

TDK-dolgozat

Ágoston Paless
Gergely Kiss



**Pázmány Péter
Catholic University**

Faculty of Information Technology
and Bionics

**Budapest University of Technology
and Economics**

Faculty of Mechanical Engineering

Lattice-based stochastic simulation of cancer cell cultures

Essay written by:

Ágoston Paless (4th Semester)
PPKE ITK Molecular Bionics Engineer

Gergely Kiss (4th Semester)
BME GPK Mechatronics Engineering

Supervisors:

Dr. Péter Tamás Boldog
HUN-REN Wigner Research Centre for Physics,
colleague

Mihály András Vághy
Pázmány Péter Catholic University, PhD student

Academic year 2023/24

Contents

1	Introduction	5
2	Mathematical background	7
2.1	The Stochastic Simulation Algorithm	7
2.2	The marginal Reduced Rate Method	9
2.3	The extended marginal Reduced Rate Method	13
3	Structure of simulation and data processing system	16
3.1	Simulation program	16
3.2	Heatmap maker	16
3.3	3D visualizer	17
3.4	Analytic tool	18
4	Implementation of the simulation software	19
4.1	PreAllocVector	19
4.2	Coordinate	20
4.3	Cell	20
4.4	Slot	20
4.5	Lattice	20
4.6	Cells	20
4.7	State	21
4.8	Reactions	21
4.9	Saver	22
4.10	Simulation	24
5	Toy model	25
5.1	Cell types	25
5.2	Chemotherapy	25
5.3	Reactions	25
5.4	Initial state	26
5.5	Equilibrium cell count	26
5.6	Early stop mechanism	27

6	Results	28
6.1	Effect of volume exclusion on tumor structure	28
6.2	Parameter analysis of drug interventions	29
6.3	Time course of single trajectories	31
6.4	Runtime	32
7	Summary	35
	References	37
	Appendix	38

Acknowledgments

We would like to thank our supervisors: Mihály András Vághy and Dr. Péter Boldog for the tremendous support and guidance that they gave us throughout the past two semesters. We are really grateful for them.

We would like to thank the Faculty of Information Technology and Bionics of Pázmány Péter Catholic University for providing the necessary hardware to carry out our computations.

Foreword

This thesis was prepared for the Tudományos Diákkor Konferencia (TDK) held at the Faculty of Information Technology and Bionics of Pázmány Péter Catholic University on June 21, 2024. The thesis was finalized on January 14., 2025.

The results presented in this thesis are exclusively based on the research achievements of the co-authors up to the date of the Faculty TDK.

1 Introduction

In recent decades chemical reaction networks (CRNs), universal descriptors of nonnegative dynamics [1], [2], gained increasing attention due to their usefulness in modeling nonnegative phenomena and extensive theoretical background. CRNs, often referred to as kinetic systems, consist of a set of species, a set of complexes as (integer-)valued linear combinations of species and a set of elementary reactions that describe the transformation of a given complex into another. We assume that a reaction can only decrease the concentrations of the reactants; that is, there should be no negative cross-effects, and that a reaction cannot take place if the concentration of any reactant is zero. Mathematically, the transition function of each reaction is assumed to be nonnegative and to be zero if the concentration of any reactant is zero, ensuring the invariance of the nonnegative orthant. We note, that while the original motivation of such models was to describe chemical reactions, the possible applications can include various phenomena such as traffic models [3], cellular systems [4], gene regulatory networks [5] or epidemic models [6]. The main assumptions of the deterministic formulation of such models is that the number of particles exceed the classical macroscopic limit [7]–[9] and that they are kinetically homogeneous (well-mixed) [10].

When the number of particles do not exceed the macroscopic limit we must resort to the stochastic formulation of chemical kinetics. Then, in contrast with reaction *rate* in the deterministic formulation, we have reaction *probabilities* (per unit time) and the system takes the form of a discrete Markov chain or master equation [11]. We note, that in some sense the stochastic formulation is superior to the discrete formulation as they are equivalent when both are valid physically and the former is sometimes valid when the deterministic approach is not. However, if there are more than a few species and reactions, typically neither approaches can be solved analytically and we must use numerical methods [12]. While the numerical solution of the deterministic formulation is usually straightforward using well-known time stepping techniques, the numerical simulation of the underlying master equation corresponding to the stochastic formulation is usually infeasible. Motivated by this, the celebrated Gillespie algorithm, often called stochastic simulation algorithm (SSA), was developed based on Monte Carlo techniques in [11], [13], and later many variants were introduced [14]–[17].

While these algorithms address the problem of low particle number, in certain applications the kinetic homogeneity assumption might still be violated. In particular, the spatial structure of cell population models play an important role in the evolution of the system [18]. Lattice

based extensions of the Gillespie algorithm were introduced in [19], [20] and further developed in terms of performance in [21]. In this work we extend these models into a three-dimensional setting in an efficient implementation.

Our main motivation comes from the increasing popularity of personalized medicine [22], [23] and digital twins [24]–[26] along with the rise of high performance computing (HPC). The stochastic formulation, extensive Monte Carlo simulation and scenario analysis of tumorous growth and especially their reaction to medicine might not only help the design of personalized treatment plans but also help drug discovery.

2 Mathematical background

2.1 The Stochastic Simulation Algorithm

In this section, we introduce the Gillespie algorithm [8], [11], [17] which will later serve as the basis for our mathematical model and simulation algorithm. The stochastic simulation algorithm (SSA) was formulated by Daniel T. Gillespie [11] and independently by Hungarian researchers János Tóth, Péter Érdi and was implemented by Vera Hárs [27].

2.1.1 Variables and quantities

The state variables in the SSA were originally introduced within the context of chemical reactions. We consider chemical species $S_1, \dots, S_N$ of amount $(X_1, \dots, X_N) \in \mathbb{N}_0^N$ each and reactions $R_1, \dots, R_M$ between them. For convenience we note that $\mu = 1, 2, \dots, M$ stands for the reaction index; thus M is the number of reactions, $i = 1, 2, \dots, N$ is the index of species, and N is the total number of species. Reaction R_μ defines a change $(\Delta X_1, \dots, \Delta X_N)_\mu$ in the population of chemical species.

The so called propensity a_μ of reaction R_μ is given by the formula $a_\mu = c_\mu \cdot h_\mu$, where $c_\mu \in \mathbb{R}$ is the rate constant and h_μ is the number of distinct molecular reactant combinations available in the given state of the system. For example, the number of distinct molecular reactant combinations for reactions of the form $S_1 + S_2 \rightarrow S_3$ and $2S_1 + S_2 \rightarrow S_3$ are $\binom{X_1}{1} \cdot \binom{X_2}{1} = X_1 \cdot X_2$ and $\binom{X_1}{2} \cdot \binom{X_2}{1} = \frac{1}{2} X_1 \cdot (X_1 - 1) \cdot X_2$, respectively.

The sum of the propensities is defined as $a = \sum_{\mu=1}^M a_\mu$ and it can be shown [11] that the waiting time τ between consecutive reactions is of exponential distribution with parameter a and the following reaction occurs with probability $P(R_\mu)$:

$$\tau \sim \text{Exp}(a), \quad P(R_\mu) = \frac{a_\mu}{a}. \quad (1)$$

2.1.2 Steps of the SSA

The algorithm starts by initializing variables and prescribing a halting condition. The halting condition can be simply prescribing a maximal simulated time or a more complicated condition on the state. For example, we might want to stop the simulation if a certain ratio of chem-

icals is met or if a certain chemical is eliminated entirely from the system, so that we spare computational resources.

After initialization we enter a cycle that repeats until the halting condition is met. Each cycle starts with the calculation of propensities given the current state. After having calculated the propensities we choose a reaction R_μ and waiting time τ according to Eq. (2.1.1). Then, the state variables are updated according to the selected reaction and add τ to the state time. Finally, the halting condition is checked, and the process repeats if necessary. Figure 1. shows the flowchart of the algorithm.

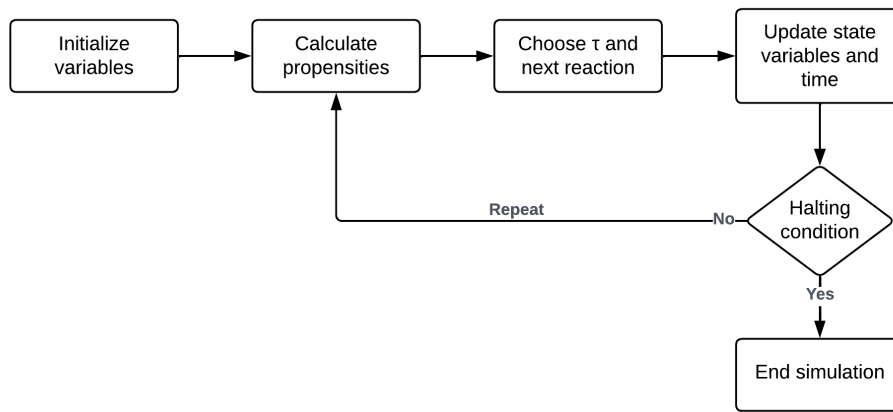


Figure 1: Flowchart of the Stochastic Simulation Algorithm

2.1.3 Advantages and disadvantages

The SSA serves as a great alternative for ordinary differential equation based deterministic models. Unlike an ODE, the SSA yields reliable results even when the number of agents is under the macroscopic limit. The inherent stochastic nature of the method and the possibility of simulating small populations allows for accounting of stochastic effects, which can lead to vastly different outcomes when the number of agents is low. This way we may estimate the empirical probabilities of certain outcomes of the process. For example, in our simulations such a scenario occurs when the tumor is small, and only a few random events determine the faith of the cluster, weather the tumor grows or is eliminated.

On the other hand, exact stochastic simulations are known to be computationally expensive. Because of the probabilistic nature of such algorithms, it is often necessary to simulate numerous realizations or trajectories, in order to gain reliable information about the probability of a given outcome or obtain the expected values of the state variables over time.

2.2 The marginal Reduced Rate Method

One of the assumptions of the Gillespie algorithm is that the system is *well-stirred*; that is, spatial organization of the chemicals is neglected. This makes it particularly difficult to use for modeling cell cultures as populations of cells usually have a well defined structure in living organisms to maintain the function of tissues and organs. To appropriately model the behavior of cell cultures, where volume exclusion plays a key role, spatial organization is introduced into the system in the form of a spatial grid, also referred to as a lattice. We adopt the description of such a system, the marginal Reduced Rate Method (mRRM), from Boldog [21].

In this framework cells and chemical species exist on a rectangular lattice of size $(n_1 \times n_2)$. Cells are microscopic entities compared to chemical particles. In realistic models it is assumed that they are subject to volume exclusion, meaning that two cells cannot occupy the same space. Consequently, cells act as physical obstacles to one another. On the other hand, chemicals are orders of magnitude smaller, allowing us to assume that they diffuse freely in the intercellular space with diffusion coefficients significantly greater than those of cells. As a result, chemical species are considered well-mixed. A distinct position on the lattice is referred to as a *slot* or a *coordinate*. Due to volume exclusion, each slot can contain only one cell at a time, allowing a cell to be uniquely identified by its position on the lattice.

The typical behaviors of living cells include reactions like proliferation, movement and apoptosis. For the sake of simplicity, we only model the mitosis and the interphase of the cell cycle and intermediate phases are omitted.

The simplest case of a lattice-based cell population model is a two-dimensional grid with von Neumann neighborhood, meaning each slot has four neighbors: top, bottom, left, and right. We say that a cell has j free neighbors if there are j adjacent slots, that do not contain a cell. We may refer to such cells as j -class cells. In a 2 dimensional von Neumann neighboring situation that also means that the cell has $4 - j$ occupied neighbors. For convenience, the number of cells in class j is denoted by N_j , and the total number of cells in the population is given by $\sum_{j=0}^4 N_j$.

