University of Sheffield

University of Sheffield

Development of a generic agent-based cell model in Python

Jessica Leatherland

Supervisor: Dawn Walker

A report submitted in fulfilment of the requirements
for the degree of BSc in Computer Science (Artificial Intelligence)

in the

Department of Computer Science

May 8, 2024

Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Jessica Leatherland

Signature: J.Leatherland

Date: 07/05/2024

Abstract

The biological cell is the basic unit comprising all living organisms; understanding it provides insight into questions across the field of biology. Computer modelling can be an invaluable tool for the progression of this understanding, enabling researchers to predict cell behaviours and interactions and explore the mechanisms behind them. Agent-based modelling, a paradigm widely used in this field, has the ability to investigate the complex emergent behaviours that arise in multicellular populations. However, models are often specialised to fit the specific objectives of a research project, resulting in a lack of generic models to build on, with researchers typically required to program their model from scratch. This project aims to create a 3D agent-based model in Python, that encodes generic cell behaviours and facilitates efficient extension for future biological research. The resulting application achieves this, with: cell agents that have their essential behaviours; a physical model implemented to keep the agents from overlapping; a GUI where the user can set up simulations; and useful features for research including graph generation of the cell population sizes, 3D visualisation of the agents, and data export options.

Acknowledgements

I would like to extend my heartfelt thanks to Dr Dawn Walker, my supervisor, for her guidance throughout this project. Her patience, kindness, and encouragement have been invaluable in supporting the completion of this dissertation.

I would also like to thank my family for their love and support throughout my degree. I love you all very much.

Special thanks go to my best friend Jasmine Hui Ping Tay, who has made my university life so much brighter, and been there through all the good and the bad. I am grateful every day that you decided to turn around and talk to me on that fateful first day of second year.

Last but not least, I would like to thank Lorenzo Venanzi. You have brought me endless joy at a time when I would never have thought it possible, and I am so very happy to have you in my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Researchers of cell biology often use computational models because they can be a valuable tool in predicting cell behaviours and interactions and exploring the mechanisms that might be behind them. Agent-based modelling and simulation (ABMS) is widely used in this field, because of its ability to explore complex emergent behaviours arising from systems of individuals [1]. However, since models quickly become tailored to fit the particular objectives of a research project, there is a lack of models and frameworks encoding generic cell behaviours for use as starting points for new models [2]. This can slow down the rate of research, because development of new models often has to start from scratch. This project aims to create a generic 3D agent-based cell model in Python that can easily be built upon to investigate specialised cell types and populations, providing a basis for future research. Emphasis will be on creating an efficient, modular, well-documented system, following best coding practices to facilitate the model's extension.

1.1 Aims and Objectives

The main aim of the project is to create a highly extensible system, encoding basic cell behaviours. The objectives that must be met to achieve this are as follows:

The model will capture the basic functions of cells following the cell cycle [3, Chp. 17]: they will be able to move through their environment in 3D space, grow, divide through mitosis, and die. The model will employ a physical correction algorithm, to prevent cells from overlapping due to any of these processes. The extracellular environment will be represented by a 3D cube with programmable substance levels - cell agents will be able to sense the substance levels at their position, and their behaviour will be affected accordingly. For the purposes of this project, there will only be one programmable substance level (oxygen concentration), but the modular coding style will facilitate adding additional layers to the grid, should this be required in future research. The user will be able to control the initial state of the environment and the cells within it before running a simulation. They will also be able to view simulation statistics graphed on the GUI, and see a 3D visualisation of the simulated environment and cell agents.

Two additional objectives will also be considered: (i) the model might be extended to simulate a particular biological scenario, in order to test how easily and effectively the model can be specialised, and how successfully it models real data; and (ii) focus might be put on trying to optimise the Python program, and possibly translating the system to Flame GPU2[1], a highly optimised, parallelised environment. The decision between these additional objectives will be made later in the project, when it is clear whether there is adequate time to complete one.

1.2 Constraints

Handling large numbers of individual agents in a model can be computationally intensive, especially in three dimensions. Python is a relatively slow programming language. It will therefore be necessary to carefully consider which techniques should be used to optimise the processes in the 3D model. To avoid the constraints imposed by the use of a specific engine, this project will be built without one.

1.3 Overview of the Report

Chapter 2 discusses the relevant literature and technologies related to this project, including: underlying cell biology; paradigms and frameworks used in existing models; and how previous agent-based models have simulated cells. Chapter 3 details the project objectives, with analysis of individual requirements and plans for their evaluation, and includes a project risk analysis. Chapter 4 presents designs for the program flow, class architecture, key algorithms, and the GUI. Chapter 5 goes into the details of the project implementation and the testing that was carried out. Chapter 6 discusses the results of the project, with an evaluation of its efficacy and ideas given for further work. Chapter 7 concludes the report.

1.4 Relationship to Degree Programme

The main areas of technical knowledge required for completing this project are in good software design principles, agent-based modelling, modelling physical forces, GUI design, producing graphs, and 3D visualisation. The Computer Science (Artificial Intelligence) Undergraduate Degree offered at the University of Sheffield has taught most of these areas. Multiple modules have focused on software design and have given the author experience in producing modular systems. Two modules taken by the author have covered ABMS, including one assessing it in an assignment. Multiple projects over the degree have included creating a GUI (although not in Python), and producing graphs. No modules undertaken by the author have focused on physics-based modelling or 3D visualisation. Overall, this degree programme has provided a good basis for much of the knowledge needed to undertake this project, but some self-learning is still required to cover all areas.

[1] https://flamegpu.com/

Chapter 2

Literature Survey

This survey presents an overview of existing literature and technology relevant to this project. The basics of cell biology and the cell cycle must be understood to inform their simulation, and so these are described in detail. An exploration of alternative paradigms used in the biological modelling field is used to justify the adoption of an ABMS approach. Examples of existing agent-based biological models are then scrutinised to identify the best known techniques for simulating cell biology. Finally, software tools and frameworks used in this field are assessed to inform whether any should be adopted.

2.1 Cell Biology

In biology, a cell is considered the smallest, most basic unit of life. It consists of a membrane enclosing a concentrated chemical solution, and has the ability to multiply itself. As is explained by Alberts, et al. [3, Chp. 1], it is these fundamental facts that make the study of cells so significant; all life is comprised of cells, and as such, understanding them and their evolution allows insight into questions across the field of biology.

2.1.1 The Cell Cycle

The cell cycle is the name given to the process by which all cells reproduce, and is described by Alberts, et al. [3, Chp. 17] as follows. Cells cyclically perform a specific sequence of procedures in which first their full contents are duplicated, and then they divide in two. To pass on the genetic information to the next generation, the DNA in each of a cell's chromosomes must be replicated to form two complete copies. The copies must be properly distributed to two daughter cells, such that each receives the whole genome. Most cells also duplicate their other internal structures, and grow to accommodate this.

The eukaryotic (animal) cell cycle is typically divided into four sequential phases: G_1, S, G_2, and M. Chromosome duplication takes place during S phase. During M phase, nuclear division (mitosis) occurs, segregating the copied chromosomes into two daughter nuclei within the original cell, before cell division (cytokinesis) takes place, splitting the cell in two, such

that each daughter cell has one of the two nuclei. The G_1 and G_2 phases are gap phases, which allow time for growth and for monitoring the internal and external environment to assess whether conditions are acceptable for proceeding with the major S and M phases. If conditions are unsuitable, for example if a cell is crowded and has no space to divide, a cell will delay progress through G_1. It can also enter into a state of quiescence (inactivity) known as the G_0 state, where it can be considered to be withdrawn from the cell cycle. Withdrawal into G_0 is generally reversible, but cells can remain in this state indefinitely. The G_1, S and G_2 phases together are known as interphase. Cell growth occurs throughout interphase.

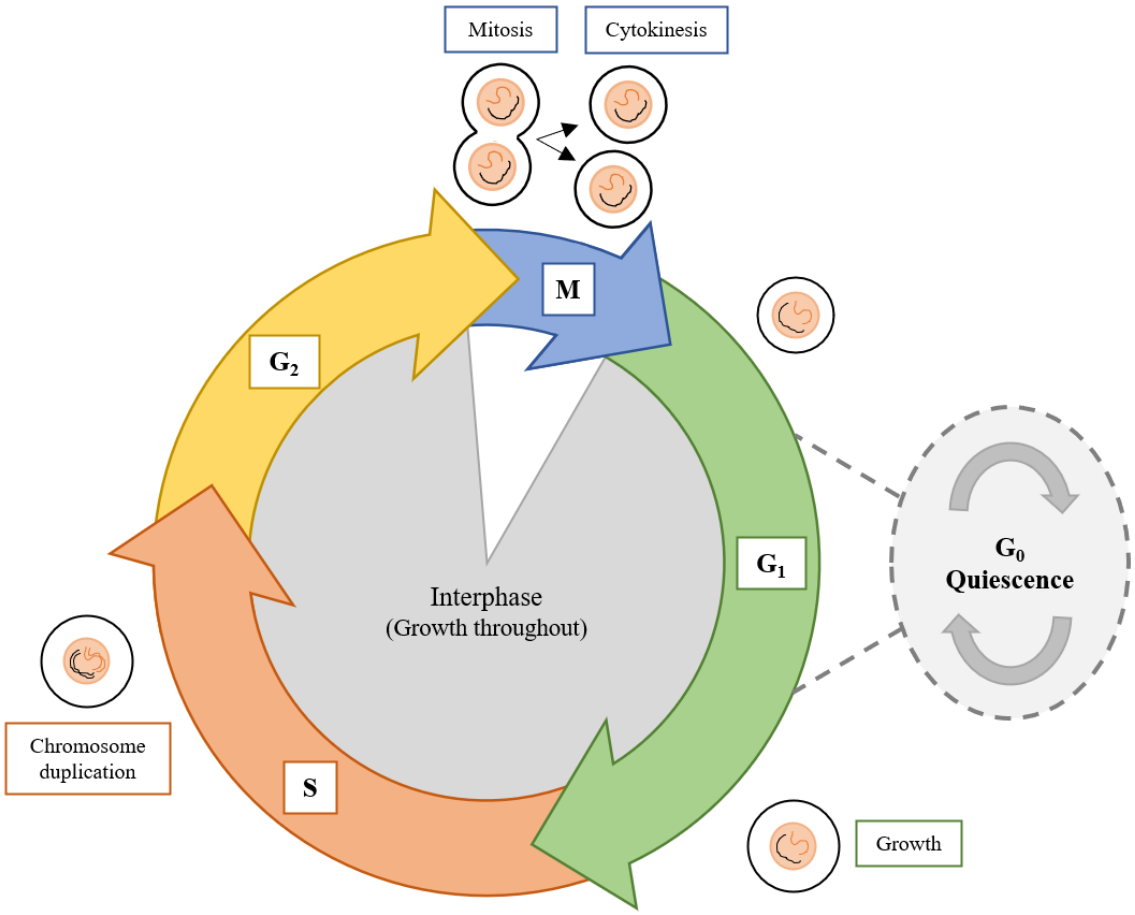


Figure 2.1: The eukaryotic cell cycle. Inspired by Alberts, et al. [3, Fig. 17-2], [3, Fig. 17-4].

The duration of different stages of the cell cycle can vary dramatically depending on the cell type and numerous intracellular and extracellular factors. For a typical mammalian cell, the total cycle time is roughly 24 hours. A possible distribution of phase times given by Cooper [4, Chp. 14], is G_1 : 11 hours, S : 8 hours, G_2 : 4 hours, and M : 1 hour. G_1 has the greatest potential variance within the same cell type as, if a cell transitions into G_0, the phase can last anywhere from several hours to several years, or indeed might last until the

cell or the organism dies.

2.1.2 Cell Death

Cell death is as integral to the growth, development, and maintenance of a multicellular organism as cell reproduction. For example, as explained by Alberts, et al. [3, Chp. 18], maintenance of tissue size relies on cell death occurring at the same rate as reproduction, and the health of an organism depends on the removal of damaged or infected cells. Alberts describes two mechanisms for cell death: apoptosis, and necrosis. In most cases, cells die in a programmed manner, systematically destroying themselves from within before being eaten by other cells. Apoptosis is the most common mechanism by which this occurs. Via this procedure, a cell shrinks, its internal structures collapse and disassemble, it breaks apart into several apoptotic bodies, and the surface of these bodies become chemically altered such that a neighbouring cell or a macrophage will engulf them, in order to stop their contents from spilling out, thus preventing an inflammatory response. Whilst the full process of apoptosis (from initial signalling to cell removal) is estimated to take between 6 and 24 hours in a living organism, the morphological changes take place in less than 2 hours [5]. Cell necrosis is the mechanism by which most cells that have experienced trauma die. It is often caused by a lack of energy, leading to a break down in the balance across the cell membrane. Such a cell will swell and burst, inducing an inflammatory response as its contents are spilled over neighbouring cells [3, Chp. 18]. This is usually a rapid process.

2.2 Modelling Paradigms

Several approaches have been used for modelling biological systems. Common paradigms used in this field are continuum-based methods and discrete (individual-based) methods. For the latter of these, cellular automata (CA)-based modelling and agent-based modelling are discussed here. There are arguments for and against the use of each of these methods which make them naturally suited to different modelling scenarios.

2.2.1 Continuum Modelling

A continuum model is one in which the system being investigated is assumed to be continuous, such that it can be described without considering the behaviour of its individual components. Instead it is characterised as a whole entity, whose mechanics and evolution are governed by mathematical equations. These are generally ordinary and partial differential equations. In biological modelling, continuum models ignore individual cells and instead simulate regions representing large multiples. See Milde, et al. [6, Fig. 2] for a diagram showing ways these regions can be characterised in a continuum model. The continuum approach has been applied to numerous tasks in biology, for example in the modelling of collective cell migration [7], and of cell-cell adhesion [8]. As detailed by Milde, et al. [6], continuum models work best on larger scales, for example in the modelling of growing tissue at the millimeter to centimeter range.

They are desirable over their individual-based counterparts for scenarios involving immense population sizes such as this, where it becomes unnecessary and unreasonable to track each individual cell [9]. Due to the lack of computation at the level of the individual, continuum models are less computationally intensive than agent-based models for large populations. However, the inevitable consequence of approximating cells by a continuum is that it prohibits the study of emergent behaviours arising at the level of individual cells [6], [10], making this an unsuitable modelling approach for this project.

2.2.2 CA Modelling

A cellular automaton is a lattice-based model, divided into discrete lattice cells (not to be confused with biological cells), which can each take one of a finite number of states. A lattice cell's state evolves at discrete time steps, governed by local transition rules based on the states of cells in a defined neighbourhood. CA models have the capacity to demonstrate complex emergent behaviours by virtue of their individual-based approach. This makes them particularly suited to the biological field, and as such they have seen many uses in biological modelling, including, for example, modelling of cell signalling [11]; predator-prey relationships [12]; and tumour growth [13]. CA modelling's greatest advantage is its inherent simplicity; CA models are both straightforward to implement and permit quick computation (which can often be highly parallelised). As noted in [14], when trying to determine rule sets to model complex biological mechanisms that are not well understood, the speed of these systems allows for examining huge numbers of parameter ranges that would be impractical to explore in more realistic models. CA models are not restricted to the level of simplicity discussed thus far, as they can be extended in a number of ways to broaden the scenarios that they can simulate. Some examples of these extensions, outlined in [15] include asynchrony, in which transition rules do not have to be applied to all lattice cells at once, allowing asynchronous updates; and heterogeneity, in which lattice cells can have different types and thus different transition rules, and/or transition rules can vary over time. Transition rules can also be heuristic-based/probabilistic. Pitfalls of CA models, arising from their dependence on discrete space, are that grid-based artifacts can emerge, and they are incapable of modelling continuous morphology or continuous migration throughout an environment. As such, they are unsuitable for modelling biological cells in cases where, as in this project, these features are necessary points of study.

