Agent-Based Simulation of Angiogenesis: Exploring Cellular Dynamics and the Role of Protrusions in Pattern Formation

BSc Computer Science

Final Project Report

Author: Ao Zhang

Supervisor: Dr Katie Bentley

Student ID: 20008509

November 17, 2023

Abstract

Collective cell communication is essential for the formation of complex tissue patterns during development. In this study, we propose an agent-based model simulation to investigate how irregular, widely branched cells affect pattern formation in developing blood vessels. We analyse the tissue-wide network architecture using graph theoretical properties and simulate the dynamical consequences for the Dll4-Notch signaling pathway. The pathway is a complex system of signaling molecules involved in cell-cell communication and the regulation of cell fate during development, which establishes a salt-and-pepper pattern of cell signaling that decides where branches in the network will form. Our simulation aims to identify how cell shape influence pattern creation during angiogenesis and gain a deeper understanding of the underlying mechanisms of collective cell communication. We also investigate how pattern dynamics change when cells have dynamically extending and retracting protrusions. Our agent-based model overcomes some of the limitations of existing models and provides insights into the interactions during pattern formation. The simulation framework provides a useful tool for investigating the complex cellular behavior during pattern formation, and can be extended to investigate other biological processes.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Ao Zhang

November 17, 2023

Acknowledgements

I would like to express my deepest gratitude and appreciation to my principal investigator, Dr. Katie Bently, for her invaluable assistance and guidance throughout this project. This project would not have been a success without her knowledge, insightful feedback, and unwavering dedication.

I would like to extend my sincerest thanks and appreciation to Leo-Laurenz Zeitler. His patience, understanding, and his willingness to provide guidance and support throughout the various stages of this project. His kindness, generosity, and his unwavering commitment to our shared goals have been truly invaluable.

Contents

Chapter 1

Introduction

During embryonic development, cells must coordinate to form complex tissue patterns. This involves the formation of organized groups of cells, with leader and follower cells that have different abilities to protrude and exert traction. These cells can also sense and respond to chemical signals and mechanical cues through cell-cell adhesions that can be either temporary or permanent (Scarpa & Mayor, 2016).

This requires endothelial cells (ECs), to communicate with each other. A specific pattern of cell signalling, like a checkerboard (See Figure 1.1), is shown during this process (Perkins, Benzinger, Arcak, & Khammash, 2020).

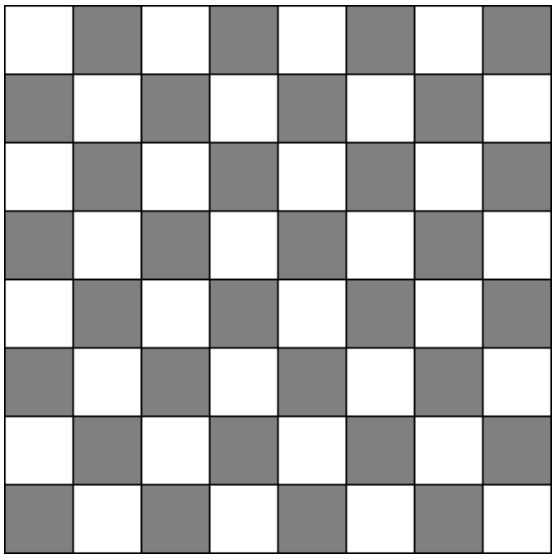

Figure 1.1: Checkerboard pattern (salt-and-pepper pattern).

However, recent observations have revealed that some cells have protrusions that could establish connections to distal cells than previously thought. How pattern dynamics change

with prongy shaped cells is still poorly understood and requires further investigation.

The existing Python code base for simulating the collective cell communication network lacks spatial information of cells and has limitations in its design for simulating cell shapes and interactions, which may not accurately represent the complex and diverse nature of cells in vivo. To overcome this limitation, a new simulation framework is essential to support simulating communication network for cells. Using a agent-based model could improve the flexibility and capabilities of the simulation when adding new features.

1.1 Aim and Objectives

1.1.1 Research Objectives

The primary objective of this project is to create an agent-based model that simulates how cells communicate during the formation of patterns in developing blood vessels. It aims to address some of the limitations of existing models for investigating collective cell communication and angiogenesis, such as the lack of spatial resolution, the oversimplification of cell behavior and interactions. The model will provide a more detailed understanding of how molecular signals interact to direct tissue patterning, especially in the context of the Dll4-Notch signaling pathway. The model will be used to explore how different parameters and conditions affect the emergence and maintenance of a salt-and-pepper pattern of cell signalling among ECs, and how this pattern influences the morphology. The model will also be validated against observations from previous studies on angiogenesis and Dll4-Notch signaling. The project will contribute to advancing the knowledge and methods for studying complex biological systems using computational modeling and simulation techniques.

1.1.2 Research Questions

- How does the agent-based simulation of collective cells improve upon existing models by obtaining the spacial information of cells and how do different parameters, including spatial and temporal aspects, as well as initial conditions, impact the pattern formation.
- How does the inclusion of prongy shaped cells with dynamic protrusions that can extend and retract, as opposed to solely relying on neighbouring cell connections, influence the pattern dynamics in the agent-based simulation of angiogenesis?
- How do network properties such as the clustering coefficient and characteristic path length

change when comparing the scenarios with and without protrusions from the cells.

1.1.3 Research Contributions

The research contributions of this project are multifaceted and encompass various aspects of agent-based modeling, pattern formation, angiogenesis, and computational biology. The main contributions can be summarized as follows:

- Development of a comprehensive agent-based model for simulating collective cell communication during angiogenesis and pattern formation in developing blood vessels. This model addresses the limitations of existing models by incorporating spatial resolution and more realistic representations of cell behavior and interactions.
- Exploration of the impact of star-shaped cells with dynamic protrusions on the pattern dynamics and network properties of the agent-based model. This investigation enhances the understanding of the role of these unique cellular structures in angiogenesis and blood vessel development.
- Extension and refinement of the agent-based model to incorporate additional features, such as varying cell sizes and molecular cues involved in angiogenesis, enabling a more accurate and detailed representation of blood vessel development in silico.
- Addressing the computational challenges and limitations associated with the agent-based simulation of angiogenesis. This research proposes strategies and approaches to improve the efficiency and accuracy of the simulation, paving the way for more advanced and large-scale modeling of complex biological systems.

This project will advance the current understanding of angiogenesis and pattern formation in developing blood vessels. The agent-based model developed in this project has the potential to serve as a powerful tool for investigating complex biological systems, facilitating novel insights and discoveries in the field of computational biology.

Chapter 2

Background

2.1 Biological Background

Angiogenesis is the process of new blood vessel formation from pre-existing vessels, which plays a crucial role in the growth and maintenance of tissues, as well as in wound healing and pathological conditions such as tumor growth (Carmeliet & Jain, 2011). The process is initiated by the activation of endothelial cells (EC), which respond to various molecular signals, including vascular endothelial growth factor (VEGF) and delta-like ligand 4 (Dll4) (Gerhardt et al., 2003). The Dll4-Notch signaling pathway is a highly conserved method for cell-cell communication, which is believed to happening by direct cell contact, is critical for the regulation of angiogenesis and blood vessel development (Geudens & Gerhardt, 2011). However, recent observations of prongy shaped cells may affect our understanding of how may cells will be neighbouring during angiogenesis and how pattern dynamic changes.

2.1.1 Endothelial Cells

ECs are involved in angiogenesis, where they respond to various molecular signals and undergo proliferation, migration, and differentiation to form new blood vessels.

ECs can display heterogeneity in terms of morphology, function, and molecular markers, depending on their location within the vascular tree and the specific tissue in which they are found (Hennigs, Matuszcak, Trepel, & Korbelin, 2021). Tip cells are specialized ECs that extend filopodia to sense and follow angiogenic cues, such as vascular endothelial growth factor (VEGF), and guide the direction of sprout growth (Gerhardt et al., 2003).

This process is essential for the growth and development of tissues and organs, but it can

also contribute to the formation of tumors. Disorder of ECs are associated with a number of diseases, including hypertension, atherosclerosis, and diabetes (Rajendran et al., 2013). Thus, understanding the function of ECs and how they can be maintained in a healthy state is an important area of research in the field of cardiovascular biology.

Recent observations of prongy shaped cells, which possess dynamic protrusions that can extend and retract to establish connections with distal cells, they could reach backwards into the preexisting network rather than in the tissue, both suggest that our understanding of EC behavior during angiogenesis may need to be refined. The presence of these cells and their unique morphological features may have significant implications for pattern dynamics, cell-cell communication, and the overall organization of ECs within developing blood vessels. Investigating the role of these star-shaped cells in the context of an agent-based model will provide valuable insights into their function and potential impact on angiogenesis and blood vessel development.

2.1.2 Dll4-Notch Signal Pathway

The Dll4-Notch signaling pathway is a complex process that plays a critical role in the regulation of angiogenesis and blood vessel development. This pathway is involved in the coordination of cell behavior and fate decisions among adjacent cells, ensuring proper vascular network formation and organization (Jakobsson et al., 2010; Ubezio et al., 2016; Suchting et al., 2007).

During angiogenesis, the Dll4-Notch signaling pathway plays a crucial role in establishing a salt-and-pepper pattern of cell signalling among ECs. This pattern determines the positions of future blood vessel branches and ensures that they are spaced correctly within the developing vascular network. Disruptions to the Dll4-Notch signaling pathway can lead to excessive, disorganized vessel sprouting (Hellstrom et al., 2007).

The Dll4-Notch pathway could be defined with the following rules (Zakirov et al., 2021):

- A constant influx of the vascular endothelial growth factor(VEGF) increases the activity of vascular endothelial growth factor receptor (VEGFR) on the surface of the cell.
- With a transcriptional delay, the gene expression of Dll4 is upregulated to the activity of VEGFR and the capacity of the cell to produce Dll4.
- Dll4 is sent through the network of cells by direct cell-cell contact, and is split equally between all connections.

- The activity of Notch, the receptor for Dll4, increases in proportion to the sum of all incoming Dll4.
- The Notch activated cells will produce less VEGFR protein after a time delay.
- for Dll4 and VEGFR: If no new VEGF is perceived, VEGFR remain inactive and no new transcription of new Dll4 is induced.

The inclusion of the Dll4-Notch signaling pathway in an agent-based simulation of angiogenesis will allow us to investigate and visualize the temporal endothelium of angiogenesis. We could use this pathway to see how different parameters and conditions influence stability of the salt-and-pepper pattern formation, as well as the impact of prongy shaped cells with dynamic protrusions on the pattern dynamic. Understanding the role of the Dll4-Notch pathway in angiogenesis and its interactions with other molecular cues and cellular behaviors will provide valuable insights into the complex processes underlying blood vessel development and tissue patterning.

2.2 Mathematical Background

Mathematical approaches play a crucial role in understanding and analyzing complex biological systems. These approaches enable researchers to capture the underlying principles governing the system, providing valuable insights into the mechanisms driving pattern formation and cellular behavior. In this study, we focus on using two key mathematical concepts:

Voronoi diagrams provide a powerful tool for segmenting space into equal region based on random seeds, which are fundamental features of many biological systems. The network properties, such as clustering coefficient and average path length, are important measures for analyzing small world networks, which are critical for investigating communication networks between cells. By leveraging these mathematical ideas, we could investigate their implications on pattern formation during angiogenesis.

2.2.1 Voronoi Diagram

Voronoi diagrams are used to divide a space into regions depending on the distances between a group of points, also known as seeds or sites. Each region, known as a Voronoi cell, is composed of all points in the space that are closer to a particular seed point than any other seed point (See Figure 2.1). Voronoi diagrams have been widely employed in various fields, including computational geometry, spatial analysis, and biological modeling (Aurenhammer, 1991).

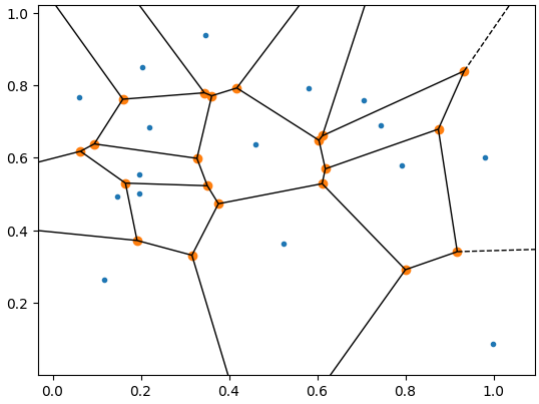

Figure 2.1: A Voronoi Diagram.

In this study, we use Voronoi diagrams to model the cells involved in angiogenesis. By incorporating Voronoi diagrams into the simulation, it is possible to investigate the role of spatial information, the presence of irregular shaped cells with dynamic protrusions along the ridges and the overall stability of the salt-and-pepper pattern formation during angiogenesis.

2.2.2 Small-world Network

The small-world network is a mathematical graph type that exhibit both high clustering coefficient and short characteristic path lengths, reflecting the properties of many real-world networks, including biological systems (Watts & Strogatz, 1998). By analyzing the properties of the cell communication network, we compare the properties that we find to small-world networks and this allows us to make conclusions about the EC network which could also provides insights into the robustness of the EC communication network.

Several network measurements can be used to characterize the small-world properties of the cell communication network, including:

Clustering coefficient: A measure of the degree to which nodes in the network tend to cluster together, forming tightly connected groups. In the context of angiogenesis, a small-world network has a high clustering coefficient and this may indicate strong local interactions between ECs, promoting coordinated behavior and pattern formation.

Characteristic path length: The average shortest path length between all pairs of nodes in the network. A short characteristic path length is observed in small-world networks and it could suggests efficient communication and signal propagation between ECs, allowing for rapid responses to changes in the microenvironment or molecular cues.

By implementing the network model by our biological assumptions and observing the net-

work properties, it is possible to investigate the type of the network and how different parameters, and initial conditions affect the connectivity between cells. Furthermore, the impact of prongy shaped cells with dynamic protrusions on the network properties can also be explored.

2.3 Computational Background

Computational models have become increasingly popular for investigating complex biological systems, including angiogenesis. Agent-based models, in particular, offer a powerful framework for simulating individual cell behavior and interactions, as well as capturing the emergent properties of the system as a whole.

For example, OpenABM-Covid19 is an agent-based simulation of the SARS-CoV-2 epidemic that includes detailed age-stratification and realistic social networks. The model can be used to evaluate the effectiveness of non-pharmaceutical interventions, such as contact tracing and vaccination programs, which allows scientists and policymakers to explore different interventions and compare their effectiveness in suppressing the COVID-19 epidemic (Hinch et al., 2021).

However, agent-based models can be computationally intensive and challenging to validate. To improve the performance of our model, we employ Just-In-Time (JIT) compilation and vectorization techniques, which can significantly enhance computational efficiency, enabling the investigation of larger systems and more complex scenarios.

2.3.1 Agent-Based Modelling

Existing models for investigating pattern formation during the development of tissue or vessels, including partial differential equation based models (Collier, Monk, Maini, & Lewis, 1996), cellular Potts models (Hirashima, Rens, & Merks, 2017), and vertex models (Alt, Ganguly, & Salbreux, 2017), among others, have advantages and limitations as follows:

- Partial differential equation based models can capture continuous changes in concentrations or distributions of signaling molecules, but generally focus on a few key variables and processes, which may not capture the full complexity of cellular interactions during pattern formation.
- Cellular Potts models can be computationally expensive, limiting the size of the systems they can simulate, especially for long time scales or large tissue sizes. However they can easily model cellular processes such as changing shape, dividing etc.

- Vertex model can explicitly incorporate cell mechanics and forces, therefore they are effective in forecasting the shapes of cells within the tissue. But it is not quite suitable for our project which is focusing on simulating the protrusions.

Our proposed agent-based model aims to address these limitations and provide a more comprehensive and adaptable framework for investigating complex tissue patterns during angiogenesis. The advantages of our agent-based model include:

- High flexibility: Agent-based models can easily incorporate multiple factors and processes, such as cell behavior, signaling pathways, and spatial information, thus enabling a more accurate representation of the underlying biological system.
- Scalability: The computational complexity of agent-based models can be adjusted according to the available resources, allowing for the study of larger tissue sizes or longer time scales without compromising the accuracy of the simulation.
- Modularity: The modular nature of agent-based models facilitates the integration of new components or the modification of existing components, enabling researchers to update or refine the model without huge effort.

Despite these advantages, there are some potential disadvantages associated with agent-based modeling that need to be considered:

- Computational cost: Depending on the complexity of the model and the number of agents, agent-based simulations can be computationally intensive, particularly for large-scale simulations.
- Validation: Validating agent-based models can be challenging due to the complexity of the biological systems they represent and the potential lack of experimental data for comparison.

Our proposed agent-based simulation has the potential to overcome some of the limitations of existing models and provide a more comprehensive and adaptable framework for investigating complex tissue patterns during angiogenesis. By incorporating key mathematical concepts such as Voronoi diagrams and small-world network properties, our model will enable the investigation of spatial information, irregular cell shapes, and cell communication networks, ultimately enhancing our understanding of the mechanisms underlying angiogenesis and pattern formation.

2.3.2 JIT and Vectorization

One way to improve the performance of the agent-based model is to use Just-In-Time (JIT) compilation and vectorization methods. These methods can enhance the computational speed of the model, allowing for the simulation of larger tissue and longer time scales.

JIT compilation is a method that optimizes the execution of a program during runtime. By compiling parts of the code that are frequently used or computationally intensive, JIT compilation can significantly speed up the simulation. There are several JIT compilers available for various programming languages, such as Numba for Python (Lam, Pitrou, & Seibert, 2015), which is the one we are using in our project.

Vectorization is another method that optimizes computational speed by using the parallel processing capabilities of modern CPUs and GPUs. Instead of performing calculations on individual elements of an array or data structure, vectorization allows for the simultaneous processing of multiple data elements using Single Instruction Multiple Data (SIMD) operations. In the context of the agent-based model, vectorization can be applied to various aspects of the simulation, such as updating cell properties, or calculating communication network nodes and edges. Using vectorization can further improve the performance of the model, allowing for the exploration of more complex systems and scenarios.

By applying JIT compilation and vectorization methods to the agent-based model of angiogenesis, we can greatly increase its computational efficiency. This will enable the investigation of larger systems and more complex scenarios, providing valuable insights into the role of the Dll4-Notch signaling pathway, prongy shaped cells, and other factors in the regulation of angiogenesis.

Chapter 3

Requirements

3.1 Agent Requirement

- Agents must contain spatial information, such as their position in the environment, in order to accurately represent their behavior and interactions. This could be important for understanding how the agents communicate with each other.
- Agents must be able to observe their environment and react to changes in it, such as signals from other agents or changes in their physical surroundings.
- Agents must be able to share their protein activity information with other agents, through mechanisms such as cell-cell communication or connecting to other cells via protrusion or a part of a protrusion.
- Agents must be able to work in a number of configurations so that diverse simulations can be conducted. This could include changing the shape, size, or behavior of the agents, as well as altering the characteristics of the environment in which they operate.

3.2 Software Requirement

- The software must have function for visualizing the pattern formation during the angiogenesis. This could include the ability to display the positions, shapes, and colors for cells, as well as functions for showing protrusions movement and analyzing pattern formation. It may also be useful to have the ability to generate images and animations of the simulation to help showing human readable results.

- The software should be able to handle large data sets and complex simulations with many cells and signaling molecules. It is vital to have the ability to scale the computation, especially when dealing with thousands or even millions of cells, each with its own set of properties and behaviors. The software should be able to handle these large data sets efficiently and without crashing.
- The software must be able to store and retrieve simulation data for analysis. This could include the ability to save and load simulation data. It may also be useful to have the ability to export data in a variety of formats for use with other software or tools.
- The software must be easy to use and allow for the modification and customization of simulation parameters and rules. It should be intuitive and well-documented, with clear instructions and examples for setting up and running simulations. It should also allow users to easily customize the simulation parameters and rules to suit specific research needs.

Chapter 4

Methods

4.1 Model Architecture

4.1.1 Overview

For our agent-based model, we use the Data-oriented Design (DOD) approach, which emphasizes the efficient organization and manipulation of data structures, as opposed to the traditional object-oriented design. DOD allows us to streamline the model's performance by optimizing cache locality, reducing memory overhead, and enhancing maintainability and extensibility.

This data-oriented design is implemented via the ***xarray*** library, which provides a convenient way to work with labeled multi-dimensional arrays (Hoyer & Hamman, 2017). It is designed to be modular, allowing for the easy integration. The core dataset is created to store various attributes of cells, such as their positions, amount of Dll4 and Active Notch and other optional properties.

Figure 4.1 shows a high level architecture graph of our agent-based model. It illustrates how the core dataset interacts with different modules that implement various aspects of the model, such as siganlling pathway and cell network.

An explanation of the core *dataset* variables is shown in Table 4.1, which comprises an organized structure of data variables and dimensions that hold various attributes and properties of the cellular environment and cells, such as their positions, levels of Dll4 and Active Notch, and other optional characteristics. This comprehensive *dataset* serves as the foundation for our agent-based model and facilitates seamless interactions with different modules that implement various aspects of the model, such as the signaling pathway and cell network.

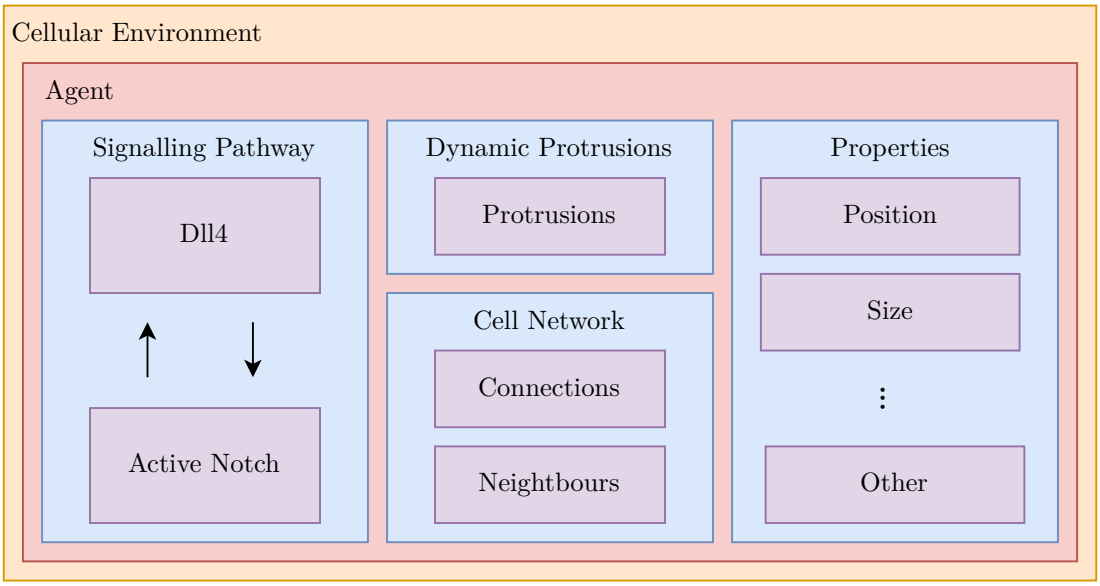

Figure 4.1: Model Architecture

Data Variables	**Dimensions**	**Description**
pos_x	$(cell)$	x-coordinates of cell positions
pos_y	$(cell)$	y-coordinates of cell positions
dll4	$(cell, time)$	Dll4 level for each cell and time point
active_notch	$(cell, time)$	Active Notch level for each cell and time point
cell_map	(map_width, map_height)	A map representing cell presence (optional)
cell_id_map	(map_width, map_height)	A map storing cell IDs (optional)
cell_neighbours	$(cell, neighbours)$	Neighbours for each cell (optional)
cell_connections	$(cell, connections)$	Connections for each cell (optional)
cell_protrusions	$(cell, protrusions)$	Protrusions for each cell (optional)
voronoi_graph	N/A	Voronoi graph of the cell positions (optional)
vor_ridge_graph	N/A	Graph of the ridge connections

Table 4.1: *dataset* architecture for the cellular environment created using xarray

4.2 Connection Network

In our agent-based model, the cell communication network plays a crucial role in facilitating cell communication and enabling the coordination of cellular behavior during angiogenesis. To simulate the cell communication network, we employed a combination of Voronoi diagrams and network graphs, which offer a natural and efficient representation of spatial relationships between cells and allow us to investigate the communication .

The Voronoi diagram is constructed based on the cell positions and is used to determine cell neighbors. Each cell is considered a node in the network graph, and the edges represent direct connections between neighboring cells. The network graph is then utilized to analyze the connectivity and structural properties of the cell network, such as the clustering coefficient or In-degree and out-degree distributions. This approach allows us to capture the complex spatial

organization and interactions between cells, which are essential for understanding the pattern dynamics.

We will continue to refine and improve our cell network representation as we gain more insights into the dynamics of angiogenesis and the specific factors influencing cell communication.

4.3 Signalling Pathway Dynamics

Due to the complexity of implementing the full Dll4-Notch signaling pathway, a simplified version of the pathway is used. This allows for easier integration and analysis while still capturing the essential features and behaviors of the pathway. In our model, we assume that VEGF and VEGFR are uniformly distributed in the tissue. We use a simple feedback loop that regulates Dll4 and Notch in the tissue which is calculate by the following way:

Initialize the Dll4 property for each cell in the dataset, with random values generated from a normal distribution with parameter mean μ and standard deviation σ.

$$Dll4_i(0) = \mathcal{N}(\mu, \sigma), \quad \text{for} \quad i = 1, 2, \ldots, N$$

Calculate the active Notch level for each cell at a given timestep, based on the average Dll4 level of its connected cells at the previous timestep.

$$ActiveNotch_i(t) = \frac{\sum_{j \in connections(i)} Dll4_j(t-1)}{|connections(i)|}, \quad \text{for} \quad i = 1, 2, \ldots, N$$

Calculate the Dll4 level for each cell at a given timestep, based on the initial Dll4 value and the active Notch level of the cell at the previous timestep. If the calculated value is less than 0, set it to 0.

$$Dll4_i(t) = \max(Dll4_i(0) - ActiveNotch_i(t-1), 0), \quad \text{for} \quad i = 1, 2, \ldots, N$$

In these expressions, N represents the total number of cells in the dataset, i represents the index of a cell, and t represents the current timestep.

The simplified Dll4-Notch signaling pathway offers several advantages. It reduces the complexity of the model, making it easier to implement and analyze. By focusing on the key aspects of the pathway, we can still capture the essential behaviors and interactions between cells that influence pattern formation during angiogenesis.

However, there are also some disadvantages associated with the simplified signaling pathway.

The main drawback is the potential loss of important details and interactions within the full Dll4-Notch signaling pathway. We may not fully capture the complexities of the system, leading to an incomplete understanding of the underlying mechanisms governing pattern formation and cell communication.

But we believe that it is still a suitable compromise for our agent-based model. The simplified pathway allows us to focus on the dynamic protrusions that could possibly affect the pattern formation in angiogenesis while maintaining computational efficiency.

4.4 Dynamic Protrusion

To model the dynamic protrusion process, we introduced a protrusion matrix in our dataset that represents the current state of protrusion for each cell. The matrix was initialized by maximum protrusion length. Furthermore, we added *ridge_availability* to our dataset attribute to regulate the use of the ridges.