2.2.1 Reaction types

Reactions are distinguished by their dependency on free or occupied neighboring slots. A reaction is *contact promoted*, if it is more likely to happen to a cell that has many neighboring cells. Similarly, a reaction is said to be *contact inhibited*, if its occurrence is less likely as the number of neighbors increases. A reaction can also be *contact independent*, when its occurrence is in-

dependent of the neighbor count of the given cell or if it is completely independent of cells (i.e. chemical reaction).

2.2.2 Propensity functions

For the description of the mRRM we use a distinct notation: we denote quantities with a (hat symbol, e.g. $\hat{a}_\mu$), so that we can use regular notation for the quantities used to describe the extended mRRM in section 2.3.

We redefine h_μ to be

$$h_\mu = \begin{cases} 1 & \text{if chemical species are not involved in } \mu, \\ \text{Nr. of distinct reactant combinations} & \text{if chemical species are involved in } \mu. \end{cases} \quad (2)$$

For ease of notation we introduce:

$$\hat{r}_\mu = \begin{cases} c_\mu \cdot h_\mu & \text{for reactions involving cells (or possibly chemical species),} \\ \frac{c_\mu \cdot h_\mu}{N} & \text{for reactions only involving chemical species.} \end{cases} \quad (3)$$

For a reaction R_μ the propensity is defined as:

$$\hat{a}_\mu = \begin{cases} \frac{\hat{r}_\mu}{4} \sum_{j=0}^4 j N_j(t) & \text{if } R_\mu \text{ is contact inhibited,} \\ \hat{r}_\mu N(t) & \text{if } R_\mu \text{ is contact independent,} \\ \frac{\hat{r}_\mu}{4} \sum_{j=0}^4 (4-j) N_j(t) & \text{if } R_\mu \text{ is contact promoted.} \end{cases} \quad (4)$$

By definition, purely chemical reactions only depend on the number of chemical species and are independent of the number of cells in any class. Thus, with a simple algebraic transformation for the propensity $\hat{a}_\mu$ of chemical reaction R_μ the formula takes the same form as in the case of contact-independent reactions:

$$\hat{a}_\mu = c_\mu h_\mu = \frac{c_\mu h_\mu}{N} N = \hat{r}_\mu N.$$

It is easy to see that if all cells have exactly 4 free neighbors, the propensity of a contact

promoted reaction is 0 and the propensity of contact inhibited reactions is maximal. Similarly, the propensity of contact promoted reactions is maximal and contact inhibited reactions are impossible if the lattice is full, so all cells have 0 free neighbors.

It has been shown [21] that, to obtain the probability of reaction R_μ within a time interval τ , we simply divide the respective propensity by the sum of all propensities, $\hat{a} = \sum_\mu \hat{a}_\mu$:

$$P(R_\mu) := P(R_\mu|\tau) = \frac{\hat{a}_\mu}{\hat{a}}. \quad (5)$$

In the mRRM we not only choose a reaction, but we also choose a cell, on which the reaction is carried out. Let A_j denote the event that the reaction acts on a j -class cell. The probability that the next reaction will act on a j -class cell given that the reaction is R_μ is given by:

$$P(A_j|R_\mu) = \begin{cases} \frac{jN_j}{\sum_{j=0}^4 jN_j} & \text{if } R_\mu \text{ is contact inhibited,} \\ \frac{N_j}{N} & \text{if } R_\mu \text{ is contact independent,} \\ \frac{(4-j)N_j}{\sum_{j=0}^4 (4-j)N_j} & \text{if } R_\mu \text{ is contact promoted.} \end{cases} \quad (6)$$

According to these equations we can perform the steps of the algorithm.

2.2.3 Steps of the mRRM

1. Initialization:

- (a) Set $t = 0$, prescribe halting condition.
- (b) Initialize lattice with size $(n_1 \times n_2)$, place $N(0) \leq (n_1 \times n_2)$ cells on the lattice and store their positions.
- (c) Store initial amounts of chemical species.

2. Calculate propensities:

Calculate $\hat{a}_\mu$, $\mu = 1, 2, \dots, M$ according to equation (4). If the sum of the propensities is zero, halt the algorithm.

3. Generate elapsed time:

Calculate τ by sampling an exponential distribution with parameter $\hat{a}$.

4. Select reaction:
Select R_μ according to equation (5).
5. Select cell:
 - (a) Choose the cell class j according to equation (6).
 - (b) Choose the target cell with uniform distribution from the chosen cell class.
 - (c) If the reaction involves a neighboring slot (free or occupied), choose target neighbor with discrete uniform distribution.
6. Execute:
 - (a) Step the time forward: $t = t + \tau$.
 - (b) Carry out reaction R_μ on the chosen j class cell.
 - (c) Update state variables.
7. Check halting condition:
If halting condition is true, end algorithm; if not, repeat from step 2.

The flowchart of the algorithm can be seen in Figure 2.

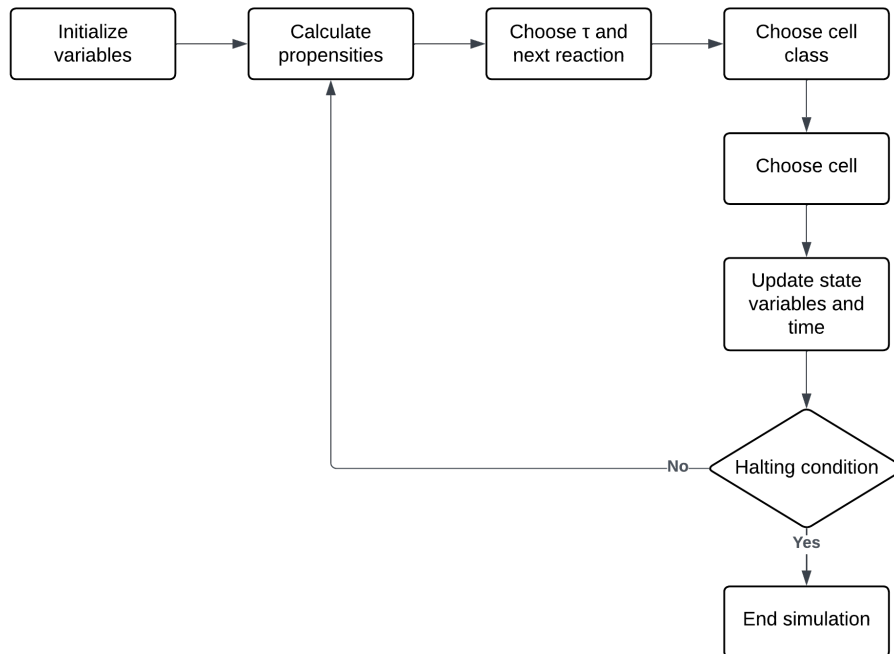


Figure 2: Flowchart of the mRRM

2.3 The extended marginal Reduced Rate Method

The mRRM is a great way to simulate homogeneous cell cultures, so if only a single type of cell is to be simulated. We extended the mRRM to include different cell types, and thus allow simulation of heterogeneous cell cultures, for example cancerous tumors.

To further increase the capabilities of the simulation algorithm we changed dimensionality to 3. Note, that in 3D, cells have 6 neighboring slots, so if a cell has j free neighbors, it has $6 - j$ occupied neighbors.

With the introduction of *cell types*, a cell is characterized by its coordinate on the lattice, its free neighbors and its type. Let $\delta = 1, 2, \dots, D$ denote cell type, where D is the number of distinct cell types. Then, $R_{\mu,\delta}$ is the event that a reaction with index μ is carried out on a cell of type δ and $a_{\mu,\delta}$ denotes the corresponding propensity.

We also introduced constants $c_{\mu,\delta}$ that are specific to a reaction and a cell type; these are referred to as *cell type constants*. Since not all cell types contribute equally to the probability of a given reaction we need to modify the propensity functions accordingly.

For ease of notation we introduce a new quantity: the *weighted cell count*. It is the weighted sum of cell counts of different types, where each term is weighted by the cell type constant $c_{\mu,\delta}$ for the given reaction and cell type. In this method there is no scaling to r_μ for purely chemical reactions, as there was in the mRRM, but in order to retain the propensity for such reactions to be $a_\mu = c_\mu h_\mu$ we define W_μ in the following way:

$$W_\mu(t) = \begin{cases} \sum_{\delta=1}^D c_{\mu,\delta} N_\delta(t) & \text{if cells are involved} \\ 1 & \text{if only chemical species are involved} \end{cases} \quad (7)$$

With these quantities, we can define the propensity functions.

$$a_\mu = \begin{cases} \frac{r_\mu}{6} \sum_{\delta=1}^D c_{\mu,\delta} \sum_{j=0}^6 j N_{\delta,j}(t) & \text{if } R_\mu \text{ is contact inhibited,} \\ r_\mu W_\mu(t) & \text{if } R_\mu \text{ is contact independent,} \\ \frac{r_\mu}{6} \sum_{\delta=1}^D c_{\mu,\delta} \sum_{j=0}^6 (6 - j) N_{\delta,j}(t) & \text{if } R_\mu \text{ is contact promoted.} \end{cases} \quad (8)$$

In the same way as in the mRRM, the probability of a reaction occurring next given the intervent

time τ is as follows:

$$P(R_\mu) := P(R_\mu|\tau) = \frac{a_\mu}{a} \quad (9)$$

We implement the same strategy to obtain the probability $P(R_{\mu,\delta}|R_\mu)$, the probability of choosing a cell of a given type when the reaction index μ is already determined. Let $a_{\mu,\delta} = \sum_{j=0}^6 a_{\mu,\delta,j}$ denote the propensity of a reaction μ happening on a cell of type δ . Then, we can calculate the above probability as:

$$P(R_{\mu,\delta}|R_\mu) = \frac{a_{\mu,\delta}}{a_\mu} \quad (10)$$

Similarly, the probability that a reaction is happening to a j -class cell, when the reaction index μ and the cell type δ are given, can be calculated as:

$$P(R_{\mu,\delta,j}|R_{\mu,\delta}) = \frac{a_{\mu,\delta,j}}{a_{\mu,\delta}} \quad (11)$$

In order to actually calculate these probabilities, we need $a_{\mu,\delta,j}$, which is the propensity for the event that a reaction μ happens to a cell of type δ and of class j .

$$a_{\mu,\delta,j} = \begin{cases} \frac{r_\mu}{6} c_{\mu,\delta,j} N_{\delta,j}(t) & \text{if } R_\mu \text{ is contact inhibited,} \\ r_\mu c_{\mu,\delta} N_{\delta,j}(t) & \text{if } R_\mu \text{ is contact independent,} \\ \frac{r_\mu}{6} c_{\mu,\delta} (6-j) N_{\delta,j}(t) & \text{if } R_\mu \text{ is contact promoted.,} \end{cases} \quad (12)$$

2.3.1 Steps of the extended mRRM

1. Initialization:

- (a) Set $t = 0$, prescribe halting condition.
- (b) Initialize lattice with size $(n_1 \times n_2 \times n_3)$. Place $N_\delta(0) \leq (n_1 \times n_2 \times n_3)$, $\delta = 1, 2, \dots, D$, $\sum_{\delta=1}^D N_\delta(0) \leq (n_1 \times n_2 \times n_3)$ cells of different type on the lattice, and store their positions.
- (c) Store initial amounts of chemical species.

2. Calculate propensities:

Calculate a_μ , $\mu = 1, 2, \dots, M$ according to (8). If the sum of the propensities is zero, halt the algorithm.

