
percent_Interior )
     type = INFECT_INT;
       else
     type = INFECT_EXT;
   }

   if( node->type == type )
      return;


   RemoveNeighborsFromTargetNodes( node );
   RemoveFromTargetNodes( node );
   /* Remove from Cell List */
   if( node->type != VACANT )
```

```
      RemoveCell(
          &node->list_Idx,
           lttc->cells[node->type],
          &lttc->n_Cells[node->type],
          &lttc->cells[node->type][lttc->n_Cells[node->type]-1]-
>list_Idx );

   node->type = type;
   /* Check neighbor's virus type */
   if( recurse == 0 ){
      for (nbrIt = 0; nbrIt < node->n_Neighbors; nbrIt++) {
     recurse = 1;
     struct node_t*  nbr = node->neighbors[nbrIt];
     enum nodeType_t nbr_Type = nbr->type;
     if( nbr_Type == INFECT_EXT || nbr_Type == INFECT_INT ){
        if( nbr->n_Nbrs[CANCER]/nbr->n_Neighbors > percent_Interior/100
)
           ChangeNodeType( nbr, INFECT_INT );
        else
           ChangeNodeType( nbr, INFECT_EXT );
     }
     recurse = 0;
      }
   }

   /* Add to list of Cells */
   if( type != VACANT ) {
      AddCell(
            node,
            lttc->cells[type],
            &lttc->n_Cells[type],
            &node->list_Idx );
      for (nbrIt = 0; nbrIt < node->n_Neighbors; nbrIt++)
         node->neighbors[nbrIt]->n_Nbrs[node->type]++;
   }
   AddToTargetNodes( node );
   AddNeighborsToTargetNodes( node );

   return;
}/* End of ChangeNodeType */
```

```
/** SeedCancer( ) ******************************************************
  Places initial cancer cells in lattice.
  Replaces lttc->center and cancer_Count of it's nearest neighbors w/
          or until distance of cancer_Radius is reached (which ever is
     "greater") with cancer cells.
**********************************************************************/
int SeedCancer( struct node_t* center,
                unsigned long cancer_Count,
                unsigned cancer_Radius ) {
   struct node_t *node_u;
   unsigned long nodeIt;
   for( nodeIt=0; nodeIt < lttc->n_Nodes; nodeIt++ ) {
      lttc->nodes[nodeIt]->visited = NOTSEEN;
      lttc->nodes[nodeIt]->distance = INT_MAX;
   }

   node_u = center;

   int count = 0;
   struct node_t *node_v;
   struct Queue * const Q = NewQueue( 8 );
   node_u->distance = 0;
   node_u->visited = SEEN;
   Enqueue( Q, node_u );
   while( (node_u = Dequeue(Q) ) ){
      if( ( node_u->distance > cancer_Radius && node_u->distance <
INT_MAX )
            && count >= cancer_Count ){
         continue;
      }
      count++;
      ChangeNodeType( node_u, CANCER );

      node_u->visited = VISITED;
      int nbrIt;
      for( nbrIt = 0; nbrIt < node_u->n_Neighbors; nbrIt++ ) {
         node_v = node_u->neighbors[nbrIt];
         if( node_v->visited == NOTSEEN ) {
            node_v->visited = SEEN;
            node_v->distance = node_u->distance + 1;
            if( Enqueue( Q, node_v ) == 0 )
               printf(" Problem growing Queue\n" );
         }
      }
   }
   DeleteQueue( Q );
   return 0;
}/* End of SeedCancer() */
```

```
/** InjectVirus( ) ***********************************************
  Changes roughly percent_Infected CANCER cells to INFECT cells
****************************************************************/
int InjectVirus( ) {
   unsigned long n_Infect = lttc->n_Cells[CANCER] *
percent_Infected/100;
   struct node_t *center;
   Point* c_point;
   switch( infect_Type ){
   case RANDOM:
      while( lttc->n_Cells[INFECT_INT]+lttc->n_Cells[INFECT_EXT]<
n_Infect ){
         unsigned long random;
         do {
            random = lttc->n_Cells[CANCER] * genrand();
         } while( random >= lttc->n_Cells[CANCER] );
         ChangeNodeType( lttc->cells[CANCER][random], INFECT_EXT );
      }
      break;
   case CENTER: {
      c_point = CenterOfMass( lttc->cells[CANCER],
                              lttc->n_Cells[CANCER] );
      center = FindNodeNear( c_point, lttc->cells[CANCER],
                                      lttc->n_Cells[CANCER] );
      VirusBallAroundNode( center, n_Infect );
      DeletePoint( c_point );
      break; }
   case MULTINODE: {
      c_point = CenterOfMass( lttc->cells[CANCER],
                              lttc->n_Cells[CANCER] );
      center = FindNodeNear( c_point, lttc->cells[CANCER],
                                      lttc->n_Cells[CANCER] );

      int It;
      for( It = 0; It < 3; It++ ) {
         double random = genrand() * 2 * M_PI;
         struct node_t* node = FindEdge( lttc->cells[CANCER],
                                         lttc->n_Cells[CANCER],
                                         center, random );
         VirusBallAroundNode( node, n_Infect/3 );
      }
      DeletePoint( c_point );
      break; }
   case PERIMETER: {
      c_point = CenterOfMass( lttc->cells[CANCER], lttc->n_Cells[CANCER]
);
      double* distance[2];
      distance[0] = malloc( lttc->n_Cells[CANCER] * sizeof(double));
      distance[1] = malloc( lttc->n_Cells[CANCER] * sizeof(double));
      unsigned long nodeIt;
      for( nodeIt=0; nodeIt < lttc->n_Cells[CANCER]; nodeIt++){
         struct node_t *cell = lttc->cells[CANCER][nodeIt];
         distance[0][nodeIt] = cell->Idx;
         distance[1][nodeIt] = Point2PointDistance( c_point, cell->pos);
```

```
      }
      MergeSort( distance, lttc->n_Cells[CANCER]);
      for( nodeIt=0; nodeIt < n_Infect; nodeIt++)
         ChangeNodeType( lttc->nodes[ (int)distance[0][nodeIt] ],
INFECT_EXT );
      free( distance[0] );
      free( distance[1] );
      DeletePoint( c_point );
      break; }
   default:
      return -1;
   }
   return 0;

}/* End of InjectVirus() */


/* MergeSort() ***************************************************
   Performs a merge sort on the N data pairs in list.
   list[1] contains the values to sort on
   list[0][i] contains a value that remains associated with list[1][i]
*****************************************************************/
void MergeSort( double** list, unsigned long N ){
   double* temp[2];
   temp[0] = malloc( N * sizeof( double ));
   temp[1] = malloc( N * sizeof( double ));
   MergeSplit( list, 0, N, temp );
   free( temp[0] );
   free( temp[1] );
   return;
}

/* MergeSplit() **************************************************
   Helper function for MergeSort().
   Should not be called outside MergeSort()
*****************************************************************/
void MergeSplit( double** list,
      unsigned long b_idx,
      unsigned long e_idx,
                double** temp ){
   if( e_idx - 1 <= b_idx )
      return; //Length 1 list is sorted

   unsigned long m_idx = ( e_idx + b_idx ) / 2;
   MergeSplit( list, b_idx, m_idx, temp );
   MergeSplit( list, m_idx, e_idx, temp );
   Merge( list, b_idx, m_idx, e_idx, temp);
   unsigned long It;
   for( It = b_idx; It < e_idx; It++ ) {
      list[0][It] = temp[0][It];
      list[1][It] = temp[1][It];
   }
}
```

```
/* Merge() *****************************************************
   Helper function for MergeSort().
   Should not be called outside MergeSort()
*****************************************************************/
void Merge( double** list, unsigned long b_idx,
                           unsigned long m_idx,
                           unsigned long e_idx,
            double** temp ){
   unsigned long It;
   unsigned long LH = b_idx;
   unsigned long RH = m_idx;

   for(It = b_idx; It < e_idx; It++){
      if( (LH < m_idx && RH >= e_idx) ||
          (LH < m_idx && list[1][LH] > list[1][RH])){
         temp[0][It] = list[0][LH];
         temp[1][It] = list[1][LH];
         LH++;
      } else {
         temp[0][It] = list[0][RH];
         temp[1][It] = list[1][RH];
         RH++;
      }
   }
   return;
}


/* FindEdge() **************************************************
   construct line L with direction "dir" that intersects start

   set node to "start"
   while( node's neighbors are all cancer )
      mark node
      find neighbor "nbr" of node closest to L
      set node to "nbr"

   returns address of node along L that is at the "edge" of the tumor
*****************************************************************/
struct node_t* FindEdge( struct node_t** nodes, unsigned long N,
                         struct node_t*  start, double dir ){
   unsigned long nodeIt = 0;
   int n_Dims = nodes[0]->pos->N;
   for( nodeIt=0; nodeIt < N; nodeIt++ )
      nodes[nodeIt]->visited = NOTSEEN;

   struct node_t* node = start;
   node->visited = VISITED;
   Vector *v = NULL;
   switch( n_Dims ) {
   case 2:
      v = NewVector(n_Dims, cos( dir ), sin( dir ));
      break;
   case 3:
```

```
      v = NewVector(n_Dims, cos( dir ), sin( dir ), start->pos-
>data[2]);
      break;
   default:
      break;
   }
   Line *l = NewLine(start->pos, v );

   while( node->n_Nbrs[CANCER] == node->n_Neighbors ){
      int nbrIt = 0;
      double s_dist, dist;
      struct node_t* temp = NULL;
      for( nbrIt = 0; nbrIt < node->n_Neighbors && temp == NULL; nbrIt+
+) {
         if( node->neighbors[nbrIt]->visited == NOTSEEN ){
            s_dist = Point2LineDistance( node->neighbors[nbrIt]->pos, l
);
            temp = node->neighbors[nbrIt];
         }
      }
      for( ; nbrIt < node->n_Neighbors; nbrIt++) {
         if( node->neighbors[nbrIt]->visited == VISITED )
            continue;

         dist = Point2LineDistance( node->neighbors[nbrIt]->pos, l );
         if( dist < s_dist ){
            temp = node->neighbors[nbrIt];
            s_dist = dist;
         }
      }
      node = temp;
      node->visited = VISITED;
   }
   DeleteVector( v );
   DeleteLine( l );

   return node;
}

/** VirusBallAroundNode() ****************************************
  Creates a densly packed ball of n_Infect infected cells centered
  around node.
*****************************************************************/
int VirusBallAroundNode( struct node_t * node, unsigned long n_Infect )
{
   unsigned long nodeIt;
   int count = 0;
   Point *point = node->pos;

   for( nodeIt = 0; nodeIt < lttc->n_Cells[CANCER]; nodeIt++ ) {
      lttc->cells[CANCER][nodeIt]->visited = NOTSEEN;
      lttc->cells[CANCER][nodeIt]->distance = UINT_MAX;
   }
   while( lttc->n_Cells[CANCER] > 0 && count < n_Infect )
```

```
   {
      if( node->type != INFECT_INT || node->type != INFECT_EXT )
         count += BFS(node, CANCER, n_Infect-count, INFECT_EXT );
      node = FindNodeNear(point, lttc->cells[CANCER],
                          lttc->n_Cells[CANCER]);
   }

   return count;
}


/* DetermineEventRecipient() ***************************************
  selects a node for action.
   if killing type T, randomly select node of type T.

