Cell-Cell Interactions Due To Differential Adhesion in The Overlapping Spheres Model

A Computatioinal Study

David Roland Korossy
St Peter's College
University of Oxford

A report presented for the module titled
Mathematical Modelling and Numerical Computation
Structured Project.
Monday 20th March, 2023

Abstract

The computational modelling of biological processes plays a crucial role in better understanding the mechanisms underpinning cell, tissue, and organ level behaviours. This helps with the prevention, treatment or even the cure of medical conditions and diseases. With recent technological advances in the screening of tissues, we now have more detailed knowledge of the properties of cells than ever. As such, cell-based models which model cells as a collection of separate cells, have become increasingly prevalent in the study of larger scale tissue-level behaviour.

In our study, we will make use of a cell-based model called the overlapping spheres model to simulate cell-adhesion according to the differential adhesion hypothesis. To ensure that we fully understand the implementation of the model, we will develop our own consistent and reliable computational framework. We will then use this to run simulations, to collect data, and to recapitulate the results of a study done in the past. Through this, we validate the reproducibility of their results and the reliability of our implementation.

Efficient neighbourhood search is pivotal in our framework, since this allows us to increase the size of our input to better reflect tissue-level behaviour. To extend on our work, we propose an upper bound for the magnitude of the displacement of cells for different perturbations, and we implement a way to measure these distances into our framework. We will conduct experiments to find the extra search radius required for calculating the neighbourhood of cells less frequently. To take our study further, we investigate the optimal frequency for performing the neighbourhood search, and we narrow down the range of noise-levels for which future studies have yet got to be conducted for. By the end our study, we will have come one step closer towards the optimal implementation of our model, and we will have gained valuable information on efficient ways of computational simulations.

Contents

Chapter 1

Introduction

The strive to understand biology on a deeper level has played a crucial role in the development of scientific knowledge in the past. From the very dawn of humanity when we have just began domesticating plants and animals to farm them in the neolithic age $12,000$ years ago [1], all the way to successfully sequencing the entire human genome in 2003 [2], we have been constantly trying to further our understanding of patterns, migration of species, the mechanics of the human body, medicine, diseases, and the cures thereof. The essence of modelling is therefore part of our nature, since we rely on our ability to make observations, and to make predictions of what decisions we should take based on those. With our current knowledge of medicine, our scientists are persistently trying to get closer to the curing, treatment or prevention of diseases and of medical conditions such as cancer, coronary heart disease, diabetes and Alzheimer's. Any significant progress in this will result not only in an improvement to our knowledge, but in no doubt, it will also result in the saving of countless many human lives ever after. Recent attempts have been made to achieve this goal through the use of mathematical, statistical and computational modelling. This helps us describe the growth of tumours, the mechanics of neurons, and it even provides a tool for the study epidemics such as COVID-19, which had a major impact on many of our lives. In the following study, we will aim to contribute to this progress by understanding a way of modelling the cell-level behaviour of tissues through a computational approach, whose applications are pivotal in early embryonic development.

According to the Differential Adhesion Hypothesis, cells prefer to interact with certain types of cells over the others. This is relevant in early-embryonic development, where cells located in epithelia, that is, a single layer of cells, can undergo cell-sorting. This phenomenon has been observed and modeled in various ways over the past years [3]. Current mathematical and computational formalism focusing on cell-level behaviour consists of a handful of models, each of which serve a specific purpose and model different scenarios. We focus on individual-based models, where cells are modeled as discrete objects that obey certain force-laws. More specifically, we will use the Overlapping Spheres (OS) model. In the following dissertation, I will present a way to model these cell-cell interactions based on the OS model. The aim is to reproduce results of computational simulations from those before me, to give a clear explanation of their underlying mathematics, to present the computational tools developed for the simulations, and to provide an analysis of my results in the study that I have conducted. This project is heavily computational and, as such, it focuses on the development of a framework, on the use of numerical methods, on designing and conducting experiments, and on optimisation.

1.1 Motivation

Computational simulations are important in scientific research. With finite time and resource availability, scientist cannot afford to conduct all the experiments they might wish to. It is optimal to first test hypotheses within a computational framework, see if their results provide correct predictions, and only then design and conduct experiments in laboratories for the hypotheses that are most likely to be correct. This saves significant amounts of money and time if implemented correctly and efficiently. There are some excellent 'open-source' softwares available to aid for this, meaning that they can be accessed by anyone willing to install it on a compatible device free of charge. There are however, some issues that arise with computational studies in modern scientific research. The first one being that not all of the source codes, nor the softwares that they were ran on to obtain the results published in papers are open source. This makes it difficult to reproduce the results in some of the studies, and it allows less room for criticism of the source code. Further, because of how specialised these tools are to the type of problems that they're implemented for, it is far less efficient to re-design a new platform to run and write code on by each institution than to use already existing ones. This process requires time and therefore money, that could have been saved if there was no need to develop an entirely new framework [4]. Open-source programs and softwares therefore help the advances in today's scientific research community. In computational biology, one of the frameworks used by researchers is called 'Chaste' (Cancer, Heart and Soft Tissue Environment), which is an open-source software tailored to simulate multi-scaled problems in biology and in physiology. It makes use of a programming language called C++, which allows for load distribution, efficient memory usage, low run-times, and thereby low computational costs. My project is based on many papers by people that used cell-based Chaste for their results. The use of this software, however, would have prevented me from fully understanding the framework, its intricate details and its ways in which it can be optimised. Hence, I have developed my own framework to recapitulate the results obtained with the use of the aforementioned software. C++ is an advanced programming language, and as someone new to the world of programming and especially new to conducting research, I chose to opt for another programming language instead. For this project, I used the open-source compiler Python3.10 for all of the obtained results, and the source code to some of the key simulations will be included in the appendix. It is worth noting that this project allows for a comparison between the framework that I developed in python and that of Osborne's and colleagues' [5], whose results I aim to reproduce. This would validate both the reproducibility of their results and the consistency of the framework.

1.2 The Differential Adhesion Hypothesis

Cell-sorting takes place when cells in a disturbed initial state arrange themselves into larger clusters of cells of the same types. This plays a crucial role in early embryonic development, since cells that are supposed to be in a certain place must be able to sort themselves and move to their intended position [6]. Attempts have been made in the past to explain this phenomenon, and one of the models that managed to pass extensive testing is called the *Differential Adhesion Hypothesis*(DAH), first advanced in the 1960s by Steinberg [3].

1.2.1 Intercellular Adhesion

Cells can 'stick' to one another, and we call this phenomenon cell-to-cell adhesion. This happens due to a multitude of physical and biochemical factors. Cell-cell adhesion molecules, such as specialised proteins called *cadherins* attached to the outer surface of cells, can form bonds with the cadherins on the surface of other cells. This helps with maintaining the structure of a tissue [7]. Further, a complex collection of proteins called the *Extra Cellular Matrix*(ECM) surround the cells. The surface of cells is also equipped with receptor proteins called *integrins*, which attach the cell to the ECM. Through these, cells are able to attract one another, bind to one another, and

form tissues. The details of how exactly this happens are intricate and are beyond the scope of my project. The main property that we shall focus on is the ability of cells to attract one another.

1.2.2 Stating the hypothesis

Consider a collection of cells of two different types, A and B. We call this a *heterotypic* aggregate of cells. Steinberg's DAH states that in a heterotypic aggregate, cells have the tendency to maximise their intercellular adhesion. Cell-adhesion is analogous to the energy released by a cell from binding to another cell, or to the ECM [6]. Due to the different nature of the proteins attached to the surfaces of cells of different types, the strength of this attractive force depends on the strength of bonds that can form between the types of cadherins. Most cadherins tend to bind to cadherins of the same type, which is called a homophilic interaction form [8]. We can therefore infer that a certain type of cells prefer contact with some cell types over others [9]. This model explains how cell-sorting takes place between two types of cells. We should note that there are certain examples where differential adhesion cannot be the driving factor of cell-sorting, as found by Moore, but the model is well-tested for many cases and thus shall not be discredited [10].

Chapter 2

The Overlapping Spheres Model

2.1 Preliminaries on modelling

We shall begin our broader description of the model used in this project by a preliminary discussion about modelling cells. There are multiple ways to model a collection of cells, and depending on the phenomenon ought to be modeled, we can use different approaches. As an example, a simple model of non-vascular tumour growth uses Partial Differential Equations (PDEs) to model the size of a tumour as a continuous quantity. This is a continuum model which depends on time and on some supply of nutrient that diffuses into the tumour from the outer domain, as outlined by Greenspan [11]. There are various other classical approaches that treat tissues as a continuum, and these usually use the assumption that the length-scale that we are working with is sufficiently large compared to the diameter of a cell. However, it can be difficult to incorporate different types of cells of a population into this model.

With recent technological advances in the imaging of cells, we now have more understanding of the properties and of the behaviour of individual cells [12]. Hence, instead of considering the aggregate of cells as a whole, we can consider each cell as an individual, and we can assign the observed attributes to each of these cells. From this, we can see how the cell-level behaviour causes changes in the whole aggregate, which we will examine in chapter 4.

2.2 A Cell-based approach

We will model a population of cells as the collection of individual cells. There are many ways of achieving this, but our current models fall into two main categories :

1. ***On-lattice models***: these discretise space into a lattice and assign our cells to available slots (possibly to multiple slots for a single cell) in the lattice. The simplest on-lattice cell model is the Cellular Automata model, which discretises 2D space into a square-latice, and assigns one cell to one square in the lattice.
2. ***Off-lattice models*** treat space as continuous, where cells are no longer confined to sites of a lattice, which provides a more realistic way of modelling the behaviour of cells. These models fall into two sub-categories:
   (a) *Cell-centre models* define the cell by its centre (a point in space), and a neighbourhood of some shape around it, such as an ellipsoid.
   (b) *Vertex models* define cells as polygons formed by some finite collection of vertices and edges.

2.3 Setting up our model

In this project, we will focus on one cell-centre model: the ***Overlapping Spheres (OS)*** model. Consider a population of cells that we model as spheres centred at a point in space. The dynamics of these cells is described by some sort of force law, that depends on the position of the cells in the population. For simplicity, we assume that cells do not die or multiply, and that they are spheres of some specified uniform radius, r_{cell}, that stays constant over time. Our aggregate is a collection of N cells, and we index each of them as cell i, for $i = 0, ..., N-1$. Note that we start from 0, because our future implementation will involve the use of a programming language that starts indexing from 0, and not from 1.

Definition 1. *We say that two cells* interact*, if they exert a force on one another.*

Definition 2. *The* neighbourhood *of cell i, $\mathcal{N}(i) = \{\ j$: cell j interacts with cell $i\ \}$.*

Let $\mathbf{r}_i = \mathbf{r}_i(t)$ denote the position of the centre of cell i at time t, and $\mathbf{F}_i = \mathbf{F}_i(\mathbf{r}_j)\ ,\ j \in \mathcal{N}(i)$ denote the force acting on it. To find the equations of motion, we apply Newton's second law to each cell. More specifically, for cell i we have:

$$\frac{d^2\mathbf{r}_i}{dt^2} = \mathbf{F}_i - \eta\frac{d\mathbf{r}_i}{dt}\,,$$

where the first term is the acceleration of the cell, the second term is the resultant force acting on the cell from all the other cells in $\mathcal{N}(i)$, and the third term is the force resisting the motion of the cell due to the viscosity of the medium that the cells are located in. We assume that this medium is highly viscous, and that our system becomes overdamped. We can therefore ignore all inertial terms, and from the dominating terms, our equations of motion become [13]:

$$\eta\frac{d\mathbf{r}_i}{dt} = \mathbf{F}_i\,. \tag{2.1}$$

This means that once we manage to find a reasonable way to model the force on a cell, we can solve the equations of motion for each cell to evaluate its position at all points in time. Now, the laws that these cells obey must have some constraints, otherwise our model would be unphysical. A good way to start setting up our model, is to note the following assumptions that we shall make regarding the properties of cells:

1. ***No two cells can be in the same position at a time***. This is because cells are solid objects, which cannot move through one another like some ghost-like particles.

2. ***If two cells are too close to one another, then they should repel each another.*** The magnitude of this repulsive force should get stronger with the size of the intersection (called the *overlap*) between two cells.

3. ***If two non-overlapping cells are close enough to one another, then they should attract each another.*** The reason behind this is as outlined in the introduction: short-range signalling takes place due to interactions between cells and the extracellular matrix.

4. ***Cells that are too far away from one another, do not interact.***[1].

Any force-law that we consider should reflect these properties. We aim to justify the validity of our force-law by closely examining the resulting behaviour of the cells under these laws.

[1] We make this assumption for our implementation, but it is not necessarily true [14]. For our model however, this is not relevant.