Investigating the impact of these unique morphological features can lead to a better understanding of the role these cells play in blood vessel development. However, the increased complexity of the model may result in longer computational times, limiting the exploration of parameter space and the study of the system's behavior under various conditions.

In order to simulate the protrusion movement, we implemented a random walk function on the Voronoi ridge graph. This function (See Algorithm 1) generates random walks, starting from specified nodes, and iteratively performs a given number of steps. In our case, we used one step for each walk to simulate a single protrusion growth or retraction step (See Figure 4.2). This introduces an element of stochasticity into the simulation. While this can help capture the inherent variability in biological systems, it might also make it challenging to precisely control the simulation and identify the key factors driving specific outcomes.

The model's performance and accuracy depend on the choice of parameters, such as the maximum protrusion length, sample rate, and the protrusion movement frequency. A poorly calibrated model may lead to unrealistic predictions. We will try different parameters to see how these factors affect the stability of pattern formation during angiogenesis. We will also compare our model predictions with real observations to validate our approach in the future.

Algorithm 1 NetworkX Random Walk

```
function NetworkXRandomWalk(graph, start_nodes, num_walks, num_steps)
    walks ← empty list
    for i in start_nodes do
        for j ← 1 to num_walks do
            curr_walk ← list with single element i
            curr ← i
            for k ← 1 to num_steps do
                neighbors ← list of neighbors of curr in graph
                if neighbors is not empty then
                    curr ← random element from neighbors
                end if
                append curr to curr_walk
            end for
            append curr_walk to walks
        end for
    end for
    return walks
end function
```

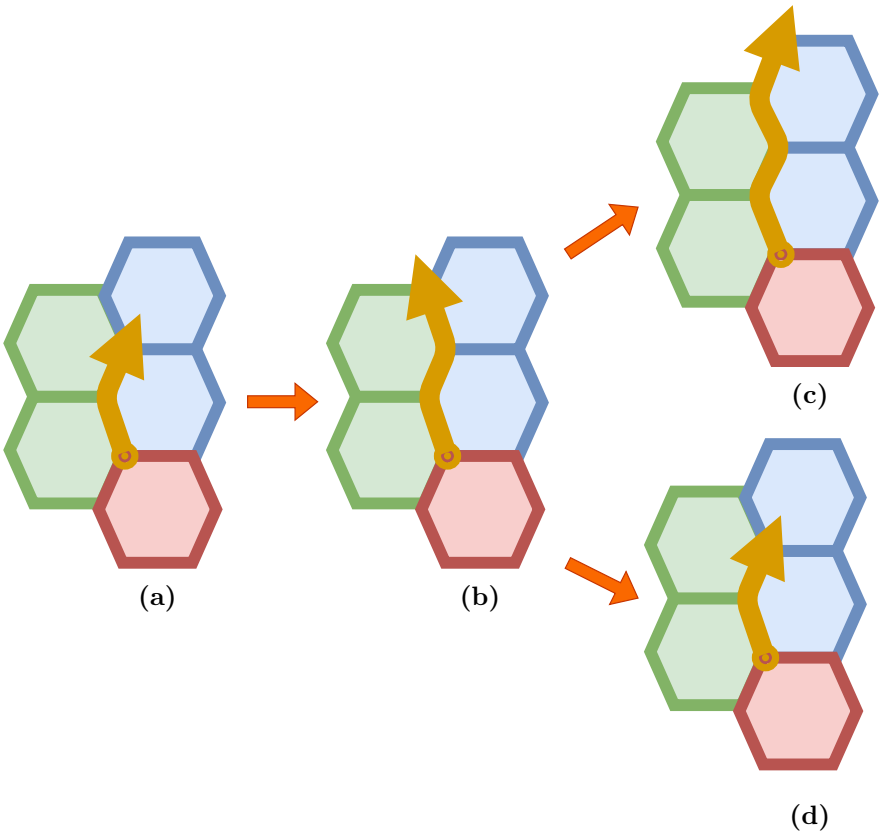

Figure 4.2: Illustration of the dynamic protrusion process using a random walk on the Voronoi ridges. For example, from (b) the protrusion can extend to (c) or retract to (d).
In (c), the blue cells and green cells have been removed from each others connection matrix.

Chapter 5

Implementation

5.1 Model Implementation

5.1.1 Pre-commit Config and Continuous Integration

As the first step for the project setup, a *pre-commit* configuration file was added to ensure consistent code quality and style throughout the project. Continuous integration were also employed to ensure the reliability and accuracy of the implemented model. The *tests.yml* file was created to set up the continuous integration workflow using *GitHub Actions*, which automated the testing and code coverage analysis for each commit. The *codecov* config file was added to further enhance the code coverage reporting, and the CI workflow was updated to use self-hosted runners for improved efficiency.

5.1.2 Initial Model Setup and Cell Representation

The initial stages of the model implementation involved setting up the fundamental structure of the simulation, which included representing cells as individual agents with unique attributes and establishing their initial configuration within the system. A critical aspect of this process was the decision to use a grid-based approach to initialize cell positions, which provided a structured and efficient method for organizing the cells within the simulation environment.

In this grid-based approach, cells were assigned to discrete positions within a 2D lattice. To introduce variability and randomness into the system, a Bernoulli distribution (See 5.1) was employed to determine the initial presence or absence of cells at each grid position. This probabilistic approach allowed for the generation of diverse initial cell positions, which in turn contributed to the exploration of various pattern formation scenarios.

$$f(k;p) = \begin{cases} p & \text{if } k = 1 \\ 1-p & \text{if } k = 0 \end{cases} \tag{5.1}$$

To model the connectivity between cells, a connection range was defined, which determined the maximum distance within which cells could establish connections with their neighbors. The initial naive implementation of the neighbor and connection matrices involved traversing all cells within the specified connection range to identify and establish connections with neighboring cells. This approach provided a straightforward method for constructing the initial cell network graph and served as a foundation for the subsequent development of more advanced connectivity and interaction mechanisms.

The combination of a grid-based approach for cell positioning, the use of a Bernoulli distribution for randomizing the initial cell presence, and the implementation of a connection range-based method for establishing cell connectivity, provided a robust and flexible starting point for the model implementation. These design choices facilitated the efficient representation and manipulation of cell attributes and interactions, setting the stage for the incorporation of more complex features and behaviors in the subsequent stages of the project.

5.1.3 Cell Communication Pathway and Network Graph

To accurately represent the connectivity between cells, we employed Voronoi diagrams, a powerful mathematical tool for partitioning a plane into regions based on the proximity to a set of points. In the context of our model, the points corresponded to the positions of individual cells, and the Voronoi regions represented the cell's shape. By constructing a Voronoi diagram for the cell positions, we were able to define the neighbors and connections between cells in a more realistic and biologically relevant manner, as opposed to relying solely on a fixed connection range.

The Voronoi diagram was generated using the efficient and robust algorithms provided by the *scipy.spatial* package. The resulting diagram consisted of a set of vertices, edges, and ridges, which defined the boundaries between neighboring Voronoi regions. The Voronoi ridges, in particular, were used to establish the cell network graph, as they represented the direct connections between neighboring cells.

With the Voronoi diagram in place, the neighbor and connection matrices were refactored to incorporate the information derived from the Voronoi regions. Instead of traversing all cells within a specified connection range, the refactored approach involved iterating through the

Voronoi ridges to identify the pairs of neighboring cells and establish their connections. This method provided a more accurate and efficient means for constructing the cell network graph, as it directly leveraged the spatial information encoded in the Voronoi diagram.

We use the simplified version of the Dll4-Notch signaling pathway as a crucial component of the model, as described in Section 2.1.2. Due to the inherent nature of the agent-based model, we were able to apply vectorization techniques to enhance the update functions for the signaling pathway. This enabled the system to process the signaling pathway interactions for multiple cells concurrently, which significantly improved the computational efficiency of the model.

5.1.4 Visualization

To facilitate the analysis and interpretation of the simulation results, various visualization functions were developed using the *Plotly* package, a powerful and versatile library for creating interactive and high-quality visualizations. The choice of *Plotly* as the visualization tool provided several benefits, including its ability to generate interactive and responsive graphs that can be easily shared and embedded in various formats, such as web applications, reports, or presentations.

The visualization functions were designed to display cell attributes, such as Dll4 and active Notch levels, and the cell network graph. These visualizations enabled a clear and intuitive representation of the simulation outcomes, allowing us to explore the emergent patterns. By leveraging the interactivity and customization features of *Plotly*, we can easily navigate through the data, zoom in on specific regions of interest, and adjust the display settings to highlight different aspects of the simulation results.

5.1.5 Protrusion Simulation

The incorporation of dynamic cell behavior was a critical aspect of the model implementation, as it allowed for the exploration of how changes in cell shape and connectivity influenced the overall pattern dynamics and the formation of the branched network structure. To achieve this, a protrusion simulation was developed, which modeled the extension and retraction of cellular protrusions along the Voronoi ridges. These protrusions played a crucial role in modulating cell-cell interactions and contributed to the emergence of complex patterns.

To simulate the dynamic behavior of cellular protrusions, we first sample some cells from the whole population and then use a random walk algorithm along the Voronoi ridges as described in 4.4. The random walk algorithm involved selecting a random direction along the Voronoi

ridge for sampled cells after a given time period.

In addition to generating protrusion movement, it also accounted for the potential effects of protrusions on cell connectivity. Specifically, the extension of a protrusion could lead to the separation of cells as described in Figure 4.2 and breaking the connections between those cells. If the protrusion retracts, their connection will be added back to their connection matrix and the cell network graph.

5.2 Simulation

Algorithm 2 Simulation Steps

```
1: Create a cell dataset
2: Initialize the Dll4 attribute
3: for each time step in the dataset do
4:     Update Dll4 and Active_Notch attributes
5:     Update the network graph
6:     Calculate Clustering_Coefficient and Characteristic_Path_Length
7: end for
8: Visualize the simulation result
```

The simulation was designed to be flexible and extensible, allowing for the incorporation of various features and behaviors throughout the model implementation. The extensibility of the simulation was achieved through the use of a modular approach, with distinct functions responsible for different aspects of the model, such as cell communication, network graph updates, and metric calculations. A sample process of conducting a simulation is shown in Algorithm 2.

The core of the simulation involved the creation of a cell dataset, which served as the primary data structure for storing and manipulating the attributes and states of individual cells. The main simulation loop iterated through the specified time steps, updating the cell attributes and network graph at each iteration. During the simulation, network metrics were calculated at each time step to evaluate the properties of the tissue-wide network architecture. The cell dataset was designed to be extensible, allowing for the addition of new attributes and features as needed. This can be added when creating the dataset or any time during the simulation. After simulation is done, various visualization functions could be used to display the cell attributes and network graph.

5.3 Software Testing

To ensure the reliability and accuracy of our agent-based model, it is crucial to perform rigorous software testing. This process helps identify potential bugs, inconsistencies, and issues that could negatively impact the model's performance and results. Varies testing framework and tools are used to evaluate and validate our model, such as *pytest* and *codecov*.

Pytest is a popular and powerful testing tool for Python that allows us to write concise and expressive test cases using simple assert statements. *Pytest* also supports parameterized testing, fixtures, plugins, and custom markers that enable us to customize and extend our testing capabilities. We used *pytest* to write unit tests for each of the model components, such as agents, signal update rule, and protrusions.

Codecov is a code coverage tool that measures how much of our source code is executed by our test cases. Code coverage is an important metric that indicates the quality and completeness of our testing efforts. *Codecov* also provides graphical reports and statistics that help us visualize and analyze our code coverage. We used *codecov* to monitor and improve our code coverage throughout the development process.

Because of the difficulty in testing the visualization functions, we did not include them in our code coverage analysis. However, we manually inspected and verified the visualization outputs to ensure they match our expectations and specifications.

5.4 Software Tools and Libraries

In this project, several software tools and libraries were used to implement the agent-based model and improve its performance. The following is a list of the main packages used in this project:

- **fsspec:** A library for interacting with various filesystems (Ruslan, Martin, & Batuhan, 2023).
- **networkx:** A library for study complex networks (Hagberg, Schult, & Swart, 2008).
- **numba:** A Just-In-Time (JIT) compiler for Python (Lam et al., 2015).
- **numpy:** A library for numerical computing in Python (Harris et al., 2020).
- **plotly:** A library for creating interactive visualizations (Plotly Technologies Inc., 2015).
- **pytest:** A testing framework for Python (Krekel et al., 2023).

- **scipy:** A library for scientific computing in Python (Virtanen et al., 2020).
- **xarray:** A library for working with labeled multi-dimensional arrays in Python (Hoyer & Hamman, 2017).

Chapter 6

Results and Evaluation

6.1 Simulation Results

6.1.1 Signaling Pathway & Pattern formation

The outcomes of our simulations revealed a strong correlation between the observed pattern formation and the predicted salt-and-pepper pattern, as endothelial cells (ECs) showed alternating high and low levels of Dll4 and active Notch signalling before stabilising (See Figure A.1). Examining the line charts revealed that irregularly shaped cells exhibited a longer stabilisation period for signal activities, indicating the influence of cell shape and direct contact with adjacent cells on pattern formation (See Figure 6.1 and 6.2).

Initially, we hypothesised that the presence of dynamic protrusions would have an effect on the stability of pattern formation during angiogenesis. This theory was based on the notion that the dynamic nature of these protrusions could make contact with distal cells, thus disrupting the stable formation of the salt-and-pepper pattern. Our simulation results supported this hypothesis, as we observed the effect of protrusions on the dynamics of patterns (See Figure 6.2 and 6.4 and A.1 - A.8).

The correlation between the stability of signalling patterns and protrusion properties, such as maximum protrusion length and sample rate, is strong and positive. The formation and stability of the salt-and-pepper pattern observed in ECs are dependent on the cellular arrangement and protrusion characteristics, according to these findings (See Figure A.9 - A.16).

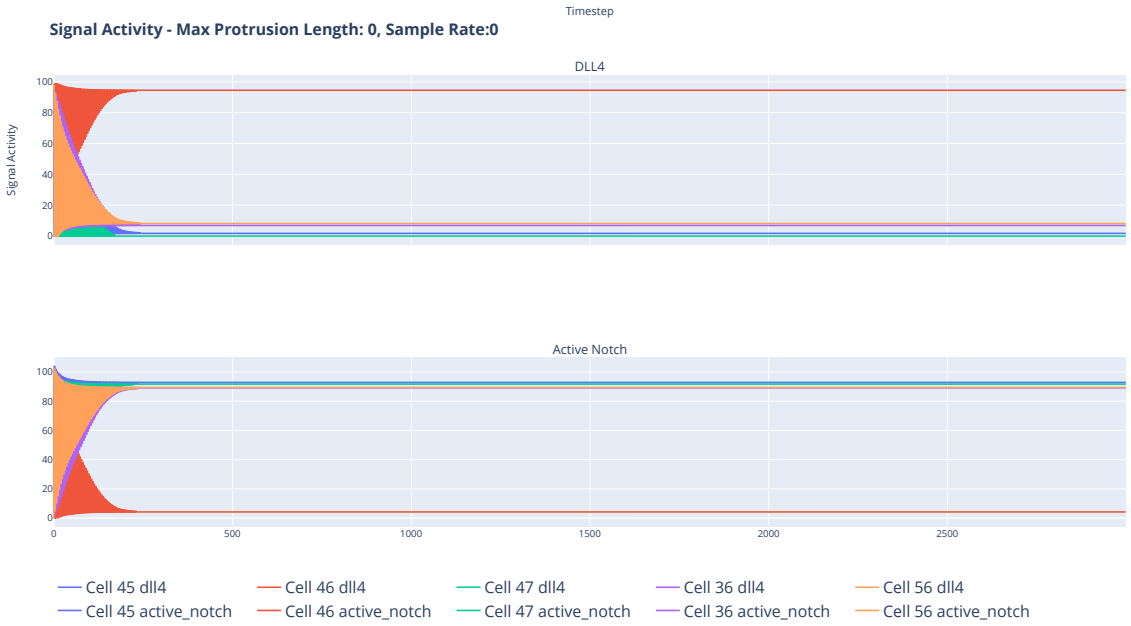

Figure 6.1: Signal Activity for **Cell 46** and its Neighbours. (All cell presence)

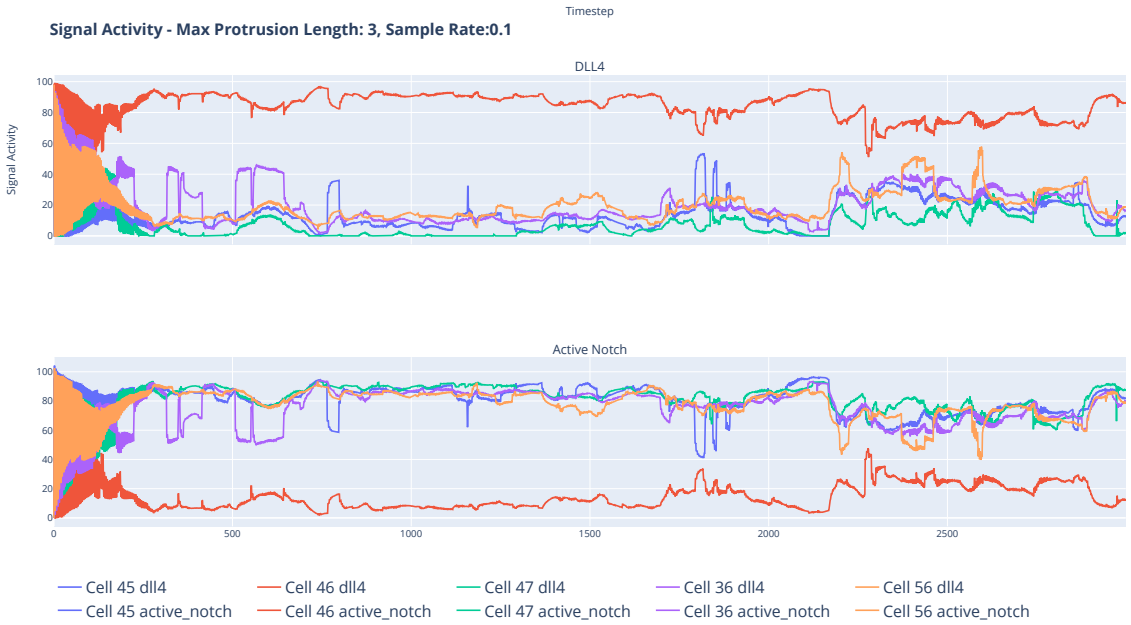

Figure 6.2: Signal Activity for **Cell 46** and its Neighbours. (All cell presence)

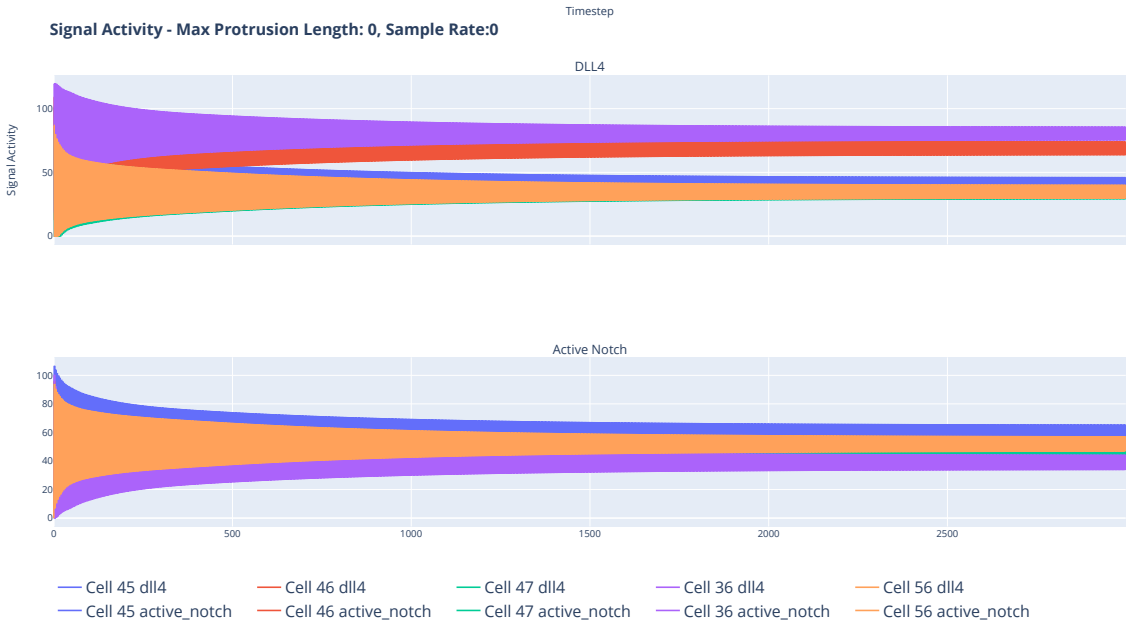

Figure 6.3: Signal Activity for **Cell 46** and its Neighbours. (Random cell presence)

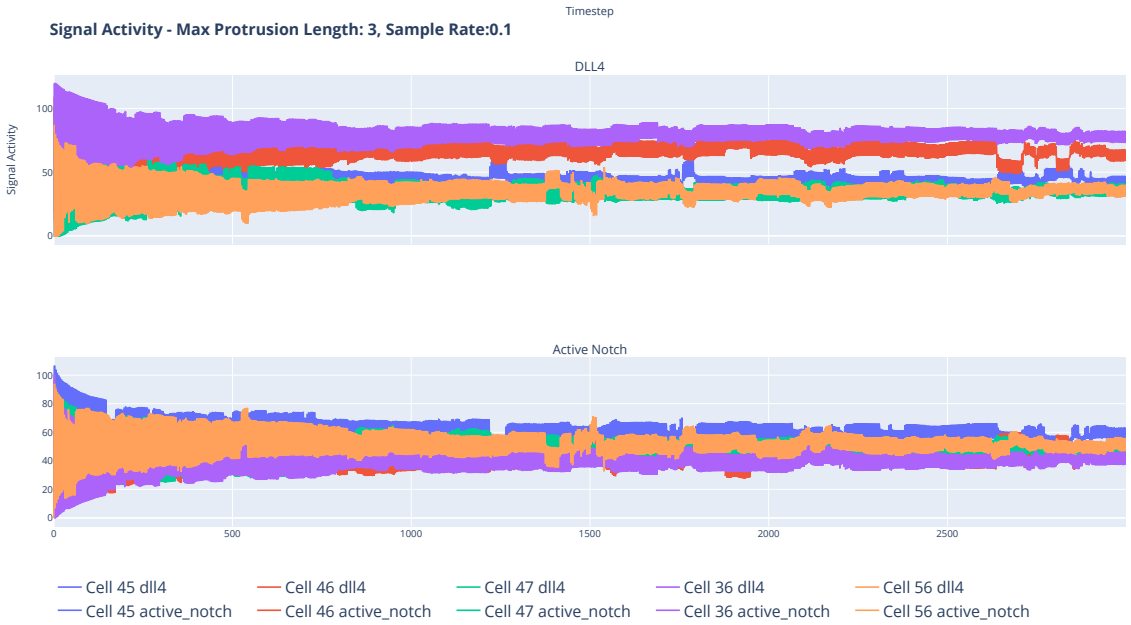

Figure 6.4: Signal Activity for **Cell 46** and its Neighbours. (Random cell presence)

6.1.2 Network Properties

Upon analyzing the network properties, it was found that the presence of dynamic protrusions led to an increase in the clustering coefficient and a decrease in the characteristic path length. A higher sample rate and longer protrusions were directly correlated with a larger clustering coefficient and shorter characteristic path length (See Figure 6.6 - 6.9). Inspection of line charts revealed that the sample rate exerted a more substantial impact on these variables as opposed to the maximum protrusion length.