2.2.3 Agent-Based Modelling

In an agent-based model (ABM), software agents move and interact within an environment based on governing rules in their programming. ABMs can be either on or off-lattice, meaning they are not restricted to being spatially discrete. The precise definition of a software agent is highly debated, however a useful characterisation of their basic features is given by Wooldridge and Jennings, [16], who identify autonomy; social ability; reactivity; and proactiveness. Here, agents are: (i) autonomous if they operate on their own and have some control over their

actions and state; (ii) have social ability if they have the capacity to interact with other agents; (iii) reactive if they perceive and react to changes in their environment; and (iv) proactive if they act not just in response to external stimuli, but are able to act towards some goal. These attributes are relevant in most cases but, as noted in Walker and Southgate, [10], since biological cells are not goal-driven, they are generally not considered to be proactive. Thus, whilst the first three features are often satisfied in the modelling of cells as software agents, the latter is not. Examples of biological ABMs include those studying granuloma formation [17], vascular adaptations [18], and cell signalling [19]. Similar to CA models, ABMs are highly suited to the study of emergent behaviour, due to their individual-based approach. ABMs also have advantages over cellular automata, in that they don't have to be discrete in space, meaning cells can be modelled to move continuously throughout an environment and to have changing morphology. ABMs are capable of modelling high spatial resolution over long temporal scales, however they are significantly more computationally intensive for large systems than comparable mathematical continuum models [2]. Similar to CA models, though, they are well suited to parallelisation, and as such their efficiency is highly dependent on their implementation.

2.3 Agent-based Modelling of Generic Cell Behaviours

There have been a number of ABMs that work at the cell level, where agents represent biological cells with a one-to-one correspondence. What follows is a discussion of three such models, focusing on the mechanisms they use to simulate generic cell behaviours (cell migration, the cell cycle, and cell death). In order that the methods used for solving cell overlap can be examined, the models chosen for scrutiny are all lattice-free and use physical modelling to govern cell positioning.

2.3.1 The Epitheliome

The Epitheliome project (see [20] for full details of the model), coded in Mathworks MatLab, couples an ABM to a simple physical model with numerical-based representation of intercellular forces. It models cells as 3D spheres on a 2D square bounded substrate, with the dimensions of the substrate and the distribution of calcium concentration throughout the substrate set by the user. The user can also set the number of cells initialised in the model and choose whether to place them using a random distribution or in chosen locations. The model works on two time scales: a slow time scale represents cell behaviours, whilst a fast time scale is used to update cell positions (representing a continuous process in the real world). The time step for each iteration is set to 30 minutes, such that the mean cell cycle times for the simulated cell types (approximately 60 hours for keratinocytes and 15 hours for urothelial cells) and their migration speed (roughly 1 µm per minute) can be well captured in the time periods. The modelled cell behaviours are: cell migration; bonding (to the substrate and other cells); spreading (once bonded to the substrate); cell growth and division (unless contact inhibited and/or insufficiently spread); and apoptosis (if cells have failed to bond to the substrate).

Cell Migration

Cells bound to the substrate but not to other cells are able to migrate, with the maximum allowed distance for migration in a single iteration being 2.5 times a cell's initial radius. New cells are initialised with a random migration direction and, based on probabilistic calculation, will usually remain migrating along the same trajectory but will sometimes choose another direction within 60° of their current direction. If a cell's path is blocked by another cell or the substrate boundary, the cell will only move as far as it can before touching the neighbour/wall. If no movement is possible in the current direction, cells change direction by 30° and attempt to move again (this is attempted up to 10 times in an iteration).

The Cell Cycle

The cell cycle length in iterations for each cell type is based on the real cell cycle time for the cell type divided by the time each iteration represents. The G1 phase length is selected from a normal distribution, with mean equal to half the cell cycle length and standard deviation equal to a tenth of that value, reflecting real observations of G1 duration. The cell cycle phases are governed by the following rule set:

1. G1: increase volume by $\frac{base\ cell\ volume}{total\ G1\ time\ steps}$ (such that the cell will have doubled in size by the end of G1), and increment internal counter
2. G1/G0 checkpoint (half-way through G1): if contact inhibited (bonded to 4 or more neighbours) or insufficiently spread (not discussed here, see [20] for further details), enter G0, else, continue in G1
3. G0: if bonded to less than four neighbours and sufficiently spread, reenter G1, else remain in G0
4. S or G2: increment internal counter
5. G2/M checkpoint: change to mitotic cell type
6. On entering M phase: assume rounded morphology and break intercellular bonds
7. On exiting M phase: divide into two identical daughter cells, overwriting the current cell with one daughter and appending the other to the end of the list of cell agents

Cell Death

In this model, cell death via apoptosis occurs if the cells fail to bind to the substrate during G1. New cells at the beginning of G1 are not bound to the substrate. At each iteration, they have a probability of binding which is biased to reflect that it is unusual for this not to occur within approximately the first hour (the first few iterations). In the rare case that cells have not bound to the substrate within G1, they shrink and are then removed from the model.

Physical Model

To minimise computation, only those cell agents that are overlapping or bonded are considered by the physical solver. Once these are identified, a force is calculated between each pair of linked cells proportional to the separation between their centres minus the sum of their radii (this is a positive attractive force if they are bonded, and a negative repulsive force if they are overlapping). Matrices are constructed for the x and y components of these forces separately, before an iterative numerical method applies these forces to find the x and y equilibrium positions of the cells independently. It is not guaranteed that a solution for 100% of the overlap is possible, so at each iteration of the physical solver, the sum of overlapping volumes for each cell is recorded in an error vector. Once the difference in the normal of this vector between two consecutive iterations is below 5%, the solver is terminated and the new cell positions are inserted into individual cell structures. Any remaining overlap is translated as squashing of the cells; cells with less than three intercellular bonds assume a more rounded morphology, reducing their spread radius to minimise the overlap.

2.3.2 BSim 2.0

BSim 2.0 (see [21] for full details) is a model for simulating bacterial populations, coded in Java. In this model, bacterial cells are Java objects that can alter their properties according to physical interactions and integration of their internal regulatory gene network. Their intracellular dynamics are controlled by ordinary and delay differential equations, and they are modelled as 3D capsular volumes with semi-rigid body mechanics that can move randomly in the environment, grow, and divide. The 3D simulation environment is a cube which can either have a boundary that cells reflect off or it can wrap. In addition to this, any 3D mesh can be put into the environment and cells can either be constrained to move only within the mesh, or treat it as the distribution of a particular chemical that affects them when they are within it.

Cell Growth and Division

Cell growth is governed by an ordinary differential equation model of rod elongation over time, with minimum and maximum length and elongation rate set as parameters. Division occurs once the original cell has grown past a set constant threshold length, with length calculated at each time step according to the Euler method for integration. When cells divide, the daughter cells are randomly moved within 5% of their current positions in order to break axial symmetry.

Physical Model

Interaction forces between cells are modelled using a semi-rigid body approach. Cell overlap is resolved by computing forces scaled according to the overlapping volumes for intersecting cells, before simultaneously relaxing the cell positions according to the computed forces. This

is done similarly to [22], in which cells are modelled as particles that can be connected to each other and/or to fixed points in the environment by massless springs, which are relaxed to separate the cells, minimising overlap.

2.3.3 PhysiCell

PhysiCell is a 3D simulator of multicellular systems written in C++ (see [23] for full details). It is parallelised using OpenMP, and is able to simulate cells in high numbers (10^6 cells is feasible on a desktop workstation, with still more possible on high-performance computing [HPC] nodes). Diffusion of substances in the environment has been handled via coupling the model to BioFVM [24], which computes vectors of diffusing substrates, so the user can tie cell behaviours to many changing environmental signals. Cells are modelled with changing volume (and sub-volumes for intracellular structures), migration, a cell cycle, and death processes. There are three separate time scales on which the model operates, to represent the real multi-temporal scales of the processes being simulated. In increasing order, there is a diffusion-level time step, at which the BioFVM environmental model is updated; a cell mechanics-level time step, at which cells move and overlap corrections are applied; and a cell-processes level time step, at which cell parameters are updated, their cell cycle/death cycle position is advanced, and their volume is updated.

Cell Migration

Cell agents have a number of user defined properties relating to motility. These are persistence time, migration speed, migration bias direction, and migration bias. Migration bias ranges from 0 to 1, with 0 being Brownian motion, and 1 being motion along the bias direction. When updating the cell velocity, the migration velocity is added to the current velocity, with the migration velocity having the potential to randomly change in each cell mechanics-level time interval with probability $\frac{mechanics\,time\,step}{persistence\,time}$. The migration velocity is based on a calculation combining the cell's speed, direction, and migration bias. There is also functionality available for biasing the migration along chemical gradients in the environment.

The Cell Cycle

The cell cycle is modelled as a collection of phases and transition rates, defined by the user. The user can also set entry and exit functions for each phase, and apply arrest functions to block phase transitions if certain conditions are met. Cell growth is controlled by systems of ordinary differential equations altering the volume and sub-volumes of each cell (the total volume is tracked along with the separate solid and fluid volumes of each of the cytoplasm and nucleus), which tend the volumes towards specified targets over time. The target solid volume is halved after cell division, doubled on beginning a new cell cycle, and set to 0 at onset of death via apoptosis (discussed later). Cell agents track their current cell cycle phase and the time they have spent in it. The cell cycle phase transitions occur with probability equal to the relevant phase transition rate multiplied by the cell processes-level time interval,

given that the cell is not arrested. The user can also make the transition rate change over time, according to any intracellular or extracellular mechanics they desire.

Cell Death

In PhysiCell, cells can die either via apoptosis or via necrosis. During apoptosis, the shrinking and blebbing of the cytoplasm, degradation of the nucleus, and active elimination of water from the cell are simulated. The relevant volume change rates are set to match the time scales of real cell volume loss in apoptotic cells, and either when cells drop below a user-set threshold volume, or after the mean duration of apoptosis has passed, they are removed from the model.

For necrosis, cytoplasmic and nuclear degradation are simulated. The cells also swell (oncosis), and when they reach a threshold volume, their membranes rupture (lysis). Again, the parameters controlling the rates of these processes are set to match expected real time scales. The trigger for entering necrosis can be deterministic, e.g., with cells instantly becoming necrotic if oxygen levels drop below a threshold, or stochastic, e.g., to represent cells surviving in low oxygen for some time, with the probability of necrosis increasing at each time step until it reaches a maximum.

2.4 ABM Software Tools and Frameworks

There are numerous tools for agent-based modelling, with popular frameworks including Swarm, MASON, Flame GPU, Repast, and NetLogo. Most ABM frameworks are written in Java, C, or C++, (including the examples provided), and are detailed in Abar's comprehensive review [25]. What follows will focus only on open source ABM tools developed specifically for Python, the programming language selected for this project, to determine their potential utility.

2.4.1 Mesa

Mesa is an ABM framework developed with the intention of being the favoured tool of researchers wanting to make ABMs in Python. The Mesa features, detailed by Kazil, et al. [26] are as follows: (i) creating a model is done using the Model class, in which the user defines the model's initial state, what happens when it runs, what occurs in the environment at each time step, and the space that the agents occupy; (ii) all agents inherit from the Agent class when they are defined; (iii) the different space types available for the environment are: continuous, single grid, multiple grids, hexagonal grid, or network grid; (iv) there is a time module, in which schedules can be created to control when agents are activated (when they can complete an action). Schedules can be: synchronous, where each agent's actions are queued, and then advance all at the same time; uniform, where agents are activated sequentially in the order that they were added to the scheduler; random, where agents are activated in a different random order in each iteration, or staged, in which agent activation

within an iteration is divided into stages, and all agents execute one stage before moving to the next; (v) the DataCollector class handles the recording, storing and exporting of data from the model and the agents. Mesa is integrated with browser-based front-end technologies (Jupyter Notebooks and Pandas) for analysis and visualisation of the data from the model.

In [25], Mesa is characterised as requiring moderate development effort, whilst only having small-medium scale computational modelling strength. This is corroborated in [26], where it is explained that accessibility and ease of use were prioritised over performance in Mesa's development, and it was initially only built for single-core processing. However, as mentioned in [27], Mesa's simple and extensible architecture has led to it having a rich community of contributors that provide extensions for it, including the possibility to run Mesa on a multi-processor system.

2.4.2 AgentPy

AgentPy is a Python 3 library for development and analysis of agent-based models. As described by Foramitti, [28], it combines the different tasks of an ABM into a single environment, and has been optimised for integration with IPython and Jupyter to allow for interactive analysis and visualisation of models created with the library. AgentPy allows the user to create their own agent types, model types, interactive simulations, and experiments, that can be run multiple times with different parameters. This, as well as data analysis of the results, can be done within a Jupyter Notebook. It also has tools for parameter sampling, Monte Carlo experiments, random number generation, parallel computing, and sensitivity analysis.

The structure of an ABM made with AgentPy is nested: agents are contained within environments, which are contained within a model, which is within an experiment. An environment can be a network, a spatial grid, or a continuous space. The model is used to initialise the environment and the agent objects within it, perform the simulation and record data from it. The experiment runs the model over multiple iterations and parameter combinations. Data output from the experiment is sent to the analysis and visualisation tools.

2.4.3 Repast4Py

Repast4Py is a recent development in the Repast Suite of agent-based modelling and simulation platforms, that builds on Repast HPC. As explained in [29], it was written in Python because of its ease of use and widespread utilisation across the sciences. Scalable performance was achieved through writing optimised ABM components in Python's C-API, and using the NumPy, Numba and PyTorch packages. Further detailed here are the features of Repast4Py, which are: (i) the global simulation is shared among a pool of processes, with agents distributed across them; (ii) all agents inherit from the R4Py_Agent object, which uniquely identifies them through storing an R4Py_AgentId containing the agent type, the process where the agent was created, and an id to distinguish agents of the same type on the same process; (iii) the whole agent population is managed by a shared context (a container based

on the semantics of a set), which provides functionality to iterate over all the agents within it, or over all agents of a specific type; (iv) all agent actions can be scheduled to occur at a specific "tick", where if action X is scheduled at tick 1, and action Y is scheduled at tick 5, X will always occur before Y, and if there are no actions scheduled on ticks 2-4, Y will occur immediately after X; (v) a shared scheduler is synchronised across all the processes. At each iteration of the shared scheduler, it determines the next tick that should be executed, pops the events that should take place at that tick from a priority queue, and executes those events, synchronising the individual schedulers for each of the processes; (vi) the possible environments (here called "shared projections") are either a grid or a continuous space. For a grid environment, agents are located in a matrix, and for a continuous environment, agent locations are expressed as floating point coordinates. Functionality of the environment includes agent movement, retrieval of agents occupying a particular location, and retrieving agent neighbours. The environments can also have a "value layer", which, for both the grid and continuous environments, is a matrix of numeric values that agents can interact with. In order to allow agents to not just access the value at their position, but affect it, the ReadWriteValueLayer class provides agents the ability to read from a read layer, and write to a write layer, and if needed, the write layer can be used to update the read layer (allowing, for example, modelling of diffusion).

2.5 Summary

This chapter has provided background for the project. Whilst restrictions inherent to the continuum and CA modelling approaches mean that they are unsuitable as tools to achieve this project's main aim, interrogation of agent-based modelling, and the techniques used to simulate cell behaviours in ABMs, has identified it as an appropriate approach. The exploration of the software tools and frameworks for agent-based modelling in Python considered here will help support decisions about which, if any, should be utilised.

Chapter 3

Requirements and Analysis

This project aims to provide a 3D agent-based model of generic cell behaviours to act as a starting point for models in future biological research. This is so that researchers can utilise their time more efficiently, focusing on specialising a model to their specific needs rather than building one from scratch. This chapter details the requirements that must be satisfied to achieve this, with consideration given to how they can be evaluated. It also discusses the ethical considerations of the project, and analyses the risks that may be faced.

3.1 Project Requirements

The overall aim of the project can be broken down into 6 key objectives:

1. Model cell agents that can migrate, follow cell cycle phases, and die
2. Implement a physical model that can resolve cell overlap
3. Allow the user to initialise a simulation, controlling the state of the initial environment and the cells within it
4. Save simulation data in an exportable format
5. Graph important simulation statistics on the GUI
6. Visualise simulations in 3D

Objectives 1-3 are critical, as the model needs to encode cell agents with generic behaviours, have some semblance of physical accuracy, and allow the user to interact with it to run simulations in order to have any utility in biological modelling. Objective 4 is required to allow users to utilise the data they get from the model outside the program. Objective 5 will give the model greater utility, as plotting statistics about the cell populations will allow the user to easily perform quantitative analysis of their simulations. Objective 6 will improve the user experience by making the simulations easier to understand, as the user will be able to visually analyse how the cell agents behave and how they interact with the environment.