2.3.1 Cell-cell interaction force-law

Consider cell i together with its neighbourhood, $\mathcal{N}(i)$. Let $\mathbf{F}_{ij}$ be the force exerted on cell i by cell j, so that the resultant force $\mathbf{F}_i$ on cell i, is given by:

$$\mathbf{F}_i = \sum_{j \in \mathcal{N}(i)} \mathbf{F}_{ij} \,. \tag{2.2}$$

We assume that $\mathbf{F}_{ij}$ acts in a direction parallel to the vector from cell i to cell j. Let $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$, and $\hat{\mathbf{r}}_{ij}$ be the corresponding unit vector. In this force-law, we say that cell i interacts with cell j if and only if the two are within a fixed distance, $r_{\max}$ of one another, beyond which cells no longer interact. We can then write the neighbourhood of cell i as $\mathcal{N}(i) = \{\, j \,:\, ||\mathbf{r}_{ij}|| < r_{\max} \,\}$. The *natural separation* between two cells in this model, s_{ij} is twice the cell radius r_{cell}, and the spring constant between two cells is μ_{ij}. In this project, we will use the force law outlined in [9], which is defined as:

$$\mathbf{F}_{ij}(t) = \begin{cases} \mu_{ij} s_{ij}(t) \hat{\mathbf{r}}_{ij}(t) \log\left(1 + \dfrac{||\mathbf{r}_{ij}(t)|| - s_{ij}(t)}{s_{ij}(t)}\right), & \text{for } \; ||\mathbf{r}_{ij}(t)|| < s_{ij}(t) \\ \mu_{ij} \left(||\mathbf{r}_{ij}(t)|| - s_{ij}(t)\right) \hat{\mathbf{r}}_{ij}(t) \exp\left(-k_c \dfrac{||\mathbf{r}_{ij}(t)|| - s_{ij}(t)}{s_{ij}(t)}\right), & \text{for } s_{ij}(t) \le ||\mathbf{r}_{ij}(t)|| \le r_{\max} \\ \mathbf{0}\,, & \text{for } ||\mathbf{r}_{ij}(t)|| \ge r_{\max}\,, \end{cases} \tag{2.3}$$

where k_c is a parameter defining the decay of the attractive force as the separation between the two cells increases. We then substitute this law into equation (2.1) to obtain the equation to be solved. To see how this force-law reflects the behaviour outlined in section (2.3), we can plot the signed magnitude of the force denoted as F_{ij} against the distance between two cells, r, for $\mu = 50$, $r_{\text{cell}} = 0.5$, $r_{\max} = 2.5$ and $k_c = 5$. The plot is shown in the figure below:

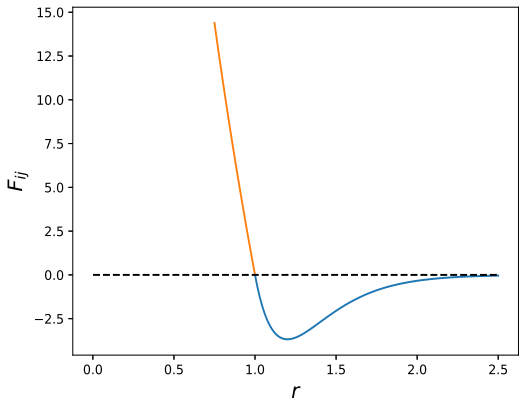


Figure 2.1: Force between two cells against the distance between their centres

As expected, we have a positive (repulsive) force between two cells when their separation is less than their natural separation, and we have a negative (attractive) force when their separation is between the natural separation and $r_{\max}$, which is taken to be larger than $s_{ij}(t)$.

We must note that the size of k_c is fine-tuned to the force: a low k_c would result in a behaviour that doesn't reflect the behaviour of cells. If k_c is too small, then the attractive force between two cells carries on being relatively strong, even after the separation between two cells increases. This can result in a scenario in which we have three collinear cells, with the outer cells getting attracted by the middle one. The issue is that since the strength of the attractive force has not decreased fast enough, the outer cells attract each other too much. This resulths in the tissue to reach a steady state where the overlap between the cells is too large, which cannot happen due to cells not being able to move through each other. An illustration of the behaviour of a system of 9 cells starting from a 3×3 square grid is shown in figure 2.2, where the motion of the cells is governed by the ODE from equation 2.1. We can see that as time goes by, the system reaches an equilibrium that is unphysical due to there being too much overlap between cells.

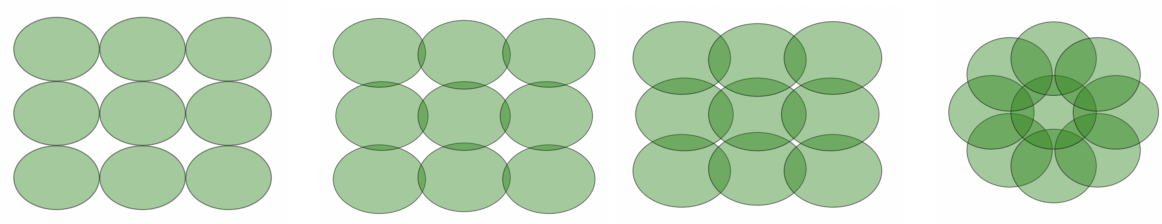


Figure 2.2: Evolution of the tissue over time with $k_c = 1$

.

Further, we can see that as the separation falls below s, not only does $\mathbf{F}_{ij}$ become repulsive, but it does so rapidly, and it becomes significantly stronger than the strength of the attractive force. This is pivotal in the stability of the model, as if the repulsive force would not get strong enough sufficiently quickly, then we could again have predictions that do not reflect the physical behaviour of cells.

I would like to briefly discuss the scaling of k_c. Let's suppose we scale our distance by α, and using the notation of $r = ||\mathbf{r}_{ij}||$ and $s = s_{ij}(t)$ between two cells for fixed i and j, we can write this scaling by letting $\tilde{r} = \alpha r$ and $\tilde{s} = \alpha s$. We want to find out how we should scale k_c. Let this scaled value of k_c be $\tilde{k}_c$. By equating the scaled force to the original one, we get:

$$(r-s)e^{-k_c\left(\frac{r-s}{s}\right)} = (r-\alpha s)e^{-\tilde{k}_c\left(\frac{r-\alpha s}{\alpha s}\right)}.$$

The only solution to this equation that is independent of r is $\alpha = 1$ and $\tilde{k}_c = k_c$. This means that we cannot scale the cell radius and still obtain the same behaviour from the attractive force by scaling k_c. This shows that the value of k_c has to be chosen carefully in order to get the desired behaviour, which is not to our surprise since the force law models the phenomenon, as opposed to the phenomenon having been observed as a result of the analysis of the model.

Thus, we now have a way of explicitly defining the force exerted on any cell i by any other cell j, which we can use to determine the total force acting on cell i to solve equation 2.1 . The force-law that we use in our model is designed to reflect the behaviour of cells interacting, and it is left for us to develop a framework that can utilise the above stated law in order to make predictions of the behaviour of tissues.

Chapter 3

Computational Simulation

Let us go back to our original ODE to be solved to get the equations of motion. This was:

$$\eta \frac{d\mathbf{r}_i(t)}{dt} = \mathbf{F}_i(t) = \sum_{j \in \mathcal{N}(i)} \mathbf{F}_{ij}(t)\,.$$

In principle, we can solve this differential equation for, say, two interacting cells analytically. This would result in the system reaching a steady state, where the separation between the two cells, $||\mathbf{r}_{ij}(t)||$ is equal to the natural separation, $s_{ij}(t)$ between them. That is, the cells would settle in a position where they just touch one another. Solving the ODE by hand gets more difficult as we increase the number of cells interacting, and it is entirely possible that there is no analytical solution to it. As a result, we use the power of computers to aid us instead.

The aim of this project is to develop a computational framework that reproduces the results in the paper from Osborne and colleagues [9], and to extend on it. I will be using Python 3.10, and I will be making use of existing libraries for my implementation of their method; NumPy [15] and SciPy [16] for handling data and for optimisation of my code, and Matplotlib [17] for visualisation. Throughout the project, I will aim to adhere myself to the principles found in Osborne's article [5], which is a guide for conducting effective computational research. I found it particularly important to constantly test and visualise the simulation as time goes by, since this helps us the evaluation of the implementation of our model. We shall now describe the mathematics underpinning our problem, and the computational framework developed for the project.

3.1 A numerical approach

In order to solve the ODE from above numerically, we use a simple Forward Euler scheme to discretise time. The Forward Euler method is commonly used to find numerical approximations to the solution of a differential equation. Given the right conditions, this approximate solution is very close to the solution that we seek [18]. The implementation of this method to our problem is as follows:

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \frac{\Delta t}{\eta} \sum_{j \in \mathcal{N}(i)} \mathbf{F}_{ij}(t)\,, \tag{3.1}$$

with a sufficiently small time-step Δt to ensure the numerical stability of the system [9]. Time is now discretised into time-steps of size Δt, and the above equation determines the position of cell i at the next time step in terms of the position of the cell, and the force acting on the cell at the

current time-step. We can evaluate this for all cells and for all time-steps, and this will achieve our goal of knowing where cells are at any given time. To assign this task to a computer, we need to break the method down to a step-by-step process that a computer can easily perform. We describe this process by the following algorithm:

Algorithm 1 The simulation

Specify the initial state of the cells in the tissue,
for each timestep **do**:
    Find the neighbourhood of all cells,
    Find the force acting on each cell,
    Update the position of each cell according to equation 3.1.

The above process successfully describes a simulation of the behaviour of cells in a tissue. The task left for the next parts of the project is to specify the ways in which we perform each step in the algorithm, and to optimise these steps. We will then aim to implement this method to a heterotypic tissue with two different cell-types.

3.2 Specifying steps

3.2.1 Initial state

The initial state of the tissue is specified by the location and by the type of cells within it. Let us think about the location of each cell first. A simple approach to assign the initial position of each cell would be to arrange them in a square grid with width L_x and height L_y, where the units of length is one cell diameter (CD). Another, more realistic approach would be to arrange the cells into a hexagonal grid which again contains $L_x \times L_y$ many cells. This approach resembles the arrangement of cells that we see in tissues. Note that even if we start from the more simple approach, we do not expect the bulk-behaviour of the system to change by a large margin as time goes by. Similarly, we could start with a square grid, let the tissue reach a state where their arrangement resembles the one from the hexagonal method, and only then start the simulation.

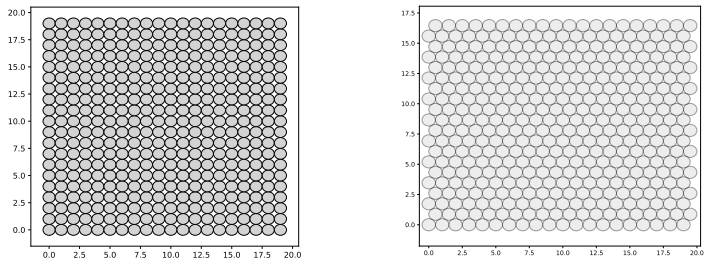


Figure 3.1: A square arrangement *(left)* and a hexagonal arrangement *(right)* for $N = 400$.

To find a way to create this initial arrangement, we would like to write a function to set up the simulation with $N =$ `num_cells` many cells. We will store the location of cells in a NumPy array with 2 columns and N rows called `cell_locations`, and we will use a datatype called *double*. This stores numbers as a 64-bit precision floating-point, which helps with our goal of performing calculations that require high precision. The position vector of cell i is in row i of `cell_locations`, that is, $\mathbf{r}_i =$ `cell_locations[i,:]`. This means that we can declare the initial position of each cell by assigning the appropriate values for the entries of the array.

Now, consider the *type* of each cell. As outlined in the introduction, we aim to examine the cell-sorting mechanism in a tissue with two different types of cells. We will assign a type to each cell, which either takes the value 0 or 1. We will store the cell types in a NumPy array of length N, and using a datatype called *int32*, which is used to store integers up to a size of $2,147,483,647$. We impose that half of the cells are of type 0 while the other half are of type 1, and the types will be randomly assigned to each cell using the `random_shuffle` function of NumPy. We define a function called `setup_simulation` that assigns the initial state of the tissue as outlined above. This can be done for the hexagonal arrangement as follows:

```
import numpy as np

num_cells = 400
radius = 0.5
cell_locations = np.zeros((num_cells, 2), dtype=np.double)
cell_types = np.zeros(num_cells, dtype=np.int32)

def setup_simulation(cell_types, cell_locations):
    """
    This function will set up the initial state of the simulation.
    """
    height = int(np.ceil(np.sqrt(num_cells * np.sqrt(3) / 2))) + 1
    width = int(np.ceil(num_cells / height))
    # Loop over the rows and columns of the hexagonal grid
    for i in range(height):
        for j in range(width):
            index = i * width + j
            if index >= num_cells:
                break
            x = (2 * j + i % 2) * r_cell
            y = i * np.sqrt(3) / 2 * r_cell
            cell_locations[index, 0] = x
            cell_locations[index, 1] = 2 * y
    # Assign cell types
    num_cells_per_type = num_cells // 2  # assuming even number of cells
    cell_types[:num_cells_per_type] = 0
    cell_types[num_cells_per_type:] = 1
    np.random.shuffle(cell_types)
```

A version of the above script was used to generate the tissue from figure 3.1 with $N = 400$, and $r_{\text{cell}} = 0.5$. We will use this later to run the simulation. Note that here, we have defined the entries of arrays as zeros first, and then updated the entries through a loop. This is a common practice in programming.