   if growing type T, randomly select from targetof(T) nodes weighted
        by the number of neighboring source nodes.
*******************************************************************/
struct node_t * DetermineEventRecipient(const struct event_t e_t ) {
   struct node_t* recp = NULL;
   double random;
   enum nodeType_t type = e_t.node_Type;

   do {
      random = genrand();
   } while( random == 1 || random == 0 );

   switch (e_t.cell_Action) {
   case KILL:
      random *= lttc->n_Cells[type];
      recp = lttc->cells[type][ (unsigned long int) random ];
      break;
   case GROW:
      {
         //Select a cell to replicate to replicates onto.
         int n_nbrIt;
         unsigned long int n_Links=0;
         for(n_nbrIt = 1; n_nbrIt <= lttc->n_Nbrs_Max; n_nbrIt++)
            n_Links += lttc->n_Target_Nodes[ type ][n_nbrIt] * n_nbrIt;
         random *= n_Links;
         n_nbrIt = 1;
         while( lttc->n_Target_Nodes[ type ][n_nbrIt]*n_nbrIt <= random
) {
            random  -= lttc->n_Target_Nodes[ type ][n_nbrIt] * n_nbrIt;
            n_nbrIt++;
         }
         recp = lttc->target_Nodes[type][n_nbrIt][(unsigned long int)
(random/n_nbrIt)];
      }
      break;
   default:
      fprintf(stderr, "Action was not KILL or GROW\n");
      return NULL;
   }
```

```
   if( recp == NULL ) {
      fprintf(stderr, "\nSelected cell is NULL.\n");
      fprintf(stderr, "lttc->time = %f\n", lttc->time );
      return NULL;
   }
   return recp;
}/* End of DetermineEventRecipient() */

/
*************************************************************************
  Print Percentage of Node Types
*************************************************************************
***/
int OutputStats() {
   /* This is coded so that the output when only Normal cells exist is
the
      same as the output created with Chetan's code */
   printf("  time: %6.2f   Density: %14.6f",
         lttc->time, (float)lttc->n_Cells[NORMAL]/lttc->n_Nodes);
   if( lttc->n_Cells[CANCER] > 0 ){
      printf("   Cancer Density: %14.6f",
                  (float)lttc->n_Cells[CANCER]/lttc->n_Nodes);
   }
   long unsigned int n_Virus = lttc->n_Cells[INFECT_INT] +
                               lttc->n_Cells[INFECT_EXT];
   if( n_Virus > 0 ) {
      if( lttc->n_Cells[CANCER] == 0 )
         printf("                                  " );
      printf("   Virus Density: %14.6f",
                  (float)n_Virus/lttc->n_Nodes);
   }
   printf("\n");


   return 0;
}/* End of OutputStats() */

/*************************************************************************
  Print Percentage of Cancer Nodes
*************************************************************************/
int OutputSummary() {
   printf(" %6.2f, ", (float)lttc->n_Cells[CANCER]/lttc->n_Nodes);

   return 0;
}/* End of OutputSummary() */

/** OutputGrid( lattice ) ****************
* Print 2d lattice to stdout as ASCII art
*****************************************/
void OutputGrid() {
   const struct node_t* cell = lttc->nodes[0];
   const float width = sqrt( lttc->n_Nodes );
   int count = 0;
```

```
   if( width != (unsigned long) width ) {
      fprintf( stderr, "Lattice isn't square. Refusing to print grid.\n"
);
      return;
   }

   unsigned long nodeIt;
   for( nodeIt=0; nodeIt < lttc->n_Nodes; nodeIt++ ) {
      cell = lttc->nodes[nodeIt];
      switch (cell->type) {
      case NORMAL:
         printf("\x1b[32m .\x1b[0m ");
         break;
      case CANCER:
         printf("\x1b[31m c\x1b[0m ");
         break;
      case INFECT_INT:
         printf(" i ");
         break;
      case INFECT_EXT:
         printf(" I ");
         break;
      default:
         printf(" _ ");
         break;
      }
      count++;
      if( count%(int)width%10 == 0 )
         printf(" | ");
      if( count % (int)width == 0 ){
         printf("\n");
    if(count % ((int)width*10) == 0 ){
       int i;
       for( i=0; i<width + width/10; i++ )
          printf(" = ");
       putchar('\n');
    }
      }
   }
   return;
}/* End of OutputGrid() */
```

```
/** OutputLatticeState( lattice )*********************************
* Print current lattice state to file for analysis
*****************************************************************/
void OutputLatticeState(char* filename) {
   FILE* fh;
   const struct node_t* cell = lttc->nodes[0];

   fh = fopen( filename, "w" );
   if( fh == NULL ) {
        fprintf(stderr, "Failed to open %s for output. %s\n",
              filename, strerror(errno));
        exit(-1);
   }

   unsigned long nodeIt;
   for( nodeIt=0; nodeIt < lttc->n_Nodes; nodeIt++ ) {
      cell = lttc->nodes[nodeIt];

      int dimIt;
      for( dimIt=0; dimIt < lttc->n_Dimensions; dimIt++)
         fprintf(fh, "%f ", cell->pos->data[dimIt]);

      switch (cell->type) {
      case NORMAL:
         fprintf(fh, " 1 \n");
         break;
      case CANCER:
         fprintf(fh, " 2 \n");
         break;
      case INFECT_INT:
      case INFECT_EXT:
         fprintf(fh, " 3 \n");
         break;
      default: //NORMAL
         fprintf(fh, " 0 \n");
      }
   }
   fclose( fh );

   return;
}/* End of OutputLatticeState() */


/** DetermineEventType( lattice ) ********************************
* Calculate the probabilities of event types and randomly select one
* 6 bins:
*   3 KILL are sized rate_i * n_Cells_i;
*   3 GROW are sized rate_i * (n_links from i to
TargetOf(i))/n_Nbrs_Avg
* Adjust lattice time
*****************************************************************/
struct event_t DetermineEventType( ){

   double ev_Bin[ N_CELL_TYPES ][ N_ACTIONS ] = {{0,0},{0,0},{0,0},
```

```
{0,0}};
   double random;
   struct event_t ev;
   int actionIt;
   double summation = 0;

   int typeIt, n_NbrsIt;
   for( typeIt = NORMAL; typeIt < N_CELL_TYPES; typeIt++ ) {
      int n_Links=0;
      ev_Bin[typeIt][KILL] = rates[typeIt][KILL] * lttc-
>n_Cells[typeIt];

      for( n_NbrsIt=0; n_NbrsIt <= lttc->n_Nbrs_Max; n_NbrsIt++ ) {
         n_Links += lttc->n_Target_Nodes[ typeIt ][ n_NbrsIt ] *
n_NbrsIt;
      }
      ev_Bin[ typeIt ][ GROW ] =
               rates[ typeIt ][GROW] * (double) n_Links / lttc-
>n_Nbrs_Avg;
   }

   for( typeIt = NORMAL; typeIt < N_CELL_TYPES; typeIt++ ) {
      for(actionIt=KILL; actionIt < N_ACTIONS; actionIt++) {
         summation += ev_Bin[ typeIt ][actionIt];
         ev_Bin[ typeIt ][ actionIt ] = summation;
      }
   }

   do {
      random = genrand();
   } while (random == 0 || random == 1 );

   lttc->time += -log( random ) / ev_Bin[INFECT_EXT][KILL];
   lttc->count++;

   do {
      random = genrand();
   } while (random == 0 || random == 1 );
   random = random * ev_Bin[N_CELL_TYPES-1][N_ACTIONS-1];

   for( ev.node_Type=NORMAL; ev.node_Type < N_CELL_TYPES; ev.node_Type++
)
   {
      for(ev.cell_Action=KILL; ev.cell_Action < N_ACTIONS;
ev.cell_Action++)
      {
         if( random < ev_Bin[ev.node_Type][ev.cell_Action] ) {
            return ev;
         }
      }
   }
   fprintf(stderr, "\n\n\n\nNo event selected.\n");
   fprintf(stderr, "Number of NORMAL cells: %lu\n", lttc-
>n_Cells[NORMAL]);
```

```
   fprintf(stderr, "Number of CANCER cells: %lu\n", lttc-
>n_Cells[CANCER]);
   fprintf(stderr, "Number of INFECT_INT cells: %lu\n", lttc-
>n_Cells[INFECT_INT]);
   fprintf(stderr, "Number of INFECT_EXT cells: %lu\n", lttc-
>n_Cells[INFECT_EXT]);
   ev.node_Type=VACANT;

   return( ev );
}/* End of DetermineEventType() */


/** RemoveCell( cell_Idx, array, n_cells, index ) ********************
* Remove cell at "del_idx" from "array" by moving array[*n_cells] to
it's location. Update "n_cells" in array, "del_idx" and index of former
last node "last_idx".
*
* Doesn't perform sanity checks on arguments.
* If del_idx, n_cells, and last_idx aren't valid for array bad things
*     will happen.
*******************************************************************/
void RemoveCell (unsigned long* const del_Idx,
                 struct node_t** array,
                 unsigned long * const n_cells,
                 unsigned long * const last_Idx) {
   const int idx_a = *del_Idx;
   const int idx_b = *last_Idx;

   if( *del_Idx < *last_Idx ) {
      array[ idx_a ] = array[ idx_b ];
      *last_Idx = idx_a;
   }
   (*n_cells)--;
   array[ *n_cells ] = NULL;
   *del_Idx = NO_IDX;

   return;
}/* End of RemoveCell() */

/** AddCell( cell, array, n_Array, index ) ***************************
* Add cell to array with n_Array nodes.
* Set cell's index into array.
*******************************************************************/
void AddCell( struct node_t* const cell,
              struct node_t** array,
              unsigned long*const  n_Array,
              unsigned long*const  cell_Idx ){
   array[*n_Array] = cell;
   *cell_Idx = *n_Array;
   (*n_Array)++;
   return;
}/* End of AddCell() */
```

```
/** TargetOf( nodeType )*********************************************
* Function takes a cell type as an argument and returns its target.
* If type is invalid or has no target, program quits
*******************************************************************/
enum nodeType_t TargetOf( const enum nodeType_t type ) {
      switch( type ){
      case INFECT_INT:
      case INFECT_EXT:
         return CANCER;
      case NORMAL:  //Treat the same as CANCER.
      case CANCER:
         return VACANT;
      case VACANT:
         fprintf(stderr, "Asked for target type of VACANT node\n");
         exit(-1);
      default:
         fprintf(stderr, "Asked for target type of undefined node
type\n");
         exit(-1);
      }

}/* End of TargetOf() */

void DeleteLattice( struct lattice_t* lattice ){
   unsigned long int nodeIt = 0;
   enum nodeType_t typeIt = 0;
   int nbrIt=0;
   for( typeIt = 0; typeIt < N_CELL_TYPES; typeIt++ ) {
      for( nbrIt=0; nbrIt <= lattice->n_Nbrs_Max; nbrIt++ ) {
    free( lattice->target_Nodes[typeIt][nbrIt] );
    lattice->target_Nodes[typeIt][nbrIt] = NULL ;
      }
   }
   for( typeIt = 0; typeIt < N_CELL_TYPES; typeIt++ ) {
      free( lattice->n_Target_Nodes[typeIt] );
      lattice->n_Target_Nodes[typeIt] = NULL;
      free( lattice->target_Nodes[typeIt] );
      lattice->target_Nodes[typeIt] = NULL;
      free( lattice->cells[typeIt] );
      lattice->cells[typeIt] = NULL;