3. Generate interval time:

Calculate τ , by sampling an exponential distribution with parameter a .

4. Select reaction:

Select R_μ according to (9).

5. Select cell:

(a) Choose the cell type δ according to (10).

(b) Choose the cell class j according to (11).

(c) Choose the target cell with discrete uniform distribution from the chosen cell class and cell type.

(d) If the reaction involves a neighboring slot (free or occupied), choose target neighbor with discrete uniform distribution.

6. Execute:

(a) Update time: $t = t + \tau$.

(b) Execute reaction R_μ on the chosen cell.

(c) Update state variables.

7. Check halting condition:

If the halting condition is true, end the algorithm.

The flowchart of the extended mRRM can be seen in Figure 15 in the Appendix.

3 Structure of simulation and data processing system

The system is composed of four components: a simulation module, a heatmap generator, an analytic tool for plotting statistical data (such as reaction frequencies) and changes in cell counts over time, and a 3D visualization tool.

The simulation software is written in C++, the 3D visualization program is implemented in Unity while the heatmap generator and the analytic tool are developed in Python.

3.1 Simulation program

The simulation program consists of five main components.

The input data, which includes parameters such as the halting condition, the lattice size, and the initial state.

The biological model, which includes the considered cell types, the reactions occurring between them, and the chemical reactants present.

The runtime data structures, which are used to record the current state of the simulation. To ensure program efficiency, these structures are specifically designed to align with the simulation algorithm used. Such structures include the lattice, the class that stores cells sorted by their type and class, and the variable that stores the chemical reactants.

The simulation component, which queries the current state, executes one step of the extended marginal Reduced Rate Method, and then performs the necessary updates to the data structures.

Finally, the data-saving component, which offers multiple independent saving options to enhance efficiency.

The schematic diagram of the the simulation software can be seen in Figure 3.

3.2 Heatmap maker

This program consists of two main parts.

A `Heatmap` class, which reads the file system in the specified folder and saves the data required for the heatmap, such as the names and values of modified constants, the values to be plotted, and verifies whether the simulation successfully completed to avoid erroneous data.

The second part is a class for plotting, which formats the given `Heatmap` and plots the requested data.

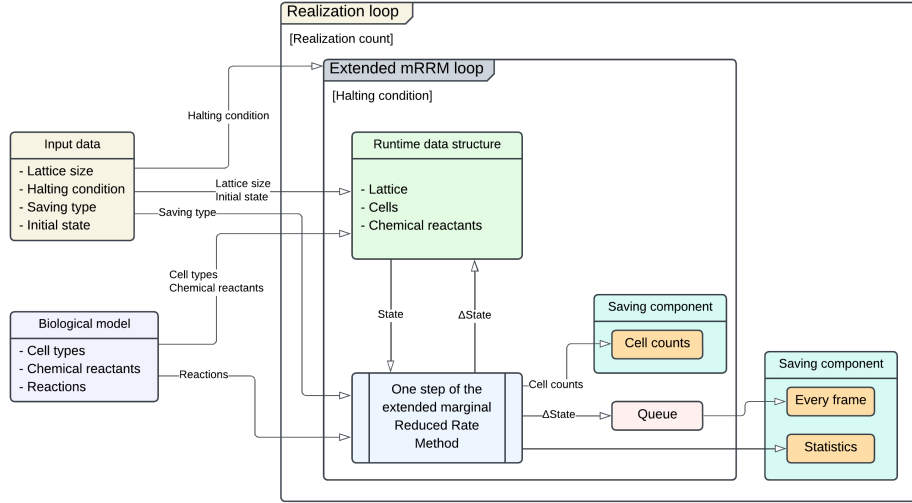


Figure 3: Schematic diagram of the simulation software, illustrating its five main components: input data, biological model, runtime data structures, simulation component and data-saving component.

3.3 3D visualizer

We developed a visualizer program in Unity to allow intuitive understanding and thus more effective model development. The Unity game engine allows users to code in C# or javascript. We made two main scripts in C#, one handling camera movement, and the other tackling reading, displaying and saving the visuals.

It allows the display of simulation data both continuously and in discrete steps. Reading has been optimized, so that even large scale simulations can be visualized on normal fps levels. Additionally, it allows users to save screenshots and timelapses as image sequences.

Two images created with this tool can be seen in Figures 4. and 5.

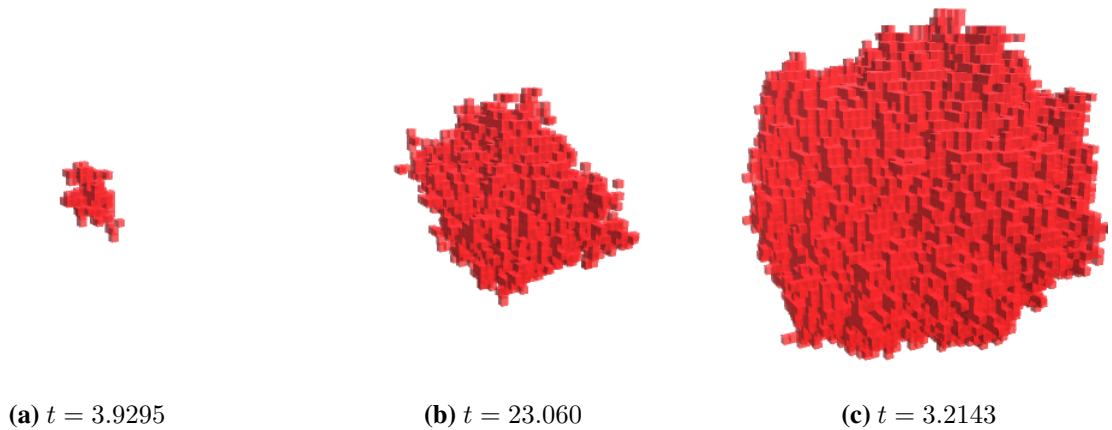


Figure 4: 3D visualization of a typical tumor progression

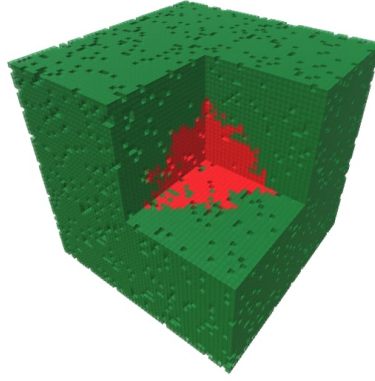


Figure 5: 3D visualization of a simulation of a tumor in a healthy tissue filled cubic simulation space with the first octant hidden.

3.4 Analytic tool

The analytic tool provides a comprehensive framework for analyzing the temporal dynamics of events and visualizing fundamental statistical metrics.

Using this tool, we can plot the death rates of individual cells at the final time point relative to the initial state through bar diagrams. Additionally, the proportions of various reactions can be represented using pie charts, providing insights into the distribution of events.

The tool further enables the visualization of cell counts or chemical reactant concentrations as a function of elapsed simulation time or iteration count. It also supports plotting the counts of different cell types within a single simulation on the same graph to facilitate comparative analysis. Moreover, specific time intervals can be highlighted to allow for detailed examination of particular phases of the simulation.

4 Implementation of the simulation software

We developed the simulation program in C++ using an object-oriented approach. In this section, we detail the classes used and their relationships.

The class diagram of the simulation software is shown in Figure 14 in the Appendix.

4.1 PreAllocVector

The `PreAllocVector` is a template class designed for storing any type of data, providing constant time access through appropriately overloaded functions.

The primary member variable is an `std::vector`, the size of which must be specified at initialization. The size should be chosen to accommodate the maximum possible amount of data, as one of our goals is to avoid copying data to a larger vector at runtime, thereby enhancing the program's efficiency. The second variable, `elementCount`, indicates the number of elements in the vector that contain valid data. Beyond this index, the vector holds uninitialized memory, which remains hidden from the user.

When adding a new item, we simply increment `elementCount` by one and insert the data at the new position. During deletion, the `remove` function not only removes the appropriate element but also moves the last valid element to the vacated position, ensuring that the data remains continuous and then decrements `elementCount`.

To enable constant-time access to the stored data, two conditions must be met. One requirement is that every element must store its index within the corresponding `PreAllocVector`. Another is the implementation of overloaded `push_back` and `remove` functions for the specific data type. In this case, the `remove` function deletes an element from the vector based on its index and updates the index of the relocated element. Similarly, the `push_back` function assigns the appropriate index value to the element upon insertion.

In our case, this approach is particularly important for storing cells, as the simulation requires the creation and deletion of cells in almost every step.

The `PreAllocVector` template class can be seen in Code snippet 1., while the implementation of the overloaded functions are presented in Code snippet 2.

4.2 Coordinate

The coordinate is a simple class designed to determine the position of individual cells on the lattice. To enhance efficiency and improve readability, we overloaded operators `=`, `==`, and `<<`.

4.3 Cell

Cells are represented as instances of the `Cell` class. Each cell stores its type, defined by an `enum`, along with its coordinates on the lattice. Additionally, each cell retains its index within the corresponding `PreAllocVector`.

The implementation of the `Cell` class is presented in Code snippet 3.

4.4 Slot

Slots represent individual cell-sized locations on the lattice. Each slot contains an `std::shared_ptr<Cell>` pointing to the cell located there, or a `nullptr` if the slot is empty (i.e., does not contain a cell). Additionally, each slot stores the number of its free neighboring slots.

4.5 Lattice

The `Lattice` class defines a three-dimensional grid where space is divided into discrete, cell-sized volumes using nested `std::vector`-s. The elements of the innermost vectors are instances of the `Slot` class. Since slots, and thus cells are queried through their coordinates, the data access time is constant in this structure.

The `Lattice` class in addition to its data storage component includes methods and overloaded operators needed for modifying data and querying the state, such as `DeleteCell()`, `AddCell()` and `GetFreeNeighbours()`. These methods ensure that the principle of data encapsulation is maintained for the member variables.

The implementation of the `Lattice` class is presented in Code snippet 4.

4.6 Cells

Propensity functions depend on how many of each type of cell there are at a given moment and how many free neighbors they have, so it is worth choosing a data structure where we can easily

count cells according to these aspects. It is also important to be able to delete and add new cells efficiently as these actions are part of almost every reaction.

Therefore, in the `Cells` class, we first sort them by type using an `std::unordered_map`, then by the number of free neighbors using an `std::vector`, and at the innermost level, we utilize `PreAllocVectors` to enable constant-time access during deletion.

In addition, the `Cells` class stores the number of cells of each type at a given time point and includes functions necessary for state updates, such as `DeleteCell()` and `AddCell()`.

When instantiating the `Cells` class, it must be associated with a `Lattice`, which will be stored as an `std::shared_ptr<Lattice>`. This ensures that the data structures are appropriately sized.

The implementation of the `Cells` class is presented in Code snippet 5.

4.7 State

The `State` class primarily serves to organize and manage data storage. It contains a `Cells` variable, the elapsed simulation time, the chemical reactants considered in the simulation and a pointer to the `Lattice`. Since our simulation assumes that the chemical reactants are well-mixed within the system, it is sufficient to store their concentrations. For this purpose, we use an `std::unordered_map<ChemicalSpecie, size_t>`.

Additionally, it includes functions required for querying and modifying the state, such as `DeleteCell()` and `AddCell()`. These functions handle the removal or addition of a cell to both the `Lattice` and the `Cells`, while also updating the neighbors of the affected cell.

The implementation of the `State` class is presented in Code snippet 6.