3.2.2 Evaluating neighbourhoods

Recall that the neighbourhood of cell i is defined as $\mathcal{N}(i) = \{ j \, : \, ||\mathbf{r}_{ij}|| < r_{\max} \}$. To find the neighbourhood of all cells, we can start by comparing the location of each cell to the location of all the other cells. We then add the cells that were within the cut-off radius to the neighbourhood of the respective cell.

Definition 3. *We say that an algorithm has a* time complexity *of $O(g(n))$, if increasing the size of the input by a factor of n will result in the limiting behaviour of the time required to execute the*

algorithm increasing by a factor of at most a constant multiple of $g(n)$. We read this as 'order $g(n)$' and we call it 'Big O notation'.

To start, we store the neighbourhoods of cells in `neighbours = [] for i in range(num_cells)`, which is initially a list of empty lists. We define the cut off radius `r_cut_off` $= r_{\max}$, denoting the distance beyond which cells are no longer included in each other's neighbourhoods. The `linalg.norm` function of the NumPy library returns the Euclidean norm of a vector, and we use this for the comparisons. We then define a function that evaluates the neighbourhoods of all cells and adds those cells to the empty lists of `neighbours`. We call this `find_neighbours(cell_locations)`, which takes `cell_locations` as an input, and we implement it as follows:

```
def find_neighbours(cell_locations):
    neighbours = [] for i in range(num_cells)
    for i in range(num_cells):
        for j in range(num_cells):
            if np.linalg.norm(cell_locations[i,:] -cell_locations[j,:]) <= r_cut_off:
                neighbours[i].append(j)
    return neighbours
```

Using the above procedure to evaluate neighbourhoods, we do N comparisons for N many cells, so if we increase the number of cells by a factor of n, then the number of calculations required to evaluate the neighbourhoods increases by a factor of n^2. This means that the time complexity of the algorithm is $O(n^2)$. Note that we could halve the number of calculations required by only looping over $j > i$ and adding `neighbours[j].append(i)` into the loop. This reduces the number of comparisons by half, but since this is just a constant multiple of the original number of calculations, our algorithm's overall time complexity would not change.

A similar way to do this would be to use the `where` function of the NumPy library, which makes it easier to perform the same procedure using :

```
def find_neighbours(cell_locations):
    neighbours = [] for i in range(num_cells)
    for i in range(num_cells):
        neighbours[i] = np.where((np.linalg.norm(cell_locations[i,:] - cell_locations,
                        axis=1) <= r_cut_off))[0]
    return neighbours
```

Unfortunately for us, while the runtime for evaluating the neighbourhoods this way is relatively lower than for the method from before without using functions form the NumPy library, this does not change the time-complexity of the algorithm. The only thing that does change, is how the computer handles the data, which happens to be faster due to the library being optimised. Improvements to the neighbourhood search will be discussed in chapter 5.

3.2.3 Calculating forces

To reflect the behaviour of cells according to the DAH, we must modify our force law defined in 2.3 . This is to ensure that two cells of different types attract each other less, but repel each other just as much as two cells that are the same type. The spring constant between two cell types will be set to $\mu = 50$, and the spring constant between two cells of different types, called the heterotypic spring constant, will be reduced to 10% of μ, meaning that $\mu_{\text{het}} = 5$. We implement this into our force-law by defining a pseudo-delta function, `delta(i,j)` inspired by the Kroneker-delta, which takes the value of 50 if two cells are the same type, and otherwise it takes the value of 5.

$$\text{delta}(i,j) = \begin{cases} \mu & \text{if type of cell i = type of cell j} \\ \mu_{\text{het}} & \text{if type of cell i} \neq \text{type of cell j} \end{cases} \tag{3.2}$$

Evaluating this function repeatedly for all interacting cells every time-step would be inefficient, so instead, we store these values in an $N \times N$ NumPy array called `Delta`, where the (i,j)th entry of the array is given by delta(i,j). We set the variables `mu` $= \mu$ and `mu_het` $= \mu_{\text{het}}$, and we define a function named `calculate_Delta`, which generates this array for us. We implement this into our framework as follows:

```
def delta(x, y):
    #This is the pseudo-delta function
    if x == y:
        return mu
    else:
        return mu_het
Delta = np.zeros((num_cells, num_cells), dtype = np.int32)
def calculate_Delta(cell_types):
    for i in range(num_cells):
        for j in range(num_cells):
            Delta[i,j] = delta(cell_types[i],cell_types[j])
    return Delta
```

We will call the `calculate_Delta` function when we will run the simulation, and then use `Delta[i,j]` in place of μ_{ij} in the attractive force. Note that μ_{ij} stays constant and takes the value of μ between all cells for the repulsive force regardless of the type of cells. The reason behind this is that the cause of the repulsive force is physical and it is due to cells being solid and not being able to pass through each other, whereas the cause of the attractive force is biochemical, stemming from protein bonds and interactions with the ECM.

To store the force on each cell, we once again use an $N \times 2$ NumPy array called `cell_forces`, where the ith row of the matrix corresponds to the force acting on cell i.
That is, `cell_forces[i,:]` $= \mathbf{F}_i$. We can then define the function that calculates the forces on all cells that we'll name `calculate_forces_each_timestep(...)`, which will take `cell_locations`, `cell_types`, `neighbours` and `Delta` as inputs. The importance of using local variables as opposed to global variables in our script is mentioned in 4.2.1 . We define this function as:

```
def calculate_forces_each_timestep(cell_locations, cell_types, neighbours, Delta):
    cell_forces = np.zeros_like(cell_locations) # Start from no forces acting on the cells.
    for i in range(num_cells):
        for j in neighbours[i]:
            if i > j:
                f = [0,0] # Start from no force being added to the force on any cells.
                r_ij = cell_locations[j, :] - cell_locations[i, :]
                if np.linalg.norm(r_ij) < 1:
                    f = (mu * ((r_ij) / np.linalg.norm(r_ij)) *
                         np.log(1 + (np.linalg.norm(r_ij) - 1)))
                elif 1 <= np.linalg.norm(r_ij) <= 2.5:
                    f = ((Delta[i,j]) * (
                                np.linalg.norm(r_ij) - 1) *
                         ((r_ij) / np.linalg.norm(r_ij)) * np.exp(-k_c * (np.linalg.norm(r_ij) - 1))
                         )
                cell_forces[i, :] += f # Update the force on cell i.
                cell_forces[j, :] += -f # Update the force on cell j, by Newton's third law.
    return cell_forces
```

Here, the variable `k_c` $= k_c$, and since Newton's third law of motion implies that $\mathbf{F}_{ij} = -\mathbf{F}_{ji}$, we could reduce the number of calculations by half as we only loop through $j < i$, and add $-\mathbf{F}_{ij}$ to the jth row of the `cell_forces` array. The loop keeps updating the rows of the array until there are no more interactions left, at which point the total force on all cells has been found. Note that every time this function is called, the force on each cell is being reset to zero. This is to ensure that forces from previous time-steps do not keep acting on the cells at future time-steps, and so we will not encounter cells marching out into the distance without interacting with other cells. This reflects the cell-medium being viscous, and inertial forces being neglected. The `calculate_forces_each_timestep` function will be used at each time-step for running the simulation, and it will be extended in later sections.

3.2.4 Running the simulation

With all the the functions from the previous subsections being defined, we have finally got all the ingredients that we need to run the simulation. We have defined the initial state of the system where we have all the positions and types of the cells, we can find all of their neighbourhoods, and we are able to determine the force acting on each cell. Now, all we have left is to find a way to run and to visualise the simulation.

Recall that the position of cell i at each timestep is defined by the law from equation 3.1, and it only depends on the position and on the force acting on cell i at the previous timestep. We aim to define a function called `run_simulation()`, that takes `cell_locations`, `neighbours` and `cell_forces` as its inputs. This function will have two purposes:

- Update the location of all cells using the forward Euler scheme.
- Visualise the state of the system to see how it evolves over time.

We use a time step size of 0.005 hours, and we aim to simulate the evolution of the system for 100 in-simulation hours. We define `simulation_finish_time = 100`, `time_step_size = 0.005` $= \Delta t$, and `num_time_steps = int(np.ceil(simulation_finish_time / time_step_size))` as global variables before running the simulation. Without loss of generality, we set the parameter $\eta = 1$, since this only sets the time-scale of the simulation and we can always change Δt to adjust for any value of η. We must call the function `find_neighbours` while we run the simulation, as the neighbourhoods of cells will change over time. In the loop, we set `cell_locations += time_step_size * cell_forces` to update the location of each cell for each time-step.

To visualise the simulation, we make use of the Matplotlib library. We want to see each cell as a circle filled in with a colour corresponding to its type. To shade the circles, we could use an existing colour map, but instead we create a new one. We do this by defining the array `colors=['green', 'purple']` and creating a listed colour map using the colours inside it, `cell_colors = ListedColormap(colors)`, which assign the colour green to type 0 and purple to type 1 cells. These are all set as global variables before running the simulation. We then create a list of circles around the centres of the cells with uniform radii of `rad_cell = 0.5`, and we shade those circles in with the colours from the colour map defined earlier. We update the figure each time-step, and if we run this over all time steps, then it will appear as though the cells are moving around, colliding with one another, and attracting/repelling each other. We have therefore successfully achieved the goal to create a frame-by-frame animation of the cells changing position according to the force law stated earlier. Although the animation helps us visualise the simulation, it is not the most time-efficient way of running our program. It turns out that the process of updating the plots each time-step takes up approximately half of the run-time, so when running the simulation, it will only be plotted every 10 in-simulation hours.

We define the `run_simulation` function in the following manner, making use of the previously defined functioins and calling them when needed:

```
def run_simulation(cell_locations, cell_forces, cell_types):
    """
    Here we perform the time stepping, update the locations and the forces, and visualise the results.
    """
    for time_step_idx in range(num_time_steps):
        # Find neighbourhoods of all cells
        neighbours = find_neighbours(cell_locations)
        # Find the force acting on each cell
        calculate_forces_each_timestep(cell_locations, cell_types, neighbours, Delta)
        # update the location of each cell
        cell_locations = cell_locations + time_step_size * cell_forces
        # Reset all the cell forces
        cell_forces = np.zeros_like(cell_locations)
        # Plot graph for time = 10,20,30,... hours
        if time_step_idx % 2000 ==0:
            fig, ax = plt.subplots()
            scat = None
            # Create a list of circles
            circles = [Circle((cell_locations[j, 0], cell_locations[j, 1]), 0.5)
                       for j in range(num_cells)]
```

```
            # Assign the colour of each cell
            colors = cell_types
            if scat is None:
                scat = PatchCollection(circles, cmap=cell_colors, alpha=0.4, edgecolors='black')
                scat.set_array(colors)
                ax.add_collection(scat)
                plt.gca().set_xlim(-10, 30)
                plt.gca().set_ylim(-10, 30)
            else:
                scat.set_paths(circles)
            # Update the plot for each frame
            scat.set_offsets(cell_locations)
            ax.set_title(f"Time: {round(time_now, 1)}")
            plt.pause(0.05)
            plt.show()
```

Now that we have defined the `run_simulation` function, we can execute the code as a script using the `if __name__ == '__main__':` condition before calling the functions:

```
if __name__ == '__main__':
    setup_simulation(cell_types, cell_locations)
    run_simulation(cell_locations, cell_forces, cell_types)
```

With this, we conclude our work on the part of our program that will act as the back-bone of our later study.

Discussion

Let us now briefly recapitulate our work in this chapter. We have successfully designed a framework implementing Differential Adhesion Hypothesis with the Overlapping Spheres model, which allows us to simulate the interactions between cells. Our simulation generates a random configuration of cells in a hexagonal grid, determines the total force acting on each cell for each time-step, it performs the time-stepping process from 3.1, and it lets us visualise the movement of cells over time. The above program proceeds precisely as described by Algorithm 1, meaning the task of creating the foundation of our computational framework has now been completed. The work done so far will underpin the study conducted in subsequent chapters.