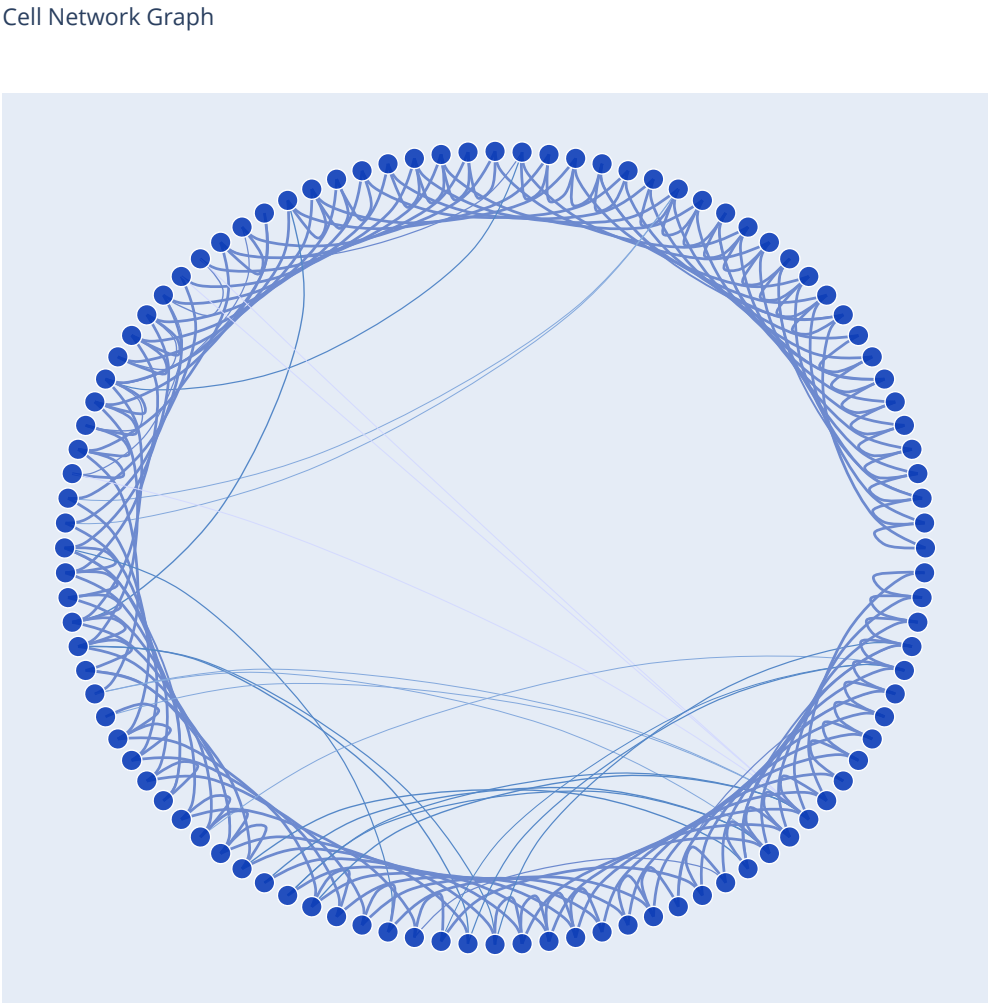

Figure 6.5: A visualization of the cell network graph

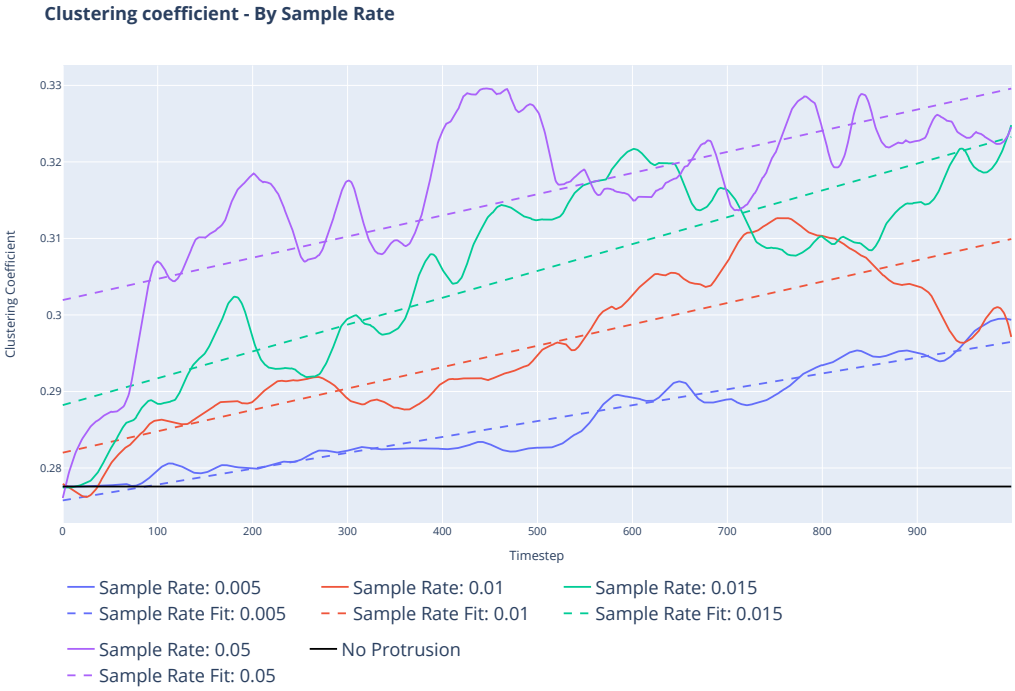

Figure 6.6: Clustering Coefficient by Sample Rate: Change over Time

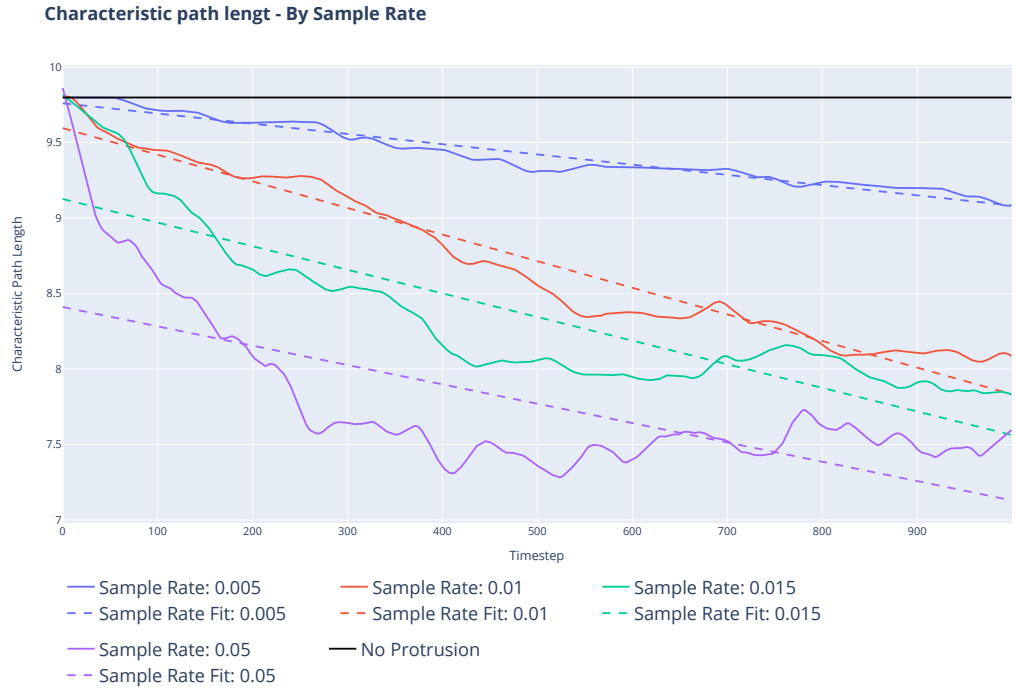

Figure 6.7: Characteristic Path Length by Max Sample Rate: Change over Time

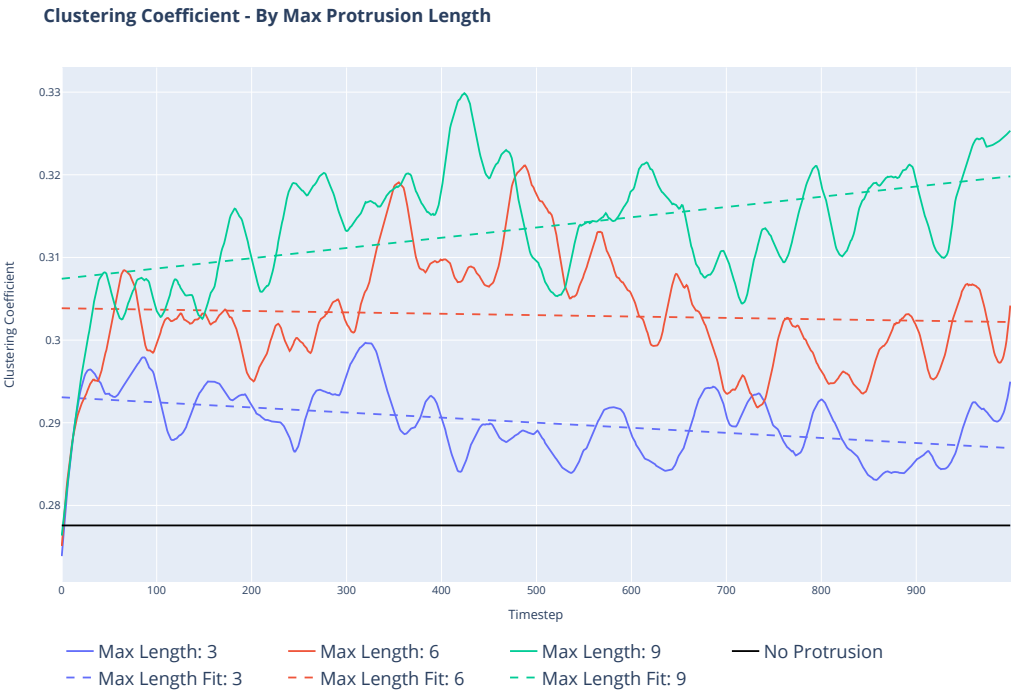

Figure 6.8: Clustering Coefficient by Max Protrusion Length: Change over Time

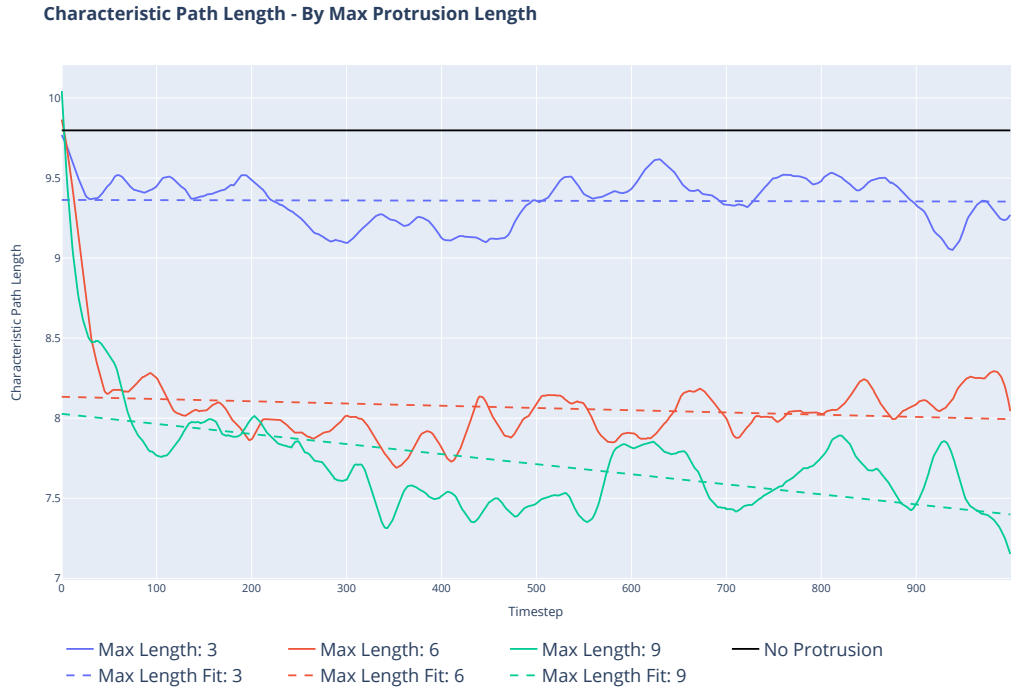

Figure 6.9: Characteristic Path Length by Max Protrusion Length: Change over Time

6.1.3 JIT & Vectorization

Using vectorized functions and JIT compilation techniques significantly improved the agent-based model's computational efficiency. Comparing the time required by the naive implementation and the guvectorized method to update Dll4 and active Notch levels disclosed that the vectorized functions were significantly faster (See Figure 6.10 - 6.13). The results demonstrate the advantages of employing vectorization techniques in the development of agent-based simulations, as it improves scalability and high performance computing, allowing researchers to investigate larger systems and more complex scenarios at lower computational costs.

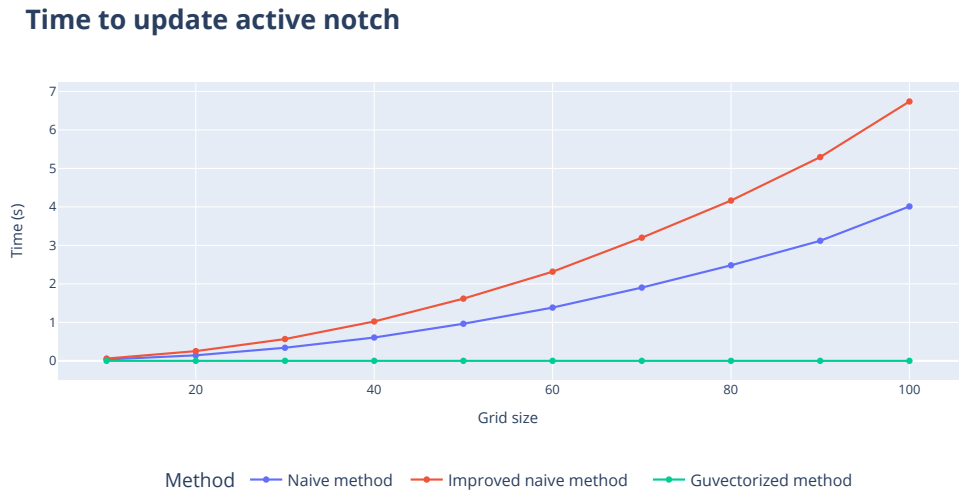

Figure 6.10: Comparison of time taken by update active notch methods for various grid sizes

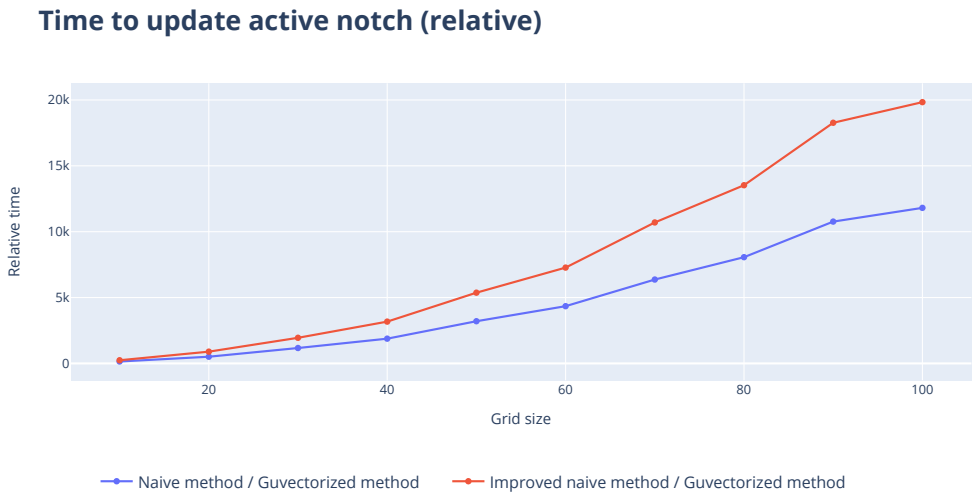

Figure 6.11: Relative time comparison of update active notch methods for various grid sizes

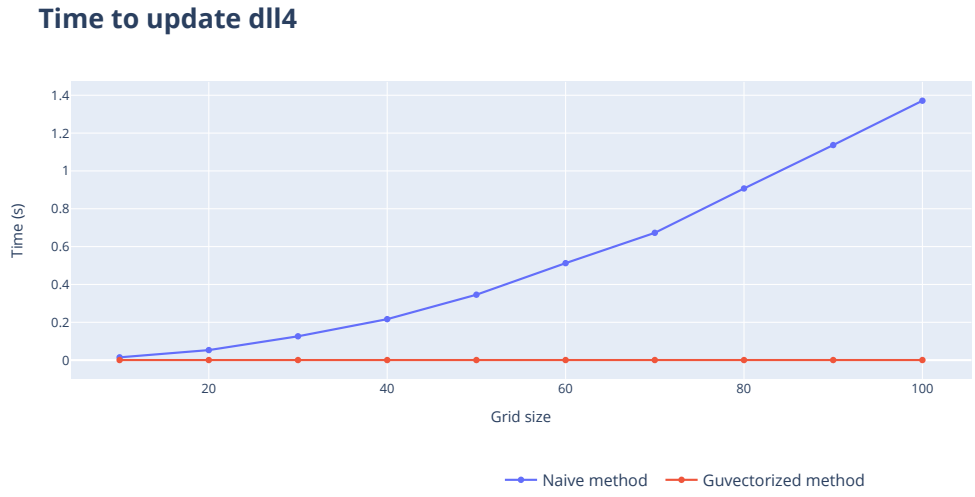

Figure 6.12: Comparison of time taken by update Dll4 methods for various grid sizes

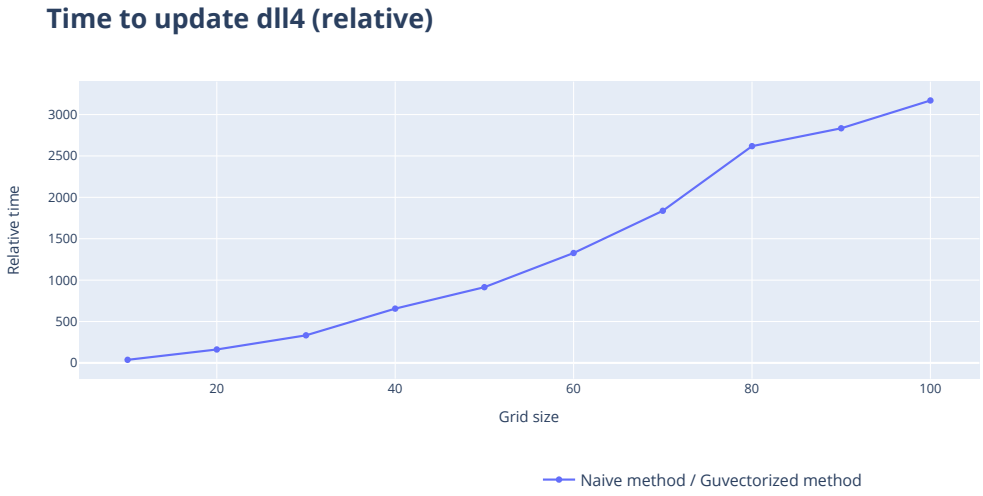

Figure 6.13: Relative time comparison of update Dll4 methods for various grid sizes

6.2 Discussion

In this project, we aimed to investigate the role of dynamic cell protrusions in pattern formation during angiogenesis through a comprehensive agent-based simulation. The simulation results presented herein provide valuable insights into the impact of irregular, widely branched cells on the development of blood vessels, as well as how these cells affect tissue-wide network architecture and the dynamics of the Dll4-Notch signaling pathway.

One of the primary findings of this project is that the presence of dynamic cell protrusions, although potentially disruptive to stable pattern formation, does not necessarily compromise the overall signaling activity within the angiogenic network. This observation suggests that the ECs

possess a certain degree of resilience and adaptability to maintain stable pattern formation even in the presence of irregularly shaped cells. This robustness could be attributed to the inherent properties of the Dll4-Notch signaling pathway, which plays a critical role in regulating cell fate during development and establishing a salt-and-pepper pattern of cell signaling. This finding warrants further investigation to establish the precise mechanisms underlying the observed robustness.

Our results also show that the implementation of dynamic protrusions leads to an increase in the clustering coefficient and a decrease in the characteristic path length of the EC network, resulting in a small-world network architecture as described by Watts and Strogatz (1998). This type of network is characterized by efficient information exchange and strong coordination among its constituent elements, which is essential for the collective behavior of ECs during angiogenesis. The emergence of a small-world network in the presence of dynamic protrusions supports the hypothesis that these structures promote local interactions and coordinated behavior among ECs, thereby affecting pattern formation and stabilization. This observation provides a basis for future studies exploring the role of other cellular and molecular factors in shaping the network architecture during angiogenesis.

In addition to the biological implications of our findings, this project also demonstrates the benefits of employing advanced computational techniques such as vectorization and just-in-time (JIT) compilation in the development of agent-based simulations. By leveraging these approaches, we achieved improved scalability and high performance, which enables researchers to investigate larger systems and more complex scenarios at a reduced computational cost. This highlights the importance of adopting cutting-edge computational methods in the study of complex biological systems, as well as the potential for interdisciplinary collaboration between computational and experimental researchers.

In conclusion, our project provides novel insights into the role of dynamic cell protrusions in pattern formation during angiogenesis, while also showcasing the power of agent-based modeling and advanced computational techniques in the investigation of complex biological processes. Our findings lay the groundwork for future studies aimed at unraveling the mechanisms underlying pattern formation and the influence of network architecture on the dynamics of the Dll4-Notch signaling pathway. Furthermore, the computational approaches employed in this study open new avenues for the development of efficient and scalable simulation tools, which will facilitate the exploration of even more complex biological systems and scenarios in the future.

Chapter 7

Conclusion and Future Work

7.1 Limitations and Future Work

Despite the valuable insights gained from our agent-based simulation, there are several limitations that need to be acknowledged and addressed in future studies.

7.1.1 Limitations

Despite the advancements made in this project, several limitations remain present, which may impact the interpretation and generalizability of the results. These limitations need to be considered when evaluating the outcomes and their implications for the understanding of pattern formation in angiogenesis.

1. Simplification of the Dll4-Notch signaling pathway: In this project, we have employed a simplified representation of the Dll4-Notch signaling pathway, which may not capture the full complexity of the molecular interactions and feedback loops involved. The reduction of such complexity might have an impact on the model's ability to fully represent the dynamics of pattern formation and cellular response to the signaling process.

2. Cell representation: The current model assumes a unchanged shape for the ECs and relies on a grid-based approach to initialize cell positions. This may not accurately reflect the diverse morphologies ECs can exhibit during angiogenesis and the irregular spatial distribution of cells observed in vivo. An improved representation of cell shape and distribution could provide a more accurate depiction of cellular interactions and their influence on pattern formation.

3. Two-dimensional representation: The model is based on a two-dimensional representation of the angiogenic network. While this simplification facilitates computational efficiency and model interpretation, it may not fully capture the complexity of the three-dimensional vascular environment, where cells can extend their protrusions and migrate in all three dimensions. This limitation may result in an over-simplified representation of the angiogenic process and could hinder the generalizability of the results to in vivo systems.

4. Lack of experimental validation: Although the model has been validated using available literature, direct experimental validation is missing. This limitation highlights the need for further experimental studies to corroborate the findings of our simulation and assess the accuracy and relevance of the model in the context of real biological systems.

7.1.2 Future Work

To address the limitations identified in this project and to further expand our understanding of pattern formation in angiogenesis, several areas of future work have been identified:

1. Comprehensive representation of the signaling pathway: Future work could focus on developing a more detailed and accurate representation of the Dll4-Notch signaling pathway by incorporating additional molecular interactions and feedback loops. This could be potentially improved by integrating differential equation models into our agent-based model which could capture continues changes in signaling cues.

2. Improved cell representation: Further research could explore the use of more advanced mathematical and computational methods to better represent the diverse morphologies and spatial distribution of ECs during angiogenesis. For example, generalized Voronoi tessellations could be employed to model the actual boundary shape of cells, allowing for the generation of more realistic cell shapes and interactions (Bock, Tyagi, Kreft, & Alt, 2010).

3. Extension to three-dimensional simulation: To more accurately capture the complexity of the angiogenic process, future work could extend the current model to a three-dimensional representation. This would enable the investigation of the role of cellular protrusions and migration in all three dimensions, providing a deeper understanding of the mechanisms driving pattern formation in angiogenesis.

4. Experimental validation: Future studies should aim to perform experimental validation of the developed model. This could involve the generation of in vitro or in vivo angiogenesis

assays to directly assess the accuracy and relevance of the model in the context of real biological systems. Such experimental validation would not only provide critical support for the findings of our simulation but also contribute to the refinement and optimization of the model, ultimately enhancing its predictive capabilities.

By addressing these limitations and expanding the scope of the model, future work has the potential to significantly advance our understanding of the complex processes underlying pattern formation in angiogenesis and to provide valuable insights into the mechanisms of collective cell communication.

7.2 Summary of Contributions

In this study, we developed and analyzed a comprehensive agent-based model to simulate pattern formation in angiogenesis, capturing the spatial and temporal dynamics of collective cell communication and the unique features of prongy shaped cells with dynamic protrusions. Our model revealed some of the key factors and parameters that influence pattern formation, stability, and network properties, providing valuable insights into the underlying mechanisms governing angiogenesis and patterning process. Furthermore, we addressed computational challenges associated with the simulation, paving the way for future advancements in the field of computational biology. Despite some limitations related to model validation and scalability, our work provide some understanding of pattern formation in angiogenesis and presents a reusable tool for future research.

Chapter 8

Legal, Social, Ethical and Professional Issues

8.1 Legal Issues

Concerning data protection and privacy, our study does not involve the processing of personal or sensitive data. Nonetheless, we take precautions to ensure that any data used or generated in the course of research is stored safely and ethically. The study adheres the British Computer Society (BCS) Code of Conduct and relevant legislation in order to fulfill the legal and ethical standards required for an undergraduate project, therefore maintain data protection and privacy concerns.

8.2 Social Issues

Our research could contribute to the domains of medicine and biology, which could have far-reaching social ramifications. By increasing our understanding of angiogenesis pattern formation, we may be able to contribute to the field of cancer research. These medical advancements may have good societal benefits by reducing patient suffering and enhancing the general population's quality of life. Although our study does not directly affect individuals and communities, it is crucial to consider the broader societal context when sharing our findings and to conduct future applications with a focus on the larger social benefits.

8.3 Ethical and Professional Issues

Maintaining the ethical integrity of our study requires that all team members get recognition for their contributions. It is crucial that our investigation be conducted with scientific rigour to ensure the validity and reproducibility of the results. Transparency reporting, which includes methods, limitations, and data analysis, will enhance the study's credibility and positively impact the scientific community. The study follows the BCS Code of Conduct's 'duty of profession', that the study respect 'professional relationship with all members with whom participated in a professional capacity' (British Computer Society, 2022). This ensures that our research meets the ethical standards expected of an undergraduate project.

References

Alt, S., Ganguly, P., & Salbreux, G. (2017). Vertex models: from cell mechanics to tissue morphogenesis [Journal Article]. *Philos Trans R Soc Lond B Biol Sci*, *372*(1720). Retrieved from https://www.ncbi.nlm.nih.gov/pubmed/28348254 doi: 10.1098/rstb.2015.0520

Bock, M., Tyagi, A. K., Kreft, J. U., & Alt, W. (2010). Generalized voronoi tessellation as a model of two-dimensional cell tissue dynamics [Journal Article]. *Bull Math Biol*, *72*(7), 1696-731. Retrieved from https://www.ncbi.nlm.nih.gov/pubmed/20082148 doi: 10.1007/s11538-009-9498-3

British Computer Society. (2022). *British computer society code of conduct* [Web Page]. Retrieved from https://www.bcs.org/media/2211/bcs-code-of-conduct.pdf

Carmeliet, P., & Jain, R. K. (2011). Molecular mechanisms and clinical applications of angiogenesis [Journal Article]. *Nature*, *473*(7347), 298-307. Retrieved from https://www.ncbi.nlm.nih.gov/pubmed/21593862 doi: 10.1038/nature10144

Collier, J. R., Monk, N. A., Maini, P. K., & Lewis, J. H. (1996). Pattern formation by lateral inhibition with feedback: a mathematical model of delta-notch intercellular signalling [Journal Article]. *J Theor Biol*, *183*(4), 429-46. (Collier, J R Monk, N A Maini, P K Lewis, J H Wellcome Trust/United Kingdom Journal Article Research Support, Non-U.S. Gov't England 1996/12/21 J Theor Biol. 1996 Dec 21;183(4):429-46. doi: 10.1006/jtbi.1996.0233.) doi: 10.1006/jtbi.1996.0233

Gerhardt, H., Golding, M., Fruttiger, M., Ruhrberg, C., Lundkvist, A., Abramsson, A., ... Betsholtz, C. (2003). Vegf guides angiogenic sprouting utilizing endothelial tip cell filopodia [Journal Article]. *J Cell Biol*, *161*(6), 1163-77. Retrieved from https://www.ncbi.nlm.nih.gov/pubmed/12810700 doi: 10.1083/jcb.200302047

Geudens, I., & Gerhardt, H. (2011). Coordinating cell behaviour during blood vessel formation [Journal Article]. *Development*, *138*(21), 4569-83. Retrieved from https://www.ncbi.nlm.nih.gov/pubmed/21965610 doi: 10.1242/dev.062323

Hagberg, A. A., Schult, D. A., & Swart, P. J. (2008, August). Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th python in science conference (scipy2008)* (pp. 11–15). Pasadena, CA USA.