   }
   for( nodeIt = 0; nodeIt < lattice->n_Nodes; nodeIt++ ) {
      DeletePoint( lattice->nodes[nodeIt]->pos );
      lattice->nodes[nodeIt]->pos = NULL ;
      free( lattice->nodes[nodeIt]->neighbors );
      lattice->nodes[nodeIt]->neighbors = NULL ;
      free( lattice->nodes[nodeIt] );
      lattice->nodes[nodeIt] = NULL ;
   }

   free( lattice->nodes );
   lattice->nodes = NULL;
   free( lattice );
```

```
   lattice = NULL;
   return;
}

void OpenFiles(){
      fpp=fopen("vpts.dat", "a+");
      fpl=fopen("vlpt.dat", "a+");
      unsigned long int n_Tumor = lttc->n_Cells[CANCER]     +
                              lttc->n_Cells[INFECT_INT] +
                            lttc->n_Cells[INFECT_EXT] ;
      fprintf(fpp," %d ",rng_Seed);
      fprintf(fpl," %d ",rng_Seed);
      fprintf(fpp," %lu ", n_Tumor);
      fprintf(fpl," %lu ", n_Tumor);
}

void CloseFiles(){
   int time_Check     = (int) floor(lttc->time);
   int time_Check_fpl = (int) floor(lttc->time);
   unsigned long int n_Tumor = lttc->n_Cells[CANCER]     +
                  lttc->n_Cells[INFECT_INT] +
                  lttc->n_Cells[INFECT_EXT] ;
   while( time_Check_fpl % 5 != 0 )
      time_Check_fpl--;

   while( time_Check < 126) {
      time_Check = time_Check+1;
      fprintf(fpp," %lu ", n_Tumor);
   }
   fprintf(fpp,"\n");
   fclose(fpp);


   if( time_Check_fpl < t_Max_Simulation ) {
      time_Check_fpl+=5;
      fprintf(fpl," %lu ", n_Tumor);
   }
   fprintf(fpl,"\n");
   fclose(fpl);
}

void ParameterFile(float time_Extinct){
   FILE *fp;
   fp=fopen("vout.dat", "a+");
   fprintf(fp," %d ",rng_Seed);
   fprintf(fp," %f ",rates[NORMAL][GROW]);
   fprintf(fp," %f ",rates[NORMAL][KILL]);
   fprintf(fp," %f ",rates[CANCER][GROW]);
   fprintf(fp," %f ",rates[CANCER][KILL]);
   fprintf(fp," %f ",rates[INFECT_INT][GROW]);
   fprintf(fp," %f ",rates[INFECT_INT][KILL]);
   fprintf(fp," %f ",rates[INFECT_EXT][GROW]);
   fprintf(fp," %f ",rates[INFECT_EXT][KILL]);
   fprintf(fp," %f ",percent_Infected);
```

```
   if( time_Extinct > 0 ) {
     fprintf(fp," %f ",time_Extinct);
   }
   else {
     fprintf(fp," %f ",lttc->time);
   }
   fprintf(fp," %d ",(int)lttc->n_Cells[NORMAL]);
   fprintf(fp," %d ",(int)lttc->n_Cells[CANCER]);
   fprintf(fp," %d ",(int)lttc->n_Cells[INFECT_INT]);
   fprintf(fp," %d ",(int)lttc->n_Cells[INFECT_EXT]);
   unsigned long int n_Virus = lttc->n_Cells[INFECT_INT] +
                lttc->n_Cells[INFECT_EXT] ;

   if( lttc->n_Cells[CANCER] == 0 ) { fprintf(fp," 1\n"); }
   if( lttc->n_Cells[CANCER]  > 0 &&
       n_Virus               == 0 ) { fprintf(fp," 2\n"); }
   if( lttc->n_Cells[CANCER]  > 0 &&
       n_Virus                > 0 ) { fprintf(fp," 3\n"); }
   fclose(fp);
   return;
}
```

Cells.h

```
#include <getopt.h>
#include <stdlib.h> /*free, exit, calloc, malloc, atoi, strtof*/
#include <math.h>   /*isfinite, sqrt, log*/
#include <stdio.h>
#include <string.h> /*strerr, strlen, strcat */
#include <errno.h>  /*errno*/
#include <limits.h> /*ULONG_MAX INT_MAX*/
#include "Queue.h"
#include "mersenne.h"
#include "cartesian.h"

#define MAX_LINE_LEN 100
#define NO_IDX ULONG_MAX
#define N_CELL_TYPES 4
enum nodeType_t   { NORMAL, CANCER, INFECT_INT, INFECT_EXT, VACANT,
N_NODE_TYPES };
enum cellAction_t { KILL, GROW, N_ACTIONS };
enum visited_t    { NOTSEEN, SEEN, VISITED };
enum infection_t  { NONE, RANDOM, CENTER, PERIMETER, MULTINODE };

/*******************/
/*** Structures  ***/
/*******************/
/* Event */
struct event_t {
   enum cellAction_t cell_Action;
   enum nodeType_t node_Type;
};

/* Cell Lattice */
struct lattice_t {
   unsigned long int *n_Target_Nodes[N_CELL_TYPES];
                              //2nd dim size: n_Nbrs_Max+1
   struct node_t *** target_Nodes[N_CELL_TYPES];
                              //size:[4][n_Nbrs_Max+1][n_Nodes]
                              //[SOURCE_Type][#target neighbors][cell
addy]

   unsigned long int  n_Cells[N_CELL_TYPES]; // # of each TYPE of Cell
   struct node_t ** cells[N_CELL_TYPES]; //size: [n_Nodes];

   unsigned long int    n_Nodes;
   struct node_t ** nodes;

   double  time;
   unsigned long int count;  /* # of events since lattice
initialization */
   unsigned long int center_Idx;
   unsigned int    n_Nbrs_Max;
   unsigned int    n_Nbrs_Avg;
   char*  coord_Filename;
   int    n_Dimensions;
```

```
};

/* Cell */
struct node_t {
   struct node_t ** neighbors; //Array size: n_Nbrs_Max
   int n_Neighbors;
   Point *pos;

   enum nodeType_t type;
   int interior; //indicates Infected cells position in tumor
   int n_Nbrs[N_CELL_TYPES]; //Number of neighbors of type

   unsigned long int targets_Idx[N_CELL_TYPES];
               // Locations(s) in target_Nodes (NO_IDX == not present)

   unsigned long int list_Idx;
               // Cell's location in cells[this.type] array NO_IDX if
empty

   unsigned long int Idx;
               // Cells' location in lttc->nodes array

   /* variables used for BFS */
   enum visited_t visited;
   unsigned int distance;
};

/*=================*/
/*=== Functions ===*/
/*=================*/
float  argtof( const char *nptr );
enum   nodeType_t TargetOf( const enum nodeType_t type );
void LinReg( float const * const X,
             float const * const Y,
         long unsigned int const N,
         float * slope,
         float * intercept );

int  SeedCancer( struct node_t* center, unsigned long int cancer_Count,
                                        unsigned int cancer_Radius );
int  InjectVirus( );
int  VirusBallAroundNode( struct node_t* node, unsigned long int
n_Infect );
int  Simulate( );
int  OutputStats();
int  OutputSummary();
struct node_t* DetermineEventRecipient( const struct event_t event);
struct event_t DetermineEventType( );
void   ChangeNodeType( struct node_t* node, enum nodeType_t type );
int  VerifyState( );
void OpenFiles();
void ParameterFile(float time_Extinct);
void CloseFiles();
```

```
/*****************************/
/* Network Related Functions */
/*****************************/
struct lattice_t* NewLattice( unsigned long int n_Nodes,
                unsigned int n_Dimensions,
                unsigned int n_Nbrs_Max,
                unsigned long int center_Idx,
                enum nodeType_t new_Type ) ;
Point* CenterOfMass( struct node_t** nodes, unsigned long int N );
struct node_t* FindNodeNear( Point* point, struct node_t ** nodelist,
                                        unsigned long int N);
int    BFS (struct node_t* node_u,
            enum nodeType_t network_Type,
        int max_Count,
        enum nodeType_t new_Type );
void   RemoveCell (unsigned long int* const del_Idx,
                  struct node_t** array,
           unsigned long int* const n_cells,
           unsigned long int* const last_Idx);
void   AddCell( struct node_t* const cell,
                struct node_t** array,
       unsigned long int* const n_array,
       unsigned long int* const cell_Idx );
unsigned long int*   GetNetworkDetails( struct node_t ** nodes,
                                        unsigned long int N );
struct node_t* FindEdge( struct node_t** nodes, unsigned long int N,
                         struct node_t* start, double dir );
void   DeleteLattice( struct lattice_t* lattice );
int    ReadDataFile( const char* const data,
                  unsigned long int * n_Nodes,
                  int* n_Dimensions,
                  unsigned int * n_Nbrs_Max,
                  unsigned long int * center_Idx ) ;
/* Uses Global lttc */
void   OutputGrid();
void   OutputLatticeState( char* filename );
int ReadNetworkFile( const char* const network,
                enum nodeType_t new_Type ) ;
int ReadCoordinateFile(  const char* const coords );
int ReadLattice(const char* const data,
                const char* const network,
       const char* const coords );


/***********************/
/* MergeSort Functions */
/***********************/
void MergeSort ( double** list, unsigned long int N );
void MergeSplit( double** list, unsigned long int b_idx,
                                unsigned long int e_idx, double** temp
);
void Merge     ( double** list, unsigned long int b_idx,
                                unsigned long int m_idx,
                      unsigned long int e_idx, double** temp );
```

```
/****************/
/***Parameters***/
/****************/
/* FIXME Should these all be global? */
extern struct lattice_t * lttc;
extern FILE*  vlpt;
extern FILE*  vpts;
extern FILE *fpp, *fpl;

/* Pass from Main to Simulate */
extern float output_Interval; /* frequency of lattice output */
extern int   print_Stats;
extern int   print_Summary;
extern int   print_Grid;
extern int   print_Tumor_Sizes;
extern int   print_Watch;
extern char* print_File;
extern int   print_Files;
extern int   verify_State;
extern int   rng_Seed;
extern int   t_Max_Simulation;
extern int   t_Infect;
extern int   do_Equalization;


/* Pass from Main to Simulate() to InjectVirus() */
extern float percent_Infected;
extern float percent_Interior;
extern enum infection_t infect_Type;

/* Pass from Main to Simulate() to DetermineEventType() */
extern float  rates[N_CELL_TYPES][N_ACTIONS];
```

Simulator.c

```
#include <stdlib.h>
#include <getopt.h>
#include <stdio.h>
#include "Cells.h"

/* Arguments */
static struct option const long_options[] = {
   /* The third value in this structure is not the default setting.
      To determine defaults, please see Parameters in Cells.h */
  /* Long Name      | Argument type    | ? | Short Name */
   {"Equalize",        no_argument,       0, 'E' },
   {"NormalGrowth",    required_argument, 0, 'N' },
   {"NormalDeath",     required_argument, 0, 'n' },
   {"CancerGrowth",    required_argument, 0, 'C' },
   {"CancerDeath",     required_argument, 0, 'c' },
   {"ViralIntGrowth",  required_argument, 0, 'I' },
   {"ViralIntDeath",   required_argument, 0, 'i' },
   {"ViralExtGrowth",  required_argument, 0, 'J' },
   {"ViralExtDeath",   required_argument, 0, 'j' },
   {"PercentInfected", required_argument, 0, 'V' },
   {"InfectionType",   required_argument, 0, 'T' },
   {"PercentInterior", required_argument, 0, 'P' },
   {"CancerRadius",    required_argument, 0, 'r' },
   {"CancerCount",     required_argument, 0, 'q' },
   {"RandomSeed",      required_argument, 0, 'R' },
   {"TimeMax",         required_argument, 0, 'm' },
   {"TimeInfect",      required_argument, 0, 'v' },
   {"OutputInterval",  required_argument, 0, 'o' },
   {"Grid",            no_argument,       0, 'g' },
   {"Stats",           no_argument,       0, 's' },
   {"Summary",         no_argument,       0, 'S' },
   {"TumorSizes",      no_argument,       0, 't' },
   {"Watch",           no_argument,       0, 'W' },
   {"File",            optional_argument, 0, 'f' },
   {"Files",           no_argument,       0, 'F' },
   {"Help",            no_argument,       0, 'H' },
   {"help",            no_argument,       0, 'h' },
   {0,                 0,                 0,  0  }
   /* Add new short options to the while condition */
   /* Update Usage() to reflect changes */
   /* Update Switch() statement to reflect changes */
};

void NormalEquilibrium();
void Usage();
```

```
/** main() ********************************************************
*   Processes commandline arguments
*   Determines when Normal equilibrium is reached
*   Calls Cancer & Virus Seeding at appropriate times
*****************************************************************/
int main (int argc, char* argv[] ) {
   char *data;
   char *network;
   char *coords;
   int optc;
   int long_index = 0;
   int cancer_Radius = 1;
   int cancer_Count = 1;