Chapter 4

Cell-Sorting

4.1 Adding a perturbation

As described in the earlier sections, we now have the foundation to simulate cell-sorting due to cell-adhesion according to the DAH. To start running the simulations and to reproduce the results from [9], we first modify the system by adding a random force to the force acting on each cell at each time-step, which corresponds to the random intrinsic movement of each cell. Let this force be $\mathbf{F}^{\text{rand}}$, and define it as:

$$\mathbf{F}^{\text{rand}} = \sqrt{\frac{2\xi}{\Delta t}}\boldsymbol{\eta}, \tag{4.1}$$

where $\boldsymbol{\eta}$ is a sample from a standard multivariate normal distribution and ξ is a parameter corresponding to the base level of perturbation. The factor of $(\Delta t)^{-1/2}$ ensures that the magnitude of the perturbation has no explicit dependence on the size of the time-step [9]. In fact, if we were to perform the time-stepping from equation 3.1 for cell i without considering the force law defined in equation 2.3, then we would get:

$$\begin{aligned}\mathbf{r}_i(t+\Delta t) &= \mathbf{r}_i(t) + \Delta t\sqrt{\frac{2\xi}{\Delta t}}\boldsymbol{\eta} \\ &= \mathbf{r}_i(t) + \sqrt{2\xi\Delta t}\boldsymbol{\eta}\,,\end{aligned}$$

which describes the diffusion of a random walker with diffusion constant ξ [4]. The implementation of this random force into our code is simple: we can use the `random.randn(2)` function from the NumPy library to generate an array of two samples from a standard normal distribution. We will examine the effect of the perturbation size on the system by multiplying ξ by the multiplier k_{pert}. This will take values in range of $[10^{-2}, 10^{2}]$ and it represents the magnitude of the perturbation in our system. We include this in the definition of `calculate_forces_each_timestep` in the following manner:

```
def calculate_forces_each_timestep(...):
    for i in range(num_cells):
        # Code for caluating forces
        cell_forces[i, :] += np.sqrt( 2* k_pert * xi / time_step_size) * np.random.randn(2)
```

Having added this perturbation term to our framework finalises our modification to the force-law, and we are now ready to run the simulation. The parameters used for all simulations are as stated in [9], and are given in table 4.1

Parameter	Description	Value
Δt	Time-step size	0.005 h
k_c	Decay of attraction force	5
μ	Spring constant	50
μ_{het}	Heterotypic spring constant	5
s	Natural cell separation	1 CD
$r_{\max}$	Force cut-off length	2.5 CD
ξ	Base level of perturbation	0.05
k_{pert}	Perturbation multiplier	$[10^{-2}, 10^{2}]$

Table 4.1: Parameters used for running simulations

4.2 The effect of perturbation

Now that we have successfully incorporated noise into our framework, we can check the validity of our implementation against the one outlined in [9]. The simulation was run for $k_{\text{pert}} = 1$ for 100 in-simulation hours, and the evolution of the tissue over time is shown in figure 4.3:

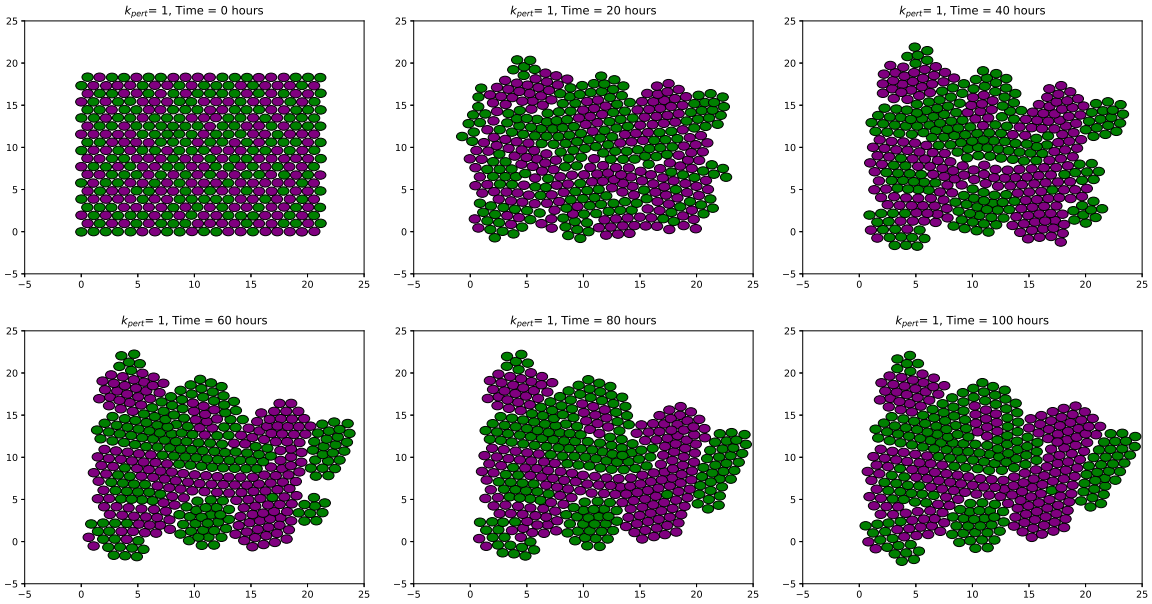


Figure 4.3: Evolution of the tissue over time with $k_{\text{pert}} = 1$

We can see that as time goes by, cells sort themselves out into a state in which there are larger clusters of cells of each type than in the initial state of the tissue. This is a steady state, since we can see that there is not much change in the tissue after 40.0 hours.

The nature of the force defined in 3.2.3 tells us that cells of the same type attract each other more than cells of different types, which explains why cell-sorting is the outcome that we observe. This is no coincidence, since the force law used in our implementation is tailored to model this self-sorting process that cells can undergo in a tissue. Looking at figure 4.3, there is a question that remains: why is it, that in the lower-left quadrant of the tissue, cells were unable to successfully sort themselves? More importantly, how would changing the level of noise affect the outcome of the system?

Let us examine the behaviour of the tissue under different sizes of perturbation. The simulation was run for a fixed initial configuration with different perturbation levels, and the outcome of each simulation at 100 hours (unless stated otherwise) is shown in the figure below.

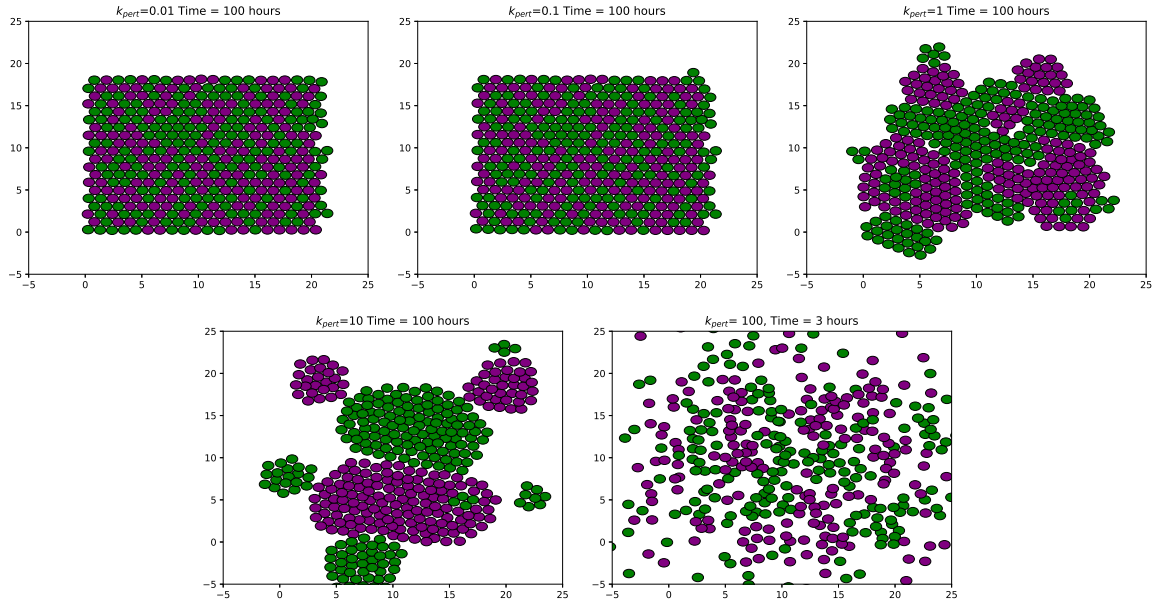


Figure 4.3: Evolution of the tissue over time with $k_{\text{pert}} = 1$

We can see from figure 4.3 that for lower values of k_{pert}, the extent of cell-sorting taking place is little to none. We see that there is a range of k_{pert}, [1, 10], that enables cell-sorting. For this range, we get large clusters of purple and green cells in the steady state. We also note that for any values beyond $k_{\text{pert}} = 100$, cell-sorting doesn't take place due to the perturbation being too high and cells disassociating as a result.

This is precisely how we expect our simulation to behave, and it reflects the behaviour of cells that we model with it. For smaller noise levels, the adhesive force between cells dominate, and the perturbation level is not sufficiently large to enable the movement of cells. For a specific range of noise, the magnitude of perturbation is sufficiently large to overcome the energy required to break the cells free from the bond with one another, and so they have enough freedom to sort. For too much noise in the system however, the magnitude of perturbation dominates, with the attractive force between cells being near-negligible, and so the attraction between two cells is not strong enough to form clusters.

4.2.1 Quantifying well-sortedness

We are now at a stage where we understand what the tissue will look like depending on the perturbation size. However, we need a more precise way of measuring how well-sorted our system is over time.

Definition 4. *The **fractional length** is defined as the total length of edges shared between cells of different types, normalised by dividing it by the length at Time* $= 0$.

If we look at the simulation with $k_{\text{pert}} = 1$ from figure 4.3, we can see that this fractional length decreases as time goes by. This suggests that we can tell how well-sorted we expect our system to be at any given time purely from the fractional length corresponding to that time. This is a good way of quantifying well-sortedness, as it does not rely on us having to look at the tissue and make an arbitrary judgement of what we classify more or less sorted.

In our implementation, we will use an analogous way of defining the fractional length, where we define it by the number of edges shared between two cells of different types. With this definition, it becomes simple to add this into our framework:

```
def calculate_forces_each_timestep(...arguments..., timestep_idx):
    fractional_lengths = np.zeros(num_time_steps)
    for i in range(num_cells):
        # ... Code for caluating forces ...
    # if they share and edge and are of different types:
    if np.linalg.norm(r_ij) <= 1:
        if cell_types[i] != cell_types[j]:
        # count the number of edges shared
            fractional_lengths[time_step_idx] += 1
```

Here, we store the fractional lengths in a NumPy array with length `num_time_steps = 20,000`, and we have made the `calculate_forces_each_timestep` function depend on the time-step, in order to update the entry of the array corresponding to the time-step. We would like to see how the fractional length changes over time for different perturbations. One way to approach this, is to run the simulation for a fixed k_{pert}, do this for a number of times in a row, take the average, and plot it against time with the 95% confidence interval above and below the mean. The runtime of this process would be high, so instead we make use of the multiprocessing tool of Python, which allows us to run all the simulations in parallel using a pool of workers. Using local variables plays a crucial role here, since we do not want to update the position of cells in a simulation using any information from another. We define the `run_simulation_wrapper()` function, that sets up and runs simulations independently of one another. We then call this function using the `Pool` function for 8 separate simulations. We save the mean and the standard deviation of the fractional length in a `.pkl` file, which allows for the plotting of multiple lines on the same plot. The implementation of the above process was done in the following manner:

```
def run_simulation_wrapper(i):
    cell_locations = np.zeros((num_cells, 2), dtype=np.double)
    cell_types = np.zeros(num_cells, dtype=np.int32)
    cell_forces = np.zeros_like(cell_locations)
    setup_simulation(cell_types, cell_locations)
    data = run_simulation(cell_locations, cell_forces, cell_types)
    return data

```

```
if __name__ == '__main__':
    with Pool() as pool:
        result = pool.map(run_simulation_wrapper, range(8))
    data_list = np.array(result)
    mean_list_over_time = np.mean(data_list, axis=0)
    n = max(mean_list_over_time)
    std_list_over_time = np.std(data_list, axis=0)

    with open('...file name of choice...', 'wb') as f:
        data = [mean_list_over_time/n, std_list_over_time/n]
        pickle.dump(data, f)
```

We have therefore implemented a way of measuring and storing the fractional length in the tissue into our framework. We are now ready to run the simulations and to finally see the relationship between fractional length and time.

The above version of the simulation was run 8 times for a range of k_{pert}, and the smoothed data was plotted with the 95% confidence interval, corresponding to mean $\pm$ 1.96 $\times$ standard deviation (for specific levels of nose) for every 100 time-step. The results are shown below:

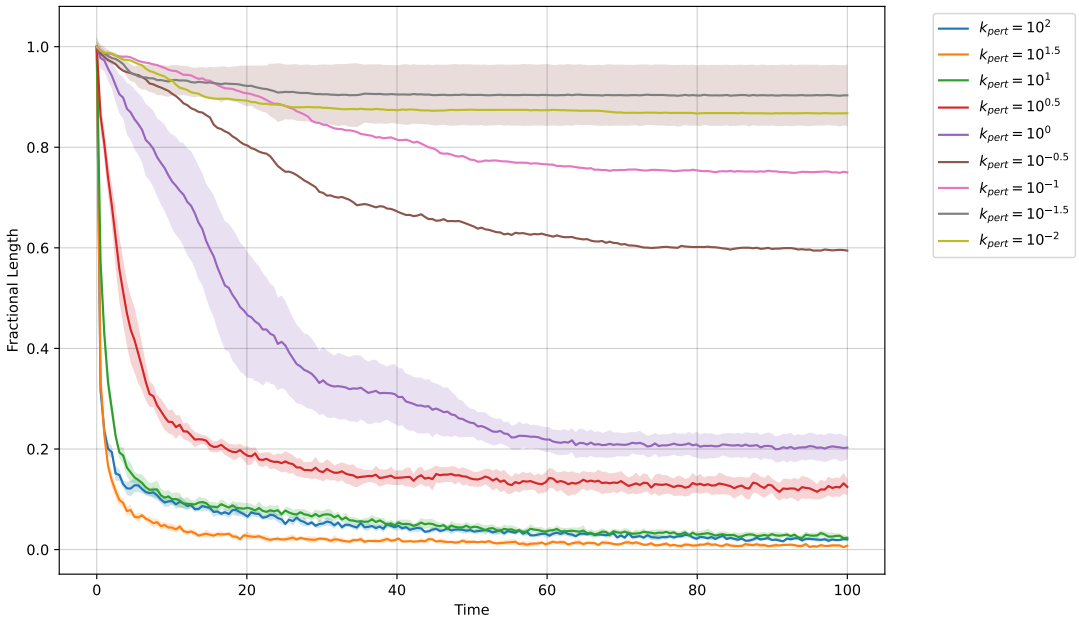
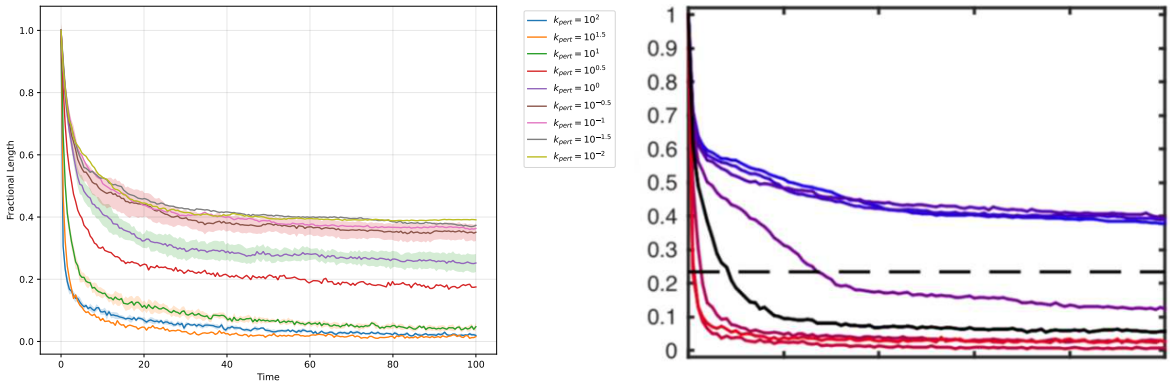


Figure 4.4: Fractional length against time.