These objectives can be further broken down into specific functional requirements, detailed in Tables 3.1 to 3.6. These have been prioritised as either mandatory (red), desirable (yellow) or optional (green). Mandatory requirements define the minimum viable product, and will all be implemented. Desirable requirements will significantly improve the quality of the system, and will all be included if possible. Optional requirements are extra features that would be beneficial to have, but will only be implemented if there is adequate time to do so.

Cell Agents		
ID	Requirement	Description
1	Seed new cell	Cells seeded in the environment are initialised with appropriate parameters according to their cell type and are tracked by the model
2	Cell cycle	Cells follow the cell cycle (with phases G1, S/G2, G0 and M), where they double in volume and divide if there is sufficient space and oxygen
3	Cell migration	As long as cells are not contact inhibited, they follow a random walk, with maximum distance at each iteration equal to their seed radius, and they cannot move outside the environment boundary
4	Cell death from hypoxia	At each iteration, if the oxygen at a cell's position is below the cell type's threshold, the cell is removed from the environment and tracked as being in a dead state
5	Cell death from age	Each time a cell divides, if its cell type ages, its age is incremented, and if it ages past its lifespan, it is removed from the environment and tracked as being in a dead state

Table 3.1: Functional requirements relating to the cell agents.

Physical Model		
ID	Requirement	Description
6	Resolve cell overlap forces	At each iteration, the physical model resolves cell overlap as best as possible by iteratively resolving forces between overlapping cells to spread them apart

Table 3.2: Functional requirements relating to the physical model.

Model Initialisation		
ID	Requirement	Description
7	Number of initial cells	The user can add cell types to the environment and set the number of initial cells of each type, up to a total of 100 cell agents
8	Environment size	The user can set the side length (in micrometres) of the environment (a 3D cube) above a minimum determined by the number of cells they have added (calculates the minimum space that the cells would fit into)
9	Random cell distribution	Cells added to the environment are initialised with random positions
10	Clustered cell distribution	When the user adds cells to the environment, they can optionally choose to place them in a cluster at a chosen location
11	Uniform oxygen levels	The user can set a uniform value for the oxygen concentration throughout the environment
12	Spatially varied oxygen levels	The user can set the detail level of the substance grid (in number of grid cells per side where min = 1 and max = grid side length / max cell agent diameter), and then choose grid cells to be the centres of oxygen sources, set their concentration, their radius of influence on other grid cells, and how quickly the concentration declines for grid cells within the radius
13	Generic cell type	A generic cell type representing normal cells is ready made for the user to add to the model
14	Cancerous cell type	A second cell type representing cancerous cells is ready made for the user to add to the model
15	Add cell types from GUI	The user can add new cell types to the model, setting all of the necessary parameters directly from the GUI, rather than needing to implement them in code

Table 3.3: Functional requirements relating to the initialisation of a model by the user.

Model Functions		
ID	Requirement	Description
16	Save model	At each iteration, information about the agents and environment necessary to reconstruct the simulation are saved in a .csv file, which can be exported by the user when a simulation has run, and is overwritten with each new simulation
17	Cancel model	The user can cancel a model while it is computing, and this takes them back to model initialisation
18	Extend model	After a simulation has run, the user can choose to extend it for a chosen number of iterations, and it then continues modelling from where it left off

Table 3.4: Functional requirements relating to model functions.

Graphical Output		
ID	Requirement	Description
19	Graph N against time	The number of cell agents in the environment at each iteration is graphed on the GUI
20	Graph reproduction and death	The reproduction and death counts at each iteration are graphed on the GUI
21	Graph cell phases	The numbers of cells in each cell cycle phase at each iteration are graphed on the GUI

Table 3.5: Functional requirements relating to the graphical output on the GUI.

Model Visualisation		
ID	Requirement	Description
22	Produce visualisation	After the user has run a model, they can click a button to produce a 3D visualisation from the agent positions and morphologies saved at each iteration
23	Cancel visualisation	The user can cancel producing a visualisation, and this returns them to the GUI where their model has been run
24	Cell colouring	Quiescent cells (in G0 phase) are visualised in a different colour to other cells
25	Highlight cells	Before the visualisation is produced, the user can choose cells to be highlighted throughout it
26	Visualise oxygen	The user can turn on/off an overlay of the substance grid which colours the grid cells in a gradient according to their oxygen concentration (with some transparency)
27	Playback controls	The visualisation has controls for frame rate, pause, play, step forward one iteration, step back one iteration, and a slider for scrubbing through the visualisation
28	Fixed camera angle	The visualisation shows a fixed angle of the 3D environment and agents within it
29	Multiple camera angles	The visualisation has multiple camera angles that can be switched between while the user is playing back the visualisation
30	Full camera orbit	The user can orbit the camera around the environment while playing back the visualisation
31	Export video	The user can export the visualisation as a video with a fixed camera angle
32	Record video	The user can record the playback of a simulation, controlling the camera angle throughout, and export this as a video

Table 3.6: Functional requirements relating to the 3D visualisation of a simulation.

Non-functional Requirements

The following non-functional requirements apply to the model as a whole:

1. The model must be written in Python
2. The model must be written using object-oriented programming (OOP), and be highly modular and extensible
3. The model must be written efficiently, and handle the computation of hundreds of cell agents in a reasonable time
4. The GUI should be responsive and intuitive

3.2 Testing and Evaluation

ABMS programs pose challenges to successful testing and evaluation. Their inherent emergent properties and stochasticity make it impossible to predict the exact outcome of initial conditions, and one cannot theoretically prove that exhibited behaviour is correct. Where experimental data are available, ABMs can be validated to an extent through calibration against expected trajectories within the data [30]. As this project does not have access to empirical data for validation, testing must rely on manual inspection of the system's behaviour, comparing it to general expected results. The tests that have been designed to evaluate each of the functional requirements are shown in Tables A.1 to A.8 in Appendix A.

There are some aspects of the system that can be assessed using specific metrics. To evaluate how well the physical model is working, the overlapping cell volume will be tracked as a percentage of total cell volume while running simulations, and the physical model will then be tuned in an attempt to minimise this. In order to test the efficiency of the program and assess areas for improvement, it will periodically be profiled using Yappi[1], a fast Python profiling library that supports inspection of multithreaded programs.

3.3 Legal and Ethical Considerations

Copyrights will be respected and intellectual property will only be used with explicit permission from the owner. Wherever code is used from an external source, appropriate licenses will be acquired and detailed in the project source code.

This project does not include activities involving human participants or data collection of any kind, and as such poses no ethical concerns.

[1] https://pypi.org/project/yappi/

3.4 Risk Analysis

The risk register given in Figure 3.1 details the potential risks for this project, and the measures that will be taken to mitigate their impact. It demonstrates that if appropriate mitigations are implemented, the risks to the project will generally be low.

Risk Register Severity Key					
Likelihood \ Impact	1: Very low	2: Low	3: Medium	4: High	5: Very High
5: Almost certain				Very high severity	
4: Very likely			High severity		
3: Likely		Medium severity			
2: Unlikely	Low severity				
1: Very unlikely					

Risk Register								
ID	Risk Description	Original risk			Mitigations Planned	Revised risk		
		Likelihood	Impact	Severity		Likelihood	Impact	Severity
1	Work could be corrupted or lost	3	4	12	Code will frequently be pushed to a Github repository and other local files made will be backed up and uploaded to a Google Drive	1	2	2
2	Delays due to illness could lead to not being able to complete the project	2	3	6	The project plan will include a two week buffer before the final deadline to allow for potential delays	2	1	2
3	Insufficient time to implement all project requirements	2	3	6	Project requirements will be implemented in order of priority (all mandatory requirements first, then all desirable, then all optional), to ensure at least the minimum viable product can be completed within the time available	2	1	2
4	The code might experience significant slow down and not scale well for large models	3	3	9	Multithreading and parallel processing libraries will be utilised to improve the performance as much as possible	2	2	4
5	Code libraries used could have bugs or require workarounds	2	2	4	Only robust and well-established libraries will be used	1	2	2

Figure 3.1: Risk register detailing the risks to the project and how they will be mitigated.

3.5 Summary

This chapter has detailed the objectives that need to be met in order to achieve the main project aim, and the requirements of each of these. An extensible 3D agent-based model will be made encoding generic cell behaviours, using a physical model to minimise cell overlap, saving data in an exportable format, and implementing a GUI where users can: initialise models, view graphical outputs of simulation statistics, and view a 3D visualisation of the environment and cell agents. The system will be evaluated using manual tests and profiling. Legal and ethical concerns have been considered, and the risks to the project have been analysed, with appropriate mitigative measures identified in order to reduce risk severity.

Chapter 4

Design

This chapter details the designs for the project system, which capture all of the project's mandatory requirements.

4.1 Architecture

The design for the cell agent class structure is shown in Figure 4.1. The Cell Type class is an abstract class, providing the base structure that all concrete cell types will need to implement for them to operate in simulations. The Generic Cell class is one of these cell types, and extends the base behaviour from the abstract class by including interaction with oxygen in the environment and a lifespan. Each instance of a cell type has a Cell Body, which contains its physical attributes and methods to act on these.

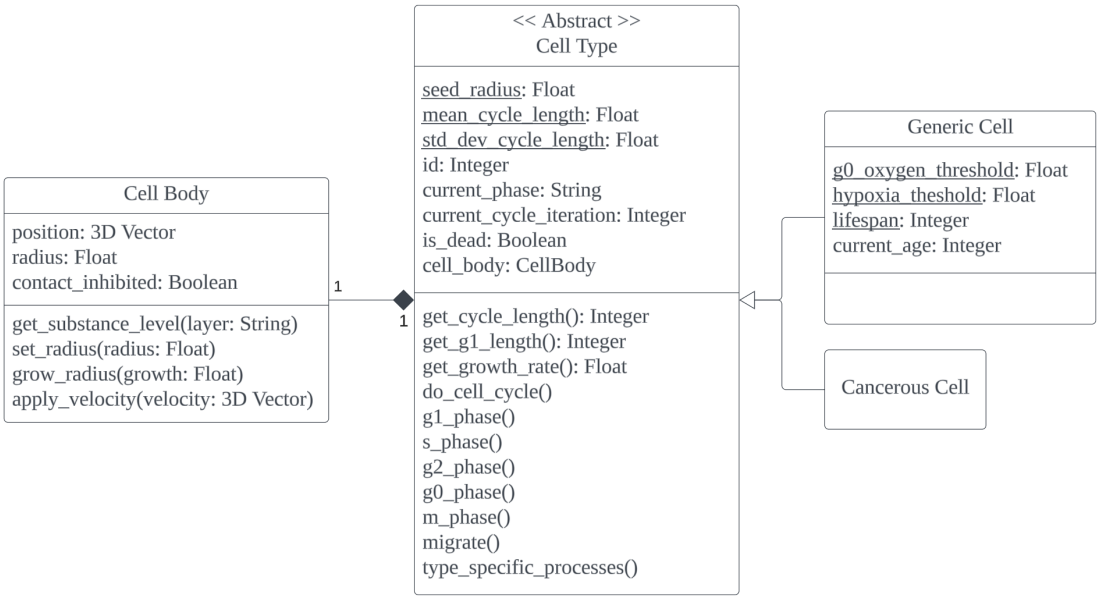


Figure 4.1: Class diagram showing the design for the cell agents.

The program flow will follow what is shown in Figure 4.2. The user will be able to change the simulation parameters as they wish, then when they click run, if the parameters are valid, the simulation subroutine will run. When this is finished, the user will be able to choose whether to produce a visualisation, and if they do then that subroutine will run. The user will then be able to use the playback controls to view the visualisation frames. At any point the user can choose to start a new simulation by entering new parameters and clicking run again.

The flows for the subroutines within the simulation are shown in Figure 4.3. At each simulation iteration, the do_cell_cycle(), migrate() and type_specific_processes() functions are called on each cell agent. The decision to include "type specific processes" was made with extensibility in mind, providing a dedicated point in the simulation loop where cell agents can have extra behaviours. For the Generic Cell class, this is the point where cells can die from hypoxia.

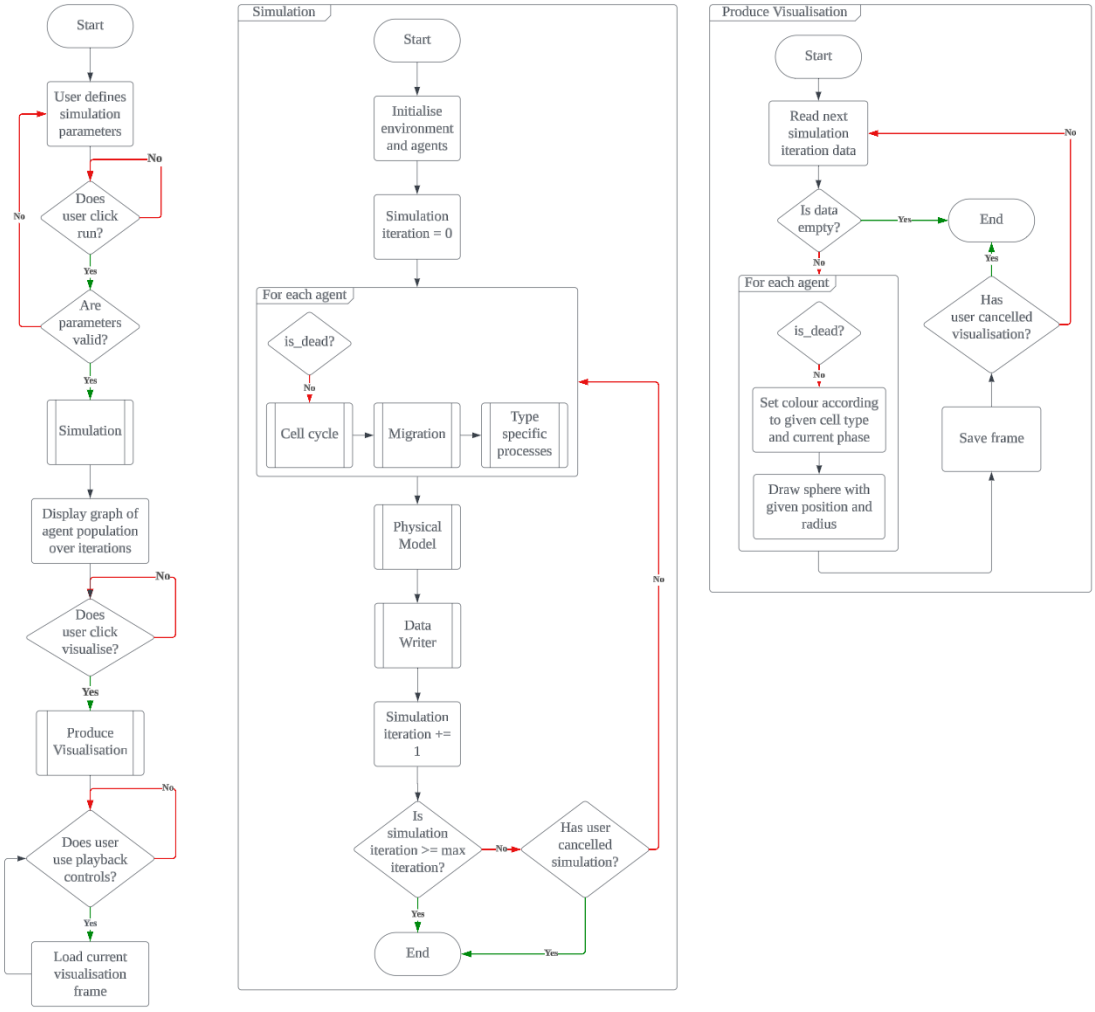


Figure 4.2: Flowcharts showing the proposed design for the overall program flow.