4.8 Reactions

The simulation software provides only abstract base classes for reactions, without any directly instantiable ones. To set up the biological model, the necessary reactions must be implemented by deriving them from these and defining their specific functionalities. Depending on the type of reaction, we can choose from three base classes: `WellMixedReaction`, `CellAndWellMixedReaction`, and `SingleCellReaction`. Their hierarchy, along with their data members and member functions, is shown in Figure 6.

Each reaction includes a `GetPropensity` and an `Effect` member function.

When defining the `GetPropensity` function, we need to specify the function that determines the propensity value of the given reaction, as these functions will be called when selecting the next event.

The `Effect` functions basically do three things: delete a cell, create a cell or reduce the concentration of the drug. Reactions will be combinations of these operations. For example, programmed cell death or an event where a chemical reactant kills a cell will result in the deletion of a cell, while cell division will involve the creation of a cell. Cell movement will be represented by deleting the original cell and creating a new one in the selected neighboring `Slot`. The execution of such reactions, which consist of multiple operations, naturally takes more time, meaning their frequency significantly impacts runtime performance.

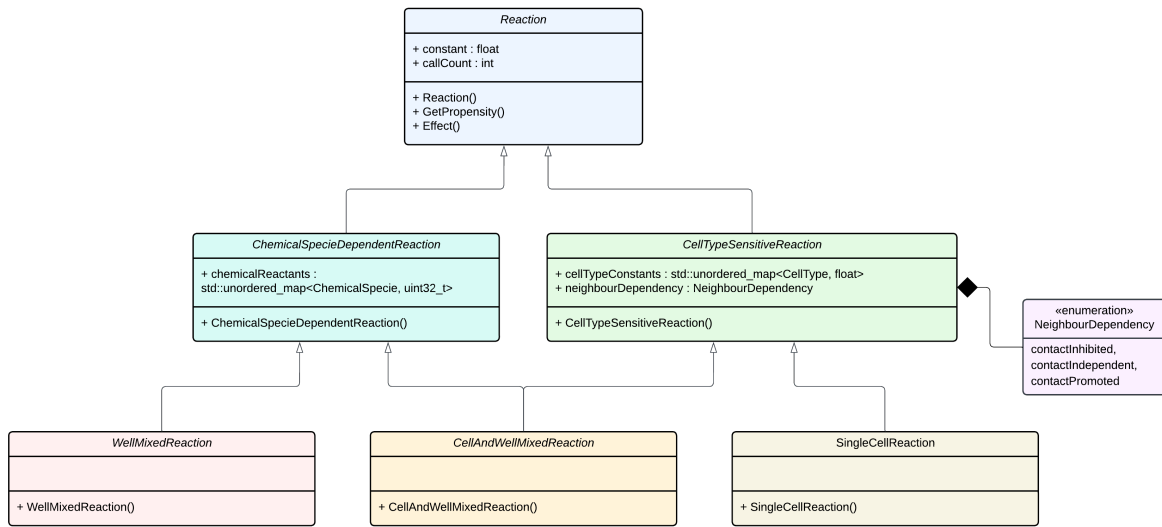


Figure 6: Hierarchy of the reaction inheritance

4.9 Saver

Since I/O operations, such as writing to a file, are time-consuming processes, the software provides three independent saving modes. These modes can be enabled or disabled when specifying the simulation parameters, ensuring that only the data relevant to our requirements is saved.

In addition to these files, two text files are generated to facilitate automated data import and to save the values of modified parameters when creating heatmaps.

The file structure can be seen in Figure 7.

The `SaveRealizationData` class contains the files required for saving the simulation

results, their file paths, the specified saving mode, and the functions necessary for performing the save operation.

The saver class can be seen in Code snippet 7.

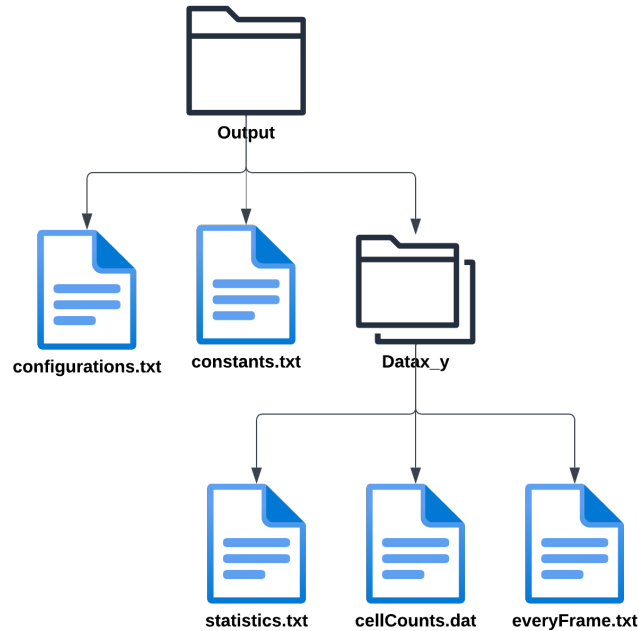


Figure 7: The figure illustrates the organization of files used for saving simulation results, facilitating automated data loading and processing for other tools. Within the Datax_y folder, the three main output files are displayed, while the files containing the simulation settings are located at the same directory level.

4.9.1 Saving cell counts

The purpose of this format is to be able to plot the number of cells of each cell type as a function of time. In each iteration, we save the time elapsed since the previous step, along with the counts of each cell type. Since the length of the output data remains constant in each step, we save this in a binary file, thereby increasing the saving speed. We output this data after every step during the simulation.

4.9.2 Saving statistics

The purpose of this method is to save data that is easier and faster to compute at runtime and store in a partially processed form. This includes, among others, the number of times a reaction has occurred or the cell death rate, which can be used to create heatmaps and other diagrams. These data are written out at the end of each realization.

4.9.3 Saving every frame

By saving every frame, we store not only the number of each cell type but also the position of every cell. We achieve this by saving the entire lattice before the first iteration. Then, in each iteration, we instantiate a `DeltaLattice` class to record at which coordinate was a cell deleted or instantiated, and if there was any changes in the concentration of a drug. Since writing to a file is a time-consuming process, these instances are stored in a `std::vector` and then written out all at once at the end of the simulation. This type of saving is used to visualize the lattice.

4.10 Simulation

Finally, the `Simulation` class encompasses the `State`, the specified reactions, the class responsible for data saving, as well as the functions and variables required for executing the extended marginal Reduced Rate Method.

Upon initializing an instance of the `Simulation` class, it copies the `State` to be able to start all realizations within a simulation from the same initial state. The `Simulation` class features a public `Simulate()` function, which is called to execute the extended marginal Reduced Rate Method.

The `Simulation` class is illustrated in Code snippet 8.

5 Toy model

To demonstrate the capabilities of our simulation software, test its performance, and verify its ability to simulate phenomena observed in cell biology, we set up a biological model.

In the model, we examine the growth of cancerous cells in the presence of healthy cells. The simulation results enable us to study two critical phenomena that cannot be captured by other differential equation-based models: volume exclusion and tumor morphology. Additionally, we will investigate the effect of a hypothetical chemotherapeutic drug on tumor growth.

5.1 Cell types

We will consider two types of cells: healthy and cancerous.

We model cancerous cells as homogeneous. All cancer cells can divide at the same rate, twice as fast as healthy cells. They die at the same rate as healthy cells

All non-cancerous cells are modeled as identical. They differ only in their position and possibly neighbor count. They can move, proliferate if space is adequate and undergo apoptosis.

5.2 Chemotherapy

One can also consider well-mixed agents, for example drug molecules, that diffuse freely in the intercellular space and are therefore said to be present in all cells (of the same type) equally. We examine the effect of a hypothetical drug with different toxicity and selectivity parameters.

5.3 Reactions

5.3.1 Apoptosis

We model cells to undergo programmed cell death with a certain probability at any given time. Healthy and tumorous cells undergo apoptosis at the same rate. We say that cells need to compete for resources and they undergo apoptosis with higher frequency when they have many neighbors and have to share the same amount of resources with more cells. Apoptosis is thus modeled contact promoted.

5.3.2 Division

A cell can divide to any of its free neighbors with equal likelihood. Division is therefore a contact inhibited reaction. Tumor cells divide at a higher rate than healthy cells: cell type constant for tumor division is twice that of healthy division.

5.3.3 Movement

Cells are allowed to move to any free position around them, considering von Neumann neighboring, with equal likelihood. Therefore `Movement` is contact inhibited. Tumor cells have a higher tendency to move, exactly 5 times that of healthy cells.

5.3.4 Drug decay

The hypothetical drug is excreted from the tumor site proportionally to the number of molecules at the site. The decay of the drug is therefore exponential.

5.3.5 Cytotoxic drug effect

When drug molecules are present, they can kill cells with a certain probability. *Drug toxicity* is represented by the respective constant for the reaction, while *Drug selectivity* is represented by the ratio of the cell type constants for the two types of cells. Note, that in this model, one molecule is enough to cause cell death.

5.4 Initial state

For tissue density an arbitrary number (90%) is selected. We assume coordinates outside the bounds of the lattice to be occupied by healthy cells at all times.

5.5 Equilibrium cell count

Since our model parameters are not fitted to experimental data, we estimate healthy tissue homeostasis parameters through simulation, assuming an average lattice fill of 90%.

Cell count in the system equilibrates based on the ratio of the division and apoptosis reaction constants, referred to as the *relative division rate*. Since we only consider these two reactions, setting the constant for apoptosis to one, allows the division constant to equal the relative division rate.

Simulations were performed on a $50 \times 50 \times 50$ lattice ($1.25 \cdot 10^5$ slots) with a stopping condition of $< 1.25 \cdot 10^7$ iterations and only healthy cells were populated. For each parameter 30 simulations were averaged.

The results can be seen in Figure 8.

It indicates that a relative division rate of approximately 9.3 maintains a 90% cell density in healthy tissue.

We repeat the simulation, with the distinction that, the initial state contains a tumor of 33 cells. The equilibrium shifts leftwards on the x-axis, since tumor cells divide twice as fast as healthy cells, and thus a lower relative division rate yields equilibrium.

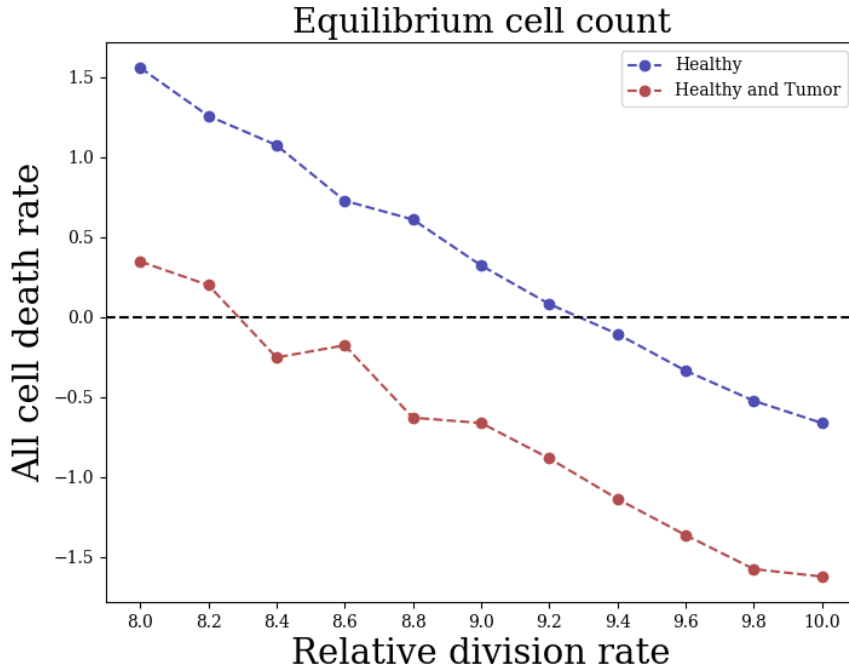


Figure 8: Death rate as a function of the relative division rate in healthy tissue in blue and in tumorous tissue in red. Initial healthy cell population is set to 90% of lattice size and initial tumor size is 33 cells. Each data point represents the average of 30 realizations.