As expected, we see that lower values of perturbation yield little to no change in fractional length, while the fractional length drops close to or below 0.2 for $k_{\text{pert}} < 1$, which is consistent with the behaviour seen from figure 4.3. We observe that if we increase the level of noise in our system, then the aggregate reaches a steady state quicker. We must note that for $k_{\text{pert}} > 10$, cells and clusters become dissociated. In this case, the low fractional length is due to the noise-term dominating as opposed to the system being well-sorted, meaning that our way of measuring well-sortedness with fractional length no longer applies for extremely large perturbations.

This behaviour agrees with the results obtained by Osborne and colleagues [9], with the only difference being that the profile of the fractional lengths suggests that they started from a square-lattice arrangement, as opposed to our hexagonal one. More precisely, it seems as though they started from a square arrangement, let the tissue reach an equilibrium with a hexagonal arrangement, and only then applied the perturbation. The simulation has also been run for the square-arrangement to test this suggestion:



(a) Our implementation with a square arrangement

(b) Osborne's and colleagues' implementation, reproduced from [9]

Figure 4.5: A comparison between our results for a square arrangement and Osborne's and colleagues' results.

Indeed, we can see that there is more similarity in appearance between the results with this implementation, while there is some structural similarity between the two results from our hexagonal arrangement[1]. We must note that in our implementation, we measure the fractional length as the number of cells of different types in contact with one another at each time. This does not take into account the overlap between cells, small gaps between different clusters in the steady state, or the angle of contact between cells. However, we have shown that with our implementation, we can obtain results that are consistent with the implementation from [9]. With this, we conclude our work in this chapter.

Discussion

Let us now review the work that we have done so far. We started by adding a noise to our model, which we have implemented into our framework. To investigate the effect of perturbation scale, we have ran the simulations for a fixed initial state while only varying the level of noise, and we deduced that cell-sorting only takes place for a specific range of k_{pert}. We have defined the fractional length, we have successfully implemented it into our framework, we have measured this for different values of k_{pert}, and we visualised our data. We have therefore successfully recapitulated the results obtained by Osborne and colleagues in [9] using our own implementation of their method. Hence, we can deduce that our implementation is successful in simulating cell-sorting according to the DAH, and that the outcome of our simulation is consistent with that previously outlined in [9]. With this, we have validated both our implementation of the model and the reproducibility of their results.

[1] See the slowly decreasing purple line from figure 4.4 and the similarly decreasing purple line in the results obtained by Osborne and colleagues from figure 4.5b .

Chapter 5

Efficient neighbourhood search

5.1 K-D Trees

Recall that the current version of our neighbourhood search has a time complexity of $O(n^2)$, meaning that if we double the size of our input, then the run time will quadruple. Instead of using this naïve comparison between all pairs of cells, we aim to implement a more efficient and a faster way of calculating the neighbourhoods. Efficient neighbourhood search is an already existing problem, and there are many well-known solutions to it. We will be using a K-Dimensional tree (K-D tree), which is a space-partitioning data structure first proposed by Bentley in 1975 [19]. In a K-D tree, each node of the tree represents data from k dimensions, and the leaves of the tree are the data-points that we are working with - in our case, this will be the location of each cell. The details of how exactly the tree is built and how the data is stored is beyond the scope of this project, but they are outlined in [20]. This data-structure provides an $O(nlog(n))$ time complexity for our neighbourhood search [21], which is significantly faster than our previous method of $O(n^2)$.

For our implementation, we shall use the `KDTrees` module of the SciPy library, which allows us to build and query the neighbourhoods of all the points in the tree [22]. We modify the previously defined `calculate_neighbourhoods` function by first building the tree, and then querying all the neighbourhoods with the `.query_ball_tree` call. This will take the following form in our framework:

```
1  def find_neighbours(cell_locations):
2      # create the K-D tree with the cell locations.
3      tree = KDTree(cell_locations)
4
5      # use the query_ball_tree method of the KDTree module to search for all points
6      # within the specified search radius of all cells.
7      neighbours = tree.query_ball_tree(tree, r_cut_off)
8      return neighbours
```

The above method returns the same list of neighbourhoods for each cell as our previous method, so the outputs of the methods are equivalent. We can use this for running our simulation notably faster, which reduces computational cost and improves the efficiency of our framework.

5.2 Improvements

With our new, faster way of calculating neighbourhoods, the question arises: what changes can we implement in order to further reduce the run-time of the simulation? In is chapter I provide a way to do this, and data collected through investigations conducted over the duration of my project will be used as the evidence supporting my claims. The following has not been previously reported in the literature.

5.2.1 Approximate neighbourhood search

We know that the constraint on the run-time of our simulation stems from searching for the neighbourhoods. As a result, a naturally emerging question is: ***could we reduce the runtime by calculating the neighbourhoods less frequently?***

Suppose we want to evaluate the neighbourhoods while skipping every n time-steps, with n being an integer ranging from 0 to 100. It is easy to assume that we can skip more time-steps to have a lower run-time. The issue with this however, is that after 100 time-steps, the neighbourhood of cell i might have changed due to cells moving into and out of it, and so we could miss interactions in the process. This is most important during the time-period in which the gradient of the fractional length curve has not yet reached a value close to 0, corresponding to a steady state (or to the dissociation of cells for $k_{\text{pert}} = 100$). Therefore, we aim to find the change in search radius to account for the cells moving out of the neighbourhood within the number of time steps skipped, depending on the level of noise in our system.

5.2.2 An upper bound

Recall that the motion of cells under our force-law resembles the diffusion of random walkers for high values of k_{pert}. Reducing the noise level introduces a damping effect to our system, which tells us that at worst, the motion of the cells in our tissue is bounded by the motion of cells in a tissue where cells move randomly. Hence, an upper bound for the expected distance travelled by a single cell can be obtained from the corresponding expected value in the case where cells move at random.

To calculate this worst-case upper bound, consider a collection of cells undergoing random motion, solely under $\mathbf{F}^{\text{rand}}$.

Let $\Delta_n \mathbf{r}_i = \mathbf{r}_i(t + n\Delta t) - \mathbf{r}_i(t + (n-1)\Delta t)$ represent the displacement of cell i in the nth two consecutive time-steps, and let $\mathbf{R}_i^{(n)} = \mathbf{r}_i(t+n\Delta t) - \mathbf{r}_i(t)$ represent the displacement of cell i after n time-steps. We wish to bound magnitude of the displacement, which corresponds to the root mean squared of the norm of $\mathbf{R}_i^{(n)}$. We can start by writing:

$$\begin{aligned}
\mathbf{R}_i^{(n)} &= \mathbf{r}_i(t + n\Delta t) - \mathbf{r}_i(t + (n-1)\Delta t) \\
&\quad + \mathbf{r}_i(t + (n-1)\Delta t) - \mathbf{r}_i(t + (n-2)\Delta t) \\
&\quad + \cdots \\
&\quad + \mathbf{r}_i(t + \Delta t) - \mathbf{r}_i(t) \\
&= \Delta_1 \mathbf{r}_i + \Delta_2 \mathbf{r}_i + \cdots + \Delta_n \mathbf{r}_i \\
&= \sum_{j=1}^{n} \Delta_j \mathbf{r}_i \,.
\end{aligned}$$

To calculate the square of the Euclidean norm squared, $\left|\left|\mathbf{R}_i^{(n)}\right|\right|^2$ denoted by $\left(R_i^{(n)}\right)^2$, we take the dot product of $\mathbf{R}_i^{(n)}$ with itself:

$$\begin{aligned}\left(R_i^{(n)}\right)^2 &= \left|\left|\mathbf{R}_i^{(n)}\right|\right|^2 \\ &= \mathbf{R}_i^{(n)} \cdot \mathbf{R}_i^{(n)} \\ &= \left(\sum_{j=1}^{n} \Delta_j \mathbf{r}_i\right) \cdot \left(\sum_{k=1}^{n} \Delta_k \mathbf{r}_i\right) \\ &= \sum_{j=1}^{n} ||\Delta_j \mathbf{r}_i||^2 \quad + \quad 2 \sum_{1 \le j < k \le n} ||\Delta_j \mathbf{r}_i||\, ||\Delta_k \mathbf{r}_i|| \cos\left(\theta_{jk}\right),\end{aligned}$$

where θ_{jk} represents the angle between the jth and the kth jump. Since the forces acting on the cells are now assumed to be random, we can deduce that the expectation of $\cos\left(\theta_{kj}\right)$ must be 0. Suppose that the expectation of the norm of the jth jump, $||\Delta_j \mathbf{r}_i||$, is some constant (this will be valid for our model, since we only consider skipping a most 100 time steps) that we shall name γ. Then, by writing the expectation as $\langle \cdot \rangle$ and using its linearity, we deduce that:

$$\left\langle \left(R_i^{(n)}\right)^2 \right\rangle = \gamma^2 n \quad \Longrightarrow \quad \sqrt{\left\langle \left(R_i^{(n)}\right)^2 \right\rangle} = \gamma\sqrt{n} \tag{5.1}$$

We have therefore deduced that the expected magnitude of the displacement after n time steps is proportional to $\sqrt{n}$. For our case, we have a damping effect induced by the attraction of cells to one another, and so it follows that the size of γ is dependent on k_{pert}.

Now, we can go back to our original problem. We make the following, important assumption: given the small time-step used in our simulation, it is highly unlikely that cells will be less than a specific distance away from each other. We take this value to be 0.9, corresponding to an overlap of 20% of the cell radius. Note that this value emerges from a physical reason, in which we assume that cells don't get so close to one another that they get flattened against each other. To find an upper-bound for $\gamma(k_{\text{pert}})$, we can use this assumption and write:

$$\begin{aligned}\gamma(k_{\text{pert}}) &= \left\langle\, ||\mathbf{r}_i(t+\Delta t) - \mathbf{r}_i(t)||\,\right\rangle \\ &= \left\langle \left|\left|\mathbf{r}_i(t) + \Delta t\left(\mathbf{F}_i + \mathbf{F}^{\text{rand}}\right) - \mathbf{r}_i(t)\right|\right| \right\rangle \qquad \text{from time-stepping} \\ &= (\Delta t)\left\langle \left|\left|\mathbf{F}_i + \mathbf{F}^{\text{rand}}\right|\right| \right\rangle \\ &\le (\Delta t)\left\langle\, ||\mathbf{F}_i||\,\right\rangle + (\Delta t)\left\langle \left|\left|\mathbf{F}^{\text{rand}}\right|\right| \right\rangle \qquad \text{by the triangle inequality}\end{aligned}$$

We then make the observation that between two consecutive time-steps, the expectation of $||\mathbf{F}_{ij}||$ is bounded by the weighted mean of the magnitude of the attractive and of the repulsive component. We can see from figure 5.1 that the neighbourhood size, $|\mathcal{N}(i)|$, is bounded by 18.

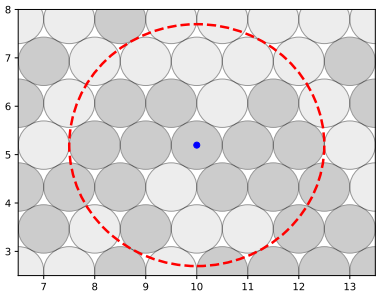


Figure 5.1: Neighbourhood of a cell, with 18 cells inside the neighbourhood.