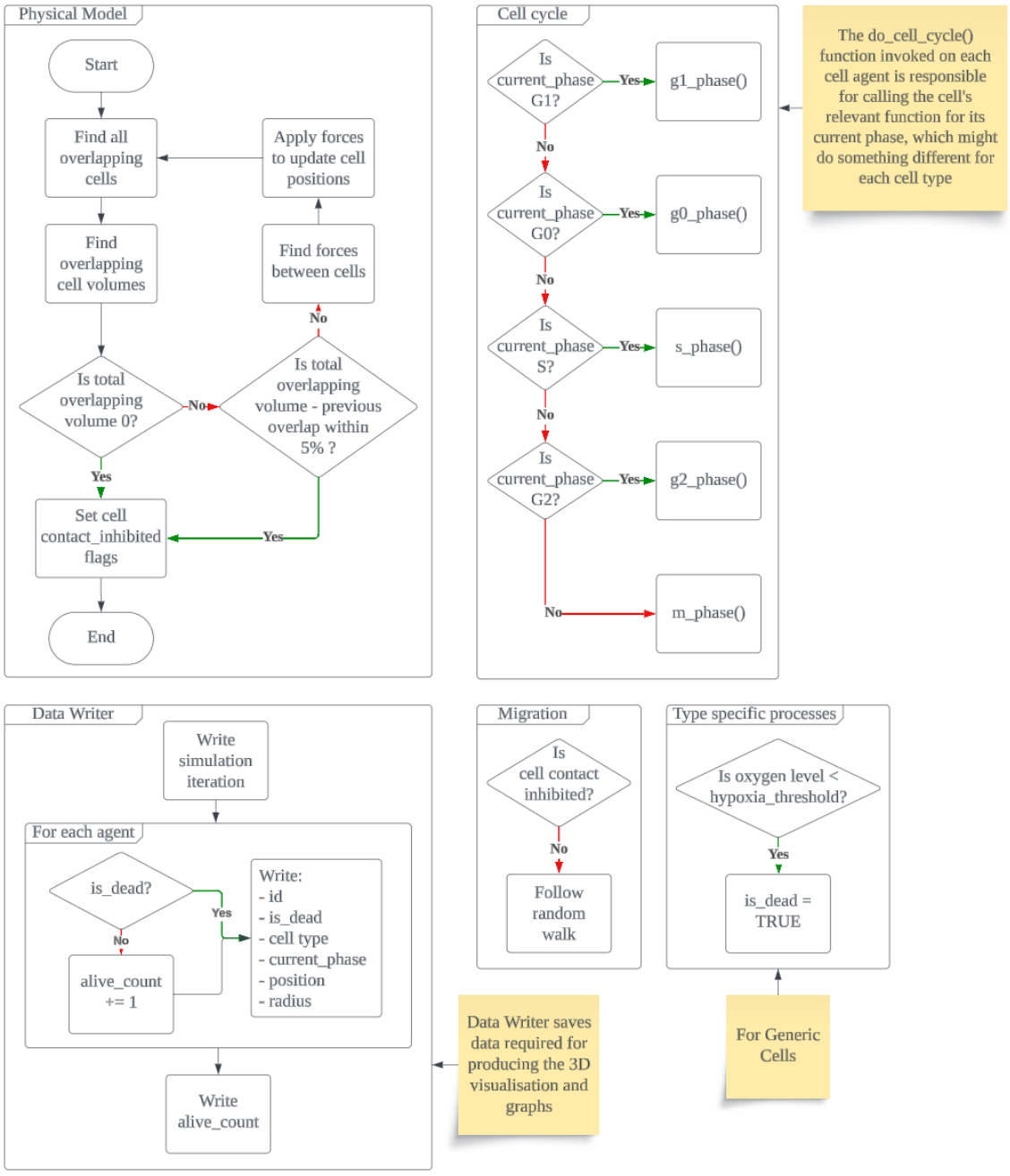


Figure 4.3: Flowcharts showing the proposed designs for subroutines of the Simulation module.

4.2 Algorithms

The cell seeding and cell cycle algorithm designs for Generic cells are displayed in Tables 4.1-4.5. When a cell is seeded in the environment, it is tracked by the model and its instance variables are initialised. This includes generating a cycle length and G1 phase length according to the Guassian distribution parameters specified in its class variables.

Seed new cell	
1.	Initialise cell position and append cell to model's list of cell agents
2.	Set cycle length according to mean and standard deviation for cell type
3.	Set G1 length with mean equal to half cell cycle length and standard deviation equal to a tenth of the mean (inspired by Epitheliome [20], as discussed in the Literature Survey)
4.	current_phase = G1
5.	current_cycle_iteration = 0
6.	current_age = 0
7.	is_dead = FALSE

Table 4.1: Pseudocode for seeding a new cell.

The cell cycle has G1, S, G2, and M phases, and an arrested G0 phase. For Generic cells, S and G2 are merged. In G1, cells grow to double their volume, and then either proceed to S/G2, or are withdrawn into G0. In S/G2, cells remain the same volume, and have a checkpoint to either proceed to M, or be withdrawn into G0. In G0, cells check at each iteration whether they can return to the cell cycle. In the M phase, cells divide in two: the parent halves its volume, and the daughter is seeded next to the it. The parent ages, and if it reaches its lifespan, it dies, otherwise its cell cycle parameters are reset.

G1 phase	
1.	current_cycle_iteration += 1
2.	Increase volume by seed_volume / G1 length
3.	IF current_cycle_iteration == G1 length:
4.	— IF oxygen level $\leq$ g0_oxygen_threshold OR contact inhibited, current_phase = G0
5.	— ELSE, current_phase = S/G2

Table 4.2: Pseudocode for the Generic Cell G1 phase.

G0 phase
1. IF NOT contact inhibited AND oxygen level > g0_oxygen_threshold:
2. — IF current_cycle_iteration == G1 length, current_phase = S/G2
3. — ELSE, current_phase = M

Table 4.3: Pseudocode for the Generic Cell G0 phase.

S/G2 phase
1. current_cycle_iteration += 1
2. IF current_cycle_iteration == cycle length - 1:
3. — IF oxygen level $\leq$ g0_oxygen_threshold OR contact inhibited, current_phase = G0
4. — ELSE, current_phase = M

Table 4.4: Pseudocode for the Generic Cell S/G2 phase.

M phase
1. Reduce volume to seed volume
2. Seed new cell next to this cell where there is space
3. current_age += 1
4. IF current_age $\geq$ lifespan
5. — is_dead = TRUE
6. ELSE
7. — Get new cycle length and new G1 length
8. — current_phase = G1
9. — current_cycle_iteration = 0

Table 4.5: Pseudocode for the Generic Cell M phase.

4.3 GUI Design

The main aim of the GUI design is to be simple and easy to use. A mockup for the initial GUI screen is shown in Figure 4.4. A widely understood user interface technique employed in this design is giving disabled buttons a more faded colour than enabled buttons to indicate to the user when they cannot be pressed. The top toolbar has a file button which reveals the export options, and the standard minimise, restore/maximise and close window buttons.

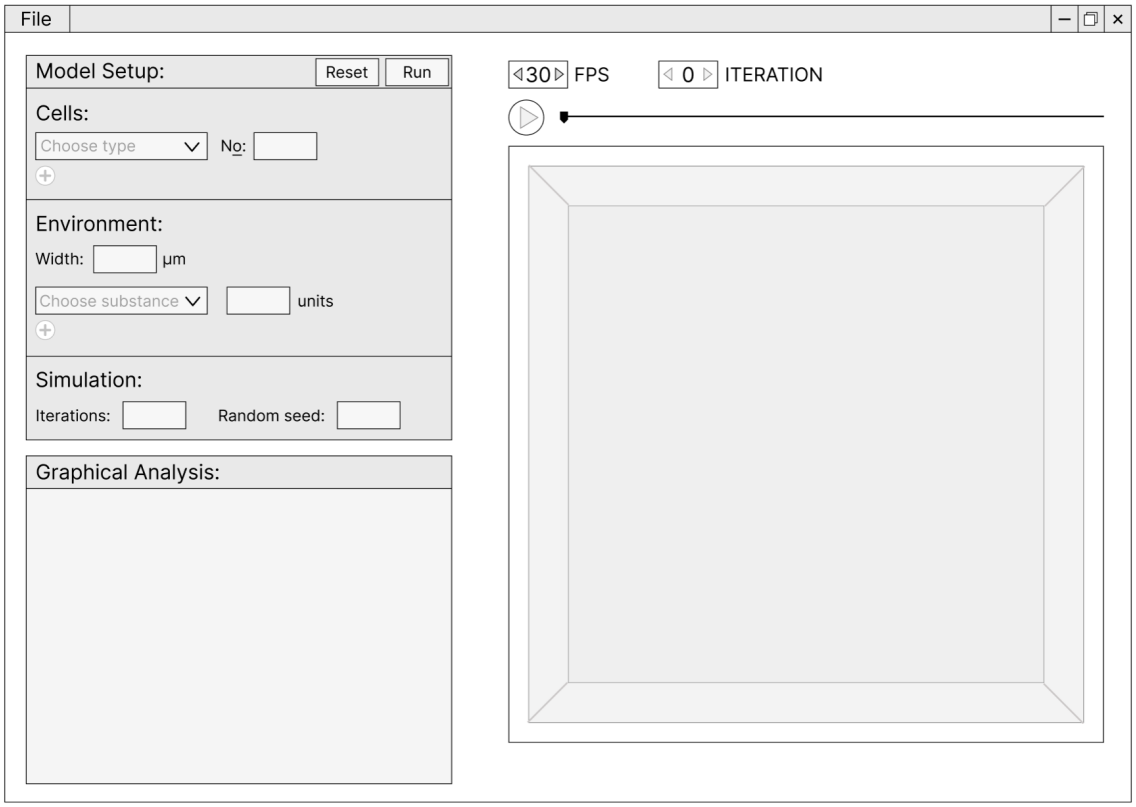


Figure 4.4: Mockup showing the design for the GUI.

The user can set simulation parameters in the Model Setup box. They can choose cell types to add with a dropdown box, which will populate with the names of all subclasses of the Cell Type abstract class. The plus sign below this will be enabled when a first cell type is chosen, and it will add another dropdown above it when pressed. The user can enter the number of initial cells of the chosen type in the box next to the dropdown. The environment has the same system for substance layers, and in the box next to the dropdown, the user can input any amount of that substance (with arbitrary units) to add to the environment. Clicking the reset button returns the model setup box to the state it is shown in currently. The model setup box will be able to scroll if its contents exceed the height of the box.

When a simulation has been run, the graphical analysis box will populate with a line graph of cell population over time, with lines for each cell type in the simulation.

When a visualisation has been produced, the current iteration in the iteration counter box is always displayed in the visualisation window. When the user presses play, the play button changes to a pause button, and vice versa. When playing, the iteration counter automatically increases at the rate defined in the frames per second box, and the marker on the timeline bar also moves accordingly. Clicking and dragging the marker on the timeline or clicking a particular point on the timeline updates the iteration counter and visualisation accordingly. When paused, the buttons in the iteration counter box can be used to step back or forwards through the iterations.

Chapter 5

Implementation and Testing

This chapter outlines how each aspect of the project was implemented, including the testing that was completed to ensure it met the acceptance criteria for the requirements. Changes to the design and requirements that were made during development are noted.

5.1 Application creation

5.1.1 Cells

The structure of the cell classes stayed much the same from the Design stage of this report. An abstract cell type class was made such that if another class is (i) either a direct or indirect subclass of it; (ii) defines the class constants; and (iii) implements the cell behaviour methods, it will automatically become available to the user to add to a simulation from the GUI. This same modular approach was used for the substance layers in the environment.

Once the cell behaviour methods had been translated from their pseudocode designs to make the GenericCell behaviour, it was then easy to add a cancerous cell type by following the same structure with a few small changes. The CancerousCell class is not affected by contact inhibition, does not require oxygen to survive or reproduce, has half the average cell cycle time of a GenericCell, and has an unlimited lifespan. These differences were chosen to create a clear distinction between the cancerous and generic cells for illustrative purposes, and to mimic the rapid proliferation rates that cancer cells exhibit.

A difference between the cell class implementations and the class diagram in the Design chapter (Figure 4.1) can be seen in the CellBody class, which has a function that was not foreseen as needed during the design stage. This function calculates how many of the environment borders the cell is in contact with, which is used in the contact inhibition algorithm described in the subsequent section.

5.1.2 Physical Model

The implemented overlap solving algorithm works as follows: (i) compare the positions of cells, accumulate the total of the overlapping distances between them, and build a dictionary

relating the cell IDs to lists of forces that should be applied to them; (ii) if there is no current overlap or the overlap has stabilised from the previous iteration of the solver, stop here; (iii) otherwise, for each cell in the cell force dictionary, apply the sum of its force list to its position; (iv) increment the solve iteration; (v) if the solve iteration has reached a maximum (set to 100 iterations), stop attempting to solve here, otherwise, repeat steps i-v; (vi) when solving has finished, find how many contacts each cell has within a certain radius (including contacts with other cells and any of the environment boundaries), and set cells with 6 or more contacts as contact inhibited.

The algorithm is determined to have stabilised if the difference in the current overlap and the overlap from the previous solve iteration is below a certain constant threshold, whilst the current overlap is also above another constant threshold to ensure that the algorithm is not cut short when removing very small overlap from the simulation.

The force that cell j applies to cell i is the unit vector pointing from cell j's centre to cell i's centre scaled by the overlap between them, which is given by the sum of their radii minus the distance between their centres. Formally, cell j with 3D vector position $\mathbf{p_j}$ and radius r_j applies a force to cell i with position $\mathbf{p_i}$ and radius r_i given by:

$$\lambda \cdot \frac{\mathbf{p_i} - \mathbf{p_j}}{|\mathbf{p_i} - \mathbf{p_j}|} \cdot (r_i + r_j + s - |\mathbf{p_i} - \mathbf{p_j}|) \tag{5.1}$$

Where s is a constant target separation used to ensure the cells separate rather than still touch at the end of the overlap solving algorithm, and λ is a constant force multiplier used to scale down the forces so that the physical model incrementally approaches the correct solution rather than making larger, more inaccurate steps. When the distance between the cells is exactly zero ($|\mathbf{p_i} - \mathbf{p_j}| = 0$), the equation would result in a divide by zero error, so in this case, the unit vector pointing from cell j to cell i is replaced by a random unit vector. When overlap is found between two cells and this force is calculated, the force is appended to cell i's force list and an equal and opposite force is appended to cell j's force list (the force vector is multiplied by -1).

All of the constants described thus far are constants of the PhysicalModel class. These were tuned to ensure the physical model separates cells in a believable way, and minimises overlap as far as possible whilst not taking too long to run. The total overlap remaining for each simulation iteration was tracked during this process. It is generally kept at zero unless there are many cramped cells or the environment isn't large enough for them to fit without overlap, but even in these cases the overlap is still kept within a reasonably low range. The values of the constants in the PhysicalModel class are given in Table C.1 in Appendix C.

There are two different implementations of the physical model. Both follow the same algorithm described above, but differ in how they choose which cells need to be compared to check for overlap. The first (PhysicalModel) uses a brute force approach, checking all possible cell pairs. This is inefficient, scaling with the square of the number of cells and also scaling with the density of the cells in the environment. In an attempt to optimise the algorithm, a second implementation was made (PhysicalModelWithLocals) which splits the environment into smaller chunks (local environments), converts each of the cells' positions

into an index relating to which local environment it falls within, populates a list for each local environment with the cells that are within them, and then a 2x2x2 window marches through the local environments, comparing cells that are within the window to see if they overlap. This is illustrated in Figure 5.1. Cell pairs that have been checked are tracked so that they are not compared multiple times if they fall into more than one position of the window. This algorithm ensures that cells that are far apart are not compared, which improves the efficiency of the physical model greatly in most circumstances. The algorithm does not scale with the number of cells, but only with their density in the environment. This can be seen in Figure 5.2, which shows the results of profiling the physical model to find out how long it took to solve a simulation iteration on average for different environment and population sizes. In the largest environment, which has room for all of the population sizes, the PhysicalModel algorithm gets slower with an increased cell population, whereas the performance of the PhysicalModelWithLocals algorithm hardly decreases. In the smaller environment which does not have room for the larger population sizes, both algorithms slow down, but the PhysicalModelWithLocals algorithm is still significantly faster than the PhysicalModel algorithm. Inspecting the values from the profiling, it can be seen that the algorithm using local environments is always at least twice as fast as the algorithm without, and often the difference is much greater than this.