   /* Process Arguments */
   while( (optc = getopt_long( argc, argv,

"hHEFN:n:C:c:I:i:J:j:m:v:gsfo:P:V:r:SR:tWq:T:",
               long_options, &long_index )) != EOF ) {
      switch( optc ) {
         case 'H':
         case 'h':
            Usage();
            break;
         case 'r':
            cancer_Radius = argtof( optarg );
            break;
         case 'q':
            cancer_Count = argtof( optarg );
            break;
         case 'E':
            do_Equalization = 1;
            break;
         case 'N':
            rates[NORMAL][GROW] = argtof( optarg );
            break;
         case 'n':
            rates[NORMAL][KILL] = argtof( optarg );
            break;
         case 'C':
            rates[CANCER][GROW] = argtof( optarg );
            break;
         case 'c':
            rates[CANCER][KILL] = argtof( optarg );
            break;
         case 'I':
            rates[INFECT_INT][GROW] = argtof( optarg );
            break;
         case 'i':
            rates[INFECT_INT][KILL] = argtof( optarg );
            break;
         case 'J':
            rates[INFECT_EXT][GROW] = argtof( optarg );
```

```
            break;
         case 'j':
            rates[INFECT_EXT][KILL] = argtof( optarg );
            break;
         case 'g':
            print_Grid = 1;
            break;
         case 'S':
            print_Summary = 1;
            break;
         case 'R':
            rng_Seed = (int) argtof( optarg );
            break;
         case 's':
            print_Stats = 1;
            break;
         case 't':
            print_Tumor_Sizes = 1;
            break;
         case 'W':
            print_Watch = 1;
            break;
         case 'P':
            percent_Interior = argtof( optarg );
            break;
         case 'o':
            output_Interval = argtof( optarg );
            break;
         case 'm':
            t_Max_Simulation = argtof( optarg );
            break;
         case 'v':
            t_Infect = argtof( optarg );
            break;
         case 'V':
            percent_Infected = argtof( optarg );
            if( percent_Infected > 100 || percent_Infected < 0 ){
               fprintf(stderr, "PercentInfected must be (0, 100)\n");
               return -1;
            }
            if( percent_Infected < 1 )
            {
               fprintf(stderr, "PercentInfected < 1.\n");
               fprintf(stderr, "Infecting <1%% of Cancer cells\n");
            }
           break;
         case 'T':
            if     ( strcmp(optarg, "RANDOM")    == 0 )
               infect_Type = RANDOM;
            else if( strcmp(optarg, "CENTER")    == 0)
               infect_Type = CENTER;
            else if( strcmp(optarg, "PERIMETER") == 0 )
               infect_Type = PERIMETER;
            else if( strcmp(optarg, "MULTI")     == 0 )
```

```
               infect_Type = MULTINODE;
            else
            {
               fprintf(stderr, "Invalid Infection Type\n");
               return 0;
            }
            break;
         case 'f':
            if( optarg )
               print_File = optarg;
            else
               print_File = "Lattice_State_";
            break;
         case 'F':
            print_Files = 1;
            break;
         case '?':
            printf("Unknown Option.\n");
            printf("Usage: %s [OPTIONS] DAT_FILE NET_FILE XYZ_FILE\n",
                  argv[0]);
            printf("Try '%s --help' for more information.\n", argv[0]);
            return 0;
         default:
            return 0;

      }
   }
   if((argc - optind) != 3 && (argc-optind) != 4 ) {
      printf("Usage: %s [OPTIONS] DAT_FILE NET_FILE XYZ_FILE\n",
argv[0]);
      printf("Try '%s --help' for more information.\n", argv[0]);
      return 0;
   }
   data    = argv[optind  ];
   network = argv[optind+1];
   coords  = argv[optind+2];
   /* Arguments Processed */


   /* Setup Simulation */
   if( print_Summary ) {
      printf(" %d, %f, %f, %f, %f, ", rng_Seed,
                           rates[NORMAL][GROW], rates[NORMAL][KILL],
                           rates[CANCER][GROW], rates[CANCER][KILL] );
      fflush(stdout);
   }
   sgenrand(rng_Seed);
   int return_Val = 0;
   return_Val += ReadLattice( data, network, coords );
   if( return_Val != 0 ) {
      fprintf( stderr, "Problem reading in Lattice\n");
      return( -1 );
   }
```

```
   /* Run Simulation until Normal Equilibrium is Reached */
   if( do_Equalization == 1 ){
      NormalEquilibrium();
      if( print_Stats ) {
         printf(" \n");
         printf("  Normal Cell Equilibrium Stopped!!\n");
         printf("  Equilibrium Value: %20.17f     \n",
                  (double) lttc->n_Cells[NORMAL]/lttc->n_Nodes);
         printf("  Restating Time and Seeding Cancer\n");
         printf(" \n");
      }
   }

   /* Time (re)starts when Cancer is seeded */
   lttc->time = 0;
   if( cancer_Count > 0 )
      SeedCancer( lttc->nodes[lttc->center_Idx], cancer_Count,
cancer_Radius);

   float time_Extinct=0;
   /*
   int time_Check = 1;
   int time_Check_fpl = 1;
   */
   if( print_Files == 1 && t_Infect != 0 ) {
      OpenFiles();
   }
   if( print_File && t_Infect != 0 ) {
      char filename[MAX_LINE_LEN];
      snprintf(filename, MAX_LINE_LEN, "%s%.1f.csv",
                     print_File, lttc->time);
      OutputLatticeState( filename );
   }

   /* Run for t_Infect Days (default 7) before injecting virus */
   int success;
   while( lttc->time < t_Infect ) {
      if( lttc->n_Cells[CANCER] == 0 ) {
         printf("\n");
         fprintf(stderr, "Cancer Cells died out before injecting
virus\n");
         exit(-1);
      }
      if( lttc->n_Cells[NORMAL] == 0 ) {
         printf("\n");
         fprintf(stderr, "Normal Cells died out before injecting
virus\n");
         exit(-1);
      }
      success = Simulate( lttc );
      if ( success == -1 ) {
         fprintf(stderr, "Simulate returned -1.\n");
      }
   }
```

```
   if( infect_Type != NONE ){
      InjectVirus( lttc );
      if( print_Grid ) {
        OutputGrid();
      }
   }

   if( print_Files == 1 && t_Infect == 0 ) {
      OpenFiles();
   }
   if( print_File && t_Infect == 0 ) {
      char filename[MAX_LINE_LEN];
      snprintf(filename, MAX_LINE_LEN, "%s%.1f.csv",
                     print_File, lttc->time);
      OutputLatticeState( filename );
   }

   /* Run all but last days of Simulation */
   int n_LinReg = 10;
   while( lttc->time <= t_Max_Simulation - n_LinReg*output_Interval ) {
      unsigned long int n_Infect = lttc->n_Cells[INFECT_INT] +
                                   lttc->n_Cells[INFECT_EXT] ;
      if( print_Files == 1 ) {
         /* Stop if Cancer or Virus Dies Out */
         if( lttc->n_Cells[CANCER] == 0 &&
                     time_Extinct == 0 ) {
            time_Extinct = lttc->time;
         }
         if(      percent_Infected  > 0 &&
                          n_Infect == 0 &&
                     time_Extinct == 0   ) {
            time_Extinct = lttc->time;
         }
      }
      if( ( ( lttc->n_Cells[CANCER] == 0 && n_Infect == 0 ) ||
            ( percent_Infected > 0 && n_Infect == 0       )    )
           && lttc->time > 126 ) {
          if( print_Stats ) {
             OutputStats( lttc );
          }
          if( print_Grid ) {
             OutputGrid();
          }


         if( print_Files == 1 ) {
            ParameterFile( time_Extinct );
         }
         break;
      }


      success = Simulate( lttc );
      if ( success == -1 ) {
```

```
      fprintf(stderr, "Simulate returned -1.\n");
   }
}

/* Run last days of Simulation */
int timeIt = 0;
int typeIt;
float output_Timer = t_Max_Simulation - n_LinReg*output_Interval ;
float *X_Vals = calloc( n_LinReg, sizeof(float) );
float *Y_Vals[N_CELL_TYPES];
for( typeIt = NORMAL; typeIt < N_CELL_TYPES; typeIt++ ) {
   Y_Vals[typeIt] = calloc( n_LinReg, sizeof(float) ) ;
}
while( lttc->time < t_Max_Simulation ) {
   /* Save Time & n_Cells every output_Interval */
   if( lttc->time > output_Timer ) {
      output_Timer += output_Interval;
      X_Vals[timeIt] = (float) lttc->time;
      for( typeIt = NORMAL; typeIt <= INFECT_EXT; typeIt++ ) {
         Y_Vals[typeIt][timeIt] = (float) lttc->n_Cells[typeIt];
      }
      timeIt++;
   }
   /* Stop if Cancer or Virus Dies Out */
   unsigned long int n_Infect = lttc->n_Cells[INFECT_INT] +
                                lttc->n_Cells[INFECT_EXT] ;
   if( ( lttc->n_Cells[CANCER] == 0 && n_Infect == 0 ) ||
       (       percent_Infected > 0 && n_Infect == 0 )    ) {
      free(X_Vals);
      X_Vals = NULL;
      for( typeIt = NORMAL; typeIt <= INFECT_EXT; typeIt++ ) {
         free(Y_Vals[typeIt]);
         Y_Vals[typeIt] = NULL;
      }

      if( print_Files == 1 ) {
         CloseFiles();
      }

      return 0;
   }
   success = Simulate( lttc );
   if ( success == -1 ) {
      fprintf(stderr, "Simulate returned -1.\n");
   }
}

/* Simulation Complete -- Wrap-Up */
if( print_Files == 1 ) {
   ParameterFile( 0.0 );
   CloseFiles();
}
if( print_Summary )
{
```

```
      float slope;
      printf(" Slopes: ");
      for( typeIt = NORMAL; typeIt <= N_CELL_TYPES; typeIt++ )
      {
         LinReg( X_Vals, Y_Vals[typeIt], n_LinReg, &slope, NULL );
         printf("%f ", slope);
      }

      printf(" Counts: ");
      for( typeIt = NORMAL; typeIt <= N_CELL_TYPES; typeIt++ )
         printf("%lu ", lttc->n_Cells[typeIt]);
      printf("\n");
   }
   free(X_Vals);
   X_Vals = NULL;
   for( typeIt = NORMAL; typeIt < N_CELL_TYPES; typeIt++ ) {
      free(Y_Vals[typeIt]);
      Y_Vals[typeIt] = NULL;
   }

   DeleteLattice( lttc );
   return 0;

}/* End of main() */

void NormalEquilibrium(){
   int success;
   float convergance_Interval = .5;
   float output_Timer = convergance_Interval;
   int timer_Reset = 0;
   int n_Avg = 5;
   int n_Cnt = 30;
   int *converge = calloc ( n_Avg*2, sizeof(int) );
   int It, conv_It;
   for( It = 0; 1; It++ ) {
      if( lttc->n_Cells[NORMAL] == 0 ) {
         printf("\n");
         fprintf(stderr, "Normal Cells died out prior to
equilibirum.\n");
         free( converge );
         exit(-1);
      }
      success = Simulate( lttc );
      if ( success == -1  ) {
         printf("\n");
         fprintf(stderr, "Simulate returned -1.\n");
         continue;
      }
      timer_Reset=0;

      /* Convergance Check.  If found: continue 30 steps & return */
      if( lttc->time > output_Timer ) {
         output_Timer += convergance_Interval;
         timer_Reset = 1;
```