5.6 Early stop mechanism

During the simulations, we check at each step whether the cancerous cells have reached the boundaries of the simulated area. Since they cannot grow beyond this boundary, we conclude that continuing the simulation beyond this point would lead to significant distortion. Therefore, the simulation is terminated. The data generated in this manner will also be considered when averaging multiple realizations. We refer to this phenomenon as *early stop*.

6 Results

6.1 Effect of volume exclusion on tumor structure

It is well known that tumors, are not comprised exclusively of cancer cells, rather the micro-environment plays a key role in allowing tumor growth and survival. These interactions are very much influenced by volume exclusion, and tumor morphology.

6.1.1 Measuring morphology

To assess how much a tumor is 'spread' we calculate the number of distinct tumor parts. Any two cancer cells are considered to be in the same part, if one can 'walk' from one to the other only stepping on tumor cells, considering von Neumann neighboring.

The number of distinct parts is calculated anytime the tumor reaches a certain size (1% of the lattice size) in terms of cell count. Distinct parts are only calculated the first time a tumor reaches a given size, so if for whatever reason the tumor shrinks to a size of measurement, we do not take the measurement again.

6.1.2 Parameters

Simulations are run on a $50 \times 50 \times 50$ lattice, with relative division rate set to 9.3, until the tumor reaches the boarder of the lattice or the maximal iteration number ($1.25 \cdot 10^7$) is met. We took measurements both when only apoptosis and division are considered and when movement is included as well.

6.1.3 Results

We found that tumor morphology is clearly different in terms of distinct tumor parts when considering healthy cells as well. This implies that considering volume-exclusion effects by healthy cells is of great importance when examining tumor morphology and structure, or interactions dependent on cell-cell contact.

Moreover, including cell movement further increases the number of distinct tumor parts. We hypothesise that increasing the reaction constant for movement would lead to further increase of distinct parts.

The simulated results are shown in Figure 9.

Morphology analysis

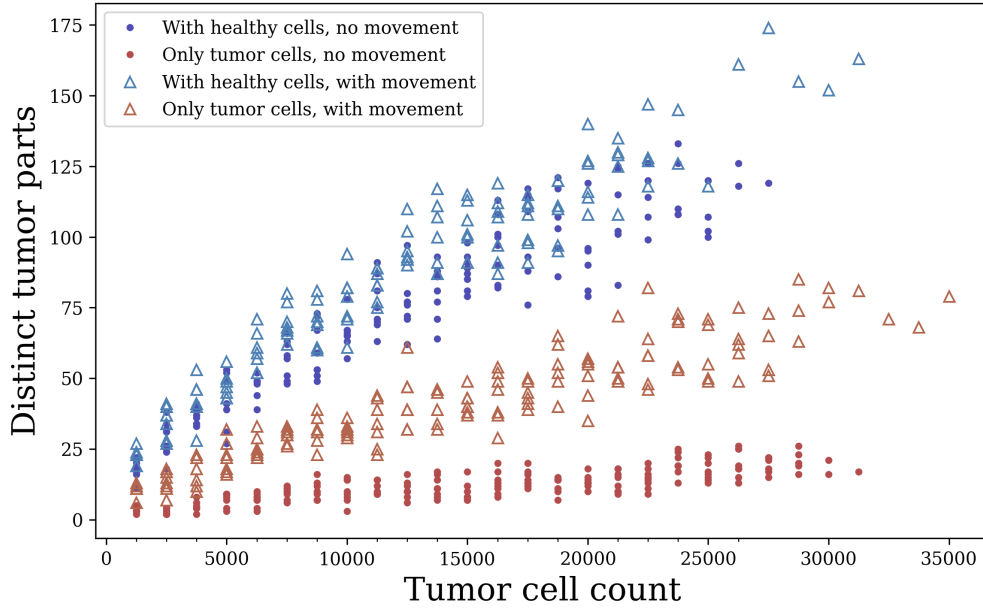


Figure 9: Distinct tumor part measurements, at fixed tumor cell count values. Simulations that are initialized with tumor cells only are in red, and those that are initialized with both tumor and healthy cells are in blue. Data points denoted with dots correspond to simulations ran without enabling cell movement, and triangles correspond to simulations where cell movement was enabled. All simulations were ran on a $50 \times 50 \times 50$ lattice, with relative division rate of 9.3, maximal iteration count of $1.25 \cdot 10^7$ and without any drug interventions.

6.2 Parameter analysis of drug interventions

Choosing the appropriate treatment and administration regime poses a challenge in clinical practice. To showcase how clinical decisions could be assisted by simulations in the future, we examine a hypothetical chemotherapy drug. We analyze how drug properties such as *toxicity* and *specificity* are connected to tumor growth and patient survival.

Simulations are run on a $50 \times 50 \times 50$ lattice, with relative division rate set to 9.3, until the maximal iteration number ($1.25 \cdot 10^7$) is met or the early stop mechanism is triggered.. On top of apoptosis, division and movement we introduce the two reactions associated with drug interventions: *Cytotoxic drug effect* and *Drug decay*.

Drug molecules are introduced periodically. Administration starts when the tumor size reaches 2000 cells, and from there a drug dose of 1000 is administered every 2.5 time units until the end of the simulation. If all tumor cells are extinct, the therapy stops, but the simulation runs until the maximal iteration limit.

While keeping all other parameters the same, we change two: the *drug toxicity* and the *drug*

specificity. To illustrate the effect of these changes in the parameters on cell growth, heatmaps are generated. Each point in the heatmaps is the average of 96 realizations, so outlier trajectories have no significant effect on the results. The x-axis shows the change in *drug specificity* while the y-axis shows the change in *drug toxicity*. These heatmaps can be seen in Figure 10.

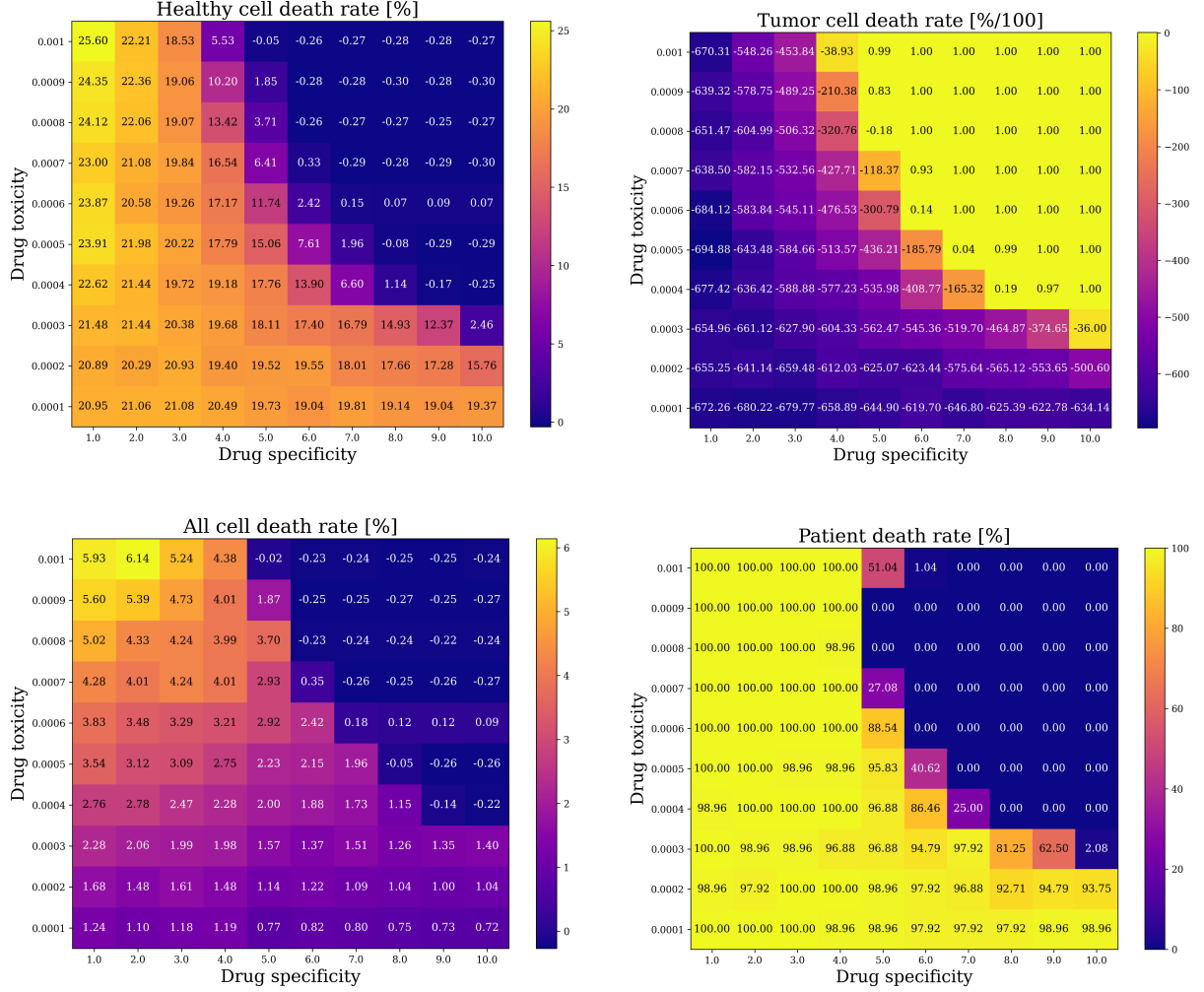


Figure 10: Death rate of healthy, tumor, all cells and patient. Each point in the heatmaps is the average of 96 realizations. All simulations were ran on a $50 \times 50 \times 50$ lattice, with relative division rate of 9.3, maximal iteration count of $1.25 \cdot 10^7$ with drug intervention. The x-axis shows the change in *specificity* while the y-axis shows the change in *toxicity*.

We observe a clear distinction between drugs that are small in toxicity and specificity and those that are highly toxic and specific. In order to achieve extinction of the entire tumor a combination of both is required, either one alone is insufficient.

We can see a sharp edge, where all cell death rates seem to change very fast for even small changes in the parameters. This is due to the fact that the therapeutic intervention stops when tumor is eliminated, but the simulation runs until the maximal iteration. This gives plenty of

time for the now tumor-free tissue to regenerate.

In order to obtain information about the harshness of these seemingly successful treatments, we examine patient's survival. For that we say that a patient survives a treatment if at any point in time during the simulated period he/she has lost less than 10% of his healthy cells compared to the initial state.

6.3 Time course of single trajectories

To have a deeper understanding of how each simulation proceeds, the number of cells and the concentration of the drug are plotted as a function of time.

It is important to note that these plots show a single realization, not an average of 96 as shown in the heatmaps. This distinction is important because the tumor detection time varies, leading to asynchronous cycles of drug delivery across different realizations. Averaging these realizations could obscure the drug's impact on cell counts.