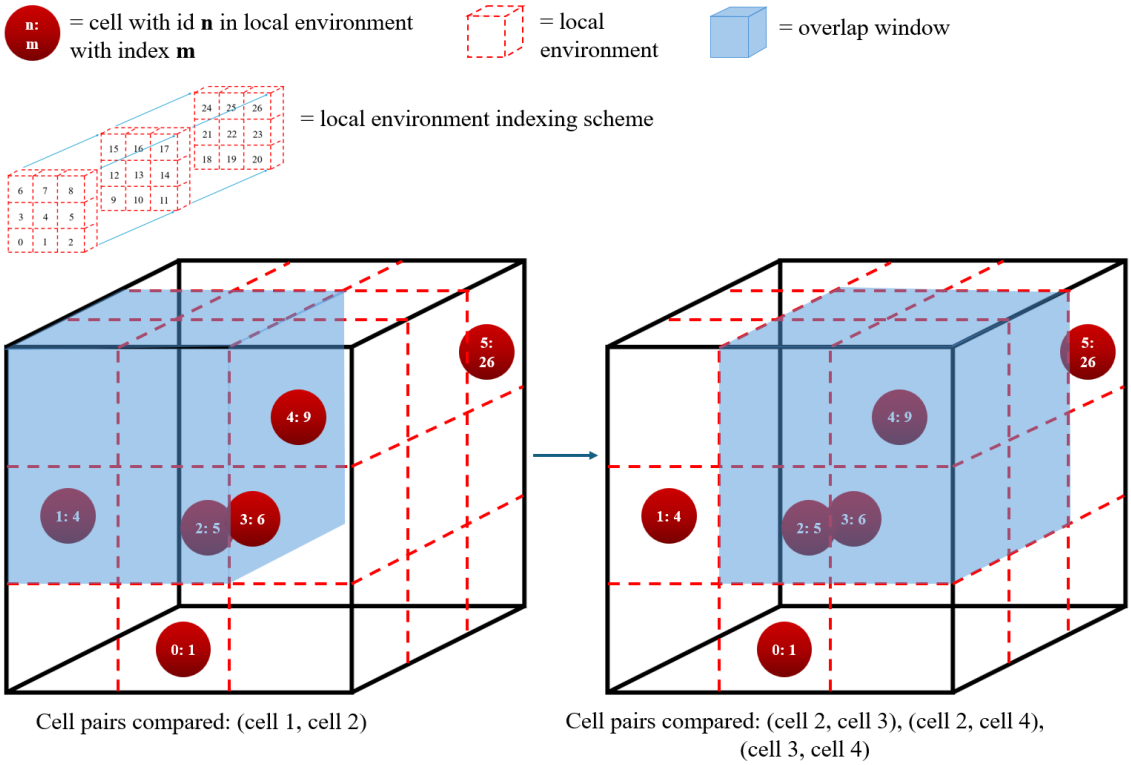
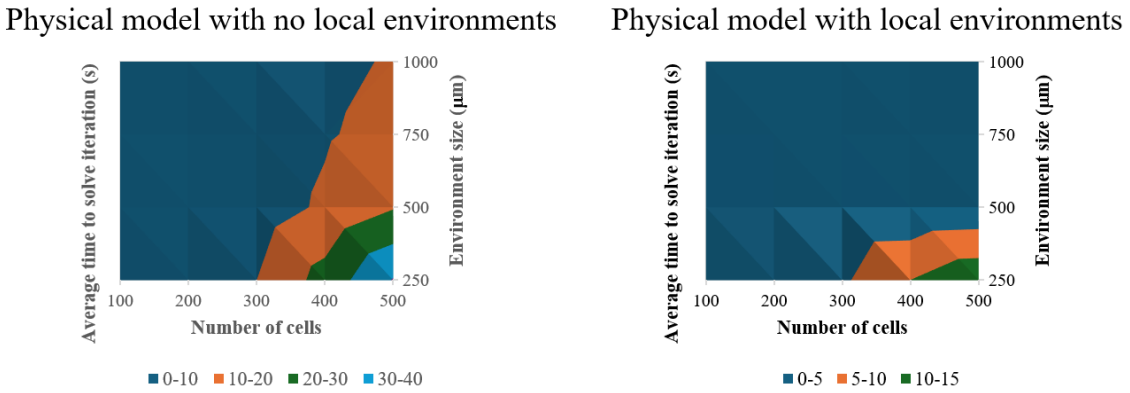


Figure 5.1: Illustration of the windowing algorithm for the PhysicalModelWithLocals overlap solver.



№ cells \ Env size (μm)	250	500	750	1000
100	0.69	0.47	0.26	0.17
200	3.51	1.98	0.68	0.81
300	9.98	4.83	3.68	2.19
400	23.77	11.57	9.04	4.09
500	40.54	19.35	13.6	12.17

№ cells \ Env size (μm)	250	500	750	1000
100	0.32	0.09	0.05	0.04
200	1.56	0.25	0.23	0.17
300	4.23	0.53	0.35	0.25
400	10.04	0.81	0.6	0.44
500	13.81	1.23	0.84	0.79

Figure 5.2: Profiling results for the two physical model algorithms for different environment sizes and cell numbers.

The minimum size of a local environment is set at twice the diameter of the largest a cell can become in the simulation (which for the Generic and Cancerous cells is slightly larger than 25μm when fully grown, meaning the local environments are a minimum of just over 50μm across). The number of local environments across one side of the environment will be the maximum number of these widths that can fit along the side. For example, for a 260μm wide environment, five widths could fit along the side, so there would be five local environments along each side, each with width $260/5 = 52\mu m$ (resulting in $5^3 = 125$ local environments in total). The performance of the algorithm is dependent on a tradeoff between needing to do the extra work of populating the local environment lists in each solve iteration and being able to carry out less cell overlap comparisons. As shown in Figure 5.3, when the environment is only big enough for three local environments widths (200μm or less), the PhysicalModelWithLocals algorithm is slightly slower than the original PhysicalModel algorithm. However, improvements in performance over the PhysicalModel algorithm are seen as soon as the environment is big enough for four local environments widths along each side. The consequence of this in the implementation is that on initialisation of a simulation, the program chooses which of the physical model classes to use based on the size of the environment, choosing the PhysicalModelWithLocals class if there is room for at least four local environment widths, and defaulting to the PhysicalModel class if this is not the case.

The profiling revealed that, on average, the physical solving algorithm takes up around 95% of the simulation run time. This means that improvements to the algorithm are very impactful in improving the performance of the model.

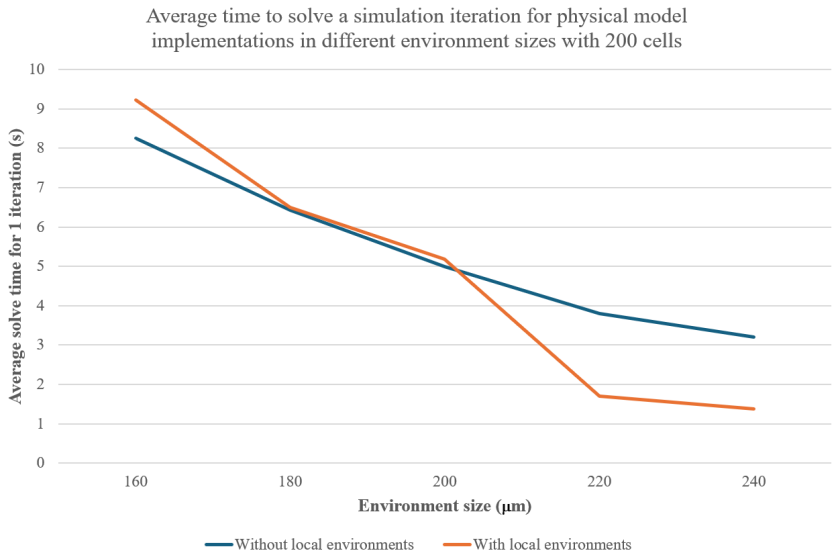


Figure 5.3: Profiling results for the two physical model algorithms for small environments.

5.1.3 GUI

The GUI was made using PySide6, a Python binding of an efficient and versatile C++ library which offers all of the GUI elements needed for this project as 'widgets' which can be arranged in different layouts. This includes a QOpenGLWidget, which was extended for implementing the visualisation (which was made using PyOpenGL, a Python binding of a fast C++ graphics library). As the author did not have prior experience with Python GUI development or with the C++ language, a program called Qt Designer was used to design the GUI layout and get the boilerplate Python code that would produce it to speed up development. This was then edited to fine tune the design and add the program logic to each of the elements. Figure 5.4 shows the results from Qt Designer. No major changes were made to the GUI design from the Design stage of this report. Screenshots of all windows and dialogs of the final application can be seen in Figures B.1 - B.5 in Appendix B.

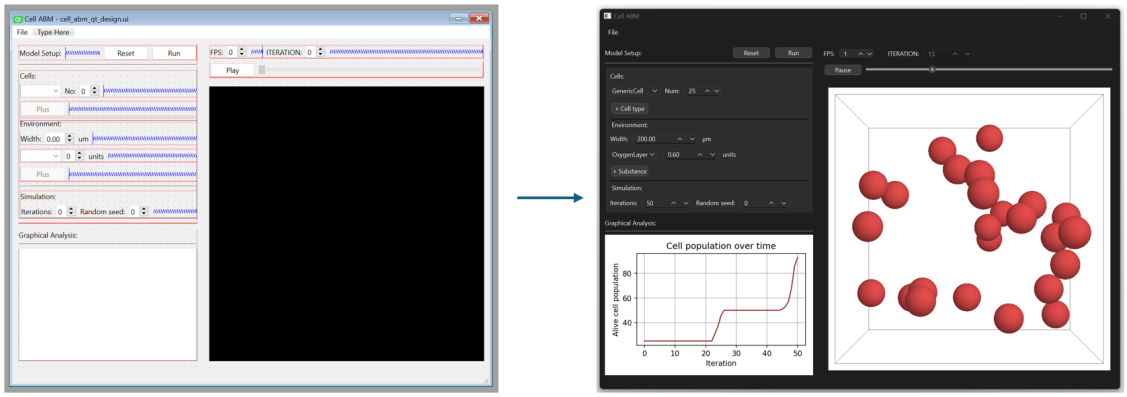


Figure 5.4: GUI design within Qt Designer and the final GUI produced from it.

5.2 Changes To Requirements During Development

A few slight alterations to the project requirements were realised during implementation of the system. It was expected that the visualisation frames could take a long time to be produced, so the user should be able to choose whether to produce them (requirement 22) and also be able to cancel this (requirement 23). The actual implementation of the visualisation with PyOpenGL performs very quickly, producing frames from the simulation data in real time as the user plays it back, so these requirements were no longer necessary and were removed. There were also requirements that placed restrictions on the user when initialising a simulation: they were not supposed to be able to add more than 100 cell agents in total (requirement 7), and they were not supposed to be able to initialise an environment too small to fit the initial cells within in without overlap (requirement 8). It was decided during development that these restrictions were against the genericity of the application, and that since the user can cancel models and the system does not crash if these restrictions are lifted, these requirements were also not necessary.

5.3 Testing

Testing is a crucial part of any project to ensure the efficacy of the system in relation to its requirements. For this project, the functional tests designed in the Requirements and Analysis stage were carried out manually to check that expected results were produced and acceptance criteria had been met. The test results are shown in Tables 5.1 to 5.7. The tests for desirable and optional requirements that were not implemented are removed, and where changes have been made to the test inputs or expected results from the Requirements and Analysis chapter due to changes in the requirements, an asterisk is shown with the requirement ID. To refer back to the original test wording, see Tables A.1 - A.8 in Appendix A. As the tables show, the application performed well in all of the tests that were carried out. There were no unexpected results and nothing caused the program to crash.

Figure 5.5 shows the visualisations produced from the physical model tests, and demonstrates that the overlap correction algorithm performs as expected, separating cells in the correct directions and to the correct degree. The graph produced in the graphical output test is displayed in Figure 5.6, and it shows that the cell proliferation works correctly and is graphed as expected, with the cell population doubling roughly every 24 iterations (the average cell cycle length for the GenericCell class).

Cell Agent Tests				
ID	Requirement	Inputs	Expected Result	Passed
1	Seed new cell	1 cell in spacious environment with high constant oxygen	Initial cell's parameters are correct, and when it divides, the new cell's parameters are correct	Yes
2	Cell cycle	1 cell in spacious environment with high oxygen	Cell grows to double volume and divides	Yes
		1 cell in spacious environment with insufficient oxygen to divide	Cell grows to double volume but does not divide	Yes
		1 cell in small environment (only big enough for it to grow) with high oxygen	Cell grows to double volume but does not divide	Yes
		Many cells in small environment with high oxygen	Population does not exceed the max that could fit in the environment	Yes
3	Cell migration	1 cell in small environment (room to move)	Cell follows random walk and does not leave the environment boundary	Yes
		1 cell in small environment (no room to move)	Cell does not move	Yes
		Many cells in small environment	Cells that are contact inhibited do not move	Yes
4	Cell death from hypoxia	1 cell in environment with low oxygen	Cell is removed from environment after first iteration and has a "dead" flag set to true	Yes
5	Cell death from age	1 cell in spacious environment with high oxygen	After cell has divided a number of times equal to its lifespan, it is removed from the environment and has a "dead" flag set to true	Yes

Table 5.1: Tests performed for the cell agent requirements.

Physical Model Tests				
ID	Requirement	Inputs	Expected Result	Passed
6	Resolve cell overlap forces	2 cells in spacious environment with overlapping initial positions (turn off cell migration and cell cycles)	Cells separate along the direction that they were overlapping after the first iteration	Yes
		3 cells in spacious environment with overlapping initial positions in triangular formation (turn off cell migration and cell cycles)	Cells spread to separate in triangular formation after the first iteration	Yes

Table 5.2: Tests performed for the physical model requirements.

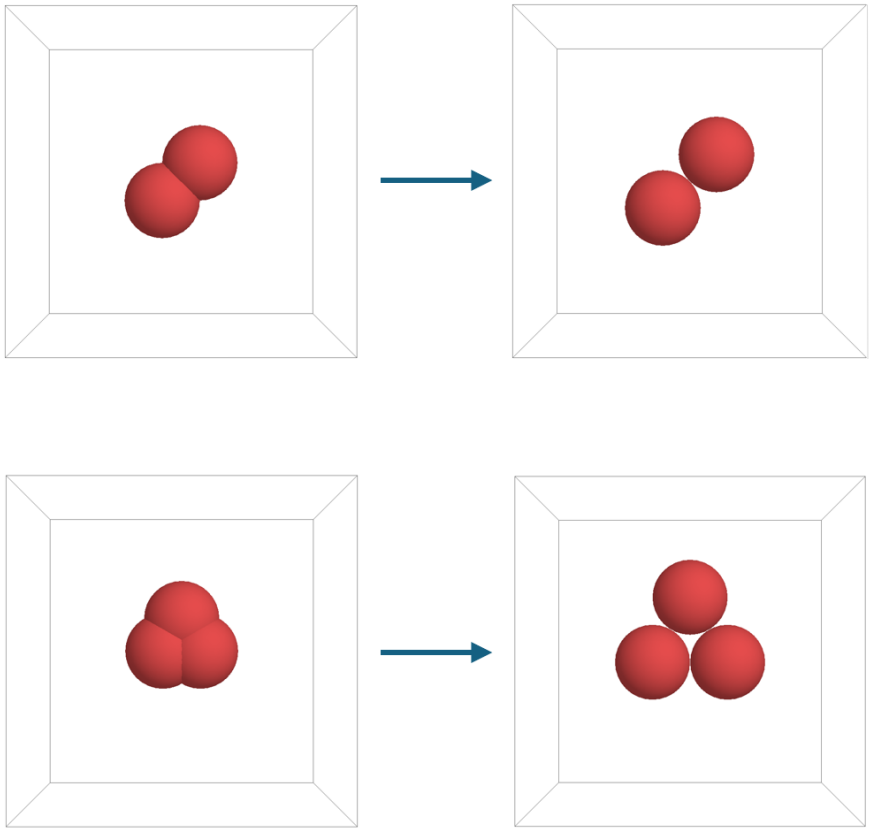


Figure 5.5: Physical model cell overlap test results (screenshotted from visualisation).