```
      }
      if( timer_Reset == 1 ) {
         converge[conv_It % (2*n_Avg)] = lttc->n_Cells[NORMAL];
         if( conv_It  == 9 ) {
            int i, sum_A = 0, sum_B = 0;
            sum_A = 0;
            sum_B = 0;
            for( i = 0; i < n_Avg; i++ ) {
               sum_A += converge[ i ];
               sum_B += converge[ i + n_Avg ];
            }

            if( sum_A < sum_B ) {
               if( print_Stats ) {
                  printf(" \n  Normal Cell Equilibrium Flagged!! \
                        %20.17f      %20.17f     \n \n",
                        (double) sum_A/(n_Avg*lttc->n_Nodes),
                        (double) sum_B/(n_Avg*lttc->n_Nodes));
               }
               int j=0;
               do{
                  success = Simulate( lttc );
                  if ( success == -1 ) {
                     fprintf(stderr, "Simulate returned -1.\n");
                     continue;
                  }
                  if( lttc->time > output_Timer ) {
                     output_Timer += convergance_Interval;
                     timer_Reset = 1;
                  }
                  if( timer_Reset == 1 )
                     j++;
                  timer_Reset = 0;
               } while ( j <= n_Cnt );
               break;
            }
         }
         conv_It = (conv_It+1) % (2*n_Avg);
      }
   }

   free( converge );
   converge = NULL;
   return;
}
```

```
/** Usage()*****************************************************
* Print usage information
*************************************************************/
void Usage () {
  fprintf(stderr, "Usage: Cells [option(s)] datFILE, netFILE,
coordFILE\n");
  fprintf(stderr, " The simulator options are:\n\
  -N --NormalGrowth=NUM     Growth rate of normal cells (default 0)\n\
  -n --NormalDeath=NUM      Death  rate of normal cells (default 0)\n\
  -C --CancerGrowth=NUM     Growth rate of cancer cells (default 0)\n\
  -c --CancerDeath=NUM      Death  rate of cancer cells (default 0)\n\
  -I --ViralIntGrowth=NUM   Growth rate of Interior viral  cells
(default 0)\n\
  -i --ViralIntDeath=NUM    Death  rate of Interior viral  cells
(default 0)\n\
  -J --ViralExtGrowth=NUM   Growth rate of Exterior viral  cells
(default 0)\n\
  -j --ViralExtDeath=NUM    Death  rate of Exterior viral  cells
(default 0)\n\
  -R --RandomSeed=NUM       Seed for random number generator (default
11)\n\
  -E --Equalize             Don't start time until normal population is
stable\n\
  -r --CancerRadius=NUM     Cancer ball radius in neighbors (default
1)\n\
  -q --CancerCount=NUM      Number of cancer nodes to seed (default 1)
\n\
  -v --TimeInfect=INTEGER   Time between cancer and virus seeding
(default 7)\n\
  -V --PercentInfected=NUM  Percentage of cancer nodes that become
viral\n\
  -T --InfectionType=TYPE   TYPE=[RANDOM|CENTER|PERIMETER|MULTINODE]\n\
  -P --PercentInterior=NUM  Percent of Cancer neighbors for interior\n\
  -m --TimeMax=INTEGER      Maximum time the simulator should run
(default 75)\n\
The output options are:\n\
  -o --OutputInterval=NUM   Set output frequency (default .5)\n\
  -g --Grid                 Print square 2d grid to stdout.\n\
  -s --Stats                Print stats to stdout.\n\
  -S --Summary              Print summary to stdout.\n\
  -t --TumorSizes           Print number of tumors and their size\n\
  -W --Watch                Wait for return key after every output\n\
  -F --Files                Output to vpts.dat, vlpt.dat, and
vout.dat\n\
  -f --File=BASENAME        Save lattice states to
BASENAME_time.csv\n");
  exit(-1);
}/* End of Usage() */
```

Queue.c

```
#include  "Queue.h"

/* NewQueue() ***************************************************/
/*  Size indicates initial size of array used to store Queue    */
/*  Queue will double in size as needed to make room for new objects */
/****************************************************************/
struct Queue* const NewQueue( int size ) {
   struct Queue* const Q = malloc( sizeof(struct Queue) );
   if( Q == NULL ){
      return NULL;
   }
   Q->q = calloc( size, sizeof(void*) );
   if( Q->q == NULL ){
      return NULL;
   }
   Q->tail = 0;
   Q->head = 0;
   Q->length = size;

   return Q;
}
/* DeleteQueue() ************************************************/
/* Frees memory associated with Q                               */
/****************************************************************/
void DeleteQueue( struct Queue* const Q ) {
   free( Q->q );
   free( Q );
   return;
}

/* Enqueue() ****************************************************/
/*  adds object x to Queue Q                                    */
/*  Q will double in size as needed to make room for new objects     */
/****************************************************************/
int Enqueue( struct Queue* const Q, void* x ) {
   if( Q->head == (Q->tail + 1) ) {
      if( GrowQueue( Q ) == 0 )
         return 0;
   }
   Q->q[Q->tail] = x;
   if(Q->tail == Q->length-1)
      Q->tail = 0;
   else
      Q->tail += 1;
   return 1;
}
```

```
/* Enqueue() ***************************************************/
/*  adds object x to Queue Q                                   */
/*  Q will not shrink regardless of how many elements are removed    */
/***************************************************************/
void* Dequeue( struct Queue* const Q ) {
   void* x = Q->q[Q->head];
   Q->q[Q->head] = NULL;

   if( Q->head == Q->tail )
      return NULL;
   if( Q->head == Q->length-1 )
      Q->head = 0;
   else
      Q->head += 1;

   return x;
}

/* GrowQueue() *************************************************/
/*   Doubles the size of Queue and moves elements as needed to       */
     maintain the appropriate structure                          */
/***************************************************************/
int GrowQueue( struct Queue* const Q ) {
   void** temp;
   temp = realloc(Q->q, Q->length*2 * sizeof(void *));
   if( temp == NULL ){
      return 0;
   }
   Q->q = temp;
   if( Q->tail < Q->head ){
      int It;
      for(It = 0; It < Q->tail; It++)
         Q->q[Q->length+It] = Q->q[It];

      Q->tail += Q->length;
   }
   Q->length *= 2;

   return 1;
}
```

```
/* Display() ***************************************************/
/* Outputs contents of Q and indicates where the head and tail are.  */
/*    Helpful for debugging or understanding how the Queue works.    */
/*                                                                   */
/* Assumes that Q's elements are char*.                              */
/* Add argument providing a function to print actual type.           */
/*********************************************************************/
void Display( struct Queue* const Q ) {
   int It = 0;
   for(It = 0; It < Q->length; It++) {
      if( (It <  Q->tail &&     It >= Q->head) ||
          (It <  Q->tail && Q->tail < Q->head) ||
          (It >= Q->head && Q->tail < Q->head) ) {
         char c = *(char*)Q->q[It];
         printf("%c", c );
      }
      else printf("-");
   }
   puts("");

   for(It = 0; It < Q->length; It++ ){
      if(It == Q->head)
         putchar('h');
      else if(It == Q->tail)
         putchar('t');
      else if( (It <  Q->tail &&     It >= Q->head) ||
          (It <  Q->tail && Q->tail < Q->head) ||
          (It >= Q->head && Q->tail < Q->head) )
         putchar('-');
      else
         putchar(' ');
   }
   puts("");
}
```

Queue.h

```
#ifndef DB_QUEUE_HEADER_FILE
#define DB_QUEUE_HEADER_FILE

#include <stdio.h>
#include <stdlib.h>

/******************/
/* Public Methods */
/******************/

/* Returns the address of a new Queue with space for "size" void
pointers */
struct Queue* const   NewQueue( int size );

/* Adds a new void pointer (x) to Queue Q.  If Q is full, it's size
doubles.
   Returns 0 if Q tried to grow but failed, otherwise returns 1      */
int              Enqueue( struct Queue* const Q, void* x  );

/* Returns and removes a void pointer from Queue Q. */
void*            Dequeue( struct Queue* const Q );

/* Free's Queue Q and associated memory */
void         DeleteQueue( struct Queue* const Q );

/* Displays value at each address in Q as characters separated by
spaces on one line and markers indicating head and tail on the second.
Useful primarily as a way to visualize how the Queue works.  */
/* Future version will allow specification of an argument to print
   non-character types */
void            Display( struct Queue* const Q );



/*******************/
/* Private Methods */
/*******************/

struct Queue {
   int tail;
   int head;
   int length;
   void** q;
};

/* Routine to double the size of a full Queue and
   move elements to the new location.               */
int            GrowQueue( struct Queue* const Q );

#endif
```

cartesian.c

```
#include "cartesian.h"

/*************************************/
/*************************************/
/*                                   */
/*          Public Routines          */
/*                                   */
/*************************************/
/*************************************/

/*************************************************************/
/*                           POINTS                          */
/*************************************************************/
/* Creates a new Point;  */
/* Arguments 2->N+1 are expected to be cartesian coordinates */
Point* NewPoint( short unsigned N, ... ) {
   va_list arguments;
   va_start ( arguments, N );
   Point *P = NewPoint2(N, arguments);
   va_end ( arguments );

   return P;
}
void DeletePoint( Point* P ) {
   if( P != NULL )
      free( P->data );
   free( P );
   P = NULL;
   return;
}


/*************************************************************/
/*                           VECTORS                         */
/*************************************************************/
/* Creates a new Vector;  Be sure to pass floats/doubles */
/* Arguments 2->N+1 are expected to be cartesian coordinates */
Vector* NewVector( short unsigned N, ... ) {
   va_list arguments;
   va_start ( arguments, N );
   Vector *V = (Vector *) NewPoint2(N, arguments);
   va_end ( arguments );

   return  V;
}
void DeleteVector( Vector* V ) {
   DeletePoint( (Point*) V );
   return;
}
```

```
/*************************************************************/
/*                           LINES                           */
/*************************************************************/

/* Create a new Line with slope S that intersects point P */
Line*  NewLine( Point* P, Vector* S ){
   Line *L = malloc( sizeof( *L ));
   if( L == NULL || P->N != S->N ) {
      free( L );
      L = NULL;
      return NULL;
   }

   L->N = P->N;
   L->P = P;
   L->D = S;
   return L;
}
void DeleteLine( Line* L ){
   free(L);
   L = NULL;
}


/*************************************************************/
/*                      UTILITY FUNCTIONS                    */
/*************************************************************/
double Point2PointDistance( Point* P1, Point* P2 ){
   double distance = 0;
   unsigned short It;
   for(It = 0; It < P1->N; It++){
      double temp = (P1->data[It]-P2->data[It]);
      distance +=  temp * temp;
   }

   return sqrt(distance);
}

double Point2LineDistance( Point *P, Line *L ){
   double slope=0;
   double temp1=0;
   double temp2=0;
   double distance = 0;
   double vector[P->N];
   unsigned short It;

   if( P->N != L->N )
      return 0;

   /* Calculate Slope */
   for(It = 0; It < P->N; It++){
      temp1 += L->D->data[It] * (L->P->data[It] - P->data[It]);
      temp2 += L->D->data[It] * L->D->data[It];
   }
```

```
   slope = -1 * temp1 / temp2;

   for(It = 0; It < P->N; It++)
      vector[It] = L->P->data[It] + L->D->data[It] * slope - P-
>data[It];


   for(It = 0; It < P->N; It++)
      distance += vector[It] * vector[It];

   return sqrt(distance);
}


/*************************************/
/*************************************/
/*                                   */
/*         Private Routines          */
/*                                   */
/*************************************/
/*************************************/

/* Creates a new point; */
Point* NewPoint2( short unsigned N, va_list arguments){
   /* Create Space */
   Point *P = malloc( sizeof( *P ) );
   if( P == NULL )
      return P;
   P->data = malloc( N * sizeof( *P->data ));
   if( P->data == NULL ) {
      free( P );
      P = NULL;
      return P;
   }

   /* Initialize */
   P->N = N;
   int It;
   for( It = 0; It < N; It++ )        {
      P->data[It] =  va_arg( arguments, double );
   }

   return P;
}
```

cartesian.h

```
#ifndef DB_CARTESIAN_H
#define DB_CARTESIAN_H

#include <math.h>
#include <stdarg.h>
#include <stdlib.h>

typedef struct Point Point;
typedef struct Vector Vector;
typedef struct Line Line;