Hence, an upper bound for this expectation is:

$$\begin{aligned}\left\langle\, ||\mathbf{F}_i||\,\right\rangle &= \left\langle \quad \Big|\Big| \sum_{j\in\mathcal{N}(i)} \mathbf{F}_{ij} \Big|\Big| \quad \right\rangle \\ &\leq \sum_{j\in\mathcal{N}(i)} \left\langle\, ||\mathbf{F}_{ij}||\,\right\rangle \\ &\leq 18\cdot 50\cdot \frac{1}{2.6}\cdot\left(\int_{0.9}^{1} -log(r)dr + \int_{1}^{2.5}(r-1)e^{-5(r-1)}dr\right) \\ &\approx 15.6\,.\end{aligned}$$

This upper bound is an major overestimate, since in most cases we will have a multitude of cells from different directions interacting with each cell, so we expect the magnitude of the force acting on it to be significantly less. Once this value is multiplied by the time-step size 0.005, the term coming from $\mathbf{F}_i$ becomes negligible.

To find the expectation of $\left|\left|\mathbf{F}^{\text{rand}}\right|\right|$, we should recall that the expectation of the magnitude of $\boldsymbol{\eta}$ is simply $\sqrt{2}$ [1]. With this information, we have that:

$$\begin{aligned}\left\langle \left|\left|\mathbf{F}^{\text{rand}}\right|\right| \right\rangle &= \sqrt{\frac{2\xi k_{\text{pert}}}{\Delta t}}\left\langle\, ||\boldsymbol{\eta}||\,\right\rangle \\ &= \sqrt{\frac{4\xi k_{\text{pert}}}{\Delta t}}\,.\end{aligned}$$

Hence, our approximation for the upper bound on γ becomes:

$$\gamma(k_{\text{pert}}) \leq \sqrt{4\xi\Delta t k_{\text{pert}}}\,,$$

and we conclude that

$$\sqrt{\left\langle \left(R_i^{(n)}\right)^2 \right\rangle} \leq \sqrt{4\xi\Delta t k_{\text{pert}} n}\,. \tag{5.2}$$

We have therefore found an upper bound to the magnitude of displacement of cells after n time-steps. Now we know that, at the very worst, we need to increase our search radius by $\sqrt{4\xi\Delta k_{\text{pert}} n}$ if we want to skip n time-steps between every neighbourhood calculation.

[1] This can be easily obtained from the definition of variance and using the linearity of expectations.

5.3 Experiments

5.3.1 Finding the change in search radius

Definition 5. *We say that the interaction of cell i with cell j is **caught** in a time-step, if the approximate neighbourhood search returns cell j, providing that cell j lies in the exact neighbourhood of cell i. We say that an interaction is **missed** if it is not caught.*

To test how good our upper-bound is for making sure that we do not miss more than 1% of interactions, we perform the following experiment:

- for every 10th multiple of n, find the change in radius required to not miss more than 1% of interactions with a 95% confidence level, and record the values in a table.
- plot (1) The data collected and (2) Our upper bound from 5.2 on the same diagram.

We can now proceed to implement this into our framework. We define the variable `time_steps_skipped`, which we gradually increase from 0 to 100 with increments of 10 time-steps, and we also define `search_radius_added`, which we adjust according to the observations we make. We record the average ratio of interactions caught by our approximation over time, which we will store in an array named `avg_ratio_caught`, which is a list of length `num_time_steps`. We then use the $O(n^2)$ method defined in 3.2.2 to calculate the exact neighbourhood in each time-step, which we compare to the approximate neighbourhoods. To avoid ambiguity, we rename our $O(n^2)$ function to `find_exact_neighbourhoods`, with the variable that will be updated by it being renamed to `exact_neighbourhoods`. We count the number of interactions caught for each cell, and we divide it by the size of their exact neighbourhood to get the ratio of interactions caught for each cell. We then store this in the variable `ratios`, which is a list of length `num_cells`. We then take the average of `ratios` over all cells, and we make this value the `time_step_idx`th entry of `avg_ratio_caught`. Once this has been done for all time-steps, we take the mean of `avg_ratio_caught` that we will store in `average_over_time`. We do this 10 times, we record each of the averages in `avg_for_each_simulation`, and we plot its mean, together with its 95% confidence interval. On the same plot, we plot the threshold of 0.99 to make sure that we do not miss more than 1% of interactions. We accept a value of `change_radius_added` if the entire confidence interval lies above the threshold, and we reject it otherwise. We then use the method of bisection, in which we run the next simulation for a change of radius that is half-way between two consecutive changes in radii that were accepted and rejected in any order. If two consecutive changes in radii were accepted or rejected, we simply decrease/increase the change in radius by their difference, respectively. The first two values of changes in radii will be 0 and 0.5 and if 0 is accepded, then we proceed no further. We repeat this process until we have reached an accuracy of 3 significant levels. By doing this, we achieve our goal of finding the minimum change in radius required in order to not miss more than 1% of interactions, with a 95% confidence, for each n. We record this value in a table and we plot it against our upper bound from 5.2. We make the following modifications to implement these our framework:

```
time_steps_skipped = # this is the chosen value of n.
search_radius_added = # this is the test value tto be accepted or rejected

def find_exact_neighbours(cell_locations):
    exact_neighbours = [] for i in range(num_cells)
    # replace neighbours with exact_neighbours from method in 3.2.2

def find_neighbours(cell_locations):
```

```
    neighbours = tree.query_ball_tree(tree, r_cut_off + radius_search_added)

def run_simulation(...):
    avg_ratio_caught = [0 for i in range(num_timesteps)]
    for time_step_idx in range(num_time_steps):
        # evaluate exact neighbourhoods each time-step.
        exact_neighbours = [] for i in range(num_cells)
        # evaluate approximate neighbourhoods skipping every n time-steps.
        if time_step_idx % (n+1) == 0:
            neighbours = []
            find_neighbours(...)
        """Code for running simulation without measuring fractional length"""
        # count interactions caught.
        ratios = 0 for i in range(num_cells)
        for i in range(num_cells):
            count = 0
            if j in exact_neighbours[i]:
                if j in neighbours[i]:
                    count += 1
            if len(exact_neighbours[i]) != 0:
                ratios[i] = count/len(exact_neighbours[i])
            else:
                ratios[i] = 1
        avg_ratio_caught[time_step_idx] = np.mean(ratios)
    average = np.mean(avg_ratio_caught)
    return average

avg_for_each_simulation = [0 for i in range 10]
def run_simulation_wrapper(arguments, i):
    """Same code as previously"""
    avg_for_each_simulation[i] = average

if __name__ == '__main__':
    threshhold = 0.99 for i in range(10)
    result = # use multiprocessing to run the simulation wrapper 10 times.
    avg_for_each_simulation = np.array(result)
    mean = np.mean(avg_for_each_simulation)
    standard_deviation = np.std(avg_for_each_simulation)
    # plot threshhold, mean, and 95% confidence interval.
```

Having implemented these changes into our framework enables us to take measurements of the change in radius required to not miss more than 1% of interactions. This can be adapted and be used in future works, to measure the required change for any chosen threshold for the percentage of interactions missed. It now remains for us to conduct the experiments to obtain data, and to store it for subsequent analysis.

5.3.2 Results

The change in search radius required to miss less than 1% of interactions was measured using the above method for $k_{\text{pert}} = 0.1$, $10^{-0.5}$, 1, 10 and 100. Each simulation was run for 10 hours corresponding to 2000 time-steps. The results are displayed in the figure below, where we have plotted our data with our upper bound:

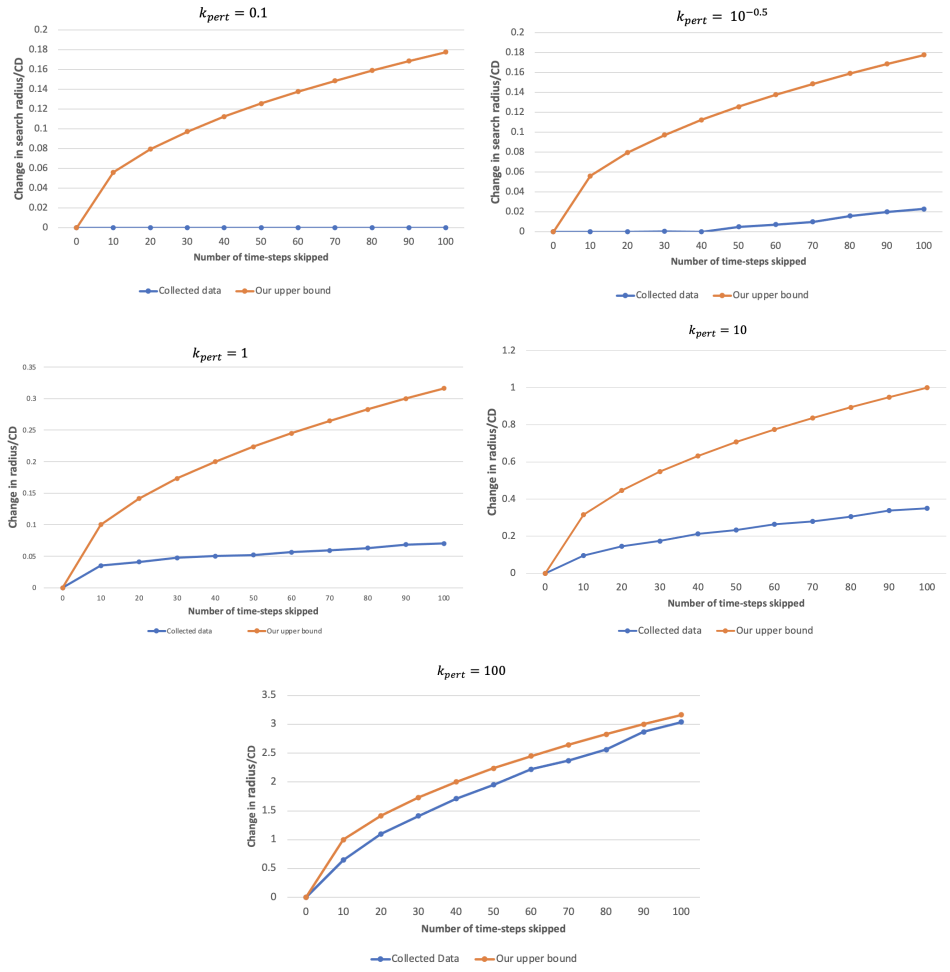


Figure 5.2: Change in radius required for missing less than 1% of interactions with a 95% confidence level, next to our upper bound.

We can now see that our upper bound from 5.2 is indeed an upper bound. As we increase k_{pert}, the blue line gets closer to the orange line, and from this we can deduce that our upper bound becomes a better fit for the measured change in radius for larger values of k_{pert}. More precisely, for any values of $k_{\text{pert}} < 0.1$, the required change in radius remains 0 until at least $n = 100$, which is due to cells not being able to move as much in 100 time-steps. For $k_{\text{pert}} = 10^{-0.5}$ this change in radius stays 0 until $n = 40$, at which point it starts increasing slowly while still staying below 0.03. For higher values of k_{pert}, we can see that the change in radius required increases to much higher values than for $k_{\text{pert}} = 1$, with its value being more than 3 for ($n = 100$, $k_{\text{pert}} = 100$), corresponding to an increase of more than 120% in the search radius size.

In the process of performing the above experiments, I noticed that the plots of our collected data appear to be a scaled version of our upper bound for $k_{\text{pert}} = 10$. Through trial and error, I found that for this specific value of k_{pert}, $\frac{1}{\sqrt{8}}$ provides a surprisingly good scaling factor to approximate our data, with there being less than 1% error between the approximation and the data. To demonstrate this, I plotted the scaled upper bound with the data collected, and this plot is shown in the figure below.

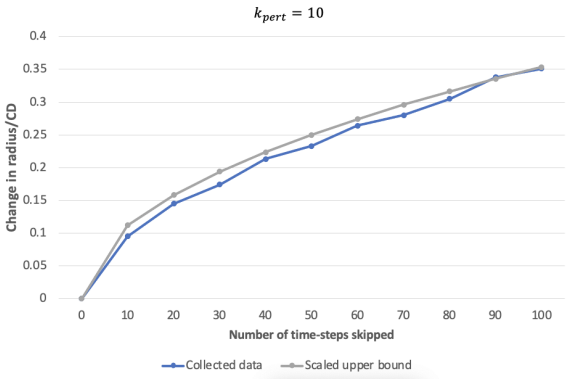


Figure 5.3: Scaled upper bound with the data collected

Discussion

Let us conclude the above conducted work. We have found a way to measure the increase in search radius required to not miss more than 1% of interactions with a 95% confidence level, which we have implemented into our framework. By conducting numerous experiments, we have obtained values for changes in search radii for a range of k_{pert}, depending on the number of timesteps skipped. We have gained valuable information, since now we know by how much we must increase the search radius if we want to skip n timesteps in the neighbourhood calculation. Future works could include taking more precise measurements for more values of k_{pert}, with larger sample sizes and for different thresholds for the interactions missed.