Harris, C. R., Millman, K. J., Van Der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... Oliphant, T. E. (2020). Array programming with numpy [Journal Article]. *Nature*, *585*(7825), 357-362. Retrieved from https://dx.doi.org/10.1038/s41586-020-2649-2 doi: 10.1038/s41586-020-2649-2

Hellstrom, M., Phng, L. K., Hofmann, J. J., Wallgard, E., Coultas, L., Lindblom, P., ... Betsholtz, C. (2007). Dll4 signalling through notch1 regulates formation of tip cells during angiogenesis [Journal Article]. *Nature*, *445*(7129), 776-80. Retrieved from https://www.ncbi.nlm.nih.gov/pubmed/17259973 doi: 10.1038/nature05571

Hennigs, J. K., Matuszcak, C., Trepel, M., & Korbelin, J. (2021). Vascular endothelial cells: Heterogeneity and targeting approaches [Journal Article]. *Cells*, *10*(10). Retrieved from https://www.ncbi.nlm.nih.gov/pubmed/34685692 doi: 10.3390/cells10102712

Hinch, R., Probert, W. J. M., Nurtay, A., Kendall, M., Wymant, C., Hall, M., ... Fraser, C. (2021). Openabm-covid19—an agent-based model for non-pharmaceutical interventions against covid-19 including contact tracing [Journal Article]. *PLOS Computational Biology*, *17*(7), 1-26. Retrieved from https://doi.org/10.1371/journal.pcbi.1009146 doi: 10.1371/journal.pcbi.1009146

Hirashima, T., Rens, E. G., & Merks, R. M. H. (2017). Cellular potts modeling of complex multicellular behaviors in tissue morphogenesis [Journal Article]. *Dev Growth Differ*, *59*(5), 329-339. Retrieved from https://www.ncbi.nlm.nih.gov/pubmed/28593653 (Hirashima, Tsuyoshi Rens, Elisabeth G Merks, Roeland M H eng Review Japan 2017/06/09 Dev Growth Differ. 2017 Jun;59(5):329-339. doi: 10.1111/dgd.12358. Epub 2017 Jun 8.) doi: 10.1111/dgd.12358

Hoyer, S., & Hamman, J. (2017). xarray: N-d labeled arrays and datasets in python [Journal Article]. *Journal of Open Research Software*. doi: 10.5334/jors.148

Jakobsson, L., Franco, C. A., Bentley, K., Collins, R. T., Ponsioen, B., Aspalter, I. M., ... Gerhardt, H. (2010). Endothelial cells dynamically compete for the tip cell position during angiogenic sprouting [Journal Article]. *Nat Cell Biol*, *12*(10), 943-53. Retrieved from https://www.ncbi.nlm.nih.gov/pubmed/20871601 doi: 10.1038/ncb2103

Krekel, H., Oliveira, B., Pfannschmidt, R., Bruynooghe, F., Laugher, B., & Bruhin, F. (2023). *pytest 7.2.1.* Retrieved from https://github.com/pytest-dev/pytest

Lam, S. K., Pitrou, A., & Seibert, S. (2015). *Numba: a llvm-based python jit compiler* [Conference Paper]. Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2833157.2833162 doi: 10.1145/2833157.2833162

Perkins, M. L., Benzinger, D., Arcak, M., & Khammash, M. (2020). Cell-in-the-loop pattern formation with optogenetically emulated cell-to-cell signaling [Journal Article]. *Nat Commun*, *11*(1), 1355. Retrieved from https://www.ncbi.nlm.nih.gov/pubmed/32170129 doi: 10.1038/s41467-020-15166-3

Plotly Technologies Inc. (2015). *Collaborative data science.* Montreal, QC: Plotly Technologies Inc. Retrieved from https://plot.ly

Rajendran, P., Rengarajan, T., Thangavel, J., Nishigaki, Y., Sakthisekaran, D., Sethi, G., & Nishigaki, I. (2013). The vascular endothelium and human diseases [Journal Article]. *Int J Biol Sci*, *9*(10), 1057-69. doi: 10.7150/ijbs.7502

Ruslan, K., Martin, D., & Batuhan, T. (2023). *fsspec 2023.3.0.* Retrieved from https://github.com/fsspec/filesystem_spec

Scarpa, E., & Mayor, R. (2016). Collective cell migration in development [Journal Article]. *J Cell Biol*, *212*(2), 143-55. Retrieved from https://www.ncbi.nlm.nih.gov/pubmed/26783298 doi: 10.1083/jcb.201508047

Suchting, S., Freitas, C., le Noble, F., Benedito, R., Breant, C., Duarte, A., & Eichmann, A. (2007). The notch ligand delta-like 4 negatively regulates endothelial tip cell formation and vessel branching [Journal Article]. *Proc Natl Acad Sci U S A*, *104*(9), 3225-30. Retrieved from https://www.ncbi.nlm.nih.gov/pubmed/17296941 doi: 10.1073/pnas.0611177104

Ubezio, B., Blanco, R. A., Geudens, I., Stanchi, F., Mathivet, T., Jones, M. L., ... Gerhardt, H. (2016). Synchronization of endothelial dll4-notch dynamics switch blood vessels from branching to expansion [Journal Article]. *Elife*, *5*. Retrieved from https://www.ncbi.nlm.nih.gov/pubmed/27074663 doi: 10.7554/eLife.12167

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., ... others (2020). Scipy 1.0: fundamental algorithms for scientific computing in python [Journal Article]. *Nature Methods*, *17*(3), 261-272. Retrieved from https://dx.doi.org/10.1038/s41592-019-0686-2 doi: 10.1038/s41592-019-0686-2

Watts, D. J., & Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks [Journal Article]. *Nature*, *393*(6684), 440-442. Retrieved from https://doi.org/10.1038/30918 doi: 10.1038/30918

Zakirov, B., Charalambous, G., Thuret, R., Aspalter, I. M., Van-Vuuren, K., Mead, T., ... Bentley, K. (2021). Active perception during angiogenesis: filopodia speed up notch selection of tip cells in silico and in vivo [Journal Article]. *Philosophical Transactions of the Royal Society B*, *376*(1821), 20190753. doi: https://doi.org/10.1098/rstb.2019.0753

Appendix A

Extra Information

A.1 Parameter Tables

Grid Size	Probability	Timesteps	Protrusion Max Length	Sample Rate
10	1	3000	0	0
10	1	3000	3	0.3
10	1	3000	3	0.6
10	1	3000	6	0.3
10	1	3000	6	0.6
10	1	3000	9	0.3
10	1	3000	9	0.6
12	0.7	3000	0	0
12	0.7	3000	3	0.3
12	0.7	3000	3	0.6
12	0.7	3000	6	0.3
12	0.7	3000	6	0.6
12	0.7	3000	9	0.3
12	0.7	3000	9	0.6

Table A.1: Simulation parameters for pattern formation analysis

Grid Size	Probability	Timesteps	Protrusion Max Length	Sample Rate
22	0.7	1000	0	0
22	0.7	1000	3	0.005
22	0.7	1000	3	0.01
22	0.7	1000	3	0.015
22	0.7	1000	3	0.05
22	0.7	1000	6	0.005
22	0.7	1000	6	0.01
22	0.7	1000	6	0.015
22	0.7	1000	6	0.05
22	0.7	1000	9	0.005
22	0.7	1000	9	0.01
22	0.7	1000	9	0.015
22	0.7	1000	9	0.05

Table A.2: Simulation parameter used for network properties analysis

A.2 Supplementary Figures

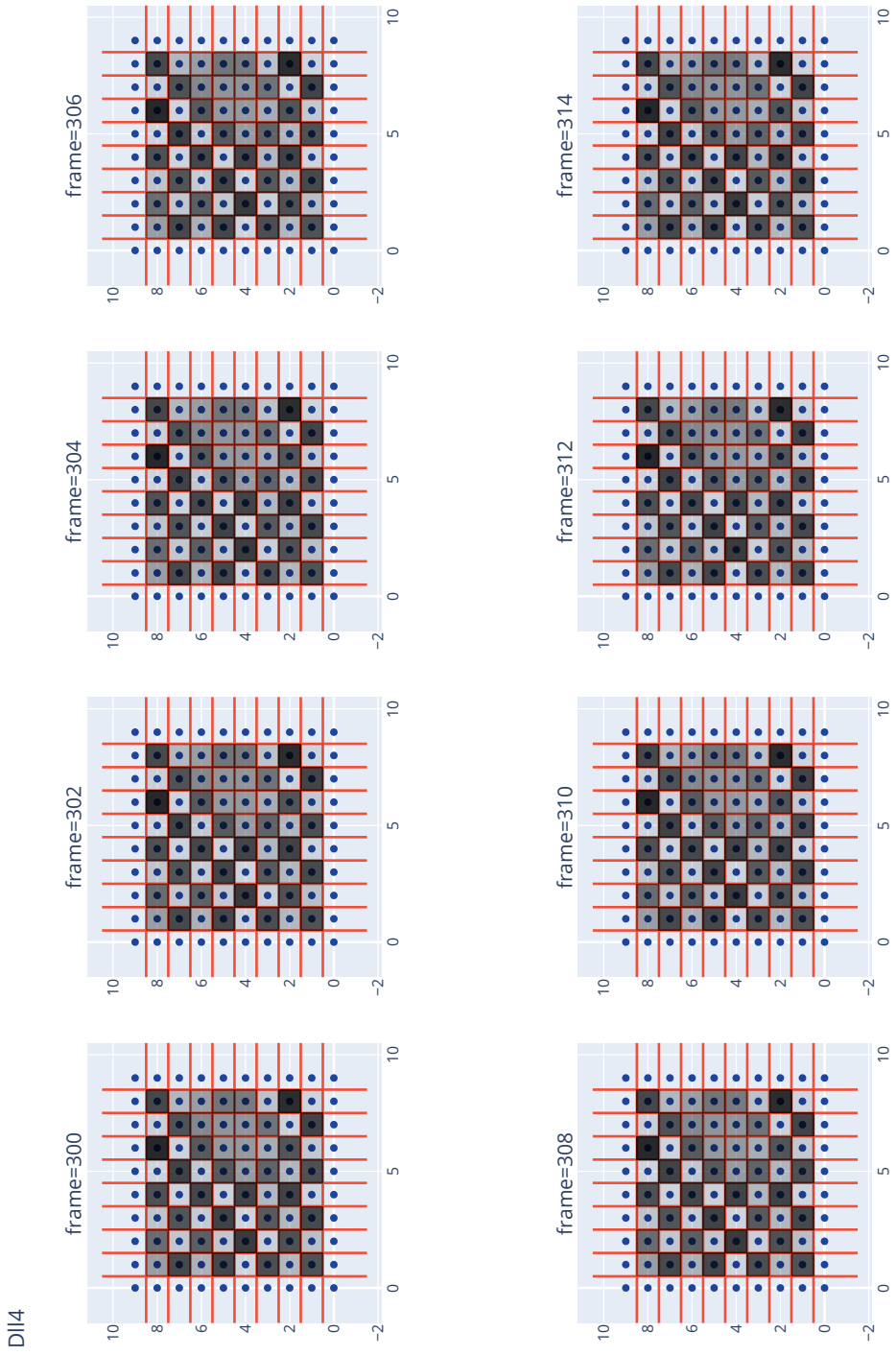

Figure A.1: Pattern change with **all cell presence** and **with out protrusion**. Frame 300 to 307. (Dll4)

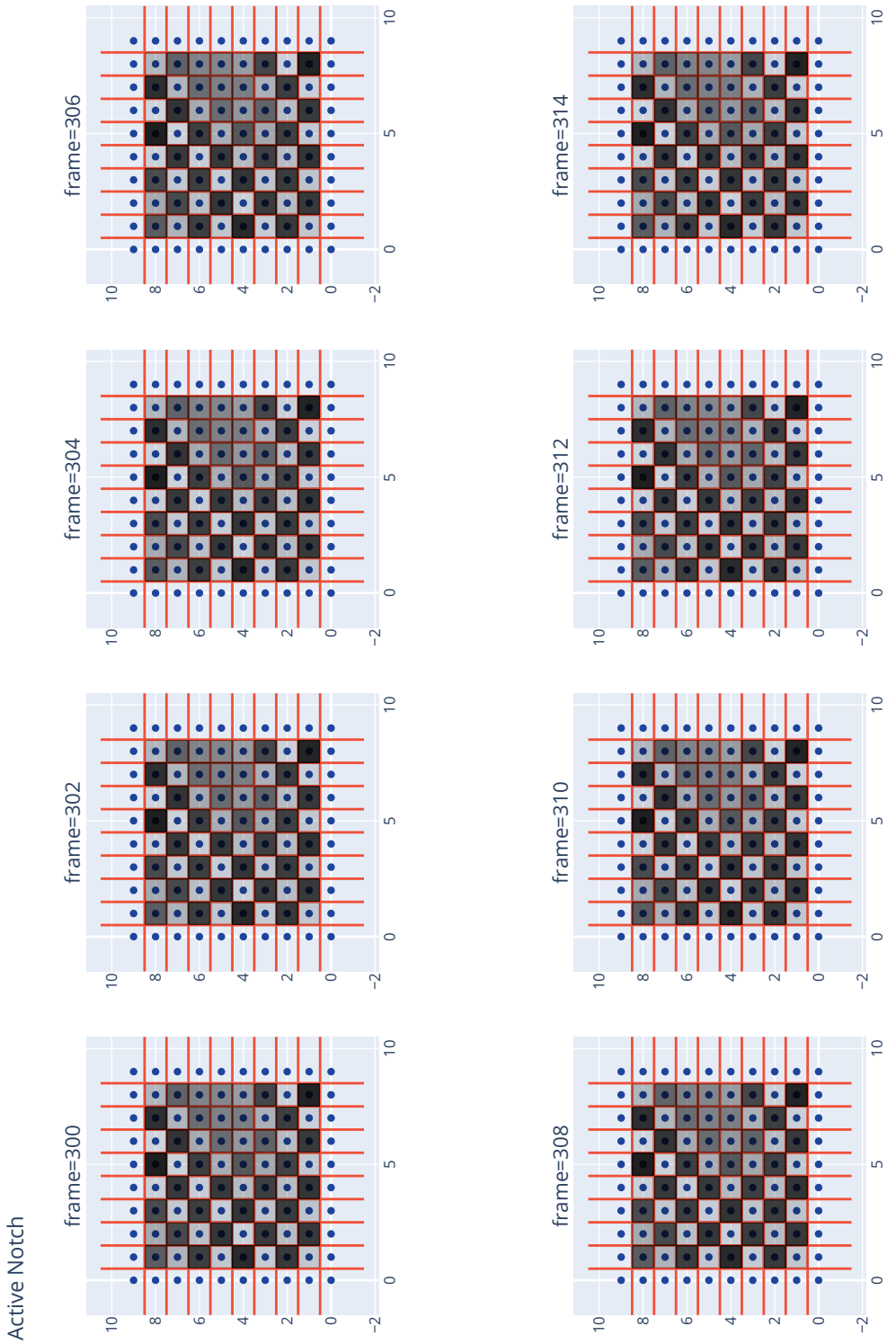

Figure A.2: Pattern change with **all cell presence** and **with out protrusion**. Frame 300 to 307. (Active Notch)

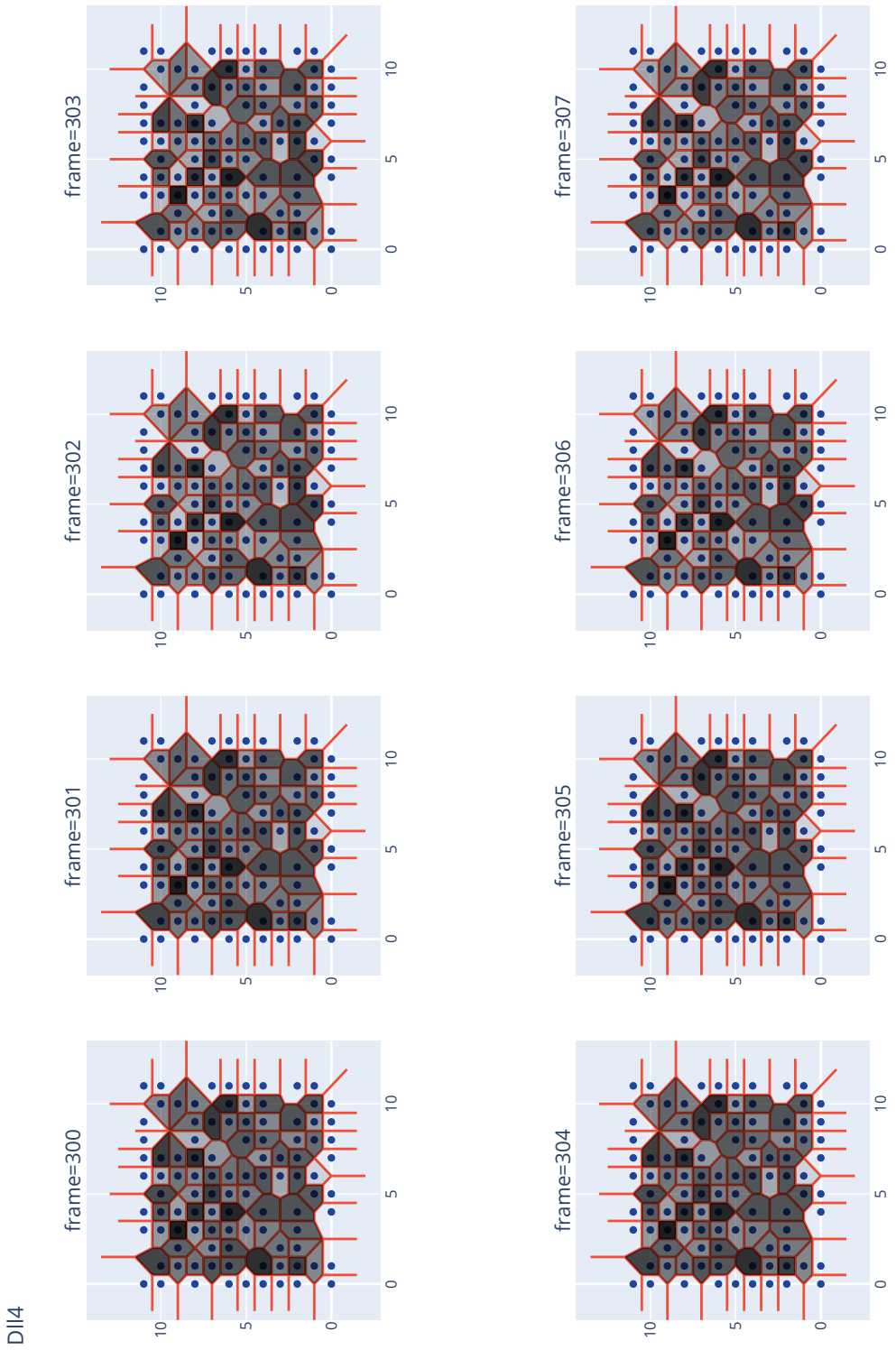

Figure A.3: Pattern change with **random cell presence** and **with out protrusion**. Frame 300 to 307. (Dll4)

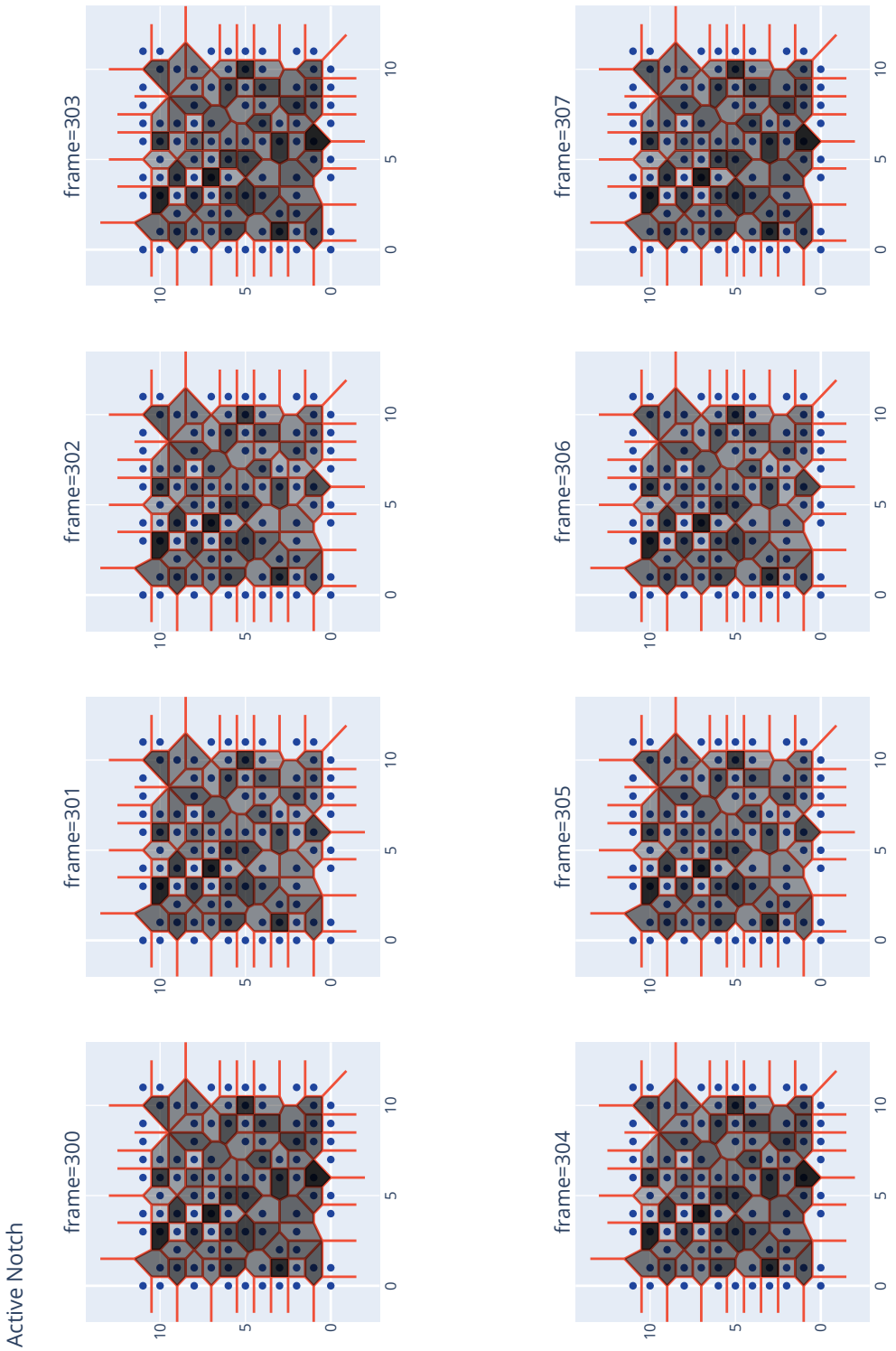

Figure A.4: Pattern change with **random cell presence** and **with out protrusion**. Frame 300 to 307. (Active Notch)

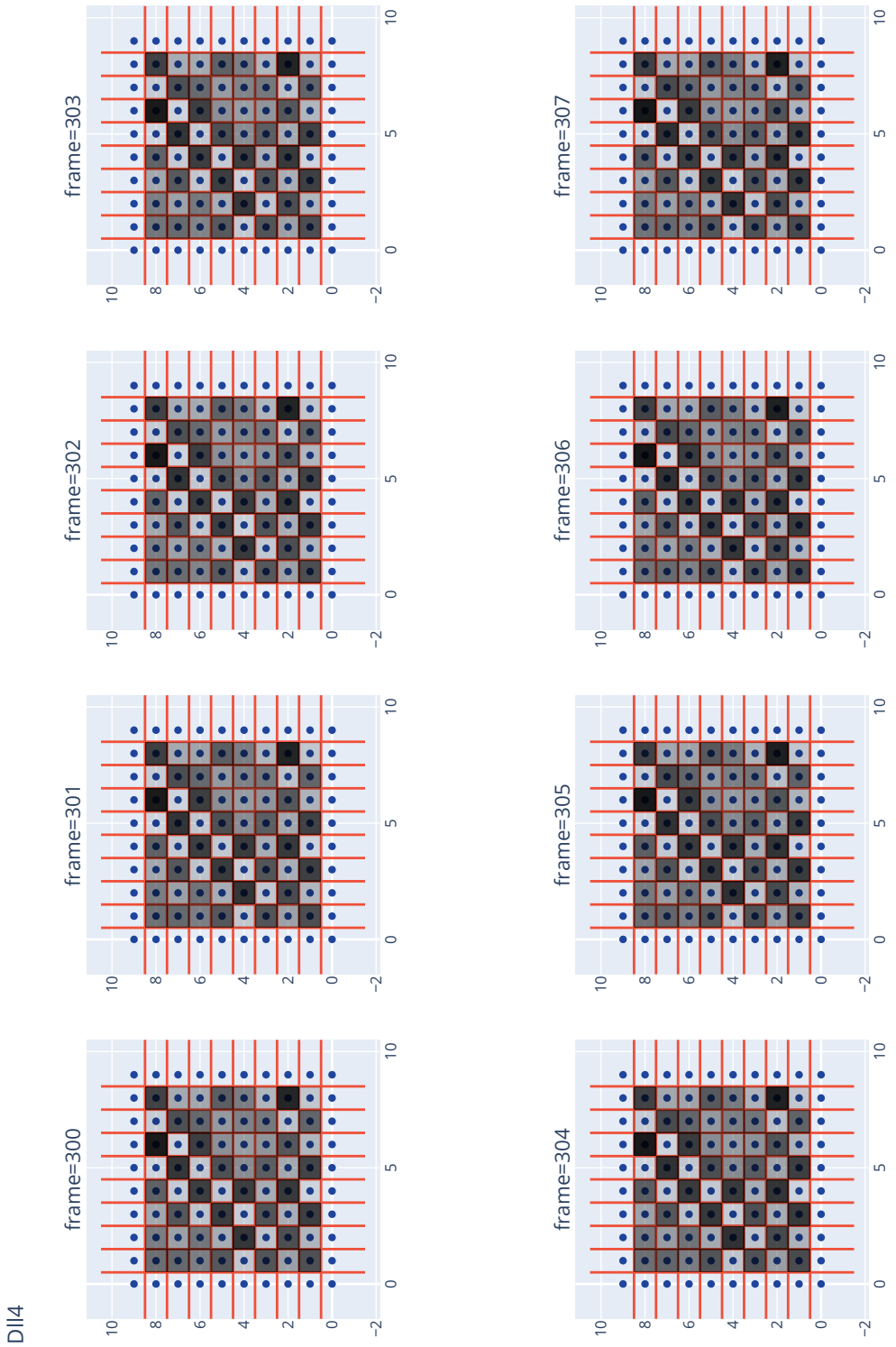

Figure A.5: Pattern change with **all cell presence** and **with protrusion**. Frame 300 to 307. (Dll4)