/* Constructors allow creation of 1D & 2D objects in N-dimensional
space.
   N is an unsigned short integer which is at least 2^16-1 */
Point* NewPoint( unsigned short N, ... /*doubles*/ ) ;
Vector* NewVector( unsigned short N, ... /*doubles*/ ) ;
Line* NewLine( Point* P, Vector* Dir ); /* P is on line,
                            Dir is vector parallel to line */

/* If P & L have different dimensions Point2LineDistance returns 0;
   Otherwise it returns the shortest distance from P to L */
double Point2LineDistance( Point *P, Line *L );
double Point2PointDistance( Point* P1, Point* P2);

/* Destructors */
void DeleteVector( Vector* V );
void DeletePoint( Point* P );
void DeleteLine( Line* L );

/* Private Members */
Point* NewPoint2( unsigned short N, va_list arguments);

struct Point {
   double  * data;
   unsigned short N;
};
struct Vector {
   double  * data;
   unsigned short N;
};
struct Line {
  Point* P;  //Point on line
  Vector* D; //Direction of Line
  unsigned short N;
};

#endif
```

```
/*************************************************************************
  Unit tests for routines in Cells.h
*************************************************************************/

#include "Cells.h"

#define RED     "\x1b[31m"
#define GREEN   "\x1b[32m"
#define RESET   "\x1b[0m"

int checkn_Nbrs();
int CheckNodeType();
int CompareLinks ( enum nodeType_t type ) ;

int main () {
   unsigned long int nodeIt;

   char* data    = "Networks/10x10.dat";
   char* network = "Networks/10x10.net";
   char* coords  = "Networks/10x10.cor";
   int return_Val = 0;
   unsigned long int* sizes = NULL;
   printf("Read in & Verify 10^2 grid     : ");
   return_Val = ReadLattice( data, network, coords );
   if( return_Val == 0 )
      return_Val = VerifyState(  );
   switch( return_Val ){
      case 0:
     printf(GREEN "PASSED\n" RESET);
     break;
       default:
     printf(RED "FAILED\n" RESET);
     break;
   }

   if( return_Val == 0 ){
   printf("Check n_Nbrs : ");
   if( checkn_Nbrs() == 0 )
      printf(GREEN "PASSED\n" RESET);
   else
      printf(RED "FAILED\n" RESET);
   fflush(stdout);
   struct node_t * center = lttc->nodes[lttc->center_Idx];
   if( return_Val == 0 )
   {
      printf("Verifying 0 Tumors with GetNetworkDetails() :");
      sizes = GetNetworkDetails( lttc->cells[CANCER], lttc-
>n_Cells[CANCER] );
      if( (sizes[0]==0) )
     printf(GREEN "PASSED\n" RESET);
      else
     printf(RED "FAILED\n" RESET);
      free( sizes );
      sizes = NULL;
```

```
      fflush(stdout);

      printf("Seeding 2 Tumors & Verify Count: ");
      SeedCancer(lttc->nodes[0], 2, 2);
      SeedCancer(center, 3, 2);
      if( lttc->n_Cells[CANCER] == 26 &&
          lttc->n_Cells[NORMAL] == 74 &&
          lttc->n_Cells[INFECT_INT] == 0  &&
          lttc->n_Cells[INFECT_EXT] == 0 )
    printf(GREEN "PASSED\n" RESET);
      else
    printf(RED "FAILED\n" RESET);
      fflush(stdout);

      printf("Verifying Tumors with GetNetworkDetails() :");
      sizes = GetNetworkDetails( lttc->cells[CANCER], lttc-
>n_Cells[CANCER] );
      if( (sizes[0]==2) && (sizes[1]==13) && (sizes[2]==13) )
    printf(GREEN "PASSED\n" RESET);
      else
    printf(RED "FAILED\n" RESET);
      free(sizes);
      sizes = NULL;
      fflush(stdout);

      printf("Verifying n_Targets[INFECT] :");
      if( lttc->n_Target_Nodes[INFECT_INT][0] == 0 &&
          lttc->n_Target_Nodes[INFECT_INT][1] == 0 &&
          lttc->n_Target_Nodes[INFECT_INT][2] == 0 &&
          lttc->n_Target_Nodes[INFECT_INT][3] == 0 &&
          lttc->n_Target_Nodes[INFECT_INT][4] == 0 &&
          lttc->n_Target_Nodes[INFECT_EXT][0] == 0 &&
          lttc->n_Target_Nodes[INFECT_EXT][1] == 0 &&
          lttc->n_Target_Nodes[INFECT_EXT][2] == 0 &&
          lttc->n_Target_Nodes[INFECT_EXT][3] == 0 &&
          lttc->n_Target_Nodes[INFECT_EXT][4] == 0 )
    printf(GREEN "PASSED\n" RESET);
      else
    printf(RED "FAILED\n" RESET);
      fflush(stdout);

      printf("Infect 2 Tumors & Verify Count: ");
      VirusBallAroundNode(center, 13);
      VirusBallAroundNode(lttc->nodes[0], 13);
      if( lttc->n_Cells[INFECT_INT] + lttc->n_Cells[INFECT_EXT] == 26 )
    printf(GREEN "PASSED\n" RESET);
      else
    printf(RED "FAILED\n" RESET);
      fflush(stdout);

      printf("Verifying with GetNetworkDetails() :");
      sizes = GetNetworkDetails( lttc->cells[CANCER], lttc-
```

```
>n_Cells[CANCER] );
      if( (sizes[0]==0) )
    printf(GREEN "PASSED\n" RESET);
      else
    printf(RED "FAILED\n" RESET);
      free(sizes);
      sizes = NULL;
      fflush(stdout);

      printf("Verifying with n_Targets[INFECT] :");
      if( lttc->n_Target_Nodes[INFECT_INT][0] == 0 &&
          lttc->n_Target_Nodes[INFECT_INT][1] == 0 &&
          lttc->n_Target_Nodes[INFECT_INT][2] == 0 &&
          lttc->n_Target_Nodes[INFECT_INT][3] == 0 &&
          lttc->n_Target_Nodes[INFECT_INT][4] == 0 &&
          lttc->n_Target_Nodes[INFECT_EXT][0] == 0 &&
          lttc->n_Target_Nodes[INFECT_EXT][1] == 0 &&
          lttc->n_Target_Nodes[INFECT_EXT][2] == 0 &&
          lttc->n_Target_Nodes[INFECT_EXT][3] == 0 &&
          lttc->n_Target_Nodes[INFECT_EXT][4] == 0 )
    printf(GREEN "PASSED\n" RESET);
      else
    printf(RED "FAILED\n" RESET);
      fflush(stdout);

      printf("Seed 2x13node tumors, then infect 14 nodes.  Verify Count
& n_Nbrs:");
      SeedCancer(lttc->nodes[0], 2, 2);
      SeedCancer(center, 3, 2);
      VirusBallAroundNode(center, 14);
      if( lttc->n_Cells[INFECT_INT] + lttc->n_Cells[INFECT_EXT] == 14 &&
                                          checkn_Nbrs() ==  0 )
    printf(GREEN "PASSED\n" RESET);
      else
    printf(RED "FAILED\n" RESET);
      fflush(stdout);
   }



   printf("Testing Viral Growth from 1 node to full & verify n_Links &
target types: ");
   SeedCancer(center, 3, 10*10);
   VirusBallAroundNode(center, 1);
   rates [NORMAL][GROW] = 0; rates [NORMAL][KILL] = 0;
   rates [CANCER][GROW] = 0; rates [CANCER][KILL] = 0;
   rates [INFECT_INT][GROW] = 10; rates [INFECT_INT][KILL] = 0;
   rates [INFECT_EXT][GROW] = 10; rates [INFECT_EXT][KILL] = 0;
   int It;
   for( It = 1; It < 10*10; It++) {
      Simulate();

      if( CheckNodeType() != 0 )
         break;
```

```
      if( CompareLinks(INFECT_EXT)+CompareLinks(INFECT_INT) != 0 )
         break;
      if( checkn_Nbrs() != 0 )
         break;
   }
   if( lttc->n_Cells[INFECT_INT]+lttc->n_Cells[INFECT_EXT] == 10*10 )
      printf(GREEN "PASSED\n" RESET);
   else
      printf(RED "FAILED\n" RESET);
   fflush(stdout);

   rates [NORMAL][GROW] =  0; rates [NORMAL][KILL] = 0;
   rates [CANCER][GROW] = 10; rates [CANCER][KILL] = 0;
   rates [INFECT_INT][GROW] =  0; rates [INFECT_INT][KILL] = 0;
   rates [INFECT_EXT][GROW] =  0; rates [INFECT_EXT][KILL] = 0;
   printf("Testing Cancer Growth Next to Virus: ");
   SeedCancer(center, 10*10, 1);
   VirusBallAroundNode(center, 13);
   ChangeNodeType(lttc->nodes[41], VACANT);
   ChangeNodeType(lttc->nodes[32], VACANT);
   ChangeNodeType(lttc->nodes[23], VACANT);
   ChangeNodeType(lttc->nodes[14], VACANT);
   ChangeNodeType(lttc->nodes[25], VACANT);
   ChangeNodeType(lttc->nodes[36], VACANT);
   ChangeNodeType(lttc->nodes[47], VACANT);
   ChangeNodeType(lttc->nodes[56], VACANT);
   ChangeNodeType(lttc->nodes[65], VACANT);
   ChangeNodeType(lttc->nodes[74], VACANT);
   ChangeNodeType(lttc->nodes[63], VACANT);
   ChangeNodeType(lttc->nodes[52], VACANT);
   {
      int error = 0;
      while (lttc->n_Cells[CANCER] < 87 )
      {
         Simulate();
         if( VerifyState() == 0 &&
             checkn_Nbrs() == 0 &&
             CheckNodeType() == 0 &&
             CompareLinks(INFECT_INT) + CompareLinks(INFECT_EXT) == 0 )
            continue;
         else{
            printf(RED "FAILED\n" RESET);
            error = 1;
            break;
         }

      }
      if( error == 0 )
         printf(GREEN "PASSED\n" RESET);
      fflush(stdout);
   }

   {
      printf("Changing nodes to Vacant then to Cancer: ");
```

```
      int error = 0;
      for( nodeIt = 0; nodeIt < lttc->n_Nodes; nodeIt++){
         struct node_t* node = lttc->nodes[nodeIt];
         ChangeNodeType( node, VACANT );
         ChangeNodeType( node, CANCER );
         if( VerifyState() == 0 )
            continue;
         else{
            printf(RED "FAILED\n" RESET);
            error = 1;
            break;
         }
      }
      if( error == 0 )
         printf(GREEN "PASSED\n" RESET);
      fflush(stdout);
   }
   {
      printf("Testing Normal Growth next to target: ");
      for( nodeIt = 0; nodeIt < lttc->n_Nodes; nodeIt++){
         struct node_t* node = lttc->nodes[nodeIt];
         ChangeNodeType( node, NORMAL );
      }
      ChangeNodeType( lttc->nodes[0], VACANT );
      ChangeNodeType( lttc->nodes[1], VACANT );
      ChangeNodeType( lttc->nodes[1], NORMAL );
      if( VerifyState() == 0 )
         printf(GREEN "PASSED\n" RESET);
      else
    printf(RED "FAILED\n" RESET);
      fflush(stdout);
   }
   {
      extern float percent_Infected;
      printf("Random Infection Count:");
      percent_Infected = 20;
      infect_Type=RANDOM;
      for( nodeIt = 0; nodeIt < lttc->n_Nodes; nodeIt++)
    ChangeNodeType( lttc->nodes[nodeIt], CANCER );
      InjectVirus();
      sizes = GetNetworkDetails( lttc->cells[INFECT_INT], lttc-
>n_Cells[INFECT_INT] );
      if( sizes[0] > 10 && lttc->n_Cells[INFECT_INT] ==
percent_Infected/100 * lttc->n_Nodes )
         printf(GREEN "PASSED\n" RESET);
      else
    printf(RED "FAILED\n" RESET);
      fflush(stdout);
      free(sizes);
      sizes = NULL;

      printf("Center Infection Count:");
      for( nodeIt = 0; nodeIt < lttc->n_Nodes; nodeIt++)
    ChangeNodeType( lttc->nodes[nodeIt], CANCER );
```