5.3.3 Optimal calculation frequency

Recall the motivating factor behind our investigation. We would like to find a way to reduce the run-time of the simulation in order to make it better suited for larger numbers of cells, which can in turn give better predictions of the behaviour of tissues. From the above investigation, we already know the change in search radius required for each time step, and so now we are finally ready to ask the following question: ***how many time-steps should we skip for the optimal run-time?***

The reason why this becomes a question, is that by decreasing the frequency of neighbourhood calculations, we must increase the search radius by some amount, resulting in each radius search to take longer to execute. The answer to this question is trivial for $k_{\text{pert}} < 0.1$, since the change in search radius remains zero until at least $n = 100$, meaning that ***it is always optimal to perform the neighbourhood search every 100 time-steps for*** $k_{pert} < 0.1$, at least for this domain of n. For higher values of k_{pert} , the answer is non-trivial, and so we would like to know the relationship between the simulation run-time and the number of time-steps skipped to find the optimal frequency of neighbourhood calculations. The simulation was run on a PC with an *AMD Ryzen 7 4800H Processor and 16GB DDR4 RAM* for 10 in-simulation hours, and it was timed with

varying n time-steps skipped in between each neighbourhood calculation. The data obtained in our previous investigation was used as the change in radius added to the search. For each n, three separate simulations were ran for each k_{pert}, and the mean of the run-times was plotted against n. The results obtained are shown in the figure below:

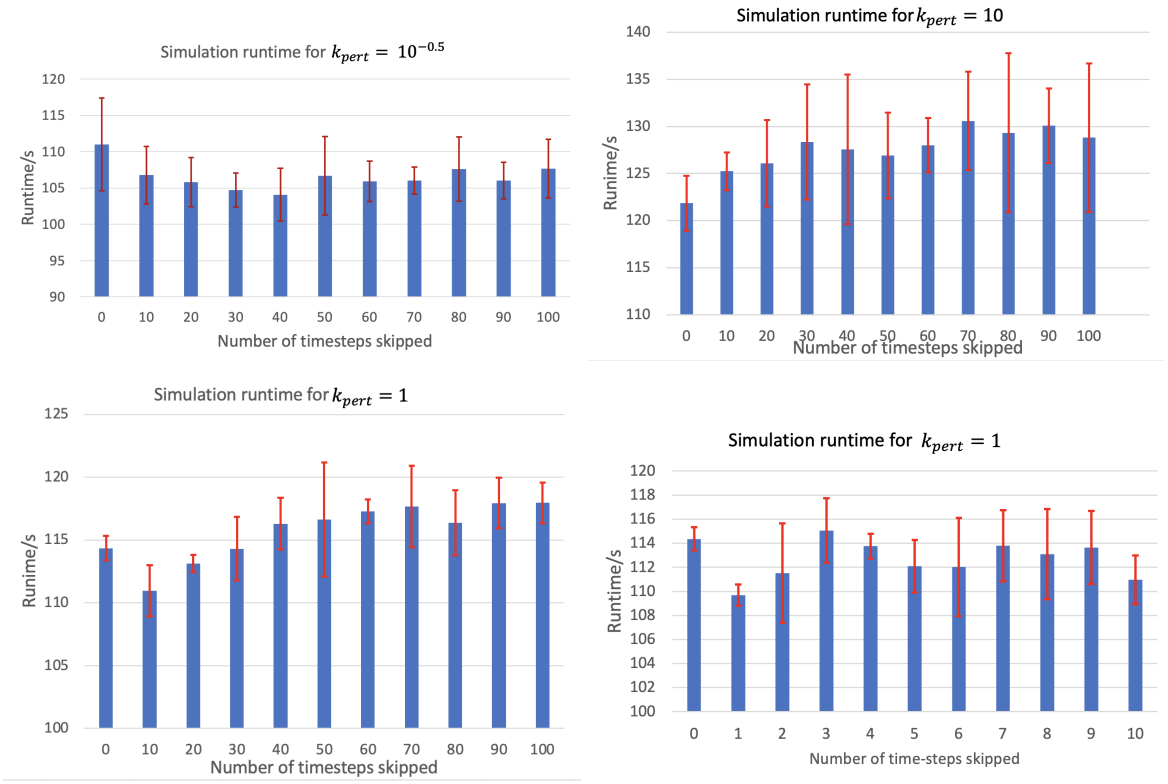


Figure 5.4: Simulation runtime against number of time-steps skipped.

We can see that for $k_{\text{pert}} = 10$, the optimal number of time-steps skipped is 0, and we know that this must hold for for $k_{\text{pert}>10}$, since the change in search radius required increases with for k_{pert}. Hence, we learned that ***it is always optimal to perform the neighbourhood search every time-step for*** $k_{pert} \geq 10$.

For $k_{\text{pert}} = 10^{-0.5}$ we see that the optimum occurs at $n = 40$, which is consistent with there being no change in radius up to $n = 40$. This suggests that for $0.1 < k_{\text{pert}}10$, we can have nontrivial optima that are between 0 and 100, which is shown for $k_{\text{pert}} = 1$, where we see a drop in the simulation run-time for $n = 10$. Having made this observation, we might want to know what happens between $n = 0$ and $n = 10$, to see whether the optimum is really 10 or not. The change in radius required has been found for this range of n, the simulations were ran for these values of n, and the results are included in figure 5.4. We see that there is another large drop in the runtime from $n = 0$ to $n = 1$ and that the runtime for $n = 1$ it is marginally less than that for $n = 10$. This provides a good candidate for the number of time-steps skipped that minimises run-time, and we can say that ***the optimum frequency is every two time-steps for*** $k_{pert} = 1$.

Hence, we have learned the optimal calculation frequency for a range of k_{pert}, we found the optima for some specific values of k_{pert}, and we have deduced that we can have non-trivial optima for the calculation frequency for $0.1 < k_{\text{pert}} < 10$. We can classify the optimum frequency depending on k_{pert} in the following manner:

k_{pert}	≤ 0.1	$0.1 < k_{\text{pert}} < 10$	≥ 10
Optimal n	100	$[0, 100]$	0

Table 5.1: Classification of the optimum frequency of neighbourhod calculations.

This concludes that the run-time can be reduced by querying the neighbourhoods less frequently for $k_{\text{pert}} < 10$, but it cannot be reduced this way for $k_{\text{pert}} \geq 10$ - something that we did not previously know. With this, we have reached the end of our work for this project.

Discussion and future works

Let us consider the different stages of cell-sorting for the values of k_{pert} where it takes place. From figure 4.4, we see that there is a time-period in which the fractional length decreases more rapidly, and there is a phase where the fractional length does not change as much, meaning that our tissue has reached a steady state where we do not expect cells to move as much as initially. The contrast between these two phases of cell sorting tells us that the change in search radius required to miss no more than 1% of interactions is in fact time-dependent. More precisely, the change in search radius could be reduced after reaching the steady state, meaning that the run-time can be even further improved. It is also worth noting that finding ways to approximate the required search radius as a function of time and of k_{pert} would be a major improvement to our current knowledge, since we could use those approximations in our search for the optimum frequencies.

Through the investigation in 5.3.2, we have found the required change in search radius for skipping n time-steps for a range of k_{pert}, and we have found the optimal calculation frequency for $k_{\text{pert}} \leq 0.1$, for $k_{\text{pert}} \geq 10$, and for some values in between. The question posed in the beginning of 5.3.3 has by no means been fully answered in this project, but we have made significant improvements to our implementation for $k_{\text{pert}} < 0.1$, for $k_{\text{pert}} = 1$ and for $k_{\text{pert}} = 0.5$, meaning that we are now one step closer to finding the optimal implementation of our model. It is also important to know that run-time cannot be reduced with less frequent neighbourhood calculations for $k_{\text{pert}} > 10$, since this allows us to focus on a much smaller range of k_{pert} values for which this optimum has yet got to be determined. Future works could include finding the optimal frequency for more values of k_{pert} in the range [0, 10], which can be made time-dependent by finding the required change in search radius for the different stages of cell sorting.

Chapter 6

Conclusion

Let us now conclude our work conducted in this project. Starting from the motivation to better understand the cell-sorting dynamics of tissues, we have used the overlapping spheres model to explain cell-sorting using the differential adhesion hypothesis. We have developed our own implementation of the model, where we have designed a framework for running simulations that can reflect the behaviour of cells undergoing this sorting process. This allows us to visualise the state of the tissue at any given time, which is important in order to know that our model reflects the real behaviour of cells in a tissue. We have incorporated a way of quantifying how well-sorted the system is over time for different levels of noise, and we have obtained results that are consistent with those outlined in [9]. Through this procedure, we have validated both our implementation of the model and the reproducibility of their results, meaning that the model is reliable and our framework is consistent.

We have improved our framework by reducing the time complexity of our simulation with the use of K-D Trees, which provides a way to perform the search in $O(nlog(n))$ time. To optimise the frequency of neighbourhood calculations, we have first found an upper bound for the required change in search radius for different configurations. We have tested how good this upper bound is by implementing a method to measure the ratio of interactions caught into our framework. Then through conducting experiments, we have measured the required change in search radius for different perturbations to the system, and we learned that our upper bound is indeed an upper bound, which becomes a better approximation for higher noise levels. To take our investigation even further, we posed the question about the trade-off between decreasing the frequency of neighbourhood calculations while using an increased search radius, and the run-time. We have found the optimum frequency of neighbourhood calculation that results in the lowest run-time for a range of perturbation levels, and we were able to classify these based on the level of noise in our system. We have also learned that with the methods used in our investigation, the run-time cannot be improved if the noise level is beyond $k_{\text{pert}} = 10$, meaning that we have significantly reduced the range of values for which future work is to be done. At the end of our work, ways to improve on our methods have been provided, and suggestions for extensions of our work have been proposed.

Seeking the optimal implementation plays a crucial role in computational scientific modelling, since this allows the handling of significantly more data, with lower run-times and with lower cost. With more efficiency and better performance, we get closer to understanding the problems in current scientific research, whose aim is to battle problems that have a major impact on our lives as humans. It then remains for us to continuously keep striving to make improvements to our models and to their implementations in the future.

Appendix A

A.1

The script of simulation used to obtain figure 4.3 is included below.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Circle
from matplotlib.collections import PatchCollection
from matplotlib.colors import ListedColormap
from scipy.spatial import KDTree
import time
from multiprocessing import Pool


"""Define some global variables """

num_cells = 400
r_cell = 0.5
r_cut_off = 2.5
k_c = 5
# base level of perturbation
xi = 0.05
# perturbation scale
k_pert = 10**(-0.5)

mu = 50

mu_het = 5

simulation_finish_time = 100

"""When will our simulation settle"""
time_settle = 100
def settle():
    global k_pert
    k_pert = 0

time_step_size = 0.005
num_time_steps = int(np.ceil(simulation_finish_time / time_step_size))
num_simulations = 1
```

```
37
38  """Defining my color map, this can be extended to more types"""
39  colors = ['green', 'purple']
40  cell_colors = ListedColormap(colors)
41
42  def delta(x, y):
43      if x == y:
44          return mu
45      else:
46          return mu_het
47
48  def find_neighbours(cell_locations):
49      tree = KDTree(cell_locations)
50      neighbours = tree.query_ball_tree(tree, r_cut_off, eps=0)
51      return neighbours
52
53
54  def setup_simulation(cell_types, cell_locations):
55      """
56      This function will set up the initial state of the simulation.
57      """
58      height = int(np.ceil(np.sqrt(num_cells * np.sqrt(3) / 2))) + 1
59      width = int(np.ceil(num_cells / height))
60      # Loop over the rows and columns of the hexagonal grid
61      for i in range(height):
62          for j in range(width):
63              index = i * width + j
64              if index >= num_cells:
65                  break
66              x = (2 * j + i % 2) * r_cell
67              y = i * np.sqrt(3) / 2 * r_cell
68              cell_locations[index, 0] = x
69              cell_locations[index, 1] = 2 * y
70      # Assign cell types
71      num_cells_per_type = num_cells // 2  # assuming even number of cells
72      cell_types[:num_cells_per_type] = 0
73      cell_types[num_cells_per_type:] = 1
74      np.random.shuffle(cell_types)
75
76  def calculate_forces_each_timestep(cell_locations, cell_forces, cell_types, neighbours,
77                                     fractional_lengths, Delta, t):
78      for i in range(num_cells):
79          for j in neighbours[i]:
80              if i > j:
81                  r_ij = cell_locations[j, :] - cell_locations[i, :]
82                  if np.linalg.norm(r_ij) < 1:
83                      f = (mu * ((r_ij) / np.linalg.norm(r_ij)) *
84                           np.log(1 + (np.linalg.norm(r_ij) - 1)))
85                      cell_forces[i, :] += f
86                      cell_forces[j, :] += -f
87
88                  elif 1 <= np.linalg.norm(r_ij) <= 2.5:
89                      f = ((Delta[i,j]) * (
```

```
                                np.linalg.norm(r_ij) - 1) *
                            ((r_ij) / np.linalg.norm(r_ij)) * np.exp(-k_c * (np.linalg.norm(r_ij) - 1))
                            )
                    cell_forces[i, :] += f
                    cell_forces[j, :] += -f
        cell_forces[i, :] += np.sqrt(  2 * k_pert * xi / time_step_size) * np.random.randn(2)
    return fractional_lengths