Model Initialisation Tests				
ID	Requirement	Inputs	Expected Result	Passed
7	Number of initial cells	Add Generic cells to the simulation	User has the option to add 1-100 Generic cells	Yes
		Add Generic and Cancerous cells to the simulation, and add 50 Generic cells	User has the option to add 1-50 Cancerous cells	N/A
8	Environment size	Add 1 Generic cell	User has the option to set the environment side length to 1 Generic cell diameter or more	Yes
		Add 20 Generic cells	User has the option to set the environment side length to 5 Generic cell diameters or more	N/A
9	Random cell distribution	Add many cells to a spacious environment	At the first iteration, cells have random non-overlapping positions	Yes
10	Clustered cell distribution	N/A	N/A	N/A
11	Uniform oxygen levels	Set oxygen concentration to 50%	Oxygen concentration is 50% throughout the environment	Yes
12	Spatially varied oxygen levels	N/A	N/A	N/A
13	Generic cell type	None	User has option to add Generic cells to the simulation	Yes
14	Cancerous cell type	None	User has option to add Cancerous cells to the simulation	Yes
15	Add cell types from GUI	N/A	N/A	N/A

Table 5.3: Tests performed for the user initialisation requirements.

Model Function Tests				
ID	Requirement	Inputs	Expected Result	Passed
16	Save model	Add 1 cell to environment and run for 50 iterations	.csv file is saved containing 50 iterations of necessary data about any cell agents in the simulation	Yes
17	Cancel model	Add many cells to the environment and run for many iterations, while model is running click "Cancel"	User is returned to initialisation with settings still in place	Yes
18	Extend model	N/A	N/A	N/A

Table 5.4: Tests performed for the model function requirements.

Graphical Output Tests				
ID	Requirement	Inputs	Expected Result	Passed
19	Graph N against time	Add 20 Generic cells to a spacious environment with high oxygen and run for 100 iterations	After the model has run, a graph of the cell agent population appears on the GUI, starting at N=20 and following the expected curve based on average Generic cell proliferation rates	Yes
20	Graph reproduction and death	N/A	N/A	N/A
21	Graph cell phases	N/A	N/A	N/A

Table 5.5: Tests performed for the graphical output requirements.

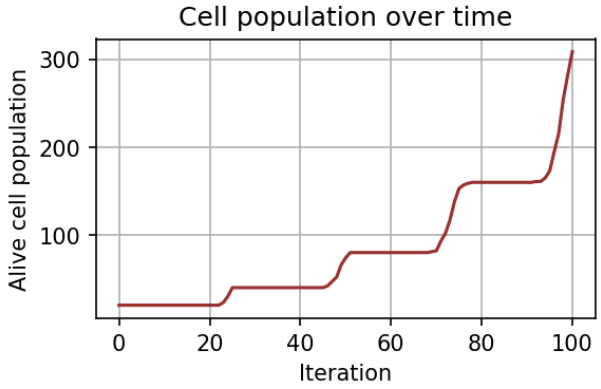


Figure 5.6: Graphical output test results showing cell population doubling every 24 iterations (on average).

Model Visualisation Tests (1)				
ID	Requirement	Inputs	Expected Result	Passed
22 *	Produce visualisation	Add 1 Generic cell in a small environment with high oxygen and run for 50 iterations	A visualisation is produced when the model finishes running with its view appropriately scaled to the environment size, and the cell/s can be seen growing, moving through their correct positions, and dividing	Yes
23	Cancel visualisation	N/A	N/A	N/A
24	Cell colouring	Add 1 Generic cell to an environment with low oxygen (too low for proliferation but not low enough to cause death) and run for 50 iterations	After the length of the Generic cell G1 phase, the cell transitions to G0 and changes colour in the visualisation	Yes
25	Highlight cells	N/A	N/A	N/A
26	Visualise oxygen	N/A	N/A	N/A

Table 5.6: Tests performed for the 3D visualisation requirements (1).

Model Visualisation Tests (2)				
ID	Requirement	Inputs	Expected Result	Passed
27 *	Playback controls	Run a simulation, play the visualisation, set the frame rate to 30fps, and then 5fps	The visualisation shows 30 model iterations every second, and then 5 iterations per second	Yes
		Run a simulation and press the pause and play buttons	When the user presses pause, the visualisation pauses on the current iteration, and when they press play, it continues from the current iteration	Yes
		Run a simulation, and press the step forward and step back buttons	When the user presses step forward, the visualisation advances one iteration, when they press step back, it goes back one iteration	Yes
		Run a simulation, and scrub through the iterations with the slider	When the user releases the slider the visualisation jumps to the selected iteration	Yes
28	Fixed camera angle	Run a simulation and produce visualisation	The view has a fixed camera angle positioned appropriately to view the full environment	Yes
29	Multiple camera angles	N/A	N/A	N/A
30	Full camera orbit	N/A	N/A	N/A
31 *	Export video	Run a simulation, click ”Export video” and choose a location to save the video	A video of the visualisation (with a fixed camera angle) is saved in the chosen location	Yes
32	Record Video	N/A	N/A	N/A

Table 5.7: Tests performed for the 3D visualisation requirements (2).

Chapter 6

Results

6.1 Outcome

This project has resulted in an extensible system that models generic cell behaviours and provides useful graphical output and visualisation to the user. This opens up the possibilities for faster research and development of cell simulation software in the future, as the model can be built upon without researchers having to develop their systems from scratch.

The model captures the cell cycle phases, cell migration, and cell death from both hypoxia and old age. It has a physical model which successfully minimises overlap between cell agents. It has a 3D environment which can have an oxygen layer added to it that the cell agents can access with a uniform substance level set by the user. The user can add generic and/or cancerous cells to a simulation, choose how many of them to initialise, and they will be randomly distributed throughout the environment. They can set the size of the environment, the number of iterations to run the model for, and a random seed. The user can run and cancel a model, and see a 3D visualisation of their simulation when it has run. The visualisation has useful playback controls for navigating the simulation iterations. The alive cell population numbers for each cell type added to a simulation can be seen graphed on the GUI. Finally, the user can export either the raw data from the most recently run simulation, or a video of the visualisation if one has been produced.

6.2 Evaluation of the System

6.2.1 Meeting of Functional Requirements

In the Requirements and Analysis stage of this report, 32 requirements were established (Tables 3.1 - 3.6) as a guide for the features that the system would need and their individual priorities. Referring back to these reveals to what extent the application has met the initial objectives of the project. Tables 6.1 - 6.6 extend the original requirements tables to show which have been achieved. The wording for requirements 7, 8 and 22 (marked with asterisks) have been altered slightly, and requirement 23 has been removed, reflecting the changes made

to the requirements during the development stage that were described in Chapter 5.

Cell Agents			
ID	Requirement	Description	Passed
1	Seed new cell	Cells seeded in the environment are initialised with appropriate parameters according to their cell type and are tracked by the model	Yes
2	Cell cycle	Cells follow the cell cycle (with phases G1, S/G2, G0 and M), where they double in volume and divide if there is sufficient space and oxygen	Yes
3	Cell migration	As long as cells are not contact inhibited, they follow a random walk, with maximum distance at each iteration equal to their seed radius, and they cannot move outside the environment boundary	Yes
4	Cell death from hypoxia	At each iteration, if the oxygen at a cell's position is below the cell type's threshold, the cell is removed from the environment and tracked as being in a dead state	Yes
5	Cell death from age	Each time a cell divides, if its cell type ages, its age is incremented, and if it ages past its lifespan, it is removed from the environment and tracked as being in a dead state	Yes

Table 6.1: Showing which functional requirements relating to the cell agents have been met.

Physical Model			
ID	Requirement	Description	Passed
6	Resolve cell overlap forces	At each iteration, the physical model resolves cell overlap as best as possible by iteratively resolving forces between overlapping cells to spread them apart	Yes

Table 6.2: Showing which functional requirements relating to the physical model have been met.

Model Initialisation			
ID	Requirement	Description	Passed
7 *	Number of initial cells	The user can add cell types to the environment and set the number of initial cells of each type, with up to a 100 of each type	Yes
8 *	Environment size	The user can set the side length (in micrometres) of the environment (a 3D cube)	Yes
9	Random cell distribution	Cells added to the environment are initialised with random positions	Yes
10	Clustered cell distribution	When the user adds cells to the environment, they can optionally choose to place them in a cluster at a chosen location	No
11	Uniform oxygen levels	The user can set a uniform value for the oxygen concentration throughout the environment	Yes
12	Spatially varied oxygen levels	The user can set the detail level of the substance grid (in number of grid cells per side where min = 1 and max = grid side length / max cell agent diameter), and then choose grid cells to be the centres of oxygen sources, set their concentration, their radius of influence on other grid cells, and how quickly the concentration declines for grid cells within the radius	No
13	Generic cell type	A generic cell type representing normal cells is ready made for the user to add to the model	Yes
14	Cancerous cell type	A second cell type representing cancerous cells is ready made for the user to add to the model	Yes
15	Add cell types from GUI	The user can add new cell types to the model, setting all of the necessary parameters directly from the GUI, rather than needing to implement them in code	No

Table 6.3: Showing which functional requirements relating to the initialisation of a model by the user have been met.

Model Functions			
ID	Requirement	Description	Passed
16	Save model	At each iteration, information about the agents and environment necessary to reconstruct the simulation are saved in a .csv file, which can be exported by the user when a simulation has run, and is overwritten with each new simulation	Yes
17	Cancel model	The user can cancel a model while it is computing, and this takes them back to model initialisation	Yes
18	Extend model	After a simulation has run, the user can choose to extend it for a chosen number of iterations, and it then continues modelling from where it left off	No

Table 6.4: Showing which functional requirements relating to model functions have been met.

Graphical Output			
ID	Requirement	Description	Passed
19	Graph N against time	The number of cell agents in the environment at each iteration is graphed on the GUI	Yes
20	Graph reproduction and death	The reproduction and death counts at each iteration are graphed on the GUI	No
21	Graph cell phases	The numbers of cells in each cell cycle phase at each iteration are graphed on the GUI	No

Table 6.5: Showing which functional requirements relating to the graphical output on the GUI have been met.

Model Visualisation			
ID	Requirement	Description	Passed
22 *	Produce visualisation	After the user has run a model, a 3D visualisation is produced from the agent positions and morphologies saved at each iteration	Yes
23	Cancel visualisation	The user can cancel producing a visualisation, and this returns them to the GUI where their model has been run	Removed
24	Cell colouring	Quiescent cells (in G0 phase) are visualised in a different colour to other cells	Yes
25	Highlight cells	Before the visualisation is produced, the user can choose cells to be highlighted throughout it	No
26	Visualise oxygen	The user can turn on/off an overlay of the substance grid which colours the grid cells in a gradient according to their oxygen concentration (with some transparency)	No
27	Playback controls	The visualisation has controls for frame rate, pause, play, step forward one iteration, step back one iteration, and a slider for scrubbing through the visualisation	Yes
28	Fixed camera angle	The visualisation shows a fixed angle of the 3D environment and agents within it	Yes
29	Multiple camera angles	The visualisation has multiple camera angles that can be switched between while the user is playing back the visualisation	No
30	Full camera orbit	The user can orbit the camera around the environment while playing back the visualisation	No
31	Export video	The user can export the visualisation as a video with a fixed camera angle	Yes
32	Record video	The user can record the playback of a simulation, controlling the camera angle throughout, and export this as a video	No

Table 6.6: Showing which functional requirements relating to the 3D visualisation of a simulation have been met.

Tables 6.1 - 6.6 demonstrate that around two thirds of the requirements have been achieved, with all of the mandatory requirements completed and one desirable requirement included (adding a cancerous cell type). Given that all mandatory requirements are achieved,

the system has the features necessary to meet the main project aims. The remaining features would not be difficult to implement from a technical perspective, but were excluded due to time constraints during the project's development phase. The reason that the cancerous cell type requirement was chosen over other desirable requirements was that it was simple to implement, and achieving it would allow for demonstration of a key potential use case of the system (i.e., research experiments that compare different cell types), as well as demonstrating the extensibility of the software. With more successful time management, or a longer development phase, the requirements could have been implemented in full.

Some initial experiments were conducted to see what the outcome would have been had requirement 10 (clustered cell distribution) been completed. This was partially to test the performance of the system under higher cell densities, and also to see whether the cells would grow in a pattern typical of real cells, to evaluate the realism of the model. Only one line of code which initialises the positions of the cell agents had to be changed to form the initial cell cluster, which shows that this would be an easy feature to add, with the only major change needed being to give the user the option to choose where to place the cluster from the GUI. The experiments conducted involved adding 20 Generic cells to the centre of a 500μm wide environment, allowing the physical solver to separate the cells into the initial cluster, and running the simulation for 100 iterations, both with and without cell migration enabled. In the experiment without cell migration, the inside of the cell cluster could be seen becoming contact inhibited and quiescent whilst the outer part of the cluster continued to divide outwards, which is exactly the expected behaviour. Both experiments reached around 300 cell population. The experiment with migration took around 1 minute to run, and the experiment without migration, which had higher cell density and more overlap to solve, took around 2 minutes to run. Whilst this is slower than previous results achieved from random cell distributions, it is still reasonably quick and is promising for the future of the model, which could be capable of running larger simulations, especially with some further optimisation. It is possible that a different physical model implementation specialised for solving cell clusters would be appropriate, as the PhysicalModelWithLocals algorithm is more suited to solving situations where the cells are distributed throughout the environment, and likely has some inefficiencies trying to solve a situation where all of the cells are clumped in one location. Videos produced from the visualisations of these experiments can be found at this link: `https://drive.google.com/drive/folders/1u7n8q-tmt8JRICveTuuzzr4uVKd5TC4S?usp=drive_link`

6.2.2 Meeting of Non-Functional Requirements

The following non-functional requirements were established in the Requirements and Analysis stage of this report:

1. The model must be written in Python
2. The model must be written using object-oriented programming (OOP), and be highly modular and extensible

3. The model must be written efficiently, and handle the computation of hundreds of cell agents in a reasonable time

4. The GUI should be responsive and intuitive

The model is written in Python using OOP, and has been designed with modularity in mind. The software has structures in place that facilitate adding new cell types and environment substance layers, and it is easy to override existing functions to provide new implementations if this is required. However, a shortcoming in the extensibility of the software is in the GUI design. As the author did not have experience with Python GUI development or the libraries that were used, a lack of a good understanding in this area meant that it was challenging to try to structure the GUI in such a way that it would be easy to add new features. As a result, the existing GUI implementation has quite a rigid structure that is not simple to extend. It would be recommended for the GUI to be reengineered to further the modularity of the system.

The model has been written with efficiency in mind, and profiling results have shown that it can handle hundreds of cell agents in reasonable time frames. The PhysicalModelWithLocals solver demonstrates significant performance enhancements over the PhysicalModel algorithm, showing that optimisation efforts were successful. The PhysicalModelWithLocals algorithm does not slow down significantly with increased cell populations, suggesting it could handle a very high agent count, but performance is more affected by cell density, and improvements could still be made in this area.

The GUI is responsive to user inputs and is constructed from standard GUI elements, so it is intuitive to use. Overall, the non-functional requirements have largely been achieved.

6.2.3 Overall Evaluation

As a whole, the application addresses the six key project objectives set out in the Requirements and Analysis stage of this report: (i) it models the essential generic behaviours of cells; (ii) it implements a physical model that resolves cell overlap; (iii) the user can initialise a simulation, controlling the initial state of the environment and cell agents; (iv) the simulation data can be exported; (v) cell agent statistics are graphed on the GUI; and (vi) the simulations are visualised in 3D. Whilst there are shortcomings with the GUI implementation, and the performance of the system can slow down with high cell densities, with all mandatory functional requirements met and the non-functional requirements achieved to a high degree, the project has largely been a success. It has the potential to improve the experience of researchers in the field of biological cell simulation, by reducing the amount of work required to model new multicellular populations.

6.3 Further Work

Whilst the system created for this project has met the main project aims, some changes and extra features could improve its utility further.