```
      infect_Type=CENTER;
      InjectVirus();
      sizes = GetNetworkDetails( lttc->cells[INFECT_INT], lttc-
>n_Cells[INFECT_INT] );
      if( sizes[0] == 1 && sizes[1] == percent_Infected/100 * lttc-
>n_Nodes )
         printf(GREEN "PASSED\n" RESET);
      else
    printf(RED "FAILED\n" RESET);
      fflush(stdout);
      free(sizes);
      sizes = NULL;

      printf("Multi Infection Count:");
      percent_Infected = 18.75;
      for( nodeIt = 0; nodeIt < lttc->n_Nodes; nodeIt++){
    enum nodeType_t type = CANCER;
    if( lttc->nodes[nodeIt]->pos->data[0] == 1  ||
        lttc->nodes[nodeIt]->pos->data[0] == 10 ||
        lttc->nodes[nodeIt]->pos->data[1] == 1  ||
        lttc->nodes[nodeIt]->pos->data[1] == 10 )
       type = NORMAL;

    ChangeNodeType( lttc->nodes[nodeIt], type );
      }
      infect_Type=MULTINODE;
      InjectVirus();
      sizes = GetNetworkDetails( lttc->cells[INFECT_INT], lttc-
>n_Cells[INFECT_INT] );
      int passed = 0;
      for(It = 1; It <= sizes[0]; It++){
    if( sizes[It] % (int) percent_Infected/3/100*64 != 0 ){
       passed = 0;
       break;
    }
    else
       passed = 1;
      }
      free(sizes);

      int n_Infected = lttc->n_Cells[INFECT_EXT] +
                       lttc->n_Cells[INFECT_INT] ;
      if( passed == 1 && n_Infected == (int) (percent_Infected/100*64) )
         printf(GREEN "PASSED\n" RESET);
      else
    printf(RED "FAILED\n" RESET);
      fflush(stdout);
      printf("Perimeter Infection Count:");
      percent_Infected = 18.75;
      for( nodeIt = 0; nodeIt < lttc->n_Nodes; nodeIt++) {
    enum nodeType_t type = CANCER;
    if( lttc->nodes[nodeIt]->pos->data[0] == 1  ||
        lttc->nodes[nodeIt]->pos->data[0] == 10 ||
        lttc->nodes[nodeIt]->pos->data[1] == 1  ||
```

```
       lttc->nodes[nodeIt]->pos->data[1] == 10 )
      type = NORMAL;

  ChangeNodeType( lttc->nodes[nodeIt], type );
    }
    infect_Type=PERIMETER;
    InjectVirus();
    n_Infected = lttc->n_Cells[INFECT_EXT] +
                 lttc->n_Cells[INFECT_INT] ;
    if( n_Infected == (int) (percent_Infected/100*64) )
  printf(GREEN "PASSED\n" RESET);
    else
  printf(RED "FAILED\n" RESET);
    fflush(stdout);

}
DeleteLattice( lttc );
}

/* Read in 10^3 Grid */
data    = "Networks/10x10x10.dat";
network = "Networks/10x10x10.net";
coords  = "Networks/10x10x10.cor";
return_Val = 0;

printf("Read in & Verify 10^3 grid     : ");
return_Val = ReadLattice( data, network, coords );
if( return_Val == 0 )
   return_Val = VerifyState(  );
switch( return_Val ){
   case 0:
  printf(GREEN "PASSED\n" RESET);
  break;
   default:
  printf(RED "FAILED\n" RESET);
  return 0;
  break;
}
   printf("Perimeter Infection Count:");
   percent_Infected = (104.0/512.0)*100.0;
   for( nodeIt = 0; nodeIt < lttc->n_Nodes; nodeIt++) {
  enum nodeType_t type = CANCER;
  if( lttc->nodes[nodeIt]->pos->data[0] == 1  ||
      lttc->nodes[nodeIt]->pos->data[0] == 10 ||
      lttc->nodes[nodeIt]->pos->data[1] == 1  ||
      lttc->nodes[nodeIt]->pos->data[1] == 10 ||
          lttc->nodes[nodeIt]->pos->data[2] == 1  ||
          lttc->nodes[nodeIt]->pos->data[2] == 10 )
        type = NORMAL;

  ChangeNodeType( lttc->nodes[nodeIt], type );
    }
    infect_Type=PERIMETER;
    InjectVirus();
```

```
      int n_Infected = lttc->n_Cells[INFECT_EXT] +
                  lttc->n_Cells[INFECT_INT] ;
      if( n_Infected == (int) (percent_Infected/100*512) )
     printf(GREEN "PASSED\n" RESET);
      else
     printf(RED "FAILED\n" RESET);
      fflush(stdout);
      OutputLatticeState( "output.csv" );
   printf("Multi Infection Count:");
   /* This also tests correct behaviour of VirusBallAroundNode( node, N
)
      when node is already infected. */
   percent_Infected = 18.75;
   for( nodeIt = 0; nodeIt < lttc->n_Nodes; nodeIt++) {
      enum nodeType_t type = CANCER;
      if( lttc->nodes[nodeIt]->pos->data[0] == 1  ||
          lttc->nodes[nodeIt]->pos->data[0] == 10 ||
          lttc->nodes[nodeIt]->pos->data[1] == 1  ||
          lttc->nodes[nodeIt]->pos->data[1] == 10 ||
          lttc->nodes[nodeIt]->pos->data[2] == 1  ||
          lttc->nodes[nodeIt]->pos->data[2] == 10 )
     type = NORMAL;

      ChangeNodeType( lttc->nodes[nodeIt], type );
   }
   infect_Type=MULTINODE;
   InjectVirus();
   sizes = GetNetworkDetails( lttc->cells[INFECT_EXT], lttc-
>n_Cells[INFECT_EXT] );
   int passed = 0;
   int It;
   for(It = 1; It <= sizes[0]; It++){
      if( sizes[It] % (int) percent_Infected/3/100*512 != 0 ){
     passed = 0;
     break;
      }
      else
     passed = 1;
   }
   free(sizes);

   n_Infected = lttc->n_Cells[INFECT_EXT] +
       lttc->n_Cells[INFECT_INT] ;
   if( passed == 1 && n_Infected == (int) (percent_Infected/100*512) )
      printf(GREEN "PASSED\n" RESET);
   else
      printf(RED "FAILED\n" RESET);
   fflush(stdout);

   printf("Multi Infection Check Center:");
   Point* point = CenterOfMass(lttc->cells[INFECT_EXT], lttc-
>n_Cells[INFECT_EXT]);
   passed = 0;
```

```
   for( It = 0; It < point->N; It++ ){
      if( abs(5.5-point->data[It]) > 1.5 ){
     passed = 0;
     break;
      }
      else passed = 1;
   }
   if( passed == 1 )
      printf(GREEN "PASSED\n" RESET);
   else
      printf(RED "FAILED\n" RESET);
   fflush(stdout);

   DeletePoint( point );
   DeleteLattice( lttc );

   data    = "Networks/3d_grid_10.dat";
   network = "Networks/3d_grid_10.net";
   coords  = "Networks/3d_grid_10.state";
   return_Val = 0;

   printf("Read in & Verify 10^3 grid     : ");
   return_Val = ReadLattice( data, network, coords );
   if( return_Val == 0 )
      return_Val = VerifyState(  );
   switch( return_Val ){
      case 0:
     printf(GREEN "PASSED\n" RESET);
     break;
      default:
     printf(RED "FAILED\n" RESET);
     break;
   }
   DeleteLattice( lttc );

   return 0;
}

int checkn_Nbrs()
{
   int typeIt, nbrIt;
   unsigned long int nodeIt;
   struct node_t *node;
   for(nodeIt = 0; nodeIt < lttc->n_Nodes; nodeIt++) {
      int n_Nbrs[N_CELL_TYPES+1] = {0,0,0,0};
      node = lttc->nodes[nodeIt];
      for( nbrIt = 0; nbrIt < node->n_Neighbors; nbrIt++ )
         n_Nbrs[ node->neighbors[nbrIt]->type ]++;

      for( typeIt=0; typeIt < N_CELL_TYPES; typeIt++) {
         if(n_Nbrs[typeIt] != node->n_Nbrs[typeIt]) {
            fprintf(stderr, "n_Nbrs count is off\n");
            return(-1);
         }
```

```
      }
   }

   return 0;
}

int CheckNodeType()
{
   int typeIt;
   for( typeIt = NORMAL; typeIt < N_CELL_TYPES; typeIt++) {
      int n_NbrsIt;
      for(n_NbrsIt = 0; n_NbrsIt <= lttc->n_Nbrs_Max; n_NbrsIt++ ){
         unsigned long int nodeIt;
         for(nodeIt = 0; nodeIt < lttc->n_Target_Nodes[typeIt]
[n_NbrsIt]; nodeIt++ ){
            if( lttc->target_Nodes[typeIt][n_NbrsIt][nodeIt]->type !=
TargetOf(typeIt))
            {
               printf("\nWrong node type in target_Nodes; count: %lu\n",
lttc->count);
               return(-1);
            }
         }
      }
   }
   return 0;
}

int CompareLinks ( enum nodeType_t type ) {
   unsigned long int IC_Links_a = 0;
   unsigned long int nodeIt;
   /* Count n_links from lttc->cells[] */
   for( nodeIt = 0; nodeIt < lttc->n_Cells[TargetOf(type)]; nodeIt++ ) {
      unsigned long int nbrIt;
      for( nbrIt = 0; nbrIt < lttc->cells[TargetOf(type)][nodeIt]-
>n_Neighbors; nbrIt++ ) {
         if( lttc->cells[TargetOf(type)][nodeIt]->neighbors[nbrIt]->type
== type )
            IC_Links_a++;
      }
   }
   unsigned long int IC_Links_b = 0;
   unsigned long int n_nbrIt;
   /* Count n_links from n_Target_Nodes[] */
   for(n_nbrIt = 0; n_nbrIt <= lttc->n_Nbrs_Max; n_nbrIt++){
      IC_Links_b += lttc->n_Target_Nodes[type][n_nbrIt] * n_nbrIt;
   }
   /* Return Difference */
   return IC_Links_a - IC_Links_b;
}
```

```
/** VerifyState()*****************************************
* Check for some inconsistancies in the lattice state.
* Useful to check changes in code haven't cause problems
*****************************************************************/
int VerifyState( ){
   struct node_t *node = lttc->nodes[0];
   int n_Nbrs[N_CELL_TYPES+1] = {0,0,0,0};
   unsigned long int n_Cells[N_CELL_TYPES+1] = {0,0,0,0};
   unsigned long int *n_Target_Nodes[N_CELL_TYPES+1];
   int nbrIt;
   int typeIt;
   unsigned long int nodeIt;
   unsigned long int count=0;

   for( typeIt = 0; typeIt < N_CELL_TYPES; typeIt++ ) {
      n_Target_Nodes[typeIt] = calloc( (lttc->n_Nbrs_Max+1), sizeof(
**n_Target_Nodes));
   }

   for( nodeIt=0; nodeIt < lttc->n_Nodes; nodeIt++ ) {
      node = lttc->nodes[nodeIt];
      count++;
      /* Assume node->type,
n_Neighbors,
neighbors */

      /* Verify node->n_Nbrs[] is correct */
      for( nbrIt = 0; nbrIt < node->n_Neighbors; nbrIt++ )
         n_Nbrs[ node->neighbors[nbrIt]->type ]++;
      for( typeIt=0; typeIt < N_CELL_TYPES; typeIt++) {
         if(n_Nbrs[typeIt] != node->n_Nbrs[typeIt]) {
            fprintf(stderr, "n_Nbrs count is off\n");
            return(-1);
         }
      }