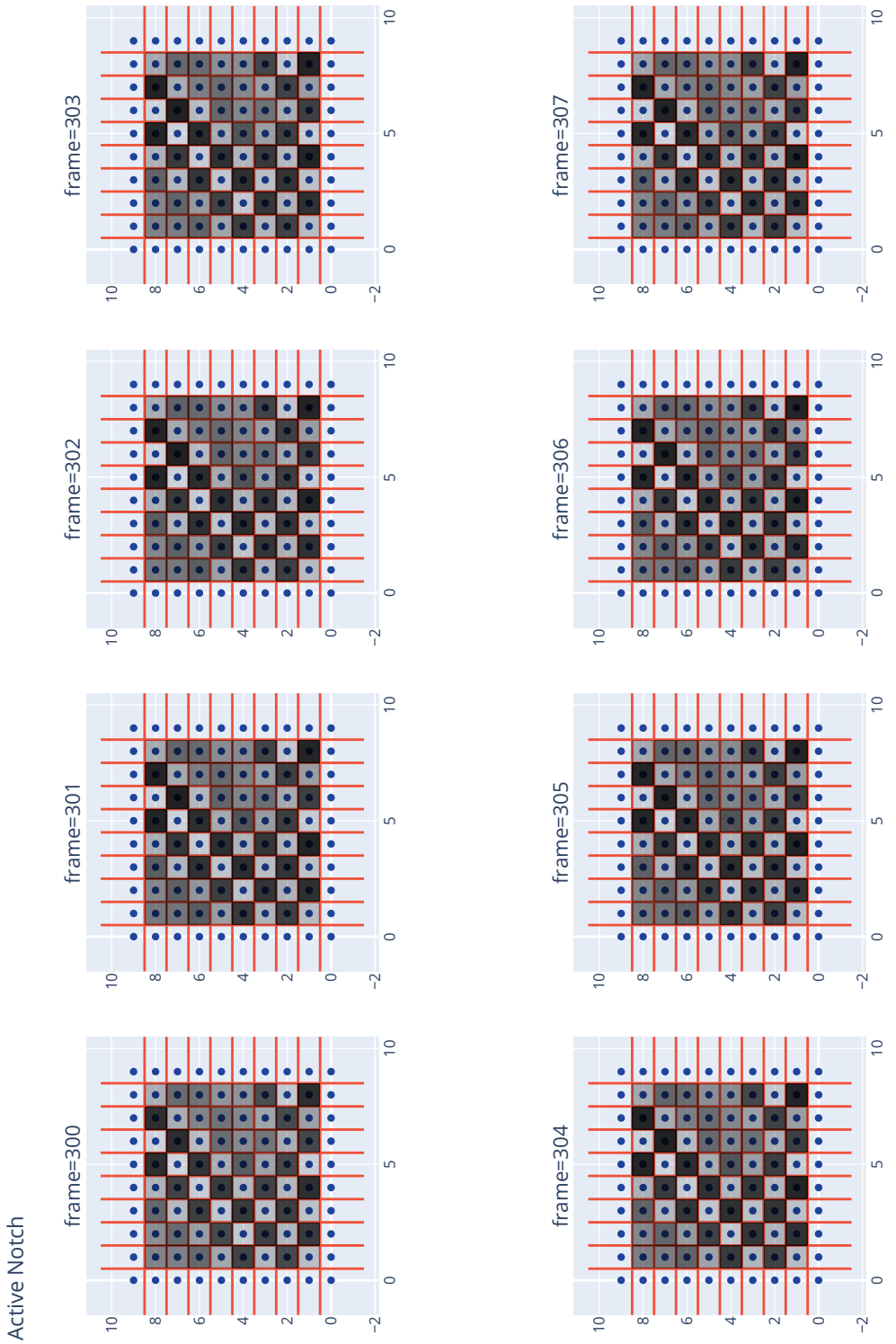

Figure A.6: Pattern change with **all cell presence** and **with protrusion**. Frame 300 to 307. (Active Notch)

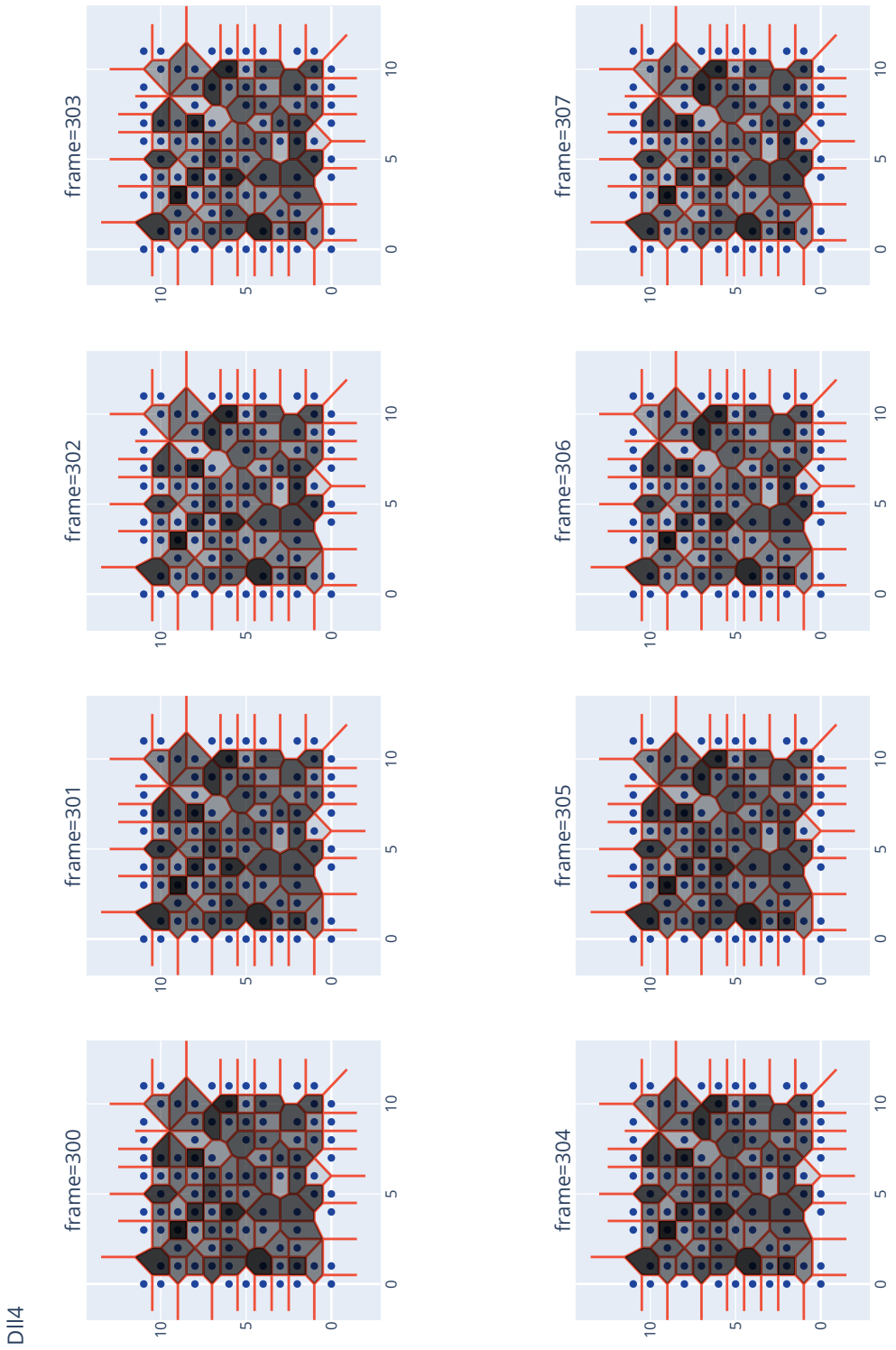

Figure A.7: Pattern change with **random cell presence** and **with protrusion**. Frame 300 to 307. (Dll4)

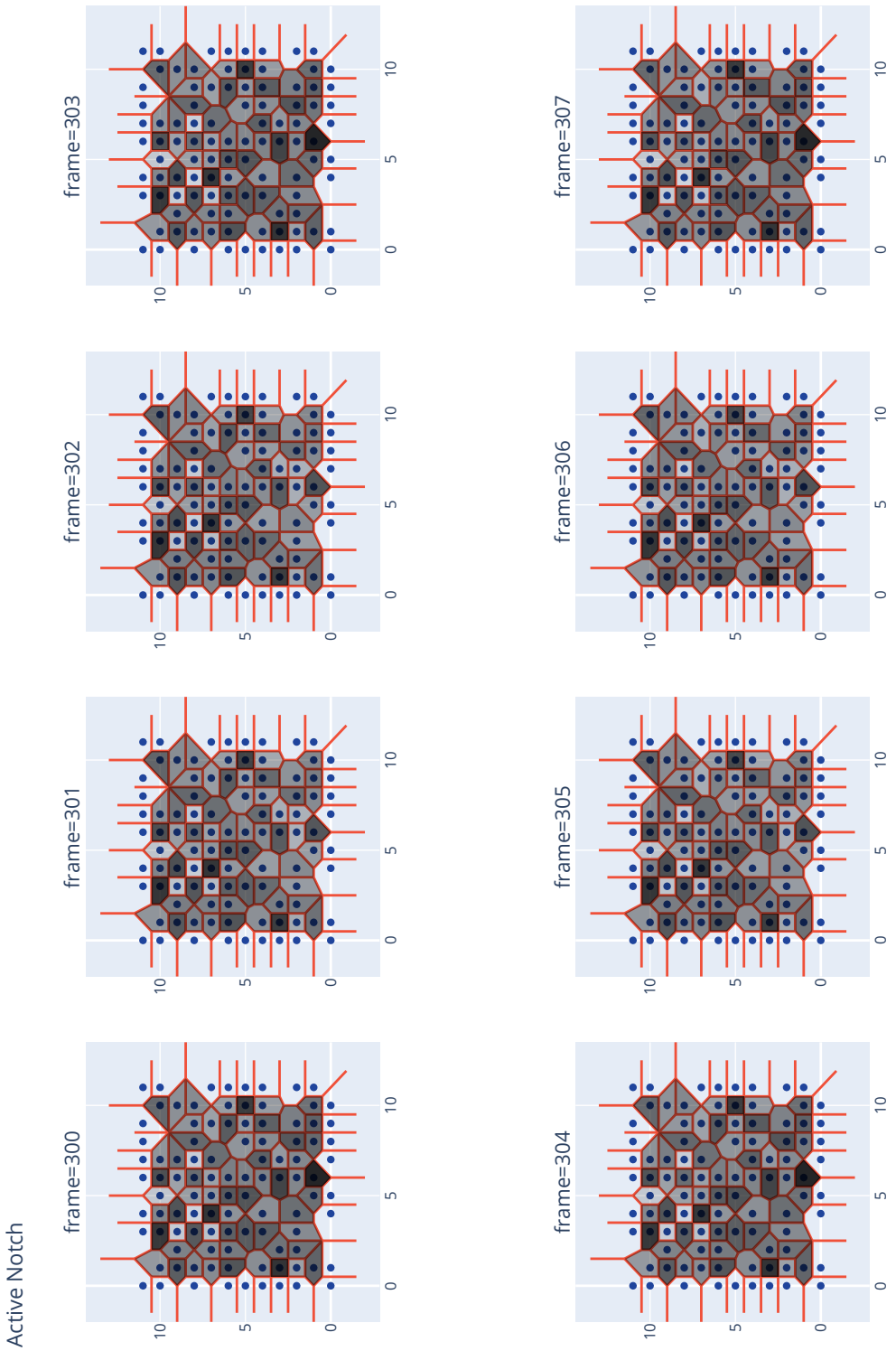

Figure A.8: Pattern change with **random cell presence** and **with protrusion**. Frame 300 to 307. (Active Notch)

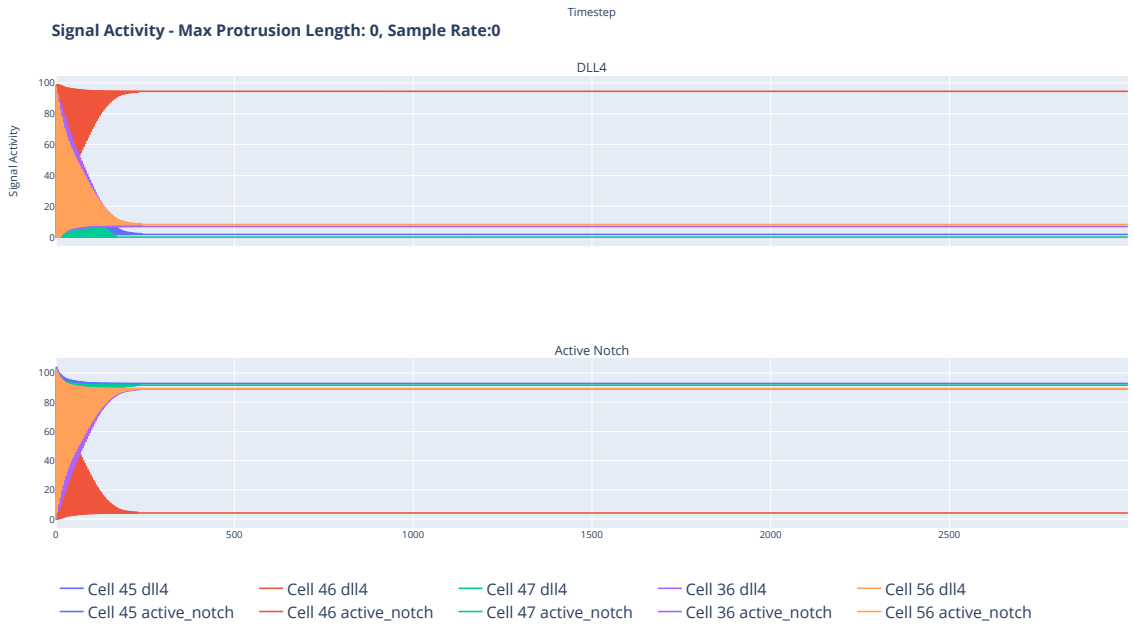

Figure A.9: Signal Activity for Cell 46 and its Neighbours.(Max Protrusion Length: 0, Sample Rate:0)

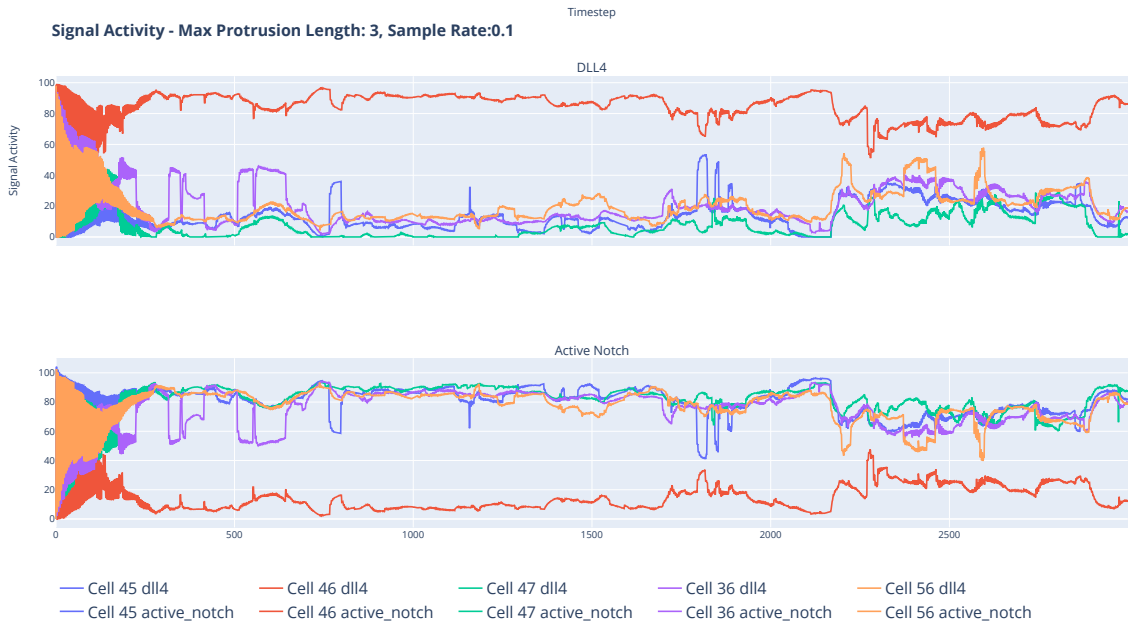

Figure A.10: Signal Activity for Cell 46 and its Neighbours.(Max Protrusion Length: 3, Sample Rate:0.1)

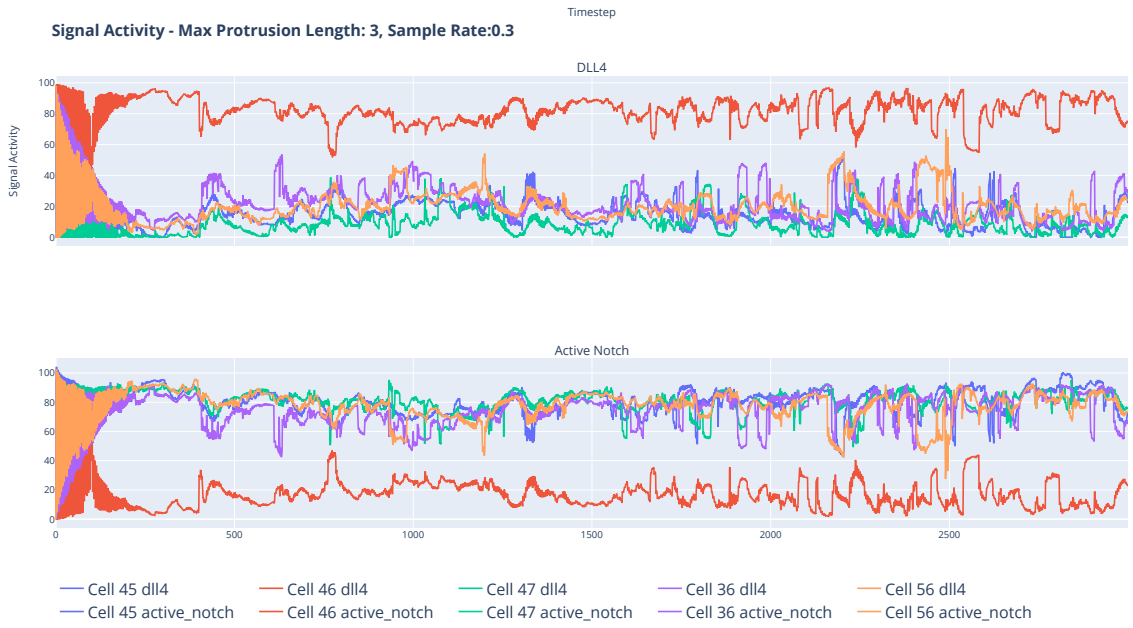

Figure A.11: Signal Activity for Cell 46 and its Neighbours.(Max Protrusion Length: 3, Sample Rate:0.3)

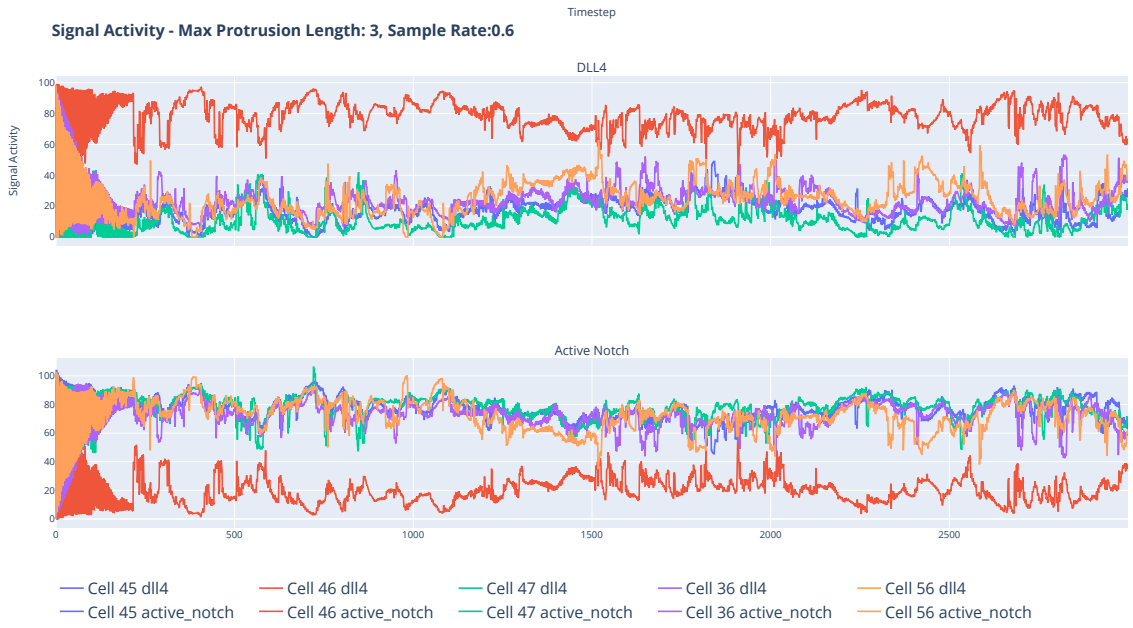

Figure A.12: Signal Activity for Cell 46 and its Neighbours.(Max Protrusion Length: 3, Sample Rate:0.6)

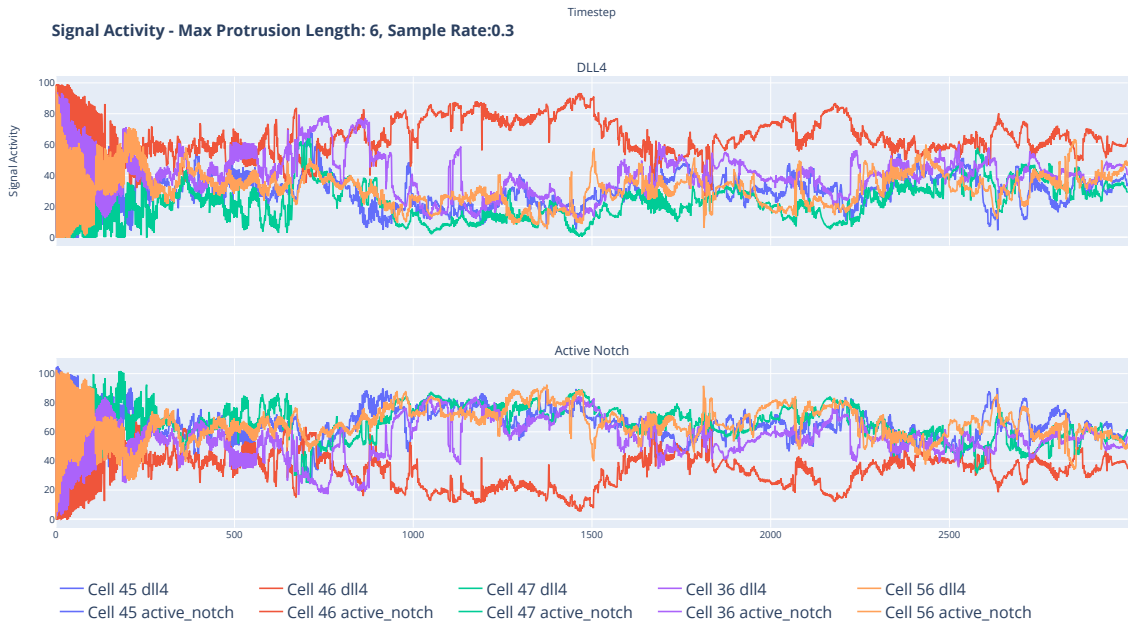

Figure A.13: Signal Activity for Cell 46 and its Neighbours.(Max Protrusion Length: 6, Sample Rate:0.3)

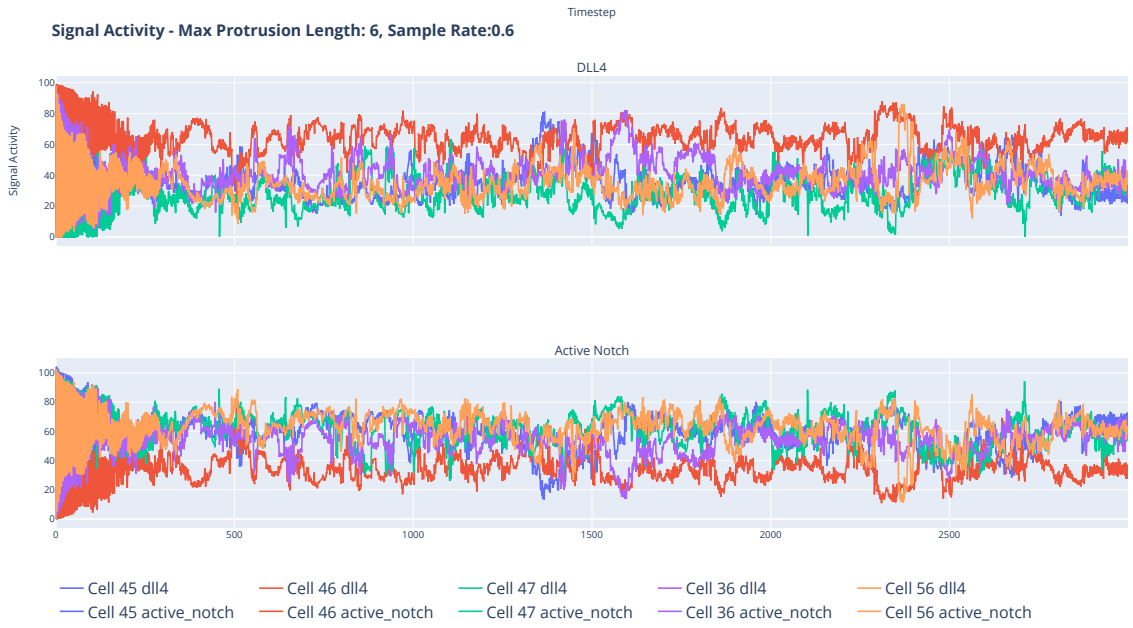

Figure A.14: Signal Activity for Cell 46 and its Neighbours.(Max Protrusion Length: 6, Sample Rate:0.6)

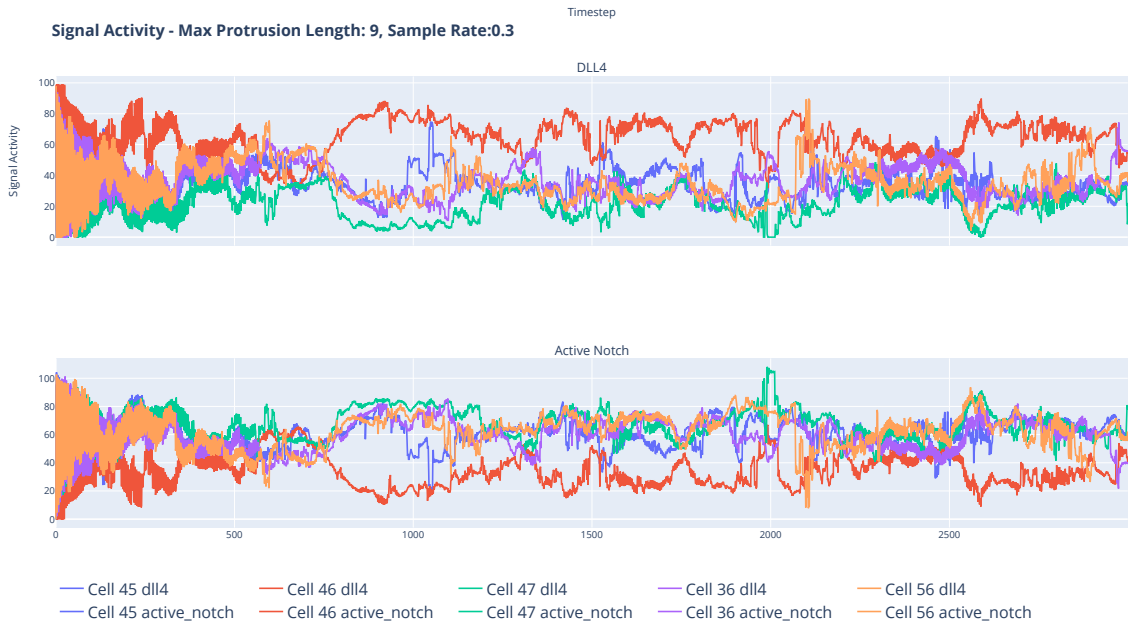

Figure A.15: Signal Activity for Cell 46 and its Neighbours.(Max Protrusion Length: 9, Sample Rate:0.3)