Delta = np.zeros((num_cells, num_cells), dtype=np.int32)

def calculate_Delta(cell_types):
    for i in range(num_cells):
        for j in range(num_cells):
            Delta[i, j] = delta(cell_types[i], cell_types[j])
    return Delta

def run_simulation(cell_locations, cell_forces, cell_types):
    print("Running simulation...")
    fractional_lengths = np.zeros(num_time_steps)
    calculate_Delta(cell_types)
    cell_locations = cell_locations + time_step_size * cell_forces
    cell_forces = np.zeros_like(cell_locations)

    fig, ax = plt.subplots()
    scat = None
    filename = "k_pert=1-t=_{:04d}.png"
    for time_step_idx in range(num_time_steps):
        # keep track of time
        time_now = time_step_idx * time_step_size
        if time_step_idx % 200 == 0:
            print(time_now)
        print(time_step_idx)
        cell_locations = cell_locations + time_step_size * cell_forces
        # Reset cell forces
        cell_forces = np.zeros_like(cell_locations)
        neighbours = find_neighbours(cell_locations)
        calculate_forces_each_timestep(cell_locations, cell_forces, cell_types,
                                       neighbours, fractional_lengths, Delta, time_step_idx)

        # plot graph for t = 10,20,30, ...
        if time_step_idx in [0, 3999, 7999, 11999, 15999, 19999]:
            # Create a list of circles with the same color as each cell
            circles = [Circle((cell_locations[j, 0], cell_locations[j, 1]), 0.5)
                       for j in range(num_cells)]
            colors = cell_types
            if scat is None:
                scat = PatchCollection(circles, cmap=cell_colors, alpha=0.4, edgecolors='black')
                scat.set_array(colors)
                ax.add_collection(scat)
                plt.gca().set_xlim(-5, 25)
                plt.gca().set_ylim(-5, 25)
            else:
                scat.set_paths(circles)
```

```
143
144             # Update the plot for each frame
145             scat.set_offsets(cell_locations)
146             ax.set_title(f"Time: {round(time_now, 1)}")
147             fig.savefig(filename.format(time_step_idx), dpi=350)
148             plt.close(fig)
149         # bring the perturbation down to see the final state of the system
150         if time_now == time_settle:
151             settle()
152
153 def run_simulation_wrapper(i):
154     k_pert = 10**(0)
155     print("We're on simulation number", i+1)
156     cell_locations = np.zeros((num_cells, 2), dtype=np.double)
157     cell_types = np.zeros(num_cells, dtype=np.int32)
158     cell_forces = np.zeros_like(cell_locations)
159     start_time = time.time()
160     setup_simulation(cell_types, cell_locations)
161     run_simulation(cell_locations, cell_forces, cell_types)
162     print("--- %s seconds ---" % (time.time() - start_time))
163
164 if __name__ == '__main__':
165     with Pool() as pool:
166         result = pool.map(run_simulation_wrapper, range(num_simulations))
167     plt.show()
```

A.2

To obtain the plots in figure 4.3, the following method has been used to generate an initial state which has been saved:

```
"""same as above, but stop after defining setup_simulation()."""
if __name__ == '__main__':
    cell_locations = np.zeros((num_cells, 2) dtype=np.double)
    cell_types = np.zeros(num_cells, dtype=np.int32)
    setup_simulation(cell_locations, cell_types)
    with open('initial_state.pkl', 'wb') as f:
        pickle.dump(data, f)
```

Running this script in a separate file generates the desired initial state, which we can use to see how the perturbation affects the final state of the simulation:

```
"""Same script as above, but we modify:"""
filename = "simulation_k_pert=_{:04d}.png"
def run_simulation(...):
    # script for time-stepping
    # (...)
    # plot end state
    ax.set_title(f"$k_{{pert}}$= {k_pert}, Time = 100 hours")
    fig.savefig(filename.format(int(rounf(np.log10(k_pert),0))), dpi=350)
```

```
def run_simulation_wrapper(i):
    k_pert = 10**(i)
    with open('initial_state.pkl', 'rb') as f:
        data = pickle.load(f)
    cell_locations, cell_types = data
    """same as above"""
if __name__ == '__main__':
    with Pool() as pool:
        result = pool.map(run_simulation_wrapper, range(-2,2))
        # run the simulation again for range(2,3) and simulation_finish_time = 3.
```

Another simulation was run, and the results are displayed below (Perhaps easier to look at due to less saturated colours being used, but these images are not vector graphics. This means that if we zoom in on the below images on a computer, then the image becomes pixelated. This is not the case for the figures in section 4, since they are stored as vector graphics and they stay smooth upon magnifying.):

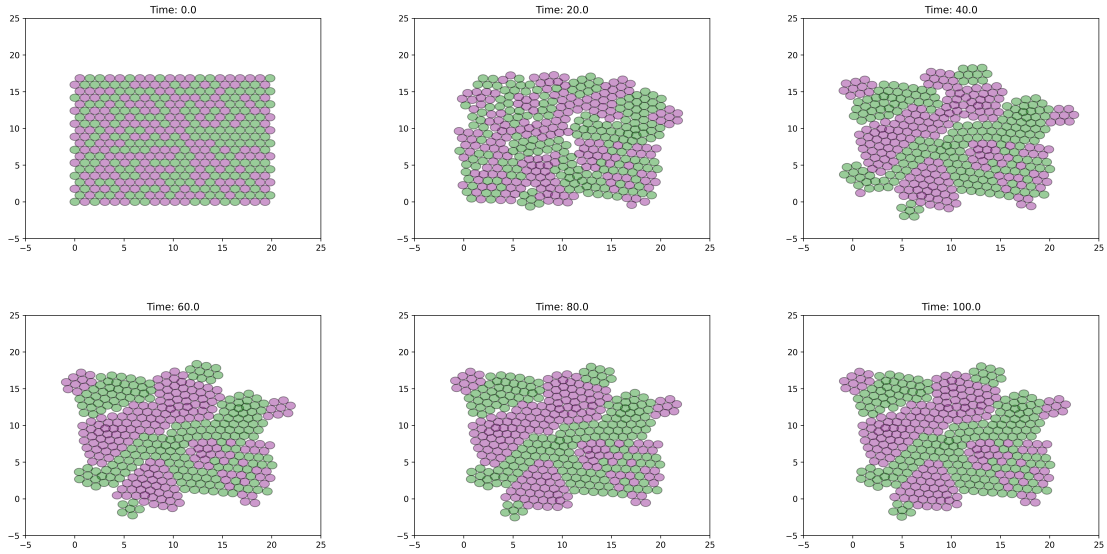


Figure A.2: Evolution of the tissue over time with $k_{\text{pert}} = 1$

.

Second simulation displaying the effect of perturbation scale on the tissue

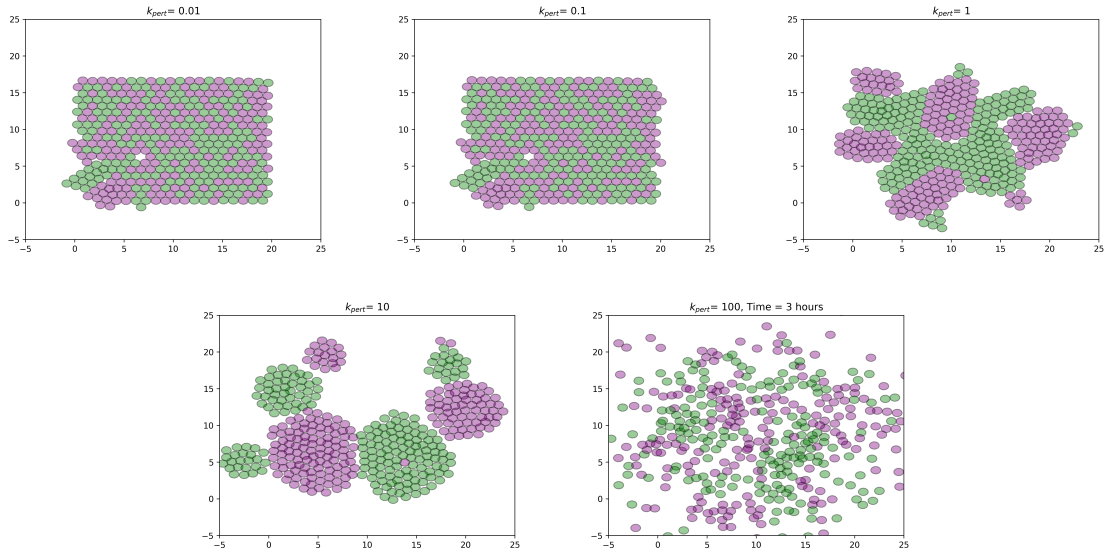


Figure A.3: Evolution of the tissue over time with $k_{\text{pert}} = 1$

.

Bibliography

[1] History.com Editors. Neolithic revolution. *HISTORY*, 2023. URL https://www.history.com/topics/pre-history/neolithic-revolution.

[2] Leroy Hood and Lee Rowen. The human genome project: Big science transforms biology and medicine. *Genome Medicine*, 5:79, 2013. ISSN 1756-994X. doi: 10.1186/gm483. URL https://doi.org/10.1186/gm483.

[3] Steinberg Malcolm S Foty, Ramsey A. The differential adhesion hypothesis: a direct evaluation. *Developmental Biology*, 278(1):255–263, 2005. ISSN 0012-1606. doi: 10.1016/j.ydbio.2004.11.012.

[4] Fergus R Cooper. *A mathematical and computational framework for modelling epithelial cell morphodynamics*. PhD thesis, University of Oxford, 2018.

[5] James M Osborne, Miguel O Bernabeu, Matteo Bruna, Ben Calderhead, Jonathan Cooper, Neil Dalchau, Sarah-Jane Dunn, Alexander G Fletcher, Marcus Freeman, Derek Groen, et al. Ten simple rules for effective computational research. *PLoS Comput Biol*, 10(3):e1003506, 2014. doi: 10.1371/journal.pcbi.1003506.

[6] S. F. Gabby Krens and Carl-Philipp Heisenberg. Cell sorting in development. In Michel Labouesse, editor, *Current Topics in Developmental Biology*, volume 95, pages 189–213. Academic Press, 2011. ISBN 9780123850652. doi: 10.1016/B978-0-12-385065-2.00006-2.

[7] Jean-Léon Maître and Carl-Philipp Heisenberg. Three functions of cadherins in cell adhesion. *Current Biology*, 23(14):R626–R633, 2013. doi: 10.1016/j.cub.2013.06.019.

[8] Mechanobiology Institute RCE. What is mechanosignaling? https://www.mechanobio.info/what-is-mechanosignaling/, n.d. Accessed: February 23, 2023.

[9] James M Osborne, Alexander G Fletcher, Joe M Pitt-Francis, Philip K Maini, and David J Gavaghan. Comparing individual-based approaches to modelling the self-organization of multicellular tissues. *PLoS Comput Biol*, 13(2):e1005387, 2017. doi: 10.1371/journal.pcbi.1005387.

[10] Robert Moore, Kathy Q Cai, Diogo O Escudero, and Xiang-Xi Xu. Cell adhesive affinity does not dictate primitive endoderm segregation and positioning during murine embryoid body formation. *Genesis: The Journal of Genetics and Development*, 47(3):185–195, 2009. doi: 10.1002/dvg.20536.

[11] HP Greenspan. Models for the growth of a solid tumour by diffusion. *Stud. Appl. Math*, 52:317–344, 1972.

[12] Zhirong Liu and Philipp Keller. Emerging imaging and genomic tools for developmental systems biology. *Dev. Cell*, 36(6):597–610, 2016. doi: 10.1016/j.devcel.2016.03.012.

[13] Pras Pathmanathan Pathmanathan, Jonathan Cooper, Alexander Fletcher, Gary Mirams, Philip Murray, James Osborne, Achim Walter, Sandra Chapman, Alan Garny, and David Gavaghan. A computational study of discrete mechanical tissue models. *Phys Biol*, 6:036001, 2009. doi: 10.1088/1478-3975/6/3/036001.

[14] Fahimeh Alisafaei, Xingyu Chen, Tamara Leahy, Paul A. Janmey, and Vivek B. Shenoy. Long-range mechanical signaling in biological systems. *Soft Matter*, 17(2):241–253, 2021. doi: 10.1039/d0sm01442g. Erratum in: Soft Matter. 2020 Dec 8;.

[15] NumPy Contributors. Numpy: The fundamental package for scientific computing with Python, 2021. URL https://numpy.org/doc/stable/index.html.

[16] SciPy. SciPy documentation. https://docs.scipy.org/doc/scipy/index.html, 2023. [Online; accessed 23-February-2023].

[17] Matplotlib Development Team. Matplotlib. https://matplotlib.org/stable/index.html, accessed February 23, 2023.

[18] Holden Lee and Jeffrey Wong. Math 361s lecture notes numerical solution of odes, 2020.

[19] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975. ISSN 0001-0782. doi: 10.1145/361002.361007.

[20] Hemant M. Kakde. Range searching using kd tree. Technical Report, 8 2005. URL https://users.cs.utah.edu/~lifeifei/cs6931/kdtree.pdf.

[21] Franco P Preparata and Michael Ian Shamos. *Computational Geometry*. Springer-Verlag, 1985.

[22] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, and Paul van Mulbregt. scipy.spatial.kdtree. https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.html, 2020. [Online; accessed 23-February-2023].