      /* Verify list_Idx is correct */
      if( node->list_Idx != NO_IDX &&
          lttc->cells[ node->type ][ node->list_Idx ] != node ) {
         fprintf(stderr, "Index to cells[] is incorrect\n");
         return(-1);
      }

      /* Verify targets_Idx[] are correct */
      for( typeIt=0; typeIt < N_CELL_TYPES; typeIt++) {
         if( node->targets_Idx[typeIt] == NO_IDX ){
         } else {
            /* Verify correct address at Idx */
            if( lttc->target_Nodes[ typeIt ]
                                  [ n_Nbrs[typeIt] ]
                                  [ node->targets_Idx[typeIt] ] != node)
{
               fprintf(stderr, "Index to target_Nodes[] is
incorrect\n");
```

```
                return(-1);
            }
            else {
                if( n_Nbrs[typeIt] == 0 ){
                    fprintf(stderr, "Node w/ nonzero target_Idx has no
neighbors that \
can replicate to it.\n");
                    return(-1);
                }

                n_Target_Nodes[ typeIt ][ n_Nbrs[typeIt] ]++;
            }
        }
    }

    n_Cells[ node->type ]++;
    n_Nbrs[0] = 0;
    n_Nbrs[1] = 0;
    n_Nbrs[2] = 0;
  }

  /* Verify lttc->n_Cells[] is correct */
  for( typeIt=0; typeIt < N_CELL_TYPES; typeIt++) {
    if( n_Cells[ typeIt ] != lttc->n_Cells[ typeIt ] ) {
        fprintf(stderr, "n_Cells[] is incorrect\n");
        return(-1);
    }
  }

  /* Verify cells[] is correct */
  for( typeIt=0; typeIt < N_CELL_TYPES; typeIt++) {
    for( nodeIt = 0; nodeIt < n_Cells[typeIt]; nodeIt++ ) {
      if( lttc->cells[typeIt][nodeIt]->type != typeIt ){
        fprintf(stderr, "Cells of wrong type in cells[].\n");
        return(-1);
      }
    }
    /* Make sure all elements after n_Cells[typeIt] are NULL */
    for( nodeIt = n_Cells[typeIt]; nodeIt < lttc->n_Nodes; nodeIt++ )
{
      if( lttc->cells[typeIt][nodeIt] != NULL ) {
        fprintf(stderr, "Not NULL cells exist in cells[%d] after
n_Cells.\n",
                typeIt);
        return(-1);
      }
    }
  }

  /* Verify lttc->n_Target_Nodes is correct */
  for( typeIt=0; typeIt < N_CELL_TYPES; typeIt++) {
    for( nbrIt=0; nbrIt < lttc->n_Nbrs_Max; nbrIt++ ){
      if( n_Target_Nodes[ typeIt ][ nbrIt ] !=
            lttc->n_Target_Nodes[ typeIt ][ nbrIt ] ) {
```

```
                fprintf(stderr, "n_Target_Nodes[] is incorrect\n");
                return(-1);
            }
        }
    }

    /* Verify lttc->target_Nodes[type][nbrs][nodes] are correct */
    for( typeIt=0; typeIt < N_CELL_TYPES; typeIt++) {
        for( nbrIt=0; nbrIt < lttc->n_Nbrs_Max; nbrIt++) {
            /* Make sure all elements after n_Target_Nodes[typeIt][nbrIt]
are NULL */
            for( nodeIt = n_Target_Nodes[typeIt][nbrIt]; nodeIt < lttc-
>n_Nodes; nodeIt++ ) {
                if( lttc->target_Nodes[typeIt][nbrIt][nodeIt] != NULL ) {
                fprintf(stderr, "Not NULL cells exist in target_Nodes[%d]
[%d] \
after n_Cells.\n", typeIt, nbrIt);
                return(-1);
                }
            }
        }
    }

    for( typeIt = 0; typeIt < N_CELL_TYPES; typeIt++ )
        free( (void*) n_Target_Nodes[typeIt] );

    for( typeIt = NORMAL; typeIt < N_CELL_TYPES; typeIt++) {
        int n_NbrsIt;
        for(n_NbrsIt = 0; n_NbrsIt <= lttc->n_Nbrs_Max; n_NbrsIt++ ){
            for(nodeIt = 0; nodeIt < lttc->n_Target_Nodes[typeIt]
[n_NbrsIt]; nodeIt++ ){
                if( lttc->target_Nodes[typeIt][n_NbrsIt][nodeIt]->type !=
TargetOf(typeIt))
                return(-1);
            }
        }
    }




    return 0;
}/* End of VerifyState() */
```

Makefile

```
Cells : Cells.o mersenne.o Queue.o cartesian.o Simulation.o
    gcc -g -o ../Cells Cells.o mersenne.o Queue.o Simulation.o
cartesian.o -lm

Cells.o : Cells.c
    gcc -Wall -c -g Cells.c -lm

Simulation.o : Simulation.c
    gcc -Wall -c -g Simulation.c -lm

Queue.o : Queue.c
    gcc -Wall -c -g Queue.c -lm

cartesian.o : cartesian.c
    gcc -Wall -c -g cartesian.c -lm

mersenne.o : mersenne.c
    gcc -Wall -c -g mersenne.c -lm

test : Cells.o mersenne.o Queue.o cartesian.o Cells-test.o
    gcc -g -o test Cells.o mersenne.o Queue.o cartesian.o Cells-test.o
-lm

Cells-test.o : Cells-test.c
    gcc -Wall -c -g Cells-test.c -lm

NetDetails : Cells.o mersenne.o Queue.o cartesian.o NetDetails.o
    gcc -g -o NetDetails Cells.o mersenne.o Queue.o cartesian.o
NetDetails.o -lm

NetDetails.o : NetDetails.c
    gcc -Wall -c -g NetDetails.c -lm


clean :
    rm -f *.o test ../Cells
```

References

Ben-Naim, E., Frachebourg, L., & Krapivsky, P. (1996). Coarsening and Persistence in the Voter Model. *Pysical Review E 53;* 3078-2087. doi:10.1103/PhysRevE.53.3078

Dingli, D., Offord, C. , Myers R., Peng, K-W., Carr, TW., Josic, K., Russell, SJ., & Bajzer, Z. (2009). Dynamics of multiple myeloma tumor therapy with a recombinant measles virus. *Cancer Gene Therapy, 16*; 873–882. doi: 10.1038/cgt.2009.40

Durrett, R., Levin, S. (1994). The importance of being discrete (and spatial). *Theoretical Population Biology, 46.3*; 363-94. doi:10.1006/tpbi.1994.1032

Kelly, E., Russell, S., (2007). History of Oncolytic Viruses: Genesis to Genetic engineering. *Molecular Therapy, 15*; 651–659 doi:10.1038/sj.mt.6300108

Lieberman, E., Hauert, C., & Nowak, M. A., (2005) Evolutionary dynamics on graphs. *Letters to Nature, 433;* 312-316. doi:10.1038/nature03211

Nishimura, T., & Matsumoto, M. (2002). Mersenne Twister with improved initialization. Retrieved January 1, 2014, from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/CODES/mt19937ar.c

Paiva, L. R., Binny, C., Ferreira, S. C., & Martins, M. L. (2009). A multiscale mathematical model for oncolytic virotherapy. *Cancer Research, 69.3* 1205-211. doi:10.1158/0008-5472.CAN-08-2173

Paiva, L. R., Martins, M. L., & Ferreira, S. C., (2011). Questing for an optimal, universal viral agent for oncolytic virotherapy. *Physical Review E, 84.4.* doi:10.1103/PhysRevE.84.041918

Reis, C. L., Pacheco J. M., Ennis, M. K., & Dingli, D., (2010). In silico evolutionary dynamics of tumor virotherapy. *Integrative Biology, 2:* 41-45. doi:10.1039/b917597k

Satō, K., Matsuda, H., & Sasaki, A. (1994). Pathogen invasion and host extinction in lattice structured populations. *Journal of Mathematical Biology*, 32(3), 251-268. doi:10.1007/BF00163881

Weisstein, E. "Pólya's Random Walk Constants." *MathWorld--A Wolfram Web Resource*. N.p., n.d. Web. 27 Feb. 2015

Wodarz D., Hofacre, A., Lau, J. W., Sun, Z., Fan, H., & Komarova, N. L. (2012). Complex spatial dynamics of oncolytic viruses in vitro: mathematical and experimental approaches. *PLoS Computational Biology, 8.6,* doi:10.1371/journal.pcbi.1002547

Appendix: Proof for Time Formula

This section shows that $\frac{-\ln(x)}{\lambda_T}$ where $x \in (0,1)$ and $\lambda_T = \sum (Adjusted\ Rates)$ is the correct formula for updating the time after a lattice event.

Obtaining a Linear Differential Equation and Initial Condition

Let $P\{X(t)=n\} = p_n(t)$ where $n \in \mathbb{N}_0$ and X(t) is a discrete random variable indicating the number of events occurring prior to time t. Assume the chance of an event happening in any short interval is independent of current and previous states when considering time intervals short enough that only one event will take place. Let λ be the expected number of events (cell division, infection or death) per unit time. Then the probability that there were no events before time $t+\Delta t$ is given by the formula $p_0(t+\Delta t) = p_0(t)(1-\lambda \Delta t)$ where $(1-\lambda \Delta t)$ is the probability that during Δt no event occurred. It is assumed that Δt is small enough that the probability of more than one event happening is 0. From this with $\Delta t \to 0$, we get the linear differential equation

$$\frac{d\,p_0(t)}{dt} = \lim_{\Delta t \to 0} \frac{p_0(t+\Delta t) - p_0(t)}{\Delta t} = \lim_{\Delta t \to 0} \frac{p_0(t)(1-\lambda \Delta t) - p_0(t)}{\Delta t} = \lim_{\Delta t \to 0} \frac{-p_0(t)\lambda \Delta t}{\Delta t} = -\lambda\, p_0(t)$$

with the initial condition that $p_0(0) = 1$, obtained by considering that we are interested in the first event since the last update, so time has been reset.

Solving Differential Equation

For easier notation let $y = p_0(t)$ then

$$\frac{dy}{dt} = -\lambda y \to \frac{dy}{dt} + \lambda y = 0 \to e^{\lambda t}\frac{dy}{dt} + e^{\lambda t}\lambda y = (ye^{\lambda t})dt = 0 \to \int (ye^{\lambda t})\,dt = 0 \to ye^{\lambda t} + C = 0$$

Solving the last part of the equation for y and substituting $p_0(t)$,

$ye^{\lambda t}+C=0 \rightarrow y=\frac{C}{e^{\lambda t}}$. The initial condition indicates that $p_0(0)=Ce^{-\lambda t}=1 \rightarrow C=1$.

Thus, $p_0(t)=e^{-\lambda t}$.

Function for Time

For this simulator we are only interested in the time of the first/next event's occurrence. The equation $p_0(t)$ gives the probability that the first event happens after time t. Therefore, if T is the time of the first event, then $P(T \le t)=1-p_0(t)=1-e^{-\lambda t}$. So T has an exponential distribution. By selecting a $F(t;\lambda)=x \in (0,1)$ and solving the exponential distribution function $F(t;\lambda)=1-e^{(-\lambda t)}$ for *t*, we get

$x=1-e^{-\lambda t} \rightarrow e^{-\lambda t}=1-x \rightarrow -\lambda t=\ln(1-x) \rightarrow t=-\ln\frac{(1-x)}{\lambda}$. Because *x* is a random variable in (0,1) we can simplify (1-x) to x. That gives the time to the next event as

$u=-\ln\frac{(x)}{\lambda}$ which is the formula used in the code when λ is the sum of the adjusted rates as defined in the section on Updating Time.