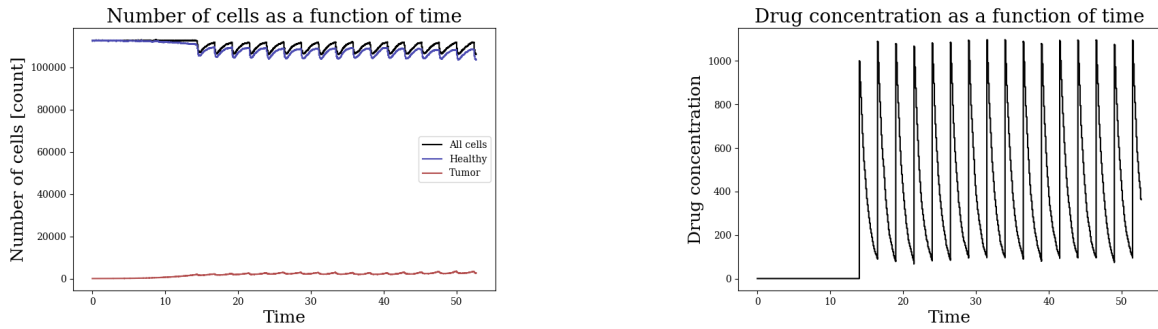


Figure 11: Time course of simulations with parameters $drug\ toxicity = 0.0006$ and $drug\ specificity = 6$. The simulation was run on a $50 \times 50 \times 50$ lattice with a relative division rate of 9.3 and a maximum iteration count of $1.25 \cdot 10^7$ under drug intervention. The x-axis represents simulated time, while the y-axes show the number of cells (left) and the concentration of drug molecules (right).

Figure 11. shows the case where $toxicity = 0.0006$ and $specificity = 6$, representing roughly the center of the heatmaps. In this case, we can observe that the drug has a significant effect on the course of the disease. Although it cannot completely eradicate the tumor cells, it can stop or at least slow down their growth. Another important observation is the continual decrease in the number of healthy cells. This suggests that although the treatment slows down the progression of the disease, it does not ultimately lead to a cure, as the patient dies either from the loss of healthy cells that have died as a result of the drug or from the growth of tumors that have survived the treatment. It should also be noted that chemotherapy treatments are typically

stopped before reaching as many doses of the drug as simulated here, with most consisting of 7-9 sub-doses. We have extended the number of doses and duration in this simulation to explore whether prolonged treatment can effectively eradicate the tumor.

6.4 Runtime

As mentioned several times in this thesis, special attention was given to maximizing the program's speed. The primary reason for this is that, due to the stochastic nature of the simulations, hundreds of realizations often need to be run to obtain reliable results. The runtime of individual realizations depends on several simulation settings, which are detailed in this section.

The simulation results presented in this section were obtained with 100 iterations per slot on a lattice of size $50 \times 50 \times 50$, resulting in a total of $50 \cdot 50 \cdot 50 \cdot 100 = 1.25 \cdot 10^7$ iterations as halting condition. Here, the factor 100 may be referred to as *reactionPerSlot*. These values are indicated with red lines in Figures 12. and 13.

6.4.1 Dependence of running time on the number of events

In this case, we adjust the *reactionPerSlot* factor, resulting in more events (reactions) occurring on the same lattice in each scenario. As expected, the runtime depends linearly on the number of iterations performed. However, we observe that after a certain point, the linear relationship breaks down and starts to fluctuate around a certain value, no longer increasing. This phenomenon is due to the early stop mechanism, described in Section 5.6.

In Figure 12., we ran simulations on a $50 \times 50 \times 50$ lattice with different *reactionPerSlot* parameters, shown on the x-axis, and measured the execution time for each realization, shown on the y-axis. The reaction constants were chosen as described in 6.3.

When plotting simulations, we would like to show the average of roughly the same number of realisations at each time point. To do this, we need to choose a *reactionPerSlot* value for which early stops occur rarely or not at all. Figure 12. shows that this is the case for *reactionPerSlot* = 100.

6.4.2 Dependence of running time on the size of the lattice

Since the iteration limit is calculated as the product of the lattice side lengths and the *reactionPerSlot* factor, running time significantly depends on lattice size. In this study, we examine

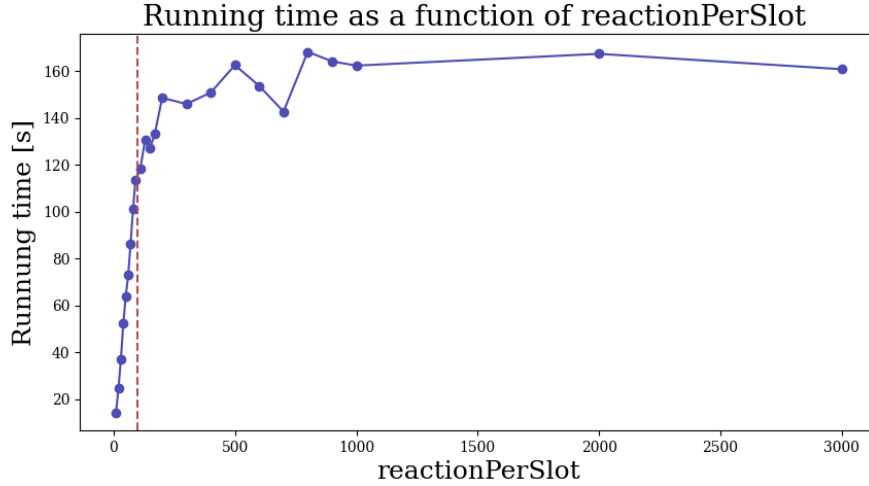


Figure 12: Selection of an appropriate *reactionPerSlot* value to ensure that when averaging multiple realizations, a consistent number of time series are available at each time point. Simulations were performed on a $50 \times 50 \times 50$ lattice with varying *reactionPerSlot* values (x-axis) and measured execution times for each realization (y-axis).

rectangular volumes, choosing equal side lengths for our 3D lattices. Therefore, in these cases, the execution time of a realizations is proportional to the cube of the side length.

It is worth noting, however, that unlike the previous scenario the function will not fluctuate around a specific value but will instead increase monotonically towards infinity. As the lattice size increases, reaching the edge of the lattice also takes more iterations.

In figure 13., we choose *reactionPerSlot* to be 100 and change the lattice side lengths from 10 up to 100, shown on the x-axis, and measured the execution time for each realization, shown on the y-axis. The reaction constants were chosen as described in 6.3.

6.4.3 Other dependencies

The runtime, in addition to the aforementioned two parameters, also depends on settings that are not selected based on these criteria.

It is greatly influenced by the types of saving methods used, as I/O operations consume significant time, and in certain saving types, this needs to be done in every iteration.

Additionally, the runtime is influenced by the constants governing the frequency of reactions, particularly the constant for *Movement*. This is because, while the *DrugDecay* reaction only involves modifying an integer variable, other reactions require placing or removing a cell on the lattice. The *Movement* reaction, however, is more time-consuming as it involves both removing the cell from its original position and placing it in a new location, significantly

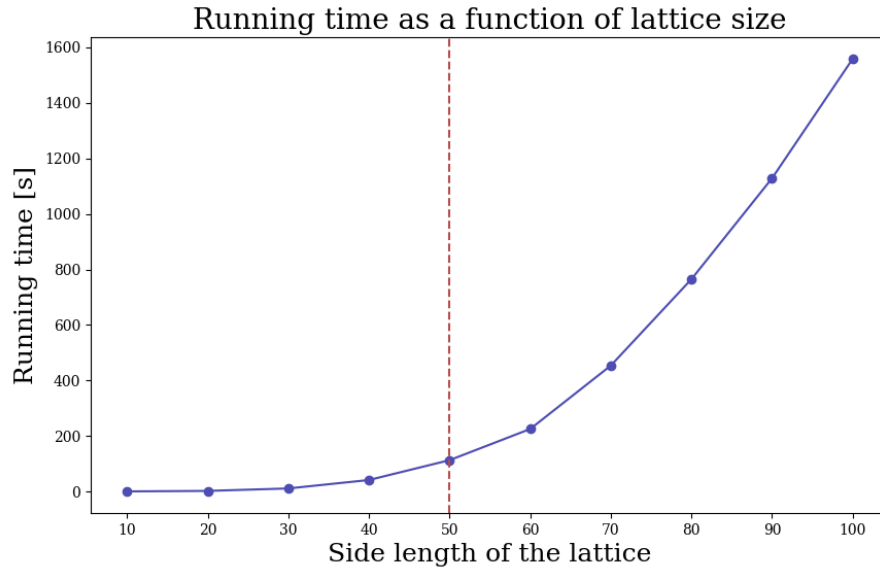


Figure 13: Dependence of execution time on lattice size. Simulations were performed with *reactionPerSlot* set to 100, varying the lattice side lengths from 10 to 100 (x-axis), and measuring the runtime for each realization (y-axis).

increasing the processing time.

7 Summary

In this thesis, we introduced our extended version of Péter Boldog’s marginal Reduced Rate Method, which enables the simulation of heterogeneous cell population dynamics in a three-dimensional space while accounting for the spatial positions of individual cells. This method enables the study of phenomena that are not feasible with other differential equation-based deterministic models, such as volume exclusion and tumor morphology.

We presented the simulation software we developed based on this method, designed to model cell population dynamics and analyze the effects of various factors, including spatial constraints and external interventions like chemotherapy. Implemented in C++ with an object-oriented approach, the program prioritizes modularity, extensibility, and computational efficiency, providing a flexible tool for studying various biological systems.

The software system is composed of four primary components: a simulation module, a 3D visualization tool, a heatmap generator, and an analytic tool. Together, these components provide a comprehensive framework for simulating, visualizing, and analyzing complex biological processes. The 3D visualization tool, implemented in Unity, offers an intuitive representation of cell arrangements, while the Python-based heatmap generator and analytic tool allow for detailed statistical analysis and parameter sensitivity studies.

During the development of the software, we placed a strong emphasis on customizability to ensure its versatility for a wide range of applications. To achieve this, the code separates the simulation component as much as possible from the biological model. Overall, the program allows users to define custom cell types and reactions, the initial state, as well as to adjust the likelihood of each cell type engaging in specific reactions.

We created a basic tumor model to showcase the capabilities of our software, which consists of two different cell types, basic cellular behaviors like movement, proliferation and apoptosis and a hypothetical chemotherapy treatment. We investigated the effect of the presence of healthy cells on tumor growth rate and morphology, as well as the impact of varying parameters of hypothetical chemotherapy drugs on a patient’s chances of recovery.

The presented framework is a major step forward in creating cell level simulations of cancer growth and treatment. We hope that this program or its derivatives, will help researchers understand cancer biology and tumor formation and pave the way for simulation-assisted personalized medicine.