The main area where the project could be improved would be in implementing the rest of the desirable and optional requirements to give the application a richer feature set. The most impactful of these additions would likely be requirement 12, which involves implementing spatially varied substance layers and allowing the user to specify the substance distribution from the GUI. This is because it would greatly increase the scope of investigations that could be made in the software, as researchers would be able to explore how cells react in varied environments with different concentrations of substances. Although not a requirement considered in this project thus far, another extension to the model could be implementing an environment that varies through time, for instance that models substance diffusion, and also allowing the cell agents to impact the substance levels in the environment. This again would broaden the scope of experiments that could be carried out in the software. The clustered cell distribution requirement discussed earlier would also have this impact, and as previously described, it would be a simple feature to add.

Various other requirements would provide quality of life improvements to the user experience in the application. Adding more graphs of different cell population statistics, such as the reproduction and death rates and the numbers of cells in different cycle phases, would facilitate understanding of the simulations and provide useful data for research. Allowing the user to highlight specific cells in the visualisation, allowing them to move the camera around the environment, and allowing them to extend a simulation further after running a model to see more iterations would also improve the user experience.

The additional project objectives that were considered for this project, but for which there was not time to explore, would be good areas to consider for future work. With access to real experimental data, the model could be extended to try and replicate it, in order to test how easily and effectively the model can be specialised, and how successfully it could be tuned to model real data. Another area of focus could be on optimising the model to make it viable for running larger experiments. The physical model, taking up around 95% of the computation time for simulations, would likely benefit from parallelisation, which could be done through utilisation of parallel processing modules in Python. Alternatively, since Python is not the best suited for parallel processing, the whole system could be translated into Flame GPU2, which is a highly optimised, parallelised environment. If successful, this would make significantly more computationally intensive experiments possible in reasonable time frames, allowing for research of larger multicellular populations.

Chapter 7

Conclusion

The goal of this project was to make an efficient, modular, 3D agent-based cell model encoding generic cell behaviours to provide a basis for future research in the field of biological cell simulation, allowing for faster development of future models.

The underlying biology behind the cell behaviours that needed to be modelled (migration, cell cycle progression, and death) was investigated in Chapter 2, which provided the knowledge necessary to design algorithms for these behaviours. Different computational modelling paradigms were also explored here, providing evidence as to why agent-based modelling is particularly suited to this project. Existing agent-based cell models and agent-based frameworks were investigated to provide understanding of the area and inspiration for the design of the system.

During the design stage, algorithms for the cell cycle phases were designed based on those seen in the Epitheliome project [20] in the literature review. A GUI mockup was also made with elements chosen to satisfy the requirements that were set out for a successful system in the Requirements and Analysis chapter.

The software was written in Python 3.11, and utilised the PySide6 and PyOpenGL libraries for the GUI and 3D visualisation. These were chosen because of their efficiency and versatility, and because the base code required for the GUI could be generated from a visual editor called Qt Designer which increased the rate of development. The final application closely resembled the mockup made in the design phase, as no major changes to the elements were required.

Largely, this project was a success. Manual testing showed that the acceptance criteria of all mandatory requirements have been met with no unexpected results, and profiling the system demonstrated that it has reasonably good performance even for large cell populations. The 6 key project objectives have been achieved: (i) essential generic cell behaviours have been implemented; (ii) a physical model successfully minimises overlap between the cell agents; (iii) the user can initialise simulations from the GUI; (iv) the user can export the simulation data; (v) graphical output of cell population data is produced on the GUI; and (vi) a 3D visualisation of the cells in the environment is produced for the user to view. The system uses OOP and is structured to be modular and extensible, meaning it is easy to add new cell

types and new substances to the environment. With all of this, the overall project aim has been achieved.

For further work, the model could be extended in numerous ways. The most impactful of these would be in the environment implementation. If the substances in the environment could be spatially and even temporally varied, it would greatly increase the scope of experiments that could be run with the model. Other ideas for future work include attempting to replicate real world data in the model to evaluate its realism and extensibility, and further optimising the model with parallelisation.

In summary, an extensible agent-based 3D cell model has been created in Python that encodes generic cell behaviours. The model can relieve some of the development burden for researchers wanting to simulate multicellular populations, thus achieving the main aim of this project.

Bibliography

[1] C. Macal and M. North, "Agent-based modeling and simulation," in *Proceedings of the 2009 Winter Simulation Conference (WSC)*, Austin, TX, USA, 2009, pp. 86–98. DOI: https://doi.org/10.1109/WSC.2009.5429318.

[2] M. Soheilypour and M. Mofrad, "Agent-based modeling in molecular systems biology," *BioEssays*, vol. 40, no. 7, Jun. 2018. DOI: https://doi.org/10.1002/bies.201800020.

[3] B. Alberts, A. Johnson, J. Lewis, D. Morgan, and M. Raff, *Molecular Biology of the Cell*, 6th. Oxford: Taylor & Francis Group, 2014, ISBN: 9780815344322. [Online]. Available: https://ebookcentral.proquest.com/lib/sheffield/detail.action?docID=5320520.

[4] G. M. Cooper, *The Cell: A Molecular Approach*, 2nd. Sunderland (MA): Sinauer Associates, 2000, ISBN: 0878931066. [Online]. Available: https://www.ncbi.nlm.nih.gov/books/NBK9876/.

[5] A. Saraste and K. Pulkki, "Morpholgic and biochemical hallmarks of apoptosis," *Cardiovascular Research*, vol. 45, no. 3, pp. 528–537, Feb. 2000. DOI: https://doi.org/10.1016/S0008-6363(99)00384-3.

[6] F. Milde, G. Tauriello, H. Haberkern, and P. Koumoutsakos, "Sem++: A particle model of cellular growth, signaling and migration," *Computational Particle Mechanics*, vol. 1, no. 2, pp. 211–227, Jun. 2014. DOI: https://doi.org/10.1007/s40571-014-0017-4.

[7] J. Arciero, Q. Mi, M. Branca, D. Hackam, and D. Swigon, "Continuum model of collective cell migration in wound healing and colony expansion," *Biopyhsical Journal*, vol. 100, no. 3, pp. 535–543, Feb. 2011. DOI: https://doi.org/10.1016/j.bpj.2010.11.083.

[8] N. Armstrong, K. Painter, and J. Sherratt, "A continuum approach to modelling cell-cell adhesion," *Journal of Theoretical Biology*, vol. 243, no. 1, pp. 98–113, Nov. 2006. DOI: https://doi.org/10.1016/j.jtbi.2006.05.030.

[9] H. Byrne and D. Drasdo, "Individual-based and continuum models of growing cell populations: A comparison," *Journal of Mathematical Biology*, vol. 58, pp. 657–687, 2008. DOI: https://doi.org/10.1007/s00285-008-0212-0.

[10] D. Walker and J. Southgate, "The virtual cell—a candidate co-ordinator for 'middle-out' modelling of biological systems," *Briefings in Bioinformatics*, vol. 10, no. 4, pp. 450–461, Jul. 2009. DOI: https://doi.org/10.1093/bib/bbp010.

[11] M. Cohen, B. Baum, and M. Miodownik, "The importance of structured noise in the generation of self-organizing tissue patterns through contact-mediated cell–cell signalling," *Journal of the Royal Society Interface*, vol. 8, no. 59, pp. 787–798, Nov. 2010, ISSN: 1742-5662. DOI: https://doi.org/10.1098/rsif.2010.0488.

[12] G. Cattaneo, A. Dennunzio, and F. Farina, "A full cellular automaton to simulate predator-prey systems," in *7th International Conference on Cellular Automata for Research and Industry*, S. El Yacoubi, B. Chopard, and S. Bandini, Eds., vol. 4173, Perpignan, France: Springer, Berlin, Heidelberg, Sep. 2006, pp. 446–451. DOI: https://doi.org/10.1007/11861201_52.

[13] D. Mallet and L. De Pillis, "A cellular automata model of tumor-immune system interactions," *Journal of Theoretical Biology*, vol. 239, no. 3, pp. 334–350, Apr. 2006. DOI: https://doi.org/10.1016/j.jtbi.2005.08.002.

[14] G. Ermentrout and L. Edelstein-Keshet, "Cellular automata approaches to biological modelling," *Journal of Theoretical Biology*, vol. 160, no. 1, pp. 97–133, Jan. 1993. DOI: https://doi.org/10.1006/jtbi.1993.1007.

[15] J. Moreira and A. Deutsch, "Cellular automaton models of tumor development: A critical review," *Advances in Complex Systems*, vol. 5, no. 2, pp. 101–337, 2002. DOI: https://doi.org/10.1142/S0219525902000572.

[16] M. Wooldridge and N. Jennings, "Intelligent agents: Theory and practice," *The Knowledge Engineering Review*, vol. 10, no. 2, pp. 115–152, Jun. 1995. DOI: https://doi.org/10.1017/S0269888900008122.

[17] J. Segovia-Juarez, S. Ganguli, and D. Kirschner, "Identifying control mechanisms of granuloma formation during m. tuberculosis infection using an agent-based model," *Journal of Theoretical Biology*, vol. 231, no. 3, pp. 357–376, Dec. 2004. DOI: https://doi.org/10.1016/j.jtbi.2004.06.031.

[18] A. Corti, M. Colombo, F. Migliavacca, J. Matas, S. Casarin, and C. Chiastra, "Multiscale computational modeling of vascular adaptation: A systems biology approach using agent-based models," *Bioengineering and Biotechnology*, vol. 9, 2021. DOI: https://doi.org/10.3389/fbioe.2021.744560.

[19] M. Pogson, R. Smallwood, E. Qwarnstrom, and M. Holcombe, "Formal agent-based modelling of intracellular chemical interactions," *Biosystems*, vol. 85, no. 1, pp. 37–45, Jul. 2006. DOI: https://doi.org/10.1016/j.biosystems.2006.02.00.

[20] D. Walker, J. Southgate, G. Hill, *et al.*, "The epitheliome: Agent-based modelling of the social behaviour of cells," *BioSystems*, vol. 76, no. 1-3, pp. 89–100, Feb. 2004. DOI: https://doi.org/10.1016/j.biosystems.2004.05.025.

[21] A. Matyjaszkiewicz, G. Fiore, F. Annunziata, *et al.*, "Bsim 2.0: An advanced agent-based cell simulator," *ACS Synthetic Biology*, vol. 6, no. 19, pp. 1969–1972, Jun. 2017. DOI: https://doi.org/10.1021/acssynbio.7b00121.

[22] T. Storck, C. Picioreanu, B. Virdis, and D. Batstone, "Variable cell morphology approach for individual-based modeling of microbial communities," *Biophysical Journal*, vol. 106, no. 9, pp. 2037–2048, May 2014. DOI: https://doi.org/10.1016/j.bpj.2014.03.015.

[23] A. Ghaffarizadeh, R. Heiland, S. Friedman, S. Mumenthaler, and P. Macklin, "Physicell: An open source physics-based cell simulator for 3-d multicellular systems," *PLOS Computational Biology*, vol. 14, no. 2, Feb. 2018. DOI: https://doi.org/10.1371/journal.pcbi.1005991.

[24] A. Ghaffarizadeh, S. Friedman, and P. Macklin, "Biofvm: An efficient, parallelized diffusive transport solver for 3-d biological simulations," *Bioinformatics*, vol. 32, no. 8, pp. 1256–1258, Apr. 2015. DOI: https://doi.org/10.1093/bioinformatics/btv730.

[25] S. Abar, G. K. Theodoropoulos, P. Lemarinier, and G. M. P. O'Hare, "Agent based modelling and simulation tools: A review of the state-of-art software," *Computer Science Review*, vol. 24, pp. 13–33, May 2017. DOI: https://doi.org/10.1016/j.cosrev.2017.03.001.

[26] J. Kazil, D. Masad, and A. Crooks, "Utilizing python for agent-based modeling: The mesa framework," in *International Conference on Social Computing, Behavioral-Cultural Modeling & Prediction and Behavior Representation in Modeling and Simulation*, 2020, pp. 308–317. DOI: Kazil, J., Masad, D., & Crooks, A. (2020). Utilizing Python for Agent-Based Modeling: The Mesa Framework. Social, Cultural, and Behavioral Modeling, 308317. https://doi.org/10.1007/978-3-030-61255-9_30.

[27] A. Antelmi, G. Cordasco, G. D'Ambrosio, D. De Vinco, and C. Spagnuolo, "Experimenting with agent-based model simulation tools," *Applied Sciences*, vol. 13, no. 1, 2023, ISSN: 2076-3417. DOI: https://doi.org/10.3390/app13010013.

[28] J. Foramitti, "Agentpy: A package for agent-based modeling in python," *The Journal of Open Source Software*, vol. 6, no. 62, Jun. 2021. DOI: https://doi.org/10.21105/joss.03065.

[29] N. Collier, J. Ozik, and E. Tatara, "Experiences in developing a distributed agent-based modelling toolkit with python," in *2020 IEEE/ACM 9th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, GA, USA: IEEE, Nov. 2020, pp. 1–12. DOI: https://doi.org/10.1109/PyHPC51966.2020.00006..

[30] P. Cooley and E. Solano, "Agent-based model (abm) validation considerations," in *Proceedings of the Third International Conference on Advances in System Simulation (SIMUL 2011)*, 2011, pp. 134–139.

Appendices

Appendix A

Designed Test Cases

Cell Agent Tests (1)			
ID	Requirement	Inputs	Expected Result
1	Seed new cell	1 cell in spacious environment with high constant oxygen	Initial cell's parameters are correct, and when it divides, the new cell's parameters are correct
2	Cell cycle	1 cell in spacious environment with high oxygen	Cell grows to double volume and divides
		1 cell in spacious environment with insufficient oxygen to divide	Cell grows to double volume but does not divide
		1 cell in small environment (only big enough for it to grow) with high oxygen	Cell grows to double volume but does not divide
		Many cells in small environment with high oxygen	Population does not exceed the max that could fit in the environment
3	Cell migration	1 cell in small environment (room to move)	Cell follows random walk and does not leave the environment boundary
		1 cell in small environment (no room to move)	Cell does not move
		Many cells in small environment	Cells that are contact inhibited do not move

Table A.1: Tests designed for the cell agent requirements (1).

Cell Agent Tests (2)			
ID	Requirement	Inputs	Expected Result
4	Cell death from hypoxia	1 cell in environment with low oxygen	Cell is removed from environment after first iteration and has a "dead" flag set to true
5	Cell death from age	1 cell in spacious environment with high oxygen	After cell has divided a number of times equal to its lifespan, it is removed from the environment and has a "dead" flag set to true

Table A.2: Tests designed for the cell agent requirements (2).

Physical Model Tests			
ID	Requirement	Inputs	Expected Result
6	Resolve cell overlap forces	2 cells in spacious environment with overlapping initial positions (turn off cell migration and cell cycles)	Cells separate along the direction that they were overlapping after the first iteration
		3 cells in spacious environment with overlapping initial positions in triangular formation (turn off cell migration and cell cycles)	Cells spread to separate in triangular formation after the first iteration

Table A.3: Tests designed for the physical model requirements.

Model Initialisation Tests			
ID	Requirement	Inputs	Expected Result
7	Number of initial cells	Add Generic cells to the simulation	User has the option to add 1-100 Generic cells
		Add Generic and Cancerous cells to the simulation, and add 50 Generic cells	User has the option to add 1-50 Cancerous cells
8	Environment size	Add 1 Generic cell	User has the option to set the environment side length to 1 Generic cell diameter or more
		Add 20 Generic cells	User has the option to set the environment side length to 5 Generic cell diameters or more
9	Random cell distribution	Add many cells to a spacious environment	At the first iteration, cells have random non-overlapping positions
10	Clustered cell distribution	Add 40 Generic cells and 20 Cancerous cells	User has the option to place each cell population in a cluster at a chosen location
11	Uniform oxygen levels	Set oxygen concentration to 50%	Oxygen concentration is 50% throughout the environment
12	Spatially varied oxygen levels	Set detail level to 5, add oxygen source to centre grid cell, set the source concentration to 50%, set source radius to 2, and set concentration decline to 20%	Centre grid cell has 50% oxygen concentration, surrounding 8 grid cells have 30% concentration and outer grid cells have 0% concentration
13	Generic cell type	None	User has option to add Generic cells to the simulation
14	Cancerous cell type	None	User has option to add Cancerous cells to the simulation
15	Add cell types from GUI	None	User has option to add a new cell type to the simulation, specify which existing cell type it extends behaviours from, and enter values for all necessary parameters

Table A.4: Tests designed for the user initialisation requirements.