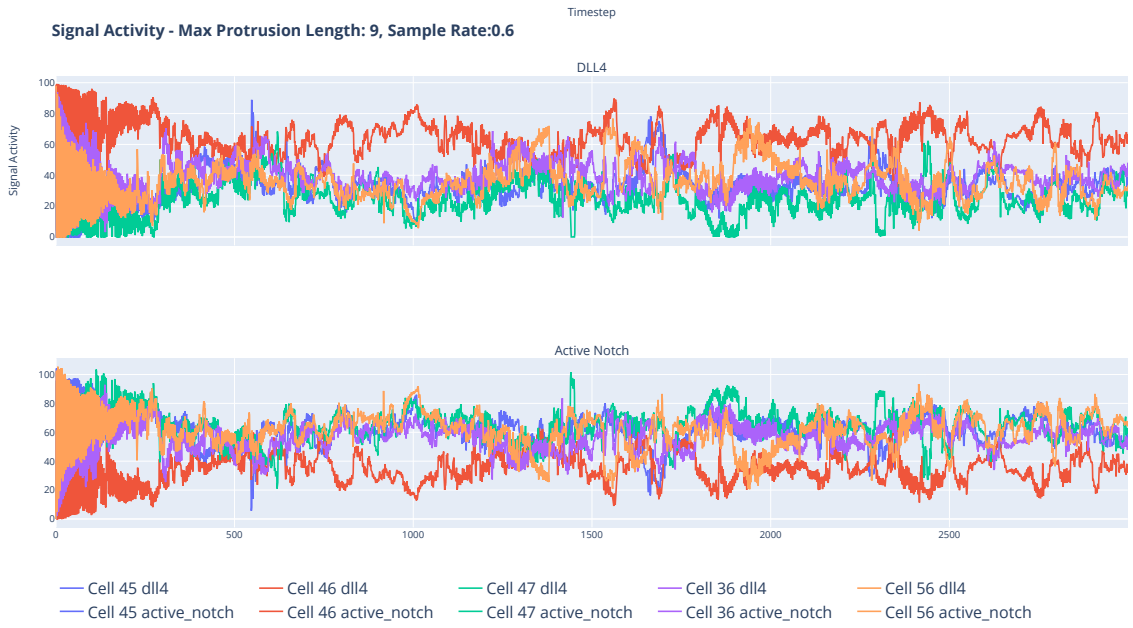

Figure A.16: Signal Activity for Cell 46 and its Neighbours.(Max Protrusion Length: 9, Sample Rate:0.6)

Appendix B

Glossary

BCS British Computing Society. 38, 39

Dll4 Delta like ligand 4. 1, 4, 6–8, 12, 15–18, 22, 23, 26, 32–34, 36, 45, 47, 49, 51