References

- [1] P. Érdi and J. Tóth, *Mathematical Models of Chemical Reactions. Theory and Applications of Deterministic and Stochastic Models*. Manchester, Princeton: Manchester University Press, Princeton University Press, 1989.
- [2] W. M. Haddad, V. Chellaboina, and Q. Hui, *Nonnegative and Compartmental Dynamical Systems*. Princeton University Press, 2010.
- [3] M. Pereira, B. Kulcsár, G. Lipták, M. Kovács, and G. Szederkényi, “The traffic reaction model: A kinetic compartmental approach to road traffic modeling,” *Transportation Research Part C: Emerging Technologies*, vol. 158, p. 104435, Jan. 2024, ISSN: 0968-090X. DOI: 10.1016/j.trc.2023.104435. [Online]. Available: <http://dx.doi.org/10.1016/j.trc.2023.104435>.
- [4] M. A. Vághy and G. Szederkényi, “Persistence and stability of generalized ribosome flow models with time-varying transition rates,” *PLOS ONE*, vol. 18, no. 7, M. Aqeel, Ed., e0288148, Jul. 2023, ISSN: 1932-6203. DOI: 10.1371/journal.pone.0288148. [Online]. Available: <http://dx.doi.org/10.1371/journal.pone.0288148>.
- [5] M. A. Vághy, I. Otero-Muras, M. Pájaro, and G. Szederkényi, “A kinetic finite volume discretization of the multidimensional pde model for gene regulatory networks,” *Bulletin of Mathematical Biology*, vol. 86, no. 2, Jan. 2024, ISSN: 1522-9602. DOI: 10.1007/s11538-023-01251-3. [Online]. Available: <http://dx.doi.org/10.1007/s11538-023-01251-3>.
- [6] T. Péni, “Nonlinear model predictive control with logic constraints for covid-19 management,” *Nonlinear Dynamics 102*, pp. 1965–1986, 2020.
- [7] P. Érdi and G. Lente, *Stochastic Chemical Kinetics*. Springer New York, 2014. DOI: 10.1007/978-1-4939-0387-0. [Online]. Available: <https://doi.org/10.1007/978-1-4939-0387-0>.
- [8] D. T. Gillespie, “Stochastic simulation of chemical kinetics,” *Annual Review of Physical Chemistry*, vol. 58, no. 1, pp. 35–55, May 2007.
- [9] T. B. Kepler and T. C. Elston, “Stochasticity in transcriptional regulation: Origins, consequences, and mathematical representations,” *Biophysical Journal*, vol. 81, no. 6, pp. 3116–3136, Dec. 2001.
- [10] M. Feinberg, *Foundations of Chemical Reaction Network Theory*. Springer, 2019.
- [11] D. T. Gillespie, “A general method for numerically simulating the stochastic time evolution of coupled chemical reactions,” *Journal of Computational Physics* 22, pp. 403–434, 1976.
- [12] J. Müller and C. Kuttler, *Methods and Models in Mathematical Biology: Deterministic and Stochastic Approaches*. Springer Berlin Heidelberg, 2015, ISBN: 9783642272516. DOI: 10.1007/978-3-642-27251-6. [Online]. Available: <http://dx.doi.org/10.1007/978-3-642-27251-6>.
- [13] D. T. Gillespie, “Exact stochastic simulation of coupled chemical reactions,” *The Journal of Physical Chemistry* 81(25), pp. 2340–2361, 1977.
- [14] Y. Cao, D. T. Gillespie, and L. R. Petzold, “Avoiding negative populations in explicit poisson tau-leaping,” *The Journal of Chemical Physics*, vol. 123, no. 5, Aug. 2005, ISSN: 1089-7690. DOI: 10.1063/1.1992473. [Online]. Available: <http://dx.doi.org/10.1063/1.1992473>.
- [15] Y. Cao, D. T. Gillespie, and L. R. Petzold, “Efficient step size selection for the tau-leaping simulation method,” *The Journal of Chemical Physics*, vol. 124, no. 4, Jan. 2006, ISSN: 1089-7690. DOI: 10.1063/1.2159468. [Online]. Available: <http://dx.doi.org/10.1063/1.2159468>.

- [16] Y. Cao, H. Li, and L. Petzold, “Efficient formulation of the stochastic simulation algorithm for chemically reacting systems,” *The Journal of Chemical Physics*, vol. 121, no. 9, pp. 4059–4067, Sep. 2004, ISSN: 1089-7690. DOI: 10.1063/1.1778376. [Online]. Available: <http://dx.doi.org/10.1063/1.1778376>.
- [17] D. T. Gillespie, “Approximate accelerated stochastic simulation of chemically reacting systems,” *The Journal of Chemical Physics*, vol. 115, no. 4, pp. 1716–1733, Jul. 2001, ISSN: 1089-7690. DOI: 10.1063/1.1378322. [Online]. Available: <http://dx.doi.org/10.1063/1.1378322>.
- [18] D. A. Charlebois and G. Balázsi, “Modeling cell population dynamics,” *In Silico Biology*, vol. 13, no. 1-2, pp. 21–39, May 2019, ISSN: 1434-3207. DOI: 10.3233/isb-180470. [Online]. Available: <http://dx.doi.org/10.3233/ISB-180470>.
- [19] R. E. Baker and M. J. Simpson, “Correcting mean-field approximations for birth-death-movement processes,” *Physical Review E*, vol. 82, no. 4, Oct. 2010, ISSN: 1550-2376. DOI: 10.1103/physreve.82.041905. [Online]. Available: <http://dx.doi.org/10.1103/PhysRevE.82.041905>.
- [20] D. C. Markham, M. J. Simpson, P. K. Maini, E. A. Gaffney, and R. E. Baker, “Incorporating spatial correlations into multispecies mean-field models,” *Physical Review E*, vol. 88, no. 5, Nov. 2013, ISSN: 1550-2376. DOI: 10.1103/physreve.88.052713. [Online]. Available: <http://dx.doi.org/10.1103/PhysRevE.88.052713>.
- [21] P. Boldog, “Exact lattice-based stochastic cell culture simulation algorithms incorporating spontaneous and contact-dependent reactions,” *arXiv:2208.04774*, 2022.
- [22] N. J. Schork, “Personalized medicine: Time for one-person trials,” *Nature*, vol. 520, no. 7549, pp. 609–611, Apr. 2015, ISSN: 1476-4687. DOI: 10.1038/520609a. [Online]. Available: <http://dx.doi.org/10.1038/520609a>.
- [23] D. C. Whitcomb, “What is personalized medicine and what should it replace?” *Nature Reviews Gastroenterology and Hepatology*, vol. 9, no. 7, pp. 418–424, May 2012, ISSN: 1759-5053. DOI: 10.1038/nrgastro.2012.100. [Online]. Available: <http://dx.doi.org/10.1038/nrgastro.2012.100>.
- [24] T. Hernandez-Boussard, P. Macklin, E. J. Greenspan, *et al.*, “Digital twins for predictive oncology will be a paradigm shift for precision cancer care,” *Nature Medicine*, vol. 27, no. 12, pp. 2065–2066, Nov. 2021, ISSN: 1546-170X. DOI: 10.1038/s41591-021-01558-5. [Online]. Available: <http://dx.doi.org/10.1038/s41591-021-01558-5>.
- [25] R. Laubenbacher, B. Mehrad, I. Shmulevich, and N. Trayanova, “Digital twins in medicine,” *Nature Computational Science*, vol. 4, no. 3, pp. 184–191, Mar. 2024, ISSN: 2662-8457. DOI: 10.1038/s43588-024-00607-6. [Online]. Available: <http://dx.doi.org/10.1038/s43588-024-00607-6>.
- [26] E. Katsoulakis, Q. Wang, H. Wu, *et al.*, “Digital twins for health: A scoping review,” *npj Digital Medicine*, vol. 7, no. 1, Mar. 2024, ISSN: 2398-6352. DOI: 10.1038/s41746-024-01073-0. [Online]. Available: <http://dx.doi.org/10.1038/s41746-024-01073-0>.
- [27] V. Hárs, “A sztochasztikus reakciókinetika néhány kérdéséről,” M.S. thesis, ELTE TTK, 1976.

Appendix

Tools used for the thesis

During the preparation of this thesis, we utilized the OpenAI ChatGPT-4 model alongside the DeepL online translator for translation purposes (from Hungarian to English). Additionally, we relied on the OpenAI ChatGPT-4 model and the Writefull tool integrated into Overleaf for grammar and language correctness checks.