Model Function Tests			
ID	Requirement	Inputs	Expected Result
16	Save model	Add 1 cell to environment and run for 50 iterations	.csv file is saved containing 50 iterations of necessary data about any cell agents in the simulation
17	Cancel model	Add many cells to the environment and run for many iterations, while model is running click "Cancel"	User is returned to initialisation with settings still in place
18	Extend model	Add 1 cell to environment and run for 10 iterations, after model has run, extend model for 20 iterations	Model runs for 20 more iterations continuing from its state at iteration 10

Table A.5: Tests designed for the model function requirements.

Graphical Output Tests			
ID	Requirement	Inputs	Expected Result
19	Graph N against time	Add 20 Generic cells to a spacious environment with high oxygen and run for 100 iterations	After the model has run, a graph of the cell agent population appears on the GUI, starting at N=20 and following the expected curve based on average Generic cell proliferation rates
20	Graph reproduction and death	Add 20 Generic cells to a spacious environment with areas of high and low oxygen and run for 100 iterations	After the model has run, a graph appears on the GUI showing the correct reproduction and death counts at each iteration
21	Graph cell phases	Add 1 Generic cell to a spacious environment with high oxygen and run for 50 iterations	After the model has run, a graph appears on the GUI with the counts of cells in each cell cycle phase, starting with the single initial cell moving through its phases at the correct times

Table A.6: Tests designed for the graphical output requirements.

Model Visualisation Tests (1)			
ID	Requirement	Inputs	Expected Result
22	Produce visualisation	Add 1 Generic cell in a small environment with high oxygen and run for 50 iterations, then click "Produce visualisation"	The visualisation view is appropriately scaled to the environment size and the cell/s can be seen growing, moving through their correct positions, and dividing
23	Cancel visualisation	Add many cells to a spacious environment with high oxygen and run for many iterations, then click "Produce visualisation", then click "Cancel"	The user is returned to the GUI at the point where their model has run
24	Cell colouring	Add 1 Generic cell to an environment with low oxygen (too low for proliferation but not low enough to cause death) and run for 50 iterations	After the length of the Generic cell G1 phase, the cell transitions to G0 and changes colour in the visualisation
25	Highlight cells	Add many cells to a spacious environment with high oxygen, select 2 cell ids, and run for 50 iterations	The cells with the chosen ids are visually distinct in the visualisation, and remain highlighted throughout the iterations
26	Visualise oxygen	Initialise oxygen concentration as specified in test 13, add cells to the environment and run for 50 iterations, then produce visualisation, and turn oxygen overlay on and off	When oxygen overlay is on, the grid cells are coloured (with some transparency) according to their oxygen concentration (centre cell has strongest colour, its surrounding cells are half as strong, and there is no colour in the outer grid cells), and cell agents can still be seen under the overlay

Table A.7: Tests designed for the 3D visualisation requirements (1).

Model Visualisation Tests (2)			
ID	Requirement	Inputs	Expected Result
27	Playback controls	Run a simulation, produce visualisation, and set the frame rate to 30fps, and then 5fps	The visualisation shows 30 model iterations every second, and then 5 iterations per second
		Run a simulation, produce visualisation, and press the pause and play buttons	When the user presses pause, the visualisation pauses on the current iteration, and when they press play, it continues from the current iteration
		Run a simulation, produce visualisation, pause, and press the step forward and step back buttons	When the user presses step forward, the visualisation advances one iteration, when they press step back, it goes back one iteration
		Run a simulation, produce visualisation, pause, and scrub through the iterations with the slider	When the user releases the slider the visualisation jumps to the selected iteration
28	Fixed camera angle	Run a simulation and produce visualisation	The view has a fixed camera angle positioned appropriately to view the full environment
29	Multiple camera angles	Run a simulation and produce visualisation, then click through camera angles while visualisation is playing	When the user clicks to a different camera angle, the camera moves and rotates accordingly
30	Full camera orbit	Run a simulation, produce visualisation, then use WASD keys	The camera moves accordingly in an orbit around the environment, staying at a fixed distance from the centre and always pointing towards it
31	Export video	Run a simulation, produce visualisation, click "Export video" and choose a location to save the video	A video of the visualisation (with a fixed camera angle) is saved in the chosen location
32	Record video	Run a simulation, produce visualisation, click "Record", use the visualisation controls, and click "Stop"	The user's manipulation of the visualisation is recorded and can then be exported

Table A.8: Tests designed for the 3D visualisation requirements (2).

Appendix B

Application Screenshots

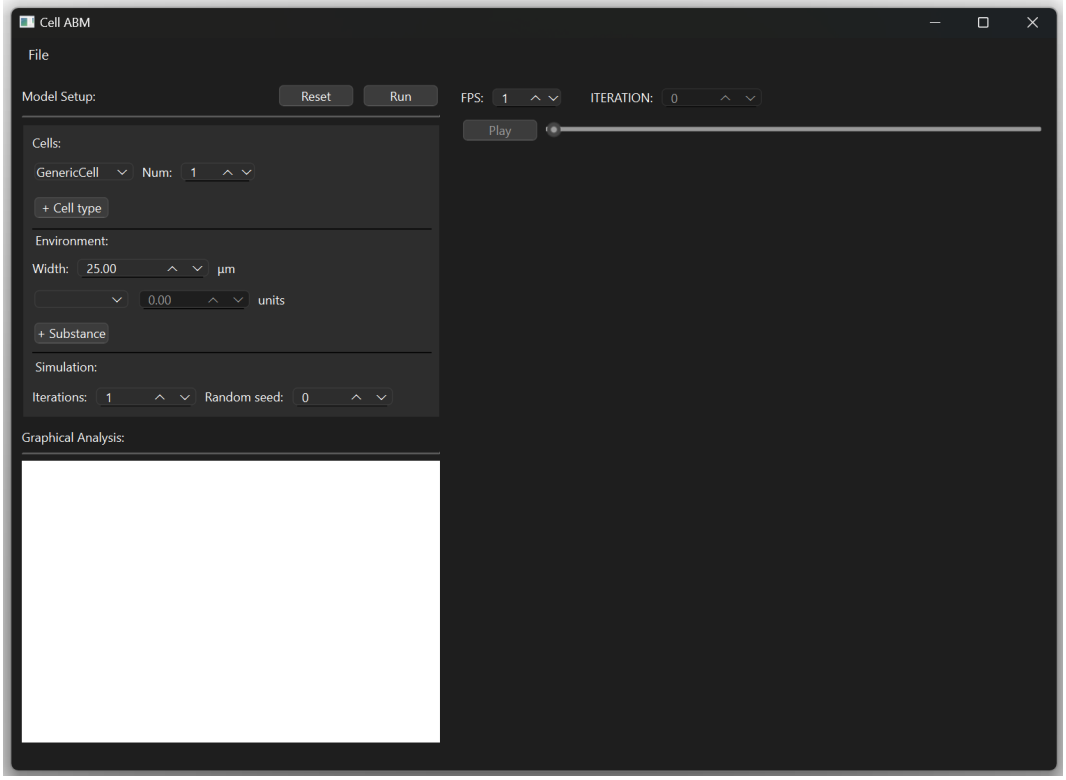


Figure B.1: Initial GUI setup on application launch.

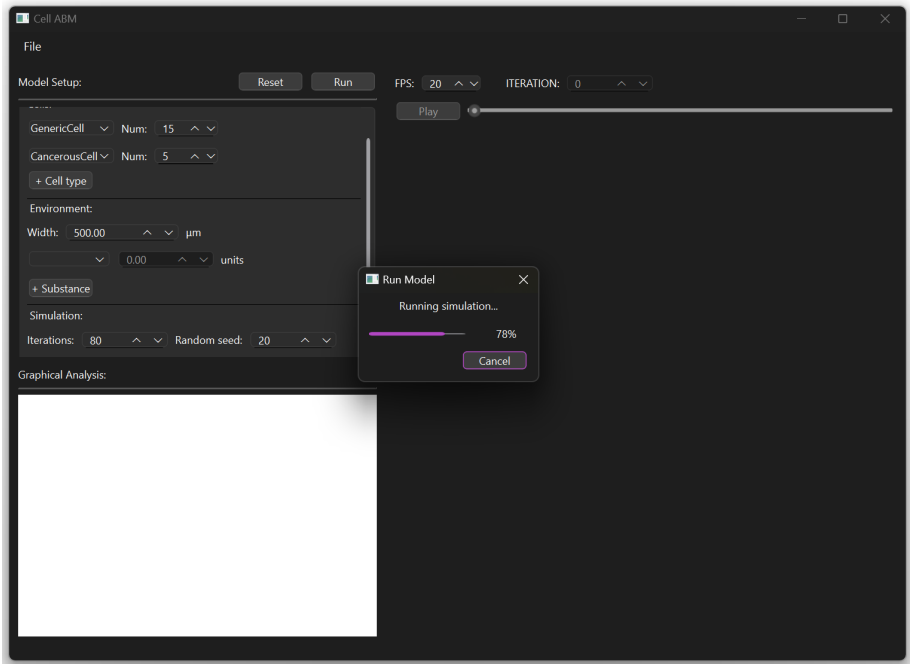


Figure B.2: Run model progress dialog appears when user runs a simulation.

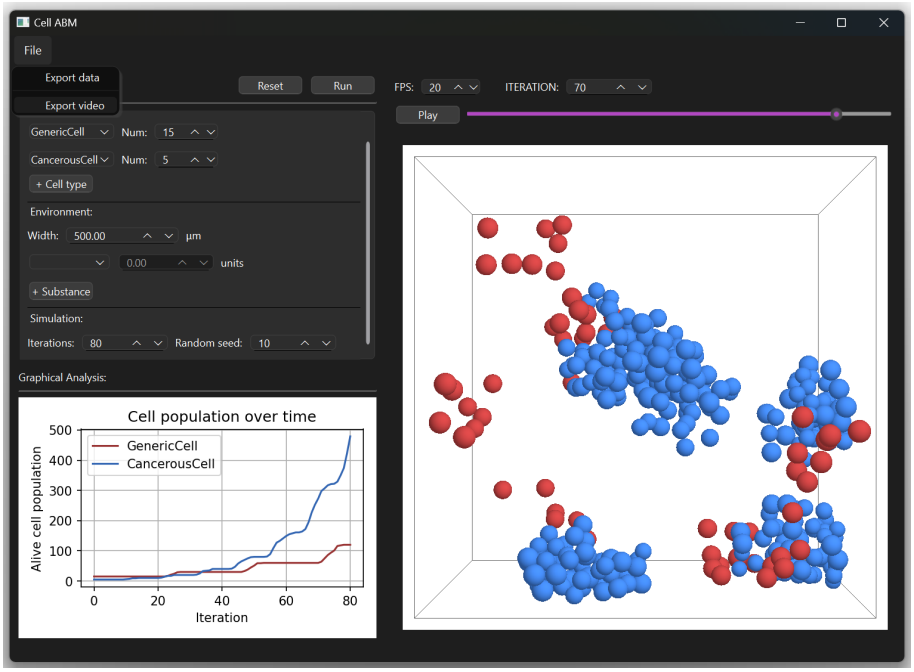


Figure B.3: Example visualisation and graphical results for model, and showing file menu.

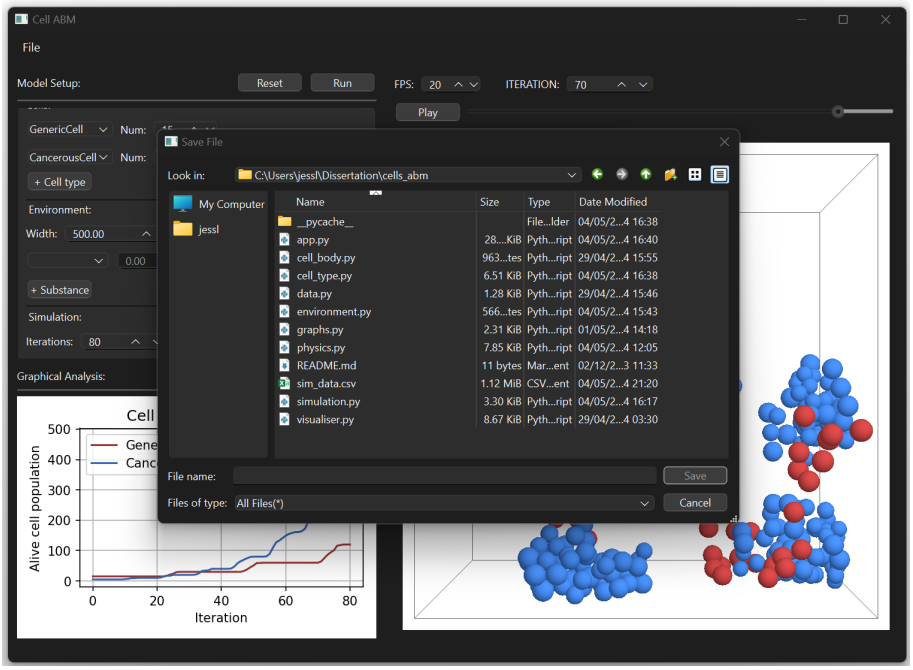


Figure B.4: File dialog appears when user chooses to export data/video.

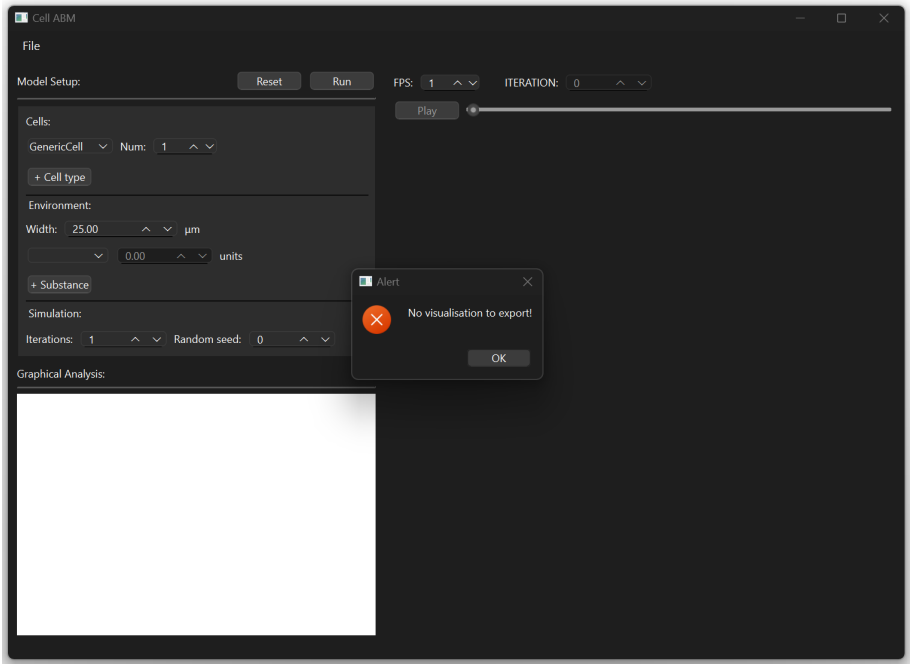


Figure B.5: Alert dialog appears when user tries to export video without a visualisation.

Appendix C

PhysicalModel class constants

PhysicalModel class constants		
Constant name	Value	Units
MAX_ITERATIONS	100	N/A
FORCE_MULTIPLIER (λ in Equation 5.1)	0.125	N/A
TARGET_SEPARATION (s in Equation 5.1)	0.125	μm
MIN_STABLE_OVERLAP_CUTOFF	5.0	μm
MIN_STABLE_OVERLAP_DIFF	0.025	μm
MIN_CONTACTS_FOR_INHIBITION	6	N/A
CONTACT_INHIBITION_RADIUS	0.5	μm

Table C.1: Constant values for the PhysicalModel class.