DOD Data-Oriented Design. 15

EC Endothelial Cell. 3, 4, 6, 7, 9, 26, 33–36

JIT Just-in-time compilation. 12, 24, 32, 34

SIMD Single Instruction Multiple Data. 12

VEGF Vascular Endothelial Growth Factor. 6–8, 17

VEGFR Vascular Endothelial Growth Factor Receptor. 7, 8, 17

Appendix C

Source Code

```
# attrs.py
import numpy as np
import xarray
from numba import float64
from numba import int32

from .jit import numba_guvectorize
from .network import get_connections


def naive_init_dll4(ds):
    """
    This defines the method for initialising the dll4 property.

    :param ds: cell dataset
    """
    for cell in range(ds.cells.shape[0]):
        ds.dll4[cell, 0] = np.random.normal(100, 10)


def naive_get_active_notch(ds, timestep):
    """
    This defines the method for getting the active notch level.
```

```
    :param ds: The cell dataset
    :param timestep: The timestep of the simulation.
    """
    num_cells = ds.cells.shape[0]
    ret = np.zeros(num_cells)
    for cell in range(num_cells):
        total_dll4 = 0
        connections = get_connections(ds, cell)
        for connection in connections:
            total_dll4 += ds.dll4[connection, timestep - 1]
        ret[cell] = total_dll4 / len(connections)
    return ret


def naive_get_dll4(ds, timestep):
    """
    This defines the method for getting the dll4 level.

    :param ds: The cell dataset
    :param timestep: The timestep of the simulation.
    """
    num_cells = ds.cells.shape[0]
    ret = np.zeros(num_cells)
    for cell in range(num_cells):
        # the dll4 level of the cell is the same value at time 0
        new_dll4 = ds.dll4[cell, 0] - ds.active_notch[cell, timestep - 1]
        ret[cell] = new_dll4 if new_dll4 > 0 else 0
    return ret


@numba_guvectorize(
    [(float64[:], float64[:])],
    "(n)->(n)",
)
def _init_dll4(dll4, ret):
    for i in range(dll4.shape[0]):
```

```
        ret[i] = np.random.normal(100, 10)


def init_dll4(ds):
    """
    Initialise all cell's dll4 attribute at time 0 in the dataset.

    :param ds: The dataset to initialise the dll4 attribute for.
    :return: The dataset with the dll4 attribute initialised.
    """
    ds.dll4[:, 0] = xarray.apply_ufunc(
        _init_dll4,
        ds.dll4[:, 0],
        input_core_dims=[["cells"]],
        dask="allowed",
        output_dtypes=[np.float64],
        output_core_dims=[["cells"]],
    )

    return ds


@numba_guvectorize(
    [(int32[:, :], float64[:], float64[:])],
    "(n, m),(n)->(n)",
)
def _get_active_notch(connection_matrix, dll4, ret):
    for cell in range(connection_matrix.shape[0]):
        total_dll4 = 0
        connections = 0
        for _ in range(connection_matrix.shape[1]):
            if connection_matrix[cell, connections] != -1:
                total_dll4 += dll4[connection_matrix[cell, connections]]
                connections += 1
        ret[cell] = connections and total_dll4 / connections or 0
```

```
def get_active_notch(ds, timestep):
    """
    Get the active notch level of each cell at a given timestep.

    :param ds: The dataset containing the cell attributes.
    :param timestep: The timestep to get the active notch for.
    :return: A 1D array of the active notch of each cell.
    """
    if timestep < 1:
        raise ValueError("Timestep must be at least 1.")

    return xarray.apply_ufunc(
        _get_active_notch,
        ds.cell_connections,
        ds.dll4[:, timestep - 1],
        input_core_dims=[["cells", "connections"], ["cells"]],
        dask="allowed",
        output_dtypes=[np.int32],
        output_core_dims=[["cells"]],
    )


def update_active_notch(ds, timestep):
    """
    Update the active notch level of each cell at a given timestep.

    :param ds: The cell dataset
    :param timestep: The timestep to get the active notch for.
    """
    ds.active_notch[:, timestep] = get_active_notch(ds, timestep)

    return ds


@numba_guvectorize(
```

```
    [(float64[:], float64[:], float64[:])],
    "(n),(n)->(n)",
)
def _get_dll4(dll4, notch, ret):
    for i in range(dll4.shape[0]):
        new_dll4 = dll4[i] - notch[i]
        ret[i] = new_dll4 if new_dll4 > 0 else 0


def get_dll4(ds, timestep):
    """
    Get the dll4 level of each cell at a given timestep.

    :param ds: The cell dataset
    :param timestep: The timestep to get the active notch for.
    :return: A 1D array of the active notch of each cell.
    """
    if timestep < 1:
        raise ValueError("Timestep must be at least 1.")

    return xarray.apply_ufunc(
        _get_dll4,
        ds.dll4[:, 0],
        ds.active_notch[:, timestep - 1],
        input_core_dims=[["cells"], ["cells"]],
        dask="allowed",
        output_dtypes=[np.float64],
        output_core_dims=[["cells"]],
    )


def update_dll4(ds, timestep):
    """
    Update the dll4 level of each cell at a given timestep.

    :param ds: The cell dataset
```

```
    :param timestep: The timestep to get the active notch for.
    """
    ds.dll4[:, timestep] = get_dll4(ds, timestep)

    return ds
```

```
# core.py
import xarray

from .util import get_voronoi_graph

DIM_CELL = "cells"
DIM_TIME = "time"
DIM_MAP_WIDTH = "map_width"
DIM_MAP_HEIGHT = "map_height"



def create_cell_dataset(
    *,
    pos_x,
    pos_y,
    vegf=None,
    vegfr=None,
    dll4=None,
    notch=None,
    active_notch=None,
    cell_map=None,
    cell_id_map=None,
    voronoi_graph=False,
):
    data_vars = {
        "pos_x": (DIM_CELL, pos_x),
        "pos_y": (DIM_CELL, pos_y),
    }

    attrs = {}
```

```
    if vegf is not None:
        data_vars["vegf"] = ((DIM_CELL, DIM_TIME), vegf)

    if vegfr is not None:
        data_vars["vegfr"] = ((DIM_CELL, DIM_TIME), vegfr)

    if dll4 is not None:
        data_vars["dll4"] = ((DIM_CELL, DIM_TIME), dll4)

    if notch is not None:
        data_vars["notch"] = ((DIM_CELL, DIM_TIME), notch)

    if active_notch is not None:
        data_vars["active_notch"] = ((DIM_CELL, DIM_TIME), active_notch)

    if cell_map is not None:
        data_vars["cell_map"] = ((DIM_MAP_WIDTH, DIM_MAP_HEIGHT), cell_map)

    if cell_id_map is not None:
        data_vars["cell_id_map"] = ((DIM_MAP_WIDTH, DIM_MAP_HEIGHT), cell_id_map)

    if voronoi_graph:
        attrs["voronoi_graph"] = get_voronoi_graph(pos_x, pos_y)

    return xarray.Dataset(data_vars=data_vars, attrs=attrs)
```

```
# dataset.py
import pathlib

import fsspec
import xarray as xr


def remove_attrs(ds):
    """
```

```
    Remove all attributes from dataset

    :param ds: xarray dataset
    :return: xarray dataset
    """
    for v in ds.variables.values():
        v.attrs = {}
    ds.attrs = {}
    return ds


def open_dataset(store, **kwargs):
    """
    Open dataset from zarr store

    :param store: zarr store
    :param kwargs: keyword arguments to pass to xarray.open_zarr
    """
    return xr.open_zarr(store, consolidated=True, concat_characters=False, **kwargs)


def save_dataset(ds, store, storage_options=None, **kwargs):
    """
    Save dataset to zarr store

    :param ds: xarray dataset
    :param store: zarr store
    :param kwargs: keyword arguments to pass to ds.to_zarr
    """
    if isinstance(store, str):
        storage_options = storage_options or {}
        store = fsspec.get_mapper(store, **storage_options)
    elif isinstance(store, pathlib.Path):
        store = str(store)
    ds = remove_attrs(ds)
    ds.to_zarr(store, consolidated=True, **kwargs)
```

```python
# distance.py
import numpy as np
from scipy.spatial.distance import cdist
from scipy.spatial.distance import pdist

from .jit import numba_njit


@numba_njit
def get_condensed_id(i, j, n):
    """
    Get the condensed index of the distance matrix

    :param i: row index
    :param j: column index
    :param n: number of elements
    """
    if i < j:
        i, j = j, i
    return n * j - j * (j + 1) // 2 + i - 1 - j


def manhattan_distance(XA, XB=None):
    """
    Calculate the Manhattan distance between points

    :param XA: first set of points
    :param XB: second set of points
    :return: Manhattan distance between points
    """
    if XB is None:
        return pdist(XA, metric="cityblock")
    else:
        return cdist(XA, XB, metric="cityblock")
```

```python
def euclidean_distance(XA, XB=None):
    """
    Calculate the Euclidean distance between points

    :param XA: first set of points
    :param XB: second set of points
    :return: Euclidean distance between points
    """
    if XB is None:
        return pdist(XA, metric="euclidean")
    else:
        return cdist(XA, XB, metric="euclidean")


@numba_njit
def manhattan_distance_numba(A, B):
    """
    Calculate the Manhattan distance between two points

    :param A: first point
    :param B: second point
    :return: Manhattan distance between two points
    """
    return np.sum(np.abs(A - B))


@numba_njit
def euclidean_distance_numba(A, B):
    """
    Calculate the Euclidean distance between two points

    :param A: first point
    :param B: second point
    :return: Euclidean distance between two points
    """
```

```
    return np.sqrt(np.sum((A - B) ** 2))
```

```
# jit.py
import logging
import os

import numba


logger = logging.getLogger(__name__)

_DISABLE_NUMBA = os.environ.get("DISABLE_NUMBA", "0")

try:
    ENABLE_NUMBA = {"0": True, "1": False}[_DISABLE_NUMBA]
except KeyError as e: # pragma: no cover
    raise KeyError("Environment variable 'DISABLE_NUMBA' must be '0' or '1'") from e

if not ENABLE_NUMBA:
    logger.warning(
        "numba globally disabled for phylokit; performance will be drastically"
        " reduced."
    )

DEFAULT_NO_PYTHON_ARGS = {
    "nopython": True,
    "cache": True,
}


def numba_njit(func, **kwargs):
    if ENABLE_NUMBA: # pragma: no cover
        return numba.jit(func, **{**DEFAULT_NO_PYTHON_ARGS, **kwargs})
    else:
        return func
```

```
def numba_jit(func, **kwargs):
    if ENABLE_NUMBA: # pragma: no cover
        return numba.jit(func, **kwargs)
    else:
        return func


def numba_guvectorize(*args, **kwargs):
    if ENABLE_NUMBA: # pragma: no cover
        return numba.guvectorize(*args, **{**DEFAULT_NO_PYTHON_ARGS, **kwargs})
    else:
        return numba.guvectorize(*args, forceobj=True, **kwargs)
```

```
# metric.py
import networkx
import numpy as np

from . import util


def characteristic_path_length(ds):
    """
    Returns the characteristic path length of the network graph.

    :param ds: The dataset.
    :return: The characteristic path length.
    """
    return networkx.average_shortest_path_length(ds.network_graph)


def clustering_coefficient(ds):
    """
    Returns the clustering coefficient of the network graph.

    :param ds: The dataset.
```

```
    :return: The clustering coefficient.
    """
    return networkx.average_clustering(ds.network_graph)


def in_degree(ds):
    """
    Returns the in-degree of the network graph.

    :param ds: The dataset.
    :return: The in-degree.
    """
    if util.is_directed(ds):
        return np.array(list(ds.network_graph.in_degree()))[:, 1]


def out_degree(ds):
    """
    Returns the out-degree of the network graph.

    :param ds: The dataset.
    :return: The out-degree.
    """
    if util.is_directed(ds):
        return np.array(list(ds.network_graph.out_degree()))[:, 1]
```

```
# network.py
import random

import networkx
import numpy as np
import xarray
from numba import float64
from numba import int32

from . import distance
```

```
from . import util
from .jit import numba_guvectorize
from .jit import numba_njit


@numba_njit
def _deprecated_get_cell_connection(pos_x, pos_y, cell_id_map, connection_range):
    """
    Get the connections for each cell in the dataset

    :param pos_x: x positions
    :param pos_y: y positions
    :param cell_id_map: map of cell ids
    :param connection_range: the maximum distance between two cells to be connected
    :return: array of connection ids
    """
    connections = np.full((connection_range * 2 + 1) ** 2, -1, dtype=np.int32)
    num_connections = 0
    for i in range(pos_x - connection_range, pos_x + connection_range + 1):
        for j in range(pos_y - connection_range, pos_y + connection_range + 1):
            if (
                util.is_in_bounds(i, j, cell_id_map.shape[0])
                and (i, j) != (pos_x, pos_y)
                and cell_id_map[i, j] != -1
            ):
                connections[num_connections] = cell_id_map[i][j]
                num_connections += 1

    return connections


@numba_guvectorize(
    [(int32[:], int32[:], int32[:, :], int32, int32[:], int32[:, :])],
    "(n),(n),(w, h),(),(c)->(n,c)",
)
def _deprecated_get_cell_connections(
```

```
    pos_x, pos_y, cell_id_map, connection_range, connection, out
):
    """
    Get the connections for each cell in the dataset

    :NOTE The dimensions for each of the parameters should be fixed as follows:
         "(n),(n),(w, h),(),(c)->(n,c)"
         where:
           :n = number of cells
           :w = width of the map
           :h = height of the map
           :c = max number of connections (connection_range * 2 + 1) ** 2

    :param pos_x: x position of the cells
    :param pos_y: y position of the cells
    :param cell_id_map: map of cell ids
    :param connection_range: the maximum distance between two cells to be connected
    :param connection: shape of the connection matrix
    :param out: output array of connections
    """
    for n in range(pos_x.shape[0]):
        out[n] = _deprecated_get_cell_connection(
            pos_x[n], pos_y[n], cell_id_map, connection_range
        )


def deprecated_get_connection_matrix_parallel(ds, connection_range):
    """
    Creates a connection matrix from a grid using a guvectorize function via xarray
        apply_ufunc
    NOTE: This function is slower than pure guvectorize function due to the data
        transfer between
           the dask workers and the scheduler

    :param ds: Dataset to create a connection matrix from
    :param connection_range: The maximum distance between two cells to be connected
```

```
    :return DataArray of connection matrix
    """
    connection = np.zeros((connection_range * 2 + 1) ** 2, dtype=np.int32)

    return xarray.apply_ufunc(
        _deprecated_get_cell_connections,
        ds.pos_x,
        ds.pos_y,
        ds.cell_id_map,
        connection_range,
        connection,
        input_core_dims=[
            [],
            [],
            ["map_width", "map_height"],
            [],
            ["connections"],
        ],
        dask="parallelized",
        output_dtypes=[np.int32],
        output_core_dims=[["connections"]],
    )


def deprecated_get_connection_matrix(ds, connection_range):
    """
    Creates a connection matrix from a grid using a guvectorize function via xarray
        apply_ufunc

    :param ds: Dataset to create a connection matrix from
    :param connection_range: The maximum distance between two cells to be connected

    :return DataArray of connection matrix
    """
    connection = np.zeros((connection_range * 2 + 1) ** 2, dtype=np.int32)
```

```
    return xarray.apply_ufunc(
        _deprecated_get_cell_connections,
        ds.pos_x,
        ds.pos_y,
        ds.cell_id_map,
        connection_range,
        connection,
        input_core_dims=[
            ["cells"],
            ["cells"],
            ["map_width", "map_height"],
            [],
            ["connections"],
        ],
        dask="allowed",
        output_dtypes=[np.int32],
        output_core_dims=[["cells", "connections"]],
    )


def deprecated_update_connection_matrix(ds, connection_range):
    """
    Updates the connection matrix in the dataset

    :param ds: Dataset to update the connection matrix in
    :param connection_range: The maximum distance between two cells to be connected

    :return Dataset with updated connection matrix
    """
    if "connections" in ds.dims:
        ds = ds.drop_dims("connections")

    ds["cell_connections"] = deprecated_get_connection_matrix(ds, connection_range)

    return ds
```

```
def init_connection_matrix(ds):
    """
    Initialise the connection matrix in the dataset

    :param ds: Dataset to initialise the connection matrix in
    :return Dataset with initialised connection matrix
    """
    _pad = 0

    if "connections" in ds.dims:
        ds = ds.drop_dims("connections")

    if "cell_protrusions" in ds:
        _pad = ds.protrusion.shape[0]

    ds["cell_connections"] = (
        ["cells", "connections"],
        np.pad(
            get_neighbour_matrix(ds),
            ((0, 0), (0, _pad)),
            "constant",
            constant_values=-1,
        ),
    )

    return ds


def get_connections(ds, cell_id):
    """
    Get the connections for a cell

    :param ds: Dataset to get the connections from
    :param cell_id: Cell to get the connections for
```

```
    :return Array of connections
    """
    if cell_id < 0 or cell_id >= ds.cells.shape[0]:
        raise IndexError("Cell id out of bounds")

    if "cell_connections" not in ds:
        raise ValueError(
            """Dataset does not contain a connection matrix,
                        try run `update_connection_matrix(ds)`"""
        )

    return util.get_valid_elements(ds.cell_connections[cell_id].data)


@numba_njit
def remove_cell_pair(connection_matrix, cell_1, cell_2):
    """
    Remove a pair of cells from the connection matrix

    :param connection_matrix: The connection matrix to remove the neighbours from
    :param cell_1: The first cell
    :param cell_2: The second cell
    """
    for i in range(connection_matrix.shape[1]):
        if connection_matrix[cell_1, i] == cell_2:
            connection_matrix[cell_1, i] = -1
    for j in range(connection_matrix.shape[1]):
        if connection_matrix[cell_2, j] == cell_1:
            connection_matrix[cell_2, j] = -1


@numba_njit
def add_cell_pair(connection_matrix, cell_1, cell_2):
    """
    Add a pair of cells to the connection matrix
```

```
    :param connection_matrix: The connection matrix to add the neighbours to
    :param cell_1: The first cell
    :param cell_2: The second cell
    """
    for i in range(connection_matrix.shape[1]):
        if connection_matrix[cell_1, i] == -1:
            connection_matrix[cell_1, i] = cell_2
            break
    for j in range(connection_matrix.shape[1]):
        if connection_matrix[cell_2, j] == -1:
            connection_matrix[cell_2, j] = cell_1
            break


@numba_njit
def _get_neighbours_matrix(num_cells, ridge_points):
    """
    Get the neighbours matrix from the ridge points

    :param num_cells: The number of cells in the grid
    :param ridge_points: The ridge points from the voronoi graph
    :return: The neighbours matrix
    """
    count_neighbours = np.bincount(ridge_points.reshape(-1))
    max_neighbours = np.max(count_neighbours)

    # initialise the connection matrix with -1
    neightbour_matrix = np.full(
        (num_cells, max_neighbours), fill_value=-1, dtype=np.int32
    )

    for cell_pairs in ridge_points:
        add_cell_pair(neightbour_matrix, cell_pairs[0], cell_pairs[1])

    return neightbour_matrix
```

```
def get_neighbour_matrix(ds):
    """
    Get the neighbours matrix from the dataset

    :param ds: Dataset to get the neighbours matrix from
    :return: cell neighbours matrix
    """
    vor = ds.voronoi_graph

    return _get_neighbours_matrix(ds.cells.shape[0], vor.ridge_points)


def update_neighbour_matrix(ds):
    """
    Update the cell neighbours matrix in the dataset

    :param ds: Dataset to update the neighbours matrix in
    :return: Dataset with updated neighbours matrix
    """
    if "neighbours" in ds.dims:
        ds.drop_dims("neighbours")

    ds["cell_neighbours"] = ("cells", "neighbours"), get_neighbour_matrix(ds)

    return ds


def get_neighbours(ds, cell_id):
    """
    Get the neighbours of a cell

    :param ds: Dataset to get the neighbours from
    :param cell_id: Cell id to get the neighbours of
    :return: Neighbours of the cell
```

```
    """
    if cell_id < 0 or cell_id >= ds.cells.shape[0]:
        raise IndexError("Cell id out of bounds")

    if "cell_neighbours" not in ds:
        raise ValueError(
            """Dataset does not contain a neighbours matrix,
                        try run `update_neighbour_matrix(ds)`"""
        )

    return util.get_valid_elements(ds.cell_neighbours[cell_id].data)


# Network graph functions


@numba_guvectorize(
    [(int32[:, :], float64[:], float64[:], float64[:, :, :])],
    "(n, m), (w), (i) -> (n, m, i)",
)
def _get_network_graph_edges(connection_matrix, weights, i, ret):
    for cell in range(connection_matrix.shape[0]):
        connections = 0
        while connections < connection_matrix.shape[1]:
            if connection_matrix[cell, connections] == -1:
                break
            ret[cell, connections, 0] = cell
            ret[cell, connections, 1] = connection_matrix[cell, connections]
            ret[cell, connections, 2] = weights[
                distance.get_condensed_id(
                    cell,
                    connection_matrix[cell, connections],
                    connection_matrix.shape[0],
                )
            ]
            connections += 1
```

```
def get_network_graph(ds, directed=False):
    """
    Get the network graph from the dataset

    :param ds: Dataset to get the network graph from
    :param directed: Whether the graph should be directed or not
    :return: Network graph
    """
    if "cell_connections" not in ds:
        init_connection_matrix(ds)

    pos = np.array([ds.pos_x, ds.pos_y]).T
    weights = distance.euclidean_distance(pos)

    if directed:
        graph = networkx.DiGraph()
    else:
        graph = networkx.Graph()

    # Add the nodes to the graph
    graph.add_nodes_from(ds.cells.data)

    # Add the edges to the graph
    ret = np.full(
        (ds.cell_connections.shape[0], ds.cell_connections.shape[1], 3),
        -1,
        dtype=np.float64,
    )

    xarray.apply_ufunc(
        _get_network_graph_edges,
        ds.cell_connections,
        weights,
        np.zeros(3, dtype=np.float64),
```

```
        ret,
        input_core_dims=[["cells", "connections"], ["weights"], [], []],
        output_core_dims=[["cells", "connections", "data"]],
        output_dtypes=[np.float64],
    )

    graph.add_weighted_edges_from(ret.reshape(-1, 3))

    # Remove the -1 edges and nodes because of
    # ‘ebunch_to_add‘ in ‘add_weighted_edges_from‘
    graph.remove_edge(-1, -1)
    graph.remove_node(-1)

    return graph


def update_network_graph(ds, directed=False):
    """
    Update the network graph in the dataset

    :param ds: Dataset to update the network graph in
    :param directed: Whether the graph should be directed or not
    :return: Dataset with updated network graph
    """

    ds.attrs["network_graph"] = get_network_graph(ds, directed)

    return ds


# Protrusion functions


def init_protrusion(ds, protrusion_max_length=3):
    """
    Initialize the protrusion matrix in the dataset and add the voronoi
```

```
    ridge dictionary and ridge availability to the attributes

    :param ds: Dataset to initialize the protrusion matrix in
    :param protrusion_max_length: Maximum length of a protrusion
    :return: Dataset with initialized protrusion matrix
    """
    if "cell_protrusions" in ds:
        ds = ds.drop_vars("cell_protrusions")

    ds["cell_protrusions"] = xarray.DataArray(
        np.full((ds.cells.shape[0], protrusion_max_length), -1, dtype=np.int32),
        dims=("cells", "protrusion"),
    )

    ridge_vertices = ds.voronoi_graph.ridge_vertices
    ridge_dict = ds.voronoi_graph.ridge_dict

    ds.cell_protrusions.attrs["inv_ridge_dict"] = {
        frozenset(v): frozenset(k) for k, v in ridge_dict.items()
    }
    ds.cell_protrusions.attrs["ridge_availability"] = {
        frozenset(v): True for _, v in ridge_dict.items()
    }

    vor_ridge_graph = networkx.Graph()

    for ridge in ridge_vertices:
        if -1 not in ridge:
            vor_ridge_graph.add_edge(ridge[0], ridge[1])

    ds.attrs["vor_ridge_graph"] = vor_ridge_graph

    return ds


def _is_neighbour(cell_0, cell_1, neighbours_matrix):
```

```
    """
    Check if two cells are neighbours

    :param cell_0: First cell
    :param cell_1: Second cell
    :param neighbours_matrix: Neighbours matrix
    """
    if cell_0 in neighbours_matrix[cell_1] and cell_1 in neighbours_matrix[cell_0]:
        return True
    else:
        return False


def _update_cell_pairs(
    inv_ridge_dict, connection_matrix, neighbours_matrix, walk, cell, operation=None
):
    """
    Update the cell pairs in the connection matrix based on the random walk and the
    operation to perform on the cell pairs

    :param inv_ridge_dict: Inverse ridge dictionary
    :param connection_matrix: Connection matrix
    :param neighbours_matrix: Neighbours matrix
    :param walk: Random walk
    :param cell: Cell to perform the operation on
    :param operation: Operation to perform on the cell pairs
    """
    associated_cells = list(inv_ridge_dict[frozenset(walk)])
    if cell not in associated_cells:
        if not _is_neighbour(
            cell, associated_cells[0], neighbours_matrix
        ) and not _is_neighbour(cell, associated_cells[1], neighbours_matrix):
            if operation == "add":
                add_cell_pair(connection_matrix, cell, associated_cells[0])
                add_cell_pair(connection_matrix, cell, associated_cells[1])
                remove_cell_pair(
```

```
                connection_matrix, associated_cells[0], associated_cells[1]
            )
        elif operation == "remove":
            remove_cell_pair(connection_matrix, cell, associated_cells[0])
            remove_cell_pair(connection_matrix, cell, associated_cells[1])
            add_cell_pair(
                connection_matrix, associated_cells[0], associated_cells[1]
            )
        else:
            raise ValueError("operation must be either 'add' or 'remove'")


def networkx_random_walk(graph, start_nodes=None, num_walks=10, num_steps=3):
    """
    Perform a random walk on the networkx graph

    :param graph: Networkx graph
    :param start_nodes: Nodes to start the random walk from
    :param num_walks: Number of random walks to perform
    :param num_steps: Number of steps to take in each random walk
    :return: List of random walks
    """
    walks = list()
    for i in start_nodes:
        for _ in range(num_walks):
            curr_walk = [i]
            curr = i
            for _ in range(num_steps):
                neighbors = list(graph.neighbors(curr))
                if len(neighbors) > 0:
                    curr = random.choice(neighbors)
                curr_walk.append(curr)
            walks.append(curr_walk)
    return walks
```

```
def simulate_protrusion_one_step(ds, start_nodes=None, sample_rate=0.2):
    """
    Simulate one step of protrusion growth or retraction in the dataset by
    performing a random walk on the voronoi ridge graph and updating the
    protrusion matrix and the cell connections matrix accordingly

    :param ds: Dataset to simulate protrusion growth or retraction in
    :param sample_rate: Rate at which to sample cells to perform protrusion
    growth or retraction on
    :return: Dataset with updated protrusion matrix and cell connections matrix
    """
    inv_ridge_dict = ds.cell_protrusions.inv_ridge_dict
    ridge_availability = ds.cell_protrusions.ridge_availability
    regions = ds.voronoi_graph.regions
    point_region = ds.voronoi_graph.point_region
    protrusion = ds.cell_protrusions.data
    cell_connections = ds.cell_connections.data
    cell_neighbours = ds.cell_neighbours.data
    num_cells = len(ds.cells)

    if start_nodes:
        num_samples = len(start_nodes)
    else:
        num_samples = int(sample_rate * num_cells)
        start_nodes = np.random.choice(num_cells, size=num_samples, replace=False)

    start_vertices = np.zeros_like(start_nodes)

    for i, cell in enumerate(start_nodes):
        if protrusion[cell, 0] == -1:
            cell_ridge_vertices = np.array(regions[point_region[cell]])
            vertices = util.get_valid_elements(cell_ridge_vertices)
            start_vertices[i] = np.random.choice(vertices, 1)
            protrusion[cell, 0] = start_vertices[i]
        else:
            start_vertices[i] = util.get_valid_elements(protrusion[cell])[-1]
```

```
walks = networkx_random_walk(
    ds.vor_ridge_graph,
    start_nodes=start_vertices,
    num_walks=1,
    num_steps=1,
)


for walk, cell in zip(walks, start_nodes):
    cell_protrusion = util.get_valid_elements(protrusion[cell])
    p = cell_protrusion.shape[0] - 1
    if p > 0 and protrusion[cell, p - 1] == walk[1]:
        protrusion[cell, p] = -1
        _update_cell_pairs(
            inv_ridge_dict,
            cell_connections,
            cell_neighbours,
            walk,
            cell,
            operation="remove",
        )
        ridge_availability[frozenset(walk)] = True
        if p == 1:
            protrusion[cell, 0] = -1
        continue
    if p + 1 == protrusion.shape[1]:
        continue
    if ridge_availability[frozenset(walk)]:
        protrusion[cell, p + 1] = walk[1]
        _update_cell_pairs(
            inv_ridge_dict,
            cell_connections,
            cell_neighbours,
            walk,
            cell,
            operation="add",
```

```
            )
            ridge_availability[frozenset(walk)] = False

    return ds


# Cell positions


@numba_njit
def get_cell_id_map(cell_map):
    """
    Get the cell ids for each cell in the grid

    :param cell_map: Cell map
    :return: Cell ids
    """
    width = cell_map.shape[0]
    height = cell_map.shape[1]
    cell_ids = np.zeros((width, height), dtype=np.int32)
    cell_id = 1
    for i in range(cell_map.shape[0]):
        for j in range(cell_map.shape[1]):
            if cell_map[i, j] == 1:
                cell_ids[i, j] = cell_id
                cell_id += 1

    return cell_ids - 1


@numba_njit
def get_cell_meshgrid(cell_map, num_cells):
    """
    Get the cell positions for each cell in the grid

    :param cell_map: Cell positions
```

```
    :param num_cells: Number of cells
    :return: Cell positions
    """
    cell_meshgrid = np.zeros((2, num_cells), dtype=np.int32)
    cell_id = 0
    for i in range(cell_map.shape[0]):
        for j in range(cell_map.shape[1]):
            if cell_map[i, j] == 1:
                cell_meshgrid[0, cell_id] = i
                cell_meshgrid[1, cell_id] = j
                cell_id += 1

    return cell_meshgrid[0], cell_meshgrid[1]


def deprecated_large_cell_grid(dim, prob=1.0):
    """
    Creates a grid with probability of generating a large cell

    :param dim: Dimension of the grid
    :param prob: probability of generating a cell
    :return: xx, yy: x and y coordinates of the grid
    """
    if prob < 0.0 or prob > 1.0:
        raise ValueError("Randomness must be between 0.0 and 1.0")

    cell_map = np.random.binomial(1, 1, size=(dim, dim)).astype(np.int32)

    large_cells = np.random.randint(0, dim, size=(int((dim**2) * prob), 2))
    for i in range(large_cells.shape[0]):
        cell_map[large_cells[i, 0], large_cells[i, 1]] = 1
        for j in range(large_cells[i, 0] - 1, large_cells[i, 0] + 2):
            for k in range(large_cells[i, 1] - 1, large_cells[i, 1] + 2):
                if util.is_in_bounds(j, k, dim) and (j, k) != (
                    large_cells[i, 0],
                    large_cells[i, 1],
```

```
            ):
                cell_map[j, k] = 0

    num_cells = np.count_nonzero(cell_map)
    cell_id_map = get_cell_id_map(cell_map).astype(np.int32)
    xx, yy = get_cell_meshgrid(cell_map, num_cells)

    return xx, yy, cell_map, cell_id_map


@numba_njit
def simple_grid(dim, prob=1.0):
    """
    Creates a simple grid with noise using numpy.meshgrid

    :param dim: Dimension of the grid
    :param prob: probability of generating a cell
    :return: xx, yy, cell_map, cell_id_map: x and y coordinates of the grid,
    cell map and cell id map of the grid
    """
    if prob < 0.0 or prob > 1.0:
        raise ValueError("Randomness must be between 0.0 and 1.0")

    cell_map = np.random.binomial(1, prob, size=(dim, dim)).astype(np.int32)
    num_cells = np.count_nonzero(cell_map)
    cell_id_map = get_cell_id_map(cell_map).astype(np.int32)
    xx, yy = get_cell_meshgrid(cell_map, num_cells)

    return xx, yy, cell_map, cell_id_map
```

```
# plot.py
import random

import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
```

```
def plotly_signal_activity(ds, cell_ids=None):
    """
    Plot the change in signal activity over time.
    Plot dll4, active_notch, vegf, vegfr in 4 separate subplots
    in the same figure.

    :param ds: The dataset.
    :return: The plot.
    """
    colors = px.colors.qualitative.Plotly
    signals = ["dll4", "active_notch"]
    if cell_ids is None:
        cell_ids = random.sample(list(ds.cells.data), 10)
    fig = make_subplots(
        rows=1,
        cols=2,
        subplot_titles=("DLL4", "Active Notch"),
        shared_xaxes=True,
        shared_yaxes=True,
    )
    for j, cell in enumerate(cell_ids):
        color = colors[j % len(colors)]
        for i, signal in enumerate(signals):
            fig.add_trace(
                go.Scatter(
                    x=ds.time,
                    y=ds[signal][cell],
                    mode="lines",
                    name=f"Cell {cell} {signal}",
                    legendgroup=f"Cell {cell}",
                    line=dict(color=color),
                ),
                row=i // 2 + 1,
                col=i % 2 + 1,
```

```
            )

    fig.update_layout(
        title="Signal Activity", xaxis_title="Time", yaxis_title="Signal", height=800
    )
    return fig
```

```
# util.py
import re

import numpy as np
from scipy.spatial import Voronoi

from .jit import numba_jit
from .jit import numba_njit


@numba_njit
def is_in_bounds(x, y, dim):
    """
    Check if x and y are in bounds of a square grid of size dim x dim

    :param x: x coordinate
    :param y: y coordinate
    :param dim: dimension of grid
    :return: True if x and y are in bounds, False otherwise
    """
    return 0 <= x < dim and 0 <= y < dim


def get_voronoi_graph(pos_x, pos_y):
    """
    Get voronoi graph from cell positions

    :param pos_x: x positions of cells
    :param pos_y: y positions of cells
```

```
    :return: voronoi graph
    """
    vor = Voronoi(np.array([pos_x, pos_y]).T)
    return vor


get_voronoi_graph = numba_jit(get_voronoi_graph, forceobj=True, cache=True)


@numba_njit
def get_valid_elements(a):
    """
    Get all elements greater than minus one from an array

    :param a: array
    :return: array of non-minus elements
    """
    return a[a > -1]


def is_directed(ds):
    """
    Check if the graph is directed

    :param ds: dataset
    :return: True if the graph is directed, False otherwise
    """
    if ds.network_graph.is_directed():
        return True
    else:
        return False


def is_valid_hex_color(color_str):
    """
    Check if a string is a valid hex color
```

```
    :param color_str: string to check
    :return: Match object if string is a valid hex color, None otherwise
    """
    regex_str = r"^#(?:[0-9a-fA-F]{3}){1,2}$"
    return re.search(regex_str, color_str)


def convert_nested_list(nested_list):
    """
    Convert a nested list to numpy array

    :param nested_list: nested list
    :return: numpy array
    """
    pad = len(max(nested_list, key=len))
    return np.array([xi + [-1] * (pad - len(xi)) for xi in nested_list])
```

```
# visualize.py
import networkx
import numpy as np
import plotly.graph_objects as go
from sklearn.preprocessing import MinMaxScaler

from .jit import numba_jit
from .jit import numba_njit
from .util import is_valid_hex_color


def _plotly_voronoi_plot_2d(vor):
    """
    This function will return the finite and infinite segments of the voronoi diagram

    NOTE: Adapted from scipy.spatial._plotutils.voronoi_plot_2d with the following
        changes:
         finite_segments and infinite_segments are 2D arrays instead of 1D arrays with
```

```
            None
        values to separate the segments

    :param vor: Voronoi diagram
    :return list, list: Finite and infinite segments of the voronoi diagram
    """
    center = vor.points.mean(axis=0)
    finite_segments = [[], []]
    infinite_segments = [[], []]
    for pointidx, simplex in zip(vor.ridge_points, vor.ridge_vertices):
        simplex = np.asarray(simplex)
        if np.all(simplex >= 0):
            finite_segments[0] += vor.vertices[simplex, 0].tolist()
            finite_segments[0].append(None)
            finite_segments[1] += vor.vertices[simplex, 1].tolist()
            finite_segments[1].append(None)
        else:
            i = simplex[simplex >= 0][0] # finite end Voronoi vertex
            t = vor.points[pointidx[1]] - vor.points[pointidx[0]] # tangent
            t /= np.linalg.norm(t)
            n = np.array([-t[1], t[0]]) # normal
            midpoint = vor.points[pointidx].mean(axis=0)
            direction = np.sign(np.dot(midpoint - center, n)) * n
            if vor.furthest_site:
                direction = -direction
            far_point = vor.vertices[i] + direction * 2
            infinite_segments[0] += [vor.vertices[i, 0], far_point[0], None]
            infinite_segments[1] += [vor.vertices[i, 1], far_point[1], None]

    return finite_segments, infinite_segments


_plotly_voronoi_plot_2d = numba_jit(_plotly_voronoi_plot_2d, forceobj=True)


def _get_voronoi_plot_scatters(
```

```
    pos_x,
    pos_y,
    labels,
    finite_segments,
    infinite_segments,
    marker_colour="#1F449C",
    line_colour="#F05039",
):
    scatters = [
        go.Scatter(
            x=pos_x,
            y=pos_y,
            mode="markers",
            name="cells",
            text=labels,
            marker=dict(color=marker_colour),
        ),
        go.Scatter(
            x=finite_segments[0] + infinite_segments[0],