Figures

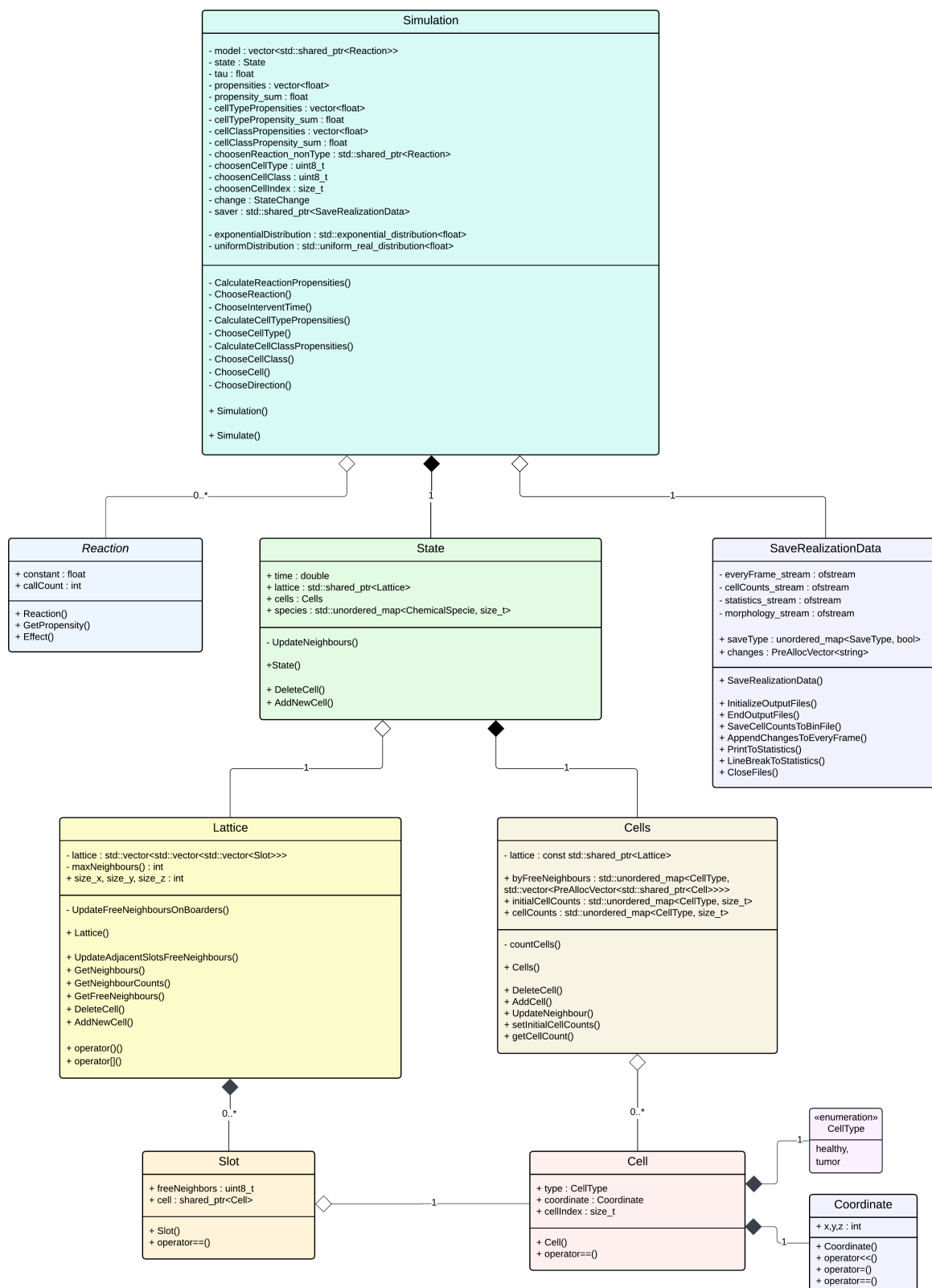


Figure 14: Class diagram of the simulation software architecture

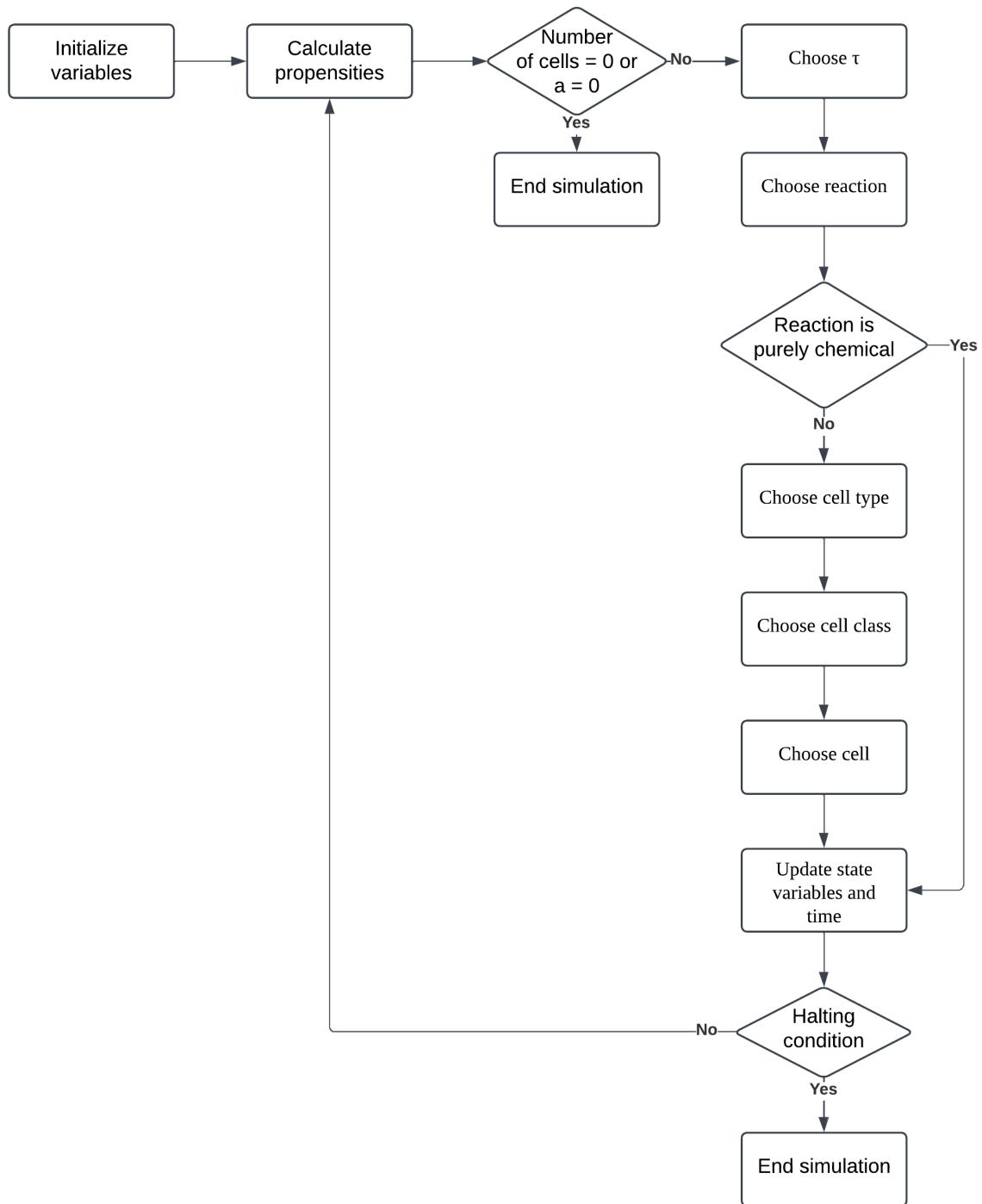


Figure 15: Flowchart of the extended mRRM

Code snippets

```
1  template <typename T>
2  class PreAllocVector
3  {
4  private:
5      size_t elementCount;
6      std::vector<T> vector;
7  public:
8      PreAllocVector<T>();
9      PreAllocVector<T>(const PreAllocVector<T>& other);
10     PreAllocVector<T>(const size_t& allocSize);
11     PreAllocVector<T>& operator=(const PreAllocVector<T>& other);
12     void remove(const T& item);
13     void remove(const size_t& index);
14     T& operator[] (size_t index);
15     size_t size() const;
16     void push_back(const T& item);
17     typename std::vector<T>::iterator begin();
18     typename std::vector<T>::iterator end();
19     void clear();
20     size_t getCapacity();
21     size_t getVectorSize();
22
23     friend std::ostream& operator<<(std::ostream& os, PreAllocVector<T>&
preAllocVector) {
24         for (size_t i = 0; i < preAllocVector.size(); i++)
25             os << preAllocVector[i] << " ";
26         os << "\n";
27         return os;
28     }
29 };
```

Listing 1: PreAllocVector class

```

1  template<>
2  inline
3  void PreAllocVector<std::shared_ptr<Cell>>::push_back(const
    std::shared_ptr<Cell>& item)
4  {
5      if (elementCount == vector.size())
6      {
7          std::cerr << "Error in PreAllocVector::push_back(T item): overflow
elementCount == vector.size()" << std::endl;
8      }
9
10     vector.at(elementCount) = item;
11     (*(vector.begin() + elementCount))->cellIndex = elementCount;
12     elementCount++;
13 }
14
15 template<>
16 inline
17 void PreAllocVector<std::shared_ptr<Cell>>::remove(const size_t& index)
18 {
19     if (index >= elementCount)
20     {
21         std::cerr << "Error in PreAllocVector::remove(size_t index): index
out of range" << std::endl;
22     }
23     else
24     {
25         *(vector.begin() + index) = *(vector.begin() + (elementCount - 1));
26         (*(vector.begin() + index))->cellIndex = index;
27         elementCount--;
28     }
29 }

```

Listing 2: Overloads for `std::shared_ptr<Cell>`

```
1 class Cell
2 {
3 private:
4 public:
5     CellType type;
6     Coordinate coordinate;
7     size_t cellIndex;
8
9     Cell();
10    Cell(const Cell& other);
11    Cell(const std::shared_ptr<Cell>& other);
12    Cell(Coordinate coordinate, CellType type);
13
14    virtual ~Cell() = default;
15
16    bool operator==(const Cell& other) const;
17 };
```

Listing 3: Cell class

```

1  class Lattice{
2      friend class Cells;
3  private:
4      std::vector<std::vector<std::vector<Slot>>> lattice;
5      int maxNeighbours;
6      void UpdateFreeNeighboursOnBoarders();
7
8  public:
9      int size_x, size_y, size_z;
10
11     Lattice();
12     Lattice(int size_x, int size_y, int size_z, int maxNeighbours);
13     Lattice(const Lattice& other);
14
15     Slot& operator()(const Coordinate& coord);
16     std::vector<std::vector<Slot>>& operator[](size_t x);
17     const std::vector<std::vector<Slot>>& operator[](size_t x) const;
18
19     void UpdateAdjacentSlotsFreeNeighbours(const Coordinate& coordinate,
20 bool added);
21
22     const std::vector<std::shared_ptr<Cell>> GetNeighbours(const
23 std::shared_ptr<Cell>& cell) const;
24     const std::map<CellType, uint8_t>
25 GetNeighbourCounts(std::shared_ptr<Cell> cell) const;
26
27     const std::vector<Coordinate> GetFreeNeighbours(const Coordinate&
28 coord) const;
29
30     void DeleteCell(const Coordinate& coord);
31     void AddCell(const std::shared_ptr<Cell>& cell);
32 };

```

Listing 4: Lattice class

```

1 class Cells
2 {
3 private:
4     const std::shared_ptr<Lattice> lattice;
5     void countCells();
6
7 public:
8     std::unordered_map<CellType,
9     std::vector<PreAllocVector<std::shared_ptr<Cell>>>> byFreeNeighbours;
10
11     std::unordered_map<CellType, size_t> initialCellCounts;
12     std::unordered_map<CellType, size_t> cellCounts;
13
14     Cells();
15     Cells(const std::shared_ptr<Lattice> lattice);
16     Cells(const Cells& other);
17
18     void DeleteCell(std::shared_ptr<Cell>& cell);
19     void AddCell(std::shared_ptr<Cell>& cell);
20
21     void UpdateNeighbour(std::shared_ptr<Cell> neighbour, bool added);
22
23     void setInitialCellCounts();
24     size_t getCellCount();
25 };

```

Listing 5: Cells class

```

1  class State
2  {
3  private:
4      void UpdateNeighbours(std::shared_ptr<Cell> cell, bool added);
5
6  public:
7      double time;
8      std::shared_ptr<Lattice> lattice;
9      Cells cells;
10     std::unordered_map<ChemicalSpecie, size_t> species;
11
12     State(std::unordered_map<ChemicalSpecie, size_t> species, uint32_t
size_x = 5, uint32_t size_y = 5, uint32_t size_z = 5);
13     State(State& other);
14
15     void DeleteCell(std::shared_ptr<Cell> cell);
16     void AddCell(std::shared_ptr<Cell> cell);
17 };

```

Listing 6: State class

```

1  class SaveRealizationData
2  {
3  private:
4      std::ofstream everyFrame_stream;
5      std::ofstream cellCounts_stream;
6      std::ofstream statistics_stream;
7  public:
8      std::unordered_map<SaveType, bool> saveType;
9      PreAllocVector<std::string> changes;
10
11     SaveRealizationData(fs::path directory_path, int realizationCount,
12 std::unordered_map<SaveType, bool> saveType);
13
14     void InitializeOutputFiles(const State& state, size_t iteraionCount =
15 0);
16     void EndOutputFiles(State& state, vector<std::shared_ptr<Reaction>>&
17 model, bool earlyStop, size_t earlyStop_iteration, double
18 simulationTime);
19     void SaveCellCountsToBinFile(std::vector<size_t>& cellCounts,
20 std::unordered_map<ChemicalSpecie, size_t> species, double& time);
21     void AppendChangesToEveryFrame();
22
23     template <typename T>
24     void PrintToStatistics(const std::vector<T>& data) {
25         if (saveType[statistics]) {
26             for_each(data.begin(), data.end(), [&](T _data) {
27 statistics_stream << _data << ' '; });
28         }
29     }
30
31     template <typename T>
32     void PrintToStatistics(const T& data) {
33         if (saveType[statistics]) {
34             statistics_stream << data << ' ';
35         }
36     }
37
38     template <typename T, typename... Args>

```

```
33     void PrintToStatistics(const T& first, const Args&... args) {
34         PrintToStatistics(first);
35         PrintToStatistics(args...);
36     }
37     void LineBreakToStatistics();
38     void CloseFiles();
39 };
```

Listing 7: Saver class

```

1  class Simulation
2  {
3  private:
4      vector<std::shared_ptr<Reaction>> model;
5      State& state;
6      vector<float> propensities;
7      float propensity_sum;
8      std::shared_ptr<Reaction> chosenReaction_nonType;
9      float tau;
10     std::exponential_distribution<float> exponentialDistribution;
11     std::uniform_real_distribution<float> uniformDistribution;
12     vector<float> cellTypePropensities;
13     float cellTypePropensity_sum;
14     uint8_t chosenCellType;
15     vector<float> cellClassPropensities;
16     float cellClassPropensity_sum;
17     uint8_t chosenCellClass;
18     size_t chosenCellIndex;
19
20     bool earlyStop;
21     size_t earlyStop_iteration;
22
23     StateChange change;
24     std::shared_ptr<SaveRealizationData> saver;
25
26     void CalculateReactionPropensities();
27     void ChooseReaction();
28     void ChooseInterventTime();
29     void CalculateCellTypePropensities
30     (std::shared_ptr<CellTypeSensitiveReaction> chosenReaction);
31     void ChooseCellType();
32     void
33     CalculateCellClassPropensities(std::shared_ptr<CellTypeSensitiveReaction>
34     chosenReaction);
35     void ChooseCellClass();
36     void ChooseCell();
37 public:

```

```
36     Simulation(vector<std::shared_ptr<Reaction>> model, State& state,  
std::shared_ptr<SaveRealizationData> saver = NULL);  
37     void Simulate(size_t iterationCount);  
38 };
```

Listing 8: Simulation class