            y=finite_segments[1] + infinite_segments[1],
            mode="lines",
            name="ridges",
            line=dict(color=line_colour),
        ),
    ]

    return scatters


def plotly_voronoi_plot_2d(
    ds, line_colour="#F05039", marker_colour="#1F449C", width=800, height=850
):
    """
    This function will return a plotly figure of the voronoi diagram

    :param ds: Dataset to plot the voronoi diagram from
```

```
    :param line_colour: Colour of the lines of the voronoi diagram
    :param marker_colour: Colour of the markers of the voronoi diagram
    :param width: Width of the plot
    :param height: Height of the plot
    :return plotly.graph_objs.Figure: Plotly figure of the voronoi diagram
    """

    if not is_valid_hex_color(line_colour) or not is_valid_hex_color(marker_colour):
        raise ValueError("Colour is not valid. Please use a valid hex colour code.")

    vor = ds.voronoi_graph

    fig = go.Figure()
    labels = [f"cell {i}" for i in range(len(ds.pos_x))]
    finite_segments, infinite_segments = _plotly_voronoi_plot_2d(vor)

    fig.add_traces(
        _get_voronoi_plot_scatters(
            ds.pos_x,
            ds.pos_y,
            labels,
            finite_segments,
            infinite_segments,
            marker_colour=marker_colour,
            line_colour=line_colour,
        )
    )

    fig.update_layout(
        title="Cell Map",
        xaxis_title="x",
        yaxis_title="y",
        width=width,
        height=height,
        autosize=False,
        margin=go.layout.Margin(
```

```
            l=40,
            r=40,
            b=85,
            t=100,
        ),
    )

    return fig


def _plotly_visualize_attr(regions, vertices, values, labels, points):
    scatters = []
    for i, region in enumerate(regions):
        if (len(region) > 0) and (-1 not in region):
            point_index = np.where(points == i)[0][0]
            x = vertices[region, 0]
            y = vertices[region, 1]
            z = values[point_index]
            scatter = go.Scatter(
                x=x,
                y=y,
                fill="toself",
                fillcolor=f"rgba(0, 0, 0, {'%.2f' % z})",
                line_color="rgba(255,255,255,0)",
                showlegend=False,
                hoverinfo="text",
                text=labels[point_index],
            )
            scatters.append(scatter)
    return scatters


def plotly_visualize_attr(ds, attr="dll4", time=0):
    """
    This function will return a plotly figure of the voronoi diagram with the attr
    coloured with a colour scale
```

```
    :param ds: Dataset to plot the voronoi diagram from
    :param attr: Attribute to plot
    :param time: Time to plot the attr at
    :return plotly.graph_objs.Figure: Plotly figure of the voronoi diagram with the
    """
    fig = plotly_voronoi_plot_2d(ds)
    labels = [f"cell {i}" for i in range(len(ds.pos_x))]

    vor = ds.voronoi_graph
    points = vor.point_region
    regions = vor.regions
    values = ds[attr].data[:, time].reshape(-1, 1)

    scaler = MinMaxScaler(feature_range=(0.3, 1))
    values = scaler.fit_transform(values)

    scatters = _plotly_visualize_attr(regions, vor.vertices, values, labels, points)
    fig.add_traces(scatters)

    return fig


def plotly_animate_attr(ds, attr="dl14"):
    """
    This function will return a plotly figure of the voronoi diagram with the attr
    coloured with a colour scale and animated over time.

    :param ds: Dataset to plot the voronoi diagram from
    :param attr: Property to colour the voronoi diagram with
    :return plotly.graph_objs.Figure: Plotly figure of the voronoi diagram with the
    """
    vor = ds.voronoi_graph
    # labels = [f"cell {i}" for i in range(len(ds.pos_x))]
    finite_segments, infinite_segments = _plotly_voronoi_plot_2d(vor)
```

```
points = vor.point_region
regions = vor.regions
values = ds[attr].data

scaler = MinMaxScaler(feature_range=(0.2, 0.9))
values = scaler.fit_transform(values)

fig_dict = {"data": [], "layout": {}, "frames": []}
fig_dict["layout"]["xaxis"] = {"title": "pos_x"}
fig_dict["layout"]["yaxis"] = {"title": "pos_y"}
fig_dict["layout"]["width"] = 900
fig_dict["layout"]["height"] = 800
fig_dict["layout"]["hovermode"] = "closest"
fig_dict["layout"]["updatemenus"] = [
    {
        "buttons": [
            {
                "args": [
                    None,
                    {
                        "frame": {"duration": 500, "redraw": False},
                        "fromcurrent": True,
                        "transition": {
                            "duration": 300,
                            "easing": "quadratic-in-out",
                        },
                    },
                ],
                "label": "Play",
                "method": "animate",
            },
            {
                "args": [
                    [None],
                    {
                        "frame": {"duration": 0, "redraw": False},
```

```
                    "mode": "immediate",
                    "transition": {"duration": 0},
                },
            ],
            "label": "Pause",
            "method": "animate",
        },
    ],
    "direction": "left",
    "pad": {"r": 10, "t": 87},
    "showactive": False,
    "type": "buttons",
    "x": 0.1,
    "xanchor": "right",
    "y": 0,
    "yanchor": "top",
}
]
sliders_dict = {
    "active": 0,
    "yanchor": "top",
    "xanchor": "left",
    "currentvalue": {
        "font": {"size": 20},
        "prefix": "Time:",
        "visible": True,
        "xanchor": "right",
    },
    "transition": {"duration": 300, "easing": "cubic-in-out"},
    "pad": {"b": 10, "t": 50},
    "len": 0.9,
    "x": 0.1,
    "y": 0,
    "steps": [],
}
fig_dict["layout"]["sliders"] = [sliders_dict]
```

```
frames = []
for i in range(ds.time.shape[0]):
    labels = [
        f"cell {j}: {attr} {'%.2f' % ds[attr].data[j, i]}"
        for j in range(len(ds.pos_x))
    ]
    frame = {"data": [], "name": str(i)}
    frame["data"] += _get_voronoi_plot_scatters(
        ds.pos_x, ds.pos_y, labels, finite_segments, infinite_segments
    )
    frame["data"] += _plotly_visualize_attr(
        regions, vor.vertices, values[:, i], labels, points
    )
    frames.append(frame)

    slider_step = {
        "args": [
            [i],
            {
                "frame": {"duration": 150, "redraw": False},
                "mode": "immediate",
                "transition": {"duration": 200},
            },
        ],
        "label": i,
        "method": "animate",
    }
    sliders_dict["steps"].append(slider_step)

fig_dict["frames"] = frames
fig = go.Figure(
    data=frames[0]["data"], frames=fig_dict["frames"], layout=fig_dict["layout"]
)

return fig
```

```
@numba_njit
def euclidean_distance(A, B):
    return np.linalg.norm(A - B)


@numba_njit
def get_idx_interv(d, D):
    k = 0
    while d > D[k]:
        k += 1
    k -= 1
    return k


@numba_njit
def de_casteljau(b, t):
    N = b.shape[0]
    if N < 2:
        raise ValueError("The control polygon must have at least two points")
    for r in range(1, N):
        b[: N - r, :] = (1 - t) * b[: N - r, :] + t * b[1 : N - r + 1, :]
    return b[0, :]


@numba_njit
def bezier_curve(b, nr):
    ret = np.zeros((nr, 2))
    t = np.linspace(0, 1, nr)
    for k in range(nr):
        ret[k, :] = de_casteljau(b, t[k])
    return ret


@numba_njit
```

```
def get_beizer_curve(A, B, dist, params, edge_colors, nr):
    d = euclidean_distance(A, B)
    K = get_idx_interv(d, dist)

    b = np.zeros((4, 2))
    b[0, :] = A
    b[1, :] = A / params[K]
    b[2, :] = B / params[K]
    b[3, :] = B

    color = edge_colors[K]
    bezier_points = bezier_curve(b, nr)
    return bezier_points, color


def plotly_cell_network_graph(ds, width=800, height=850):
    """
    This function will return a plotly figure of the cell network graph

    NOTE: Adapted from https://chart-studio.plotly.com/~notebook_demo/46.embed
          with the following changes:
            - Removed unused code
            - Rewrote the code to use numba for speed
            - Compatible with latest version of plotly

    :param ds: Dataset to plot the cell network graph from
    :param width: Width of the plot
    :param height: Height of the plot
    :return plotly.graph_objs.Figure: Plotly figure of the cell network graph
    """
    fig = go.Figure()

    width = 800
    height = 850

    DIST = np.array(
```

```
    [0.0, 0.7653668647301797, 1.4142135623730951, 1.8477590650225735, 2.0]
)
params = np.array([1.2, 1.5, 1.8, 2.1])
node_color = "rgba(0, 51, 181, 0.85)"
line_color = "#FFFFFF"
edge_colors = np.array(["#6d8acf", "#5588c8", "#84a9dd", "#d4daff"])

V = list(ds.network_graph.nodes)
E = list(ds.network_graph.edges)
labels = [f"cell {i}" for i in V]
layout = networkx.circular_layout(ds.network_graph)
Weights = list(networkx.get_edge_attributes(ds.network_graph, "weight").values())
L = len(layout)

Xn = [layout[k][0] for k in range(L)]
Yn = [layout[k][1] for k in range(L)]

for j, e in enumerate(E):
    bezier_points, color = get_beizer_curve(
        layout[e[0]], layout[e[1]], DIST, params, edge_colors, nr=5
    )
    text = "cell " + str(V[e[0]]) + " and cell " + str(V[int(e[1])])

    fig.add_trace(
        go.Scatter(
            x=bezier_points[:, 0],
            y=bezier_points[:, 1],
            mode="lines",
            line=go.scatter.Line(
                color=color,
                shape="spline",
                width=2
                / Weights[j], # The width is proportional to the edge weight
            ),
            text=text,
            hoverinfo="text",
```

```
            showlegend=False,
        )
    )

fig.add_trace(
    go.Scatter(
        x=Xn,
        y=Yn,
        mode="markers",
        marker=go.scatter.Marker(
            symbol="circle",
            size=15,
            color=node_color,
            line=go.scatter.marker.Line(color=line_color, width=1),
        ),
        text=labels,
        hoverinfo="text",
    )
)

axis = dict(
    showline=False, # hide axis line, grid, ticklabels and title
    zeroline=False,
    showgrid=False,
    showticklabels=False,
    title="",
)

fig.update_layout(
    title="Cell Network Graph",
    font_size=12,
    showlegend=False,
    autosize=False,
    width=width,
    height=height,
    xaxis=go.layout.XAxis(axis),
```

```
        yaxis=go.layout.YAxis(axis),
        margin=go.layout.Margin(
            l=40,
            r=40,
            b=85,
            t=100,
        ),
        hovermode="closest",
    )

    return fig
```

```
# test_attrs.py
import numpy as np
from numba import jit

import ec_connectivity.attrs as attrs
import ec_connectivity.core as core
import ec_connectivity.network as network


@jit
def numba_random_seed(seed):
    """
    Set the seed for the random number generator
    """
    np.random.seed(seed)


def set_random_seed(seed):
    """
    Set the seed for the random number generator
    """
    np.random.seed(seed)
    numba_random_seed(seed)
```

```
class TestInitDll4:
    def init_ds(self):
        xx, yy, _, _ = network.simple_grid(9, 1)
        num_cells = len(xx)
        timesteps = 2
        ds = core.create_cell_dataset(
            pos_x=xx,
            pos_y=yy,
            dll4=np.zeros((num_cells, timesteps)),
            active_notch=np.zeros((num_cells, timesteps)),
            voronoi_graph=True,
        )
        network.init_connection_matrix(ds)
        return ds

    def test_init_dll4(self):
        ds1 = self.init_ds()
        ds2 = self.init_ds()
        set_random_seed(0)
        attrs.init_dll4(ds1)
        set_random_seed(0)
        attrs.naive_init_dll4(ds2)
        assert np.array_equal(ds1.dll4, ds2.dll4)


class TestActiveNotch:
    def init_ds(self):
        set_random_seed(0)
        xx, yy, _, _ = network.simple_grid(9, 1)
        num_cells = len(xx)
        timesteps = 2
        ds = core.create_cell_dataset(
            pos_x=xx,
            pos_y=yy,
            dll4=np.zeros((num_cells, timesteps)),
```

```
            active_notch=np.zeros((num_cells, timesteps)),
            voronoi_graph=True,
        )
        network.init_connection_matrix(ds)
        attrs.init_dll4(ds)
        return ds

    def test_active_notch(self):
        ds1 = self.init_ds()
        ds2 = self.init_ds()

        ret1 = attrs.get_active_notch(ds1, 1)
        ret2 = attrs.naive_get_active_notch(ds2, 1)
        assert np.array_equal(ret1, ret2)


class TestDll4:
    def init_ds(self):
        set_random_seed(0)
        xx, yy, _, _ = network.simple_grid(9, 1)
        num_cells = len(xx)
        timesteps = 4
        ds = core.create_cell_dataset(
            pos_x=xx,
            pos_y=yy,
            dll4=np.zeros((num_cells, timesteps)),
            active_notch=np.zeros((num_cells, timesteps)),
            voronoi_graph=True,
        )
        network.init_connection_matrix(ds)
        attrs.init_dll4(ds)
        attrs.update_active_notch(ds, 1)
        return ds

    def test_dll4(self):
        ds1 = self.init_ds()
```

```
        ds2 = self.init_ds()


        ret1 = attrs.get_dll4(ds1, 1)
        ret2 = attrs.naive_get_dll4(ds2, 1)
        assert np.array_equal(ret1, ret2)
```

```
# test_core.py
import numpy as np
import xarray

from ec_connectivity import core
from ec_connectivity import network



class TestCore:
    def test_create_cell_dataset(self):
        xx, yy, cell_map, cell_id_map = network.simple_grid(12, 0.7)
        num_cells = len(xx)
        timesteps = 300
        ds = core.create_cell_dataset(
            pos_x=xx,
            pos_y=yy,
            vegf=np.zeros((num_cells, timesteps)),
            vegfr=np.zeros((num_cells, timesteps)),
            dll4=np.zeros((num_cells, timesteps)),
            notch=np.zeros((num_cells, timesteps)),
            active_notch=np.zeros((num_cells, timesteps)),
            cell_map=cell_map,
            cell_id_map=cell_id_map,
            voronoi_graph=True,
        )
        assert isinstance(ds, xarray.Dataset)
        assert ds.dims["cells"] == num_cells
        assert ds.dims["time"] == timesteps
```

```
# test_dataset.py
```

```
import numpy as np
import pytest
import xarray

import ec_connectivity.attrs as attrs
import ec_connectivity.core as core
import ec_connectivity.dataset as dataset
import ec_connectivity.network as network


class TestCellDataset:
    def create_test_cell_dataset():
        xx, yy, _, _ = network.simple_grid(12, 0.7)
        # TODO have a complete simulation function
        num_cells = len(xx)
        timesteps = 100
        ds = core.create_cell_dataset(
            pos_x=xx,
            pos_y=yy,
            dll4=np.zeros((num_cells, timesteps)),
            active_notch=np.zeros((num_cells, timesteps)),
            voronoi_graph=True,
        )
        ds = network.update_neighbour_matrix(ds)
        ds = network.init_protrusion(ds, protrusion_max_length=3)
        ds = network.init_connection_matrix(ds)
        ds = attrs.init_dll4(ds)
        for i in range(1, timesteps):
            ds = attrs.update_active_notch(ds, i)
            ds = attrs.update_dll4(ds, i)
            if i % 4 == 0:
                ds = network.simulate_protrusion_one_step(ds)
        return ds

    @pytest.mark.parametrize("ds", [create_test_cell_dataset()])
    def test_save_and_open_dataset(self, ds, tmp_path):
```

```
        path = tmp_path / "test.zarr"
        dataset.save_dataset(ds, path)
        ds2 = dataset.open_dataset(path)
        xarray.testing.assert_identical(ds, ds2)
```

```
# test_distance.py
from itertools import combinations

import numpy as np
import pytest

import ec_connectivity.distance as distance


class TestEuclideanDistance:
    @pytest.mark.parametrize("method", ["euclidean", "cityblock"])
    def test_distance(self, method):
        XA = np.array([[0, 0], [2, 2], [4, 4]])
        combo = list(combinations(range(len(XA)), 2))

        if method == "euclidean":
            dist_p = distance.euclidean_distance(XA)
            dist_c = distance.euclidean_distance(XA, XA)
            for i in range(dist_c.shape[0]):
                for j in range(dist_c.shape[1]):
                    if i != j:
                        p_index = distance.get_condensed_id(i, j, XA.shape[0])
                        assert dist_c[i, j] == dist_p[p_index]
                        assert dist_p[p_index] == distance.euclidean_distance_numba(
                            XA[combo[p_index][0]], XA[combo[p_index][1]]
                        )

        if method == "cityblock":
            dist_p = distance.manhattan_distance(XA)
            dist_c = distance.manhattan_distance(XA, XA)
            for i in range(dist_c.shape[0]):
```

```
            for j in range(dist_c.shape[1]):
                if i != j:
                    p_index = distance.get_condensed_id(i, j, XA.shape[0])
                    assert dist_c[i, j] == dist_p[p_index]
                    assert dist_p[p_index] == distance.manhattan_distance_numba(
                        XA[combo[p_index][0]], XA[combo[p_index][1]]
                    )
```

```
# test_metric.py
import numpy as np

from ec_connectivity import core
from ec_connectivity import metric
from ec_connectivity import network


class TestCharacteristicPathLength:
    def init_graph(self):
        np.random.seed(0)
        xx, yy, _, _ = network.simple_grid(12, 0.7)
        ds = core.create_cell_dataset(pos_x=xx, pos_y=yy, voronoi_graph=True)
        ds = network.update_network_graph(ds)
        return ds

    def test_characteristic_path_length(self):
        ds = self.init_graph()
        assert metric.characteristic_path_length(ds) == 5.786488740617181


class TestClusteringCoefficient:
    def init_graph(self):
        np.random.seed(0)
        xx, yy, _, _ = network.simple_grid(12, 0.7)
        ds = core.create_cell_dataset(pos_x=xx, pos_y=yy, voronoi_graph=True)
        ds = network.update_network_graph(ds)
        return ds
```

```
    def test_clustering_coefficient(self):
        ds = self.init_graph()
        assert metric.clustering_coefficient(ds) == 0.2753030303030302


class TestInDegree:
    def init_graph(self):
        np.random.seed(0)
        xx, yy, _, _ = network.simple_grid(12, 0.7)
        ds = core.create_cell_dataset(pos_x=xx, pos_y=yy, voronoi_graph=True)
        ds = network.update_network_graph(ds, directed=True)
        return ds

    def test_in_degree(self):
        ds = self.init_graph()
        assert metric.in_degree(ds).sum() == 482


class TestOutDegree:
    def init_graph(self):
        np.random.seed(0)
        xx, yy, _, _ = network.simple_grid(12, 0.7)
        ds = core.create_cell_dataset(pos_x=xx, pos_y=yy, voronoi_graph=True)
        ds = network.update_network_graph(ds, directed=True)
        return ds

    def test_out_degree(self):
        ds = self.init_graph()
        assert metric.out_degree(ds).sum() == 482
```

```
# test_network.py
import numpy as np
import pytest
from numba import jit
```

```
import ec_connectivity.core as core
import ec_connectivity.network as network
import ec_connectivity.util as util

POS_X = [0, 0, 1]
POS_Y = [0, 1, 1]


@jit
def numba_random_seed(seed):
    """
    Set the seed for the random number generator
    """
    np.random.seed(seed)


def set_random_seed(seed):
    """
    Set the seed for the random number generator
    """
    np.random.seed(seed)
    numba_random_seed(seed)


class TestGrid:
    def test_simple_grid(self):
        xx, yy, _, _ = network.simple_grid(3, 1)
        assert np.array_equal(xx, [0, 0, 0, 1, 1, 1, 2, 2, 2])
        assert np.array_equal(yy, [0, 1, 2, 0, 1, 2, 0, 1, 2])

    def test_simple_grid_value_error(self):
        with pytest.raises(ValueError):
            network.simple_grid(3, 1.1)

    def test_simple_grid_empty(self):
        xx, yy, _, _ = network.simple_grid(0)
```

```
        assert len(xx) == 0
        assert len(yy) == 0


class TestNeighbour:
    def init_ds(self):
        xx, yy, _, _ = network.simple_grid(3, 1)
        ds = core.create_cell_dataset(pos_x=xx, pos_y=yy, voronoi_graph=True)
        return ds

    def remove_cell(self, ds, cell_id):
        ds = ds.drop_isel(cells=cell_id)
        ds.attrs["voronoi_graph"] = util.get_voronoi_graph(ds.pos_x, ds.pos_y)
        return ds

    def test_update_neighbour_matrix(self):
        ds = self.init_ds()
        ds = network.update_neighbour_matrix(ds)
        assert np.array_equal(network.get_neighbours(ds, 4), [1, 7, 5, 3])
        assert np.array_equal(network.get_neighbours(ds, 1), [0, 2, 4])
        assert np.array_equal(network.get_neighbours(ds, 2), [5, 1])
        ds = self.remove_cell(ds, 4)
        ds = network.update_neighbour_matrix(ds)
        assert np.array_equal(network.get_neighbours(ds, 1), [2, 0, 3, 4])

    def test_get_neighbours_out_of_bounds(self):
        ds = self.init_ds()
        ds = network.update_neighbour_matrix(ds)
        with pytest.raises(IndexError):
            network.get_neighbours(ds, len(ds.cells) + 1)

    def test_no_neighbour_matrix_error(self):
        ds = self.init_ds()
        with pytest.raises(ValueError):
            network.get_neighbours(ds, 0)
```

```
class TestConnection:
    def init_ds(self):
        ds = core.create_cell_dataset(pos_x=POS_X, pos_y=POS_Y, voronoi_graph=True)
        ds = network.update_neighbour_matrix(ds)
        return ds

    def test_init_connection(self):
        ds = self.init_ds()
        ds = network.init_connection_matrix(ds)
        assert np.array_equal(ds.cell_connections, [[1, 2], [2, 0], [1, 0]])

    def test_get_connection(self):
        ds = self.init_ds()
        ds = network.init_connection_matrix(ds)
        assert np.array_equal(network.get_connections(ds, 0), [1, 2])
        assert np.array_equal(network.get_connections(ds, 1), [2, 0])
        assert np.array_equal(network.get_connections(ds, 2), [1, 0])

    def test_get_connection_out_of_bounds(self):
        ds = self.init_ds()
        ds = network.init_connection_matrix(ds)
        with pytest.raises(IndexError):
            network.get_connections(ds, 3)

    def test_no_connection_matrix_error(self):
        ds = self.init_ds()
        with pytest.raises(ValueError):
            network.get_connections(ds, 0)


class TestProtrusion:
    def init_ds(self):
        xx, yy, _, _ = network.simple_grid(3, 1)
        ds = core.create_cell_dataset(pos_x=xx, pos_y=yy, voronoi_graph=True)
        ds = network.update_neighbour_matrix(ds)
```

```
    ds = network.init_connection_matrix(ds)
    return ds

def test_init_protrusion(self):
    ds = self.init_ds()
    ds = network.init_protrusion(ds, protrusion_max_length=3)
    assert np.array_equal(ds.cell_protrusions, np.full((9, 3), -1, dtype=np.int32))
    ds = network.init_protrusion(ds, protrusion_max_length=2)
    assert np.array_equal(ds.cell_protrusions, np.full((9, 2), -1, dtype=np.int32))

def test_init_protrusion_connection(self):
    ds = self.init_ds()
    ds = network.init_protrusion(ds, protrusion_max_length=3)
    ds = network.init_connection_matrix(ds)
    assert (
        ds.connections.shape[0] == ds.neighbours.shape[0] + ds.protrusion.shape[0]
    )

def test_is_neighbour(self):
    ds = self.init_ds()
    assert network._is_neighbour(0, 1, ds.cell_neighbours)
    assert not network._is_neighbour(0, 2, ds.cell_neighbours)
    assert network._is_neighbour(1, 2, ds.cell_neighbours)

def test_simulate_protrusion(self):
    ds = self.init_ds()
    ds = network.init_protrusion(ds, protrusion_max_length=3)
    ds = network.simulate_protrusion_one_step(ds, start_nodes=[0])
    assert np.array_equal(ds.cell_protrusions[0], [0, 1, -1]) or np.array_equal(
        ds.cell_protrusions[0], [0, 2, -1]
    )
    ds = network.simulate_protrusion_one_step(ds, start_nodes=[0])
    assert (
        np.array_equal(ds.cell_protrusions[0], [0, 1, 3])
        or np.array_equal(ds.cell_protrusions[0], [0, 2, 3])
        or np.array_equal(ds.cell_protrusions[0], [-1, -1, -1])
```

```
    )


def test_update_cell_pairs(self):
    ds = self.init_ds()
    ds = network.init_protrusion(ds, protrusion_max_length=3)
    ds = network.init_connection_matrix(ds)
    network._update_cell_pairs(
        ds.cell_protrusions.inv_ridge_dict,
        ds.cell_connections.data,
        ds.cell_neighbours.data,
        walk=[2, 3],
        cell=0,
        operation="add",
    )
    assert np.array_equal(ds.cell_connections[0], [3, 1, 4, 7, -1, -1, -1])
    assert np.array_equal(ds.cell_connections[4], [1, -1, 5, 3, 0, -1, -1])
    assert np.array_equal(ds.cell_connections[7], [8, 6, -1, 0, -1, -1, -1])

    network._update_cell_pairs(
        ds.cell_protrusions.inv_ridge_dict,
        ds.cell_connections.data,
        ds.cell_neighbours.data,
        walk=[3, 2],
        cell=0,
        operation="remove",
    )
    assert np.array_equal(ds.cell_connections[0], [3, 1, -1, -1, -1, -1, -1])
    assert np.array_equal(ds.cell_connections[4], [1, 7, 5, 3, -1, -1, -1])
    assert np.array_equal(ds.cell_connections[7], [8, 6, 4, -1, -1, -1, -1])

    with pytest.raises(ValueError):
        network._update_cell_pairs(
            ds.cell_protrusions.inv_ridge_dict,
            ds.cell_connections.data,
            ds.cell_neighbours.data,
            walk=[3, 2],
```

```
                cell=0,
                operation="wrong",
            )


class TestNetworkGraph:
    def init_ds(self):
        xx, yy, _, _ = network.simple_grid(3, 1)
        ds = core.create_cell_dataset(pos_x=xx, pos_y=yy, voronoi_graph=True)
        return ds

    def test_update_network_graph(self):
        ds = self.init_ds()
        ds = network.update_network_graph(ds)
        assert ds.network_graph.number_of_nodes() == 9
        assert ds.network_graph.number_of_edges() == 12
        ds = network.update_network_graph(ds, directed=True)
        assert ds.network_graph.__class__.__name__ == "DiGraph"
```

```
# test_util.py
import random

import numpy as np

import ec_connectivity.core as core
import ec_connectivity.network as network
import ec_connectivity.util as util


class TestIsInBound:
    def test_is_in_bound(self):
        assert util.is_in_bounds(0, 0, 2)
        assert not util.is_in_bounds(2, 0, 2)
        assert not util.is_in_bounds(-1, 0, 2)
```

```
class TestGetElement:
    def test_get_element(self):
        a = np.array([1, 2, 3, -1, 5, -1, -1])
        assert np.array_equal(util.get_valid_elements(a), np.array([1, 2, 3, 5]))

    def test_multiple(self):
        values = np.random.randint(0, 100, size=(20,))
        input_arr = np.append(values, np.full(10, -1))
        np.random.shuffle(input_arr)
        assert np.array_equal(util.get_valid_elements(input_arr).sort(), values.sort())



class TestIsDirected:
    def init_ds(self):
        xx, yy, _, _ = network.simple_grid(3, 1)
        ds = core.create_cell_dataset(pos_x=xx, pos_y=yy, voronoi_graph=True)
        return ds

    def test_is_directed(self):
        ds = self.init_ds()
        ds = network.update_network_graph(ds)
        assert not util.is_directed(ds)
        ds = network.update_network_graph(ds, directed=True)
        assert util.is_directed(ds)



class TestIsValidHexColor:
    def generate_random_color(self):
        return "#" + "".join([random.choice("0123456789ABCDEF") for j in range(6)])

    def test_valid_color(self):
        for _ in range(20):
            assert util.is_valid_hex_color(self.generate_random_color())
        assert not util.is_valid_hex_color("1234567")
        assert not util.is_valid_hex_color("123456")
        assert not util.is_valid_hex_color("%$#@92")
```

```
class TestConvertNestedList:
    def test_convert_nested_list(self):
        a = [[1, 2, 3], [3, 4], []]
        assert np.array_equal(
            util.convert_nested_list(a), np.array([[1, 2, 3], [3, 4, -1], [-1, -1, -1]])
        )
```

```
# requirement/CI.txt
pytest==7.2.1
pytest-xdist==2.5.0
numpy>=1.21.6
fsspec>=2022.11.0
networkx>=2.6.3
numba>=0.56.4
scipy>=1.7.3
xarray>=0.20.2
coverage>=7.2.1
pytest-cov>=4.0.0
zarr>=2.14.2
```

```
# .coveragerc
[run]
omit =
    */tests/*
    */__init__.py
    */visualize.py
    */plot.py
    */jit.py

[report]
exclude_lines =
    pragma: no cover
    deprecated
    @numba_guvectorize
```

```
    @numba_vectorize
    @guvectorize
    @vectorize
```

```
# flake8
[flake8]
# Using black's default line length of 88
max-line-length = 88
select = C,E,F,W,B,B950
extend-ignore = E203, E501
```

```
# .gitattributes
evaluation/** linguist-vendored
```

```
# .gitignore
figures
.idea
cpp-code
data
figures_old

# Created by
    https://www.toptal.com/developers/gitignore/api/python,visualstudiocode,pycharm,macos,linux,windows,jupyt
# Edit at
    https://www.toptal.com/developers/gitignore?templates=python,visualstudiocode,pycharm,macos,linux,windows

### JupyterNotebooks ###
# gitignore template for Jupyter Notebooks
# website: http://jupyter.org/

.ipynb_checkpoints
*/.ipynb_checkpoints/*

# IPython
profile_default/
ipython_config.py
```

```
# Remove previous ipynb_checkpoints
#   git rm -r .ipynb_checkpoints/

### Linux ###
*~

# temporary files which can be created if a process still has a handle open of a
    deleted file
.fuse_hidden*

# KDE directory preferences
.directory

# Linux trash folder which might appear on any partition or disk
.Trash-*

# .nfs files are created when an open file is removed but is still being accessed
.nfs*

### macOS ###
# General
.DS_Store
.AppleDouble
.LSOverride

# Icon must end with two \r
Icon


# Thumbnails
._*

# Files that might appear in the root of a volume
.DocumentRevisions-V100
.fseventsd
```

```
.Spotlight-V100
.TemporaryItems
.Trashes
.VolumeIcon.icns
.com.apple.timemachine.donotpresent

# Directories potentially created on remote AFP share
.AppleDB
.AppleDesktop
Network Trash Folder
Temporary Items
.apdisk

### macOS Patch ###
# iCloud generated files
*.icloud

### PyCharm ###
# Covers JetBrains IDEs: IntelliJ, RubyMine, PhpStorm, AppCode, PyCharm, CLion,
    Android Studio, WebStorm and Rider
# Reference: https://intellij-support.jetbrains.com/hc/en-us/articles/206544839

# User-specific stuff
.idea/**/workspace.xml
.idea/**/tasks.xml
.idea/**/usage.statistics.xml
.idea/**/dictionaries
.idea/**/shelf

# AWS User-specific
.idea/**/aws.xml

# Generated files
.idea/**/contentModel.xml

# Sensitive or high-churn files
```

```
.idea/**/dataSources/
.idea/**/dataSources.ids
.idea/**/dataSources.local.xml
.idea/**/sqlDataSources.xml
.idea/**/dynamic.xml
.idea/**/uiDesigner.xml
.idea/**/dbnavigator.xml

# Gradle
.idea/**/gradle.xml
.idea/**/libraries

# Gradle and Maven with auto-import
# When using Gradle or Maven with auto-import, you should exclude module files,
# since they will be recreated, and may cause churn. Uncomment if using
# auto-import.
# .idea/artifacts
# .idea/compiler.xml
# .idea/jarRepositories.xml
# .idea/modules.xml
# .idea/*.iml
# .idea/modules
# *.iml
# *.ipr

# CMake
cmake-build-*/

# Mongo Explorer plugin
.idea/**/mongoSettings.xml

# File-based project format
*.iws

# IntelliJ
out/
```

```
# mpeltonen/sbt-idea plugin
.idea_modules/

# JIRA plugin
atlassian-ide-plugin.xml

# Cursive Clojure plugin
.idea/replstate.xml

# SonarLint plugin
.idea/sonarlint/

# Crashlytics plugin (for Android Studio and IntelliJ)
com_crashlytics_export_strings.xml
crashlytics.properties
crashlytics-build.properties
fabric.properties

# Editor-based Rest Client
.idea/httpRequests

# Android studio 3.1+ serialized cache file
.idea/caches/build_file_checksums.ser

### PyCharm Patch ###
# Comment Reason:
    https://github.com/joeblau/gitignore.io/issues/186#issuecomment-215987721

# *.iml
# modules.xml
# .idea/misc.xml
# *.ipr

# Sonarlint plugin
# https://plugins.jetbrains.com/plugin/7973-sonarlint
```

```
.idea/**/sonarlint/

# SonarQube Plugin
# https://plugins.jetbrains.com/plugin/7238-sonarqube-community-plugin
.idea/**/sonarIssues.xml

# Markdown Navigator plugin
# https://plugins.jetbrains.com/plugin/7896-markdown-navigator-enhanced
.idea/**/markdown-navigator.xml
.idea/**/markdown-navigator-enh.xml
.idea/**/markdown-navigator/

# Cache file creation bug
# See https://youtrack.jetbrains.com/issue/JBR-2257
.idea/$CACHE_FILE$

# CodeStream plugin
# https://plugins.jetbrains.com/plugin/12206-codestream
.idea/codestream.xml

# Azure Toolkit for IntelliJ plugin
# https://plugins.jetbrains.com/plugin/8053-azure-toolkit-for-intellij
.idea/**/azureSettings.xml

### Python ###
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# C extensions
*.so

# Distribution / packaging
.Python
build/
```

```
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
share/python-wheels/
*.egg-info/
.installed.cfg
*.egg
MANIFEST

# PyInstaller
#  Usually these files are written by a python script from a template
#  before PyInstaller builds the exe, so as to inject date/other infos into it.
*.manifest
*.spec

# Installer logs
pip-log.txt
pip-delete-this-directory.txt

# Unit test / coverage reports
htmlcov/
.tox/
.nox/
.coverage
.coverage.*
.cache
nosetests.xml
coverage.xml
```

```
*.cover
*.py,cover
.hypothesis/
.pytest_cache/
cover/

# Translations
*.mo
*.pot

# Django stuff:
*.log
local_settings.py
db.sqlite3
db.sqlite3-journal

# Flask stuff:
instance/
.webassets-cache

# Scrapy stuff:
.scrapy

# Sphinx documentation
docs/_build/

# PyBuilder
.pybuilder/
target/

# Jupyter Notebook

# IPython

# pyenv
#   For a library or package, you might want to ignore these files since the code is
```

```
#   intended to run in multiple environments; otherwise, check them in:
# .python-version

# pipenv
#   According to pypa/pipenv#598, it is recommended to include Pipfile.lock in version
     control.
#   However, in case of collaboration, if having platform-specific dependencies or
     dependencies
#   having no cross-platform support, pipenv may install dependencies that don't work,
     or not
#   install all needed dependencies.
#Pipfile.lock

# poetry
#   Similar to Pipfile.lock, it is generally recommended to include poetry.lock in
     version control.
#   This is especially recommended for binary packages to ensure reproducibility, and
     is more
#   commonly ignored for libraries.
#
     https://python-poetry.org/docs/basic-usage/#commit-your-poetrylock-file-to-version-control
#poetry.lock

# pdm
#   Similar to Pipfile.lock, it is generally recommended to include pdm.lock in
     version control.
#pdm.lock
#   pdm stores project-wide configurations in .pdm.toml, but it is recommended to not
     include it
#   in version control.
#   https://pdm.fming.dev/#use-with-ide
.pdm.toml

# PEP 582; used by e.g. github.com/David-OConnor/pyflow and github.com/pdm-project/pdm
__pypackages__/
```

```
# Celery stuff
celerybeat-schedule
celerybeat.pid

# SageMath parsed files
*.sage.py

# Environments
.env
.venv
env/
venv/
ENV/
env.bak/
venv.bak/

# Spyder project settings
.spyderproject
.spyproject

# Rope project settings
.ropeproject

# mkdocs documentation
/site

# mypy
.mypy_cache/
.dmypy.json
dmypy.json

# Pyre type checker
.pyre/

# pytype static type analyzer
.pytype/
```

```
# Cython debug symbols
cython_debug/

# PyCharm
#  JetBrains specific template is maintained in a separate JetBrains.gitignore that can
#  be found at https://github.com/github/gitignore/blob/main/Global/JetBrains.gitignore
#  and can be added to the global gitignore or merged into this file. For a more
    nuclear
#  option (not recommended) you can uncomment the following to ignore the entire idea
    folder.
#.idea/

### VisualStudioCode ###
.vscode/*
!.vscode/settings.json
!.vscode/tasks.json
!.vscode/launch.json
!.vscode/extensions.json
!.vscode/*.code-snippets

# Local History for Visual Studio Code
.history/

# Built Visual Studio Code Extensions
*.vsix

### VisualStudioCode Patch ###
# Ignore all local history of files
.history
.ionide

### Windows ###
# Windows thumbnail cache files
Thumbs.db
Thumbs.db:encryptable
```

```
ehthumbs.db

ehthumbs_vista.db


# Dump file

*.stackdump


# Folder config file

[Dd]esktop.ini


# Recycle Bin used on file shares

$RECYCLE.BIN/


# Windows Installer files

*.cab

*.msi

*.msix

*.msm

*.msp


# Windows shortcuts

*.lnk


# End of

    https://www.toptal.com/developers/gitignore/api/python,visualstudiocode,pycharm,macos,linux,windows,jupyt
```

```
# .pre-commit-config.yaml

repos:

  - repo: https://github.com/pre-commit/pre-commit-hooks

    rev: v4.4.0

    hooks:

      - id: trailing-whitespace

      - id: check-merge-conflict

      - id: debug-statements

      - id: mixed-line-ending

      - id: check-case-conflict

      - id: check-yaml
```

```
  - repo: https://github.com/asottile/reorder_python_imports
    rev: v3.9.0
    hooks:
      - id: reorder-python-imports
  - repo: https://github.com/asottile/pyupgrade
    rev: v3.3.1
    hooks:
      - id: pyupgrade
        args: [--py3-plus, --py37-plus]
  - repo: https://github.com/psf/black
    rev: 22.12.0
    hooks:
      - id: black
  - repo: https://github.com/PyCQA/flake8
    rev: 3.9.2
    hooks:
      - id: flake8
        additional_dependencies: ["flake8-bugbear==22.6.22", "flake8-builtins==1.5.3"]
  - repo: https://github.com/asottile/blacken-docs
    rev: v1.12.1
    hooks:
      - id: blacken-docs
        args: [--skip-errors]
        additional_dependencies: [black==22.12.0]
        language_version: python3
```

```
# codecov.yml
comment:
  layout: "reach, diff, flags, files"
coverage:
  status:
    project:
      default:
        target: 100%
        threshold: 5%
```

```
# pytest.ini
[pytest]
minversion = 6.0
testpaths =
    tests
```

```
# ,github/workflows/tests.yml
# For more information see:
    https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-python

name: Tests

on:
  pull_request:
    branches:
    - "main"
    - "!eval/**"
    - "!**/visualize"
    - "!**/plot"

jobs:
  pre-commit:
    name: Lint
    runs-on: self-hosted
    steps:
      - name: Cancel Previous Runs
        uses: styfle/cancel-workflow-action@0.11.0
        with:
          access_token: ${{ github.token }}

      - uses: actions/checkout@v2
      - uses: actions/setup-python@v4
        with:
          python-version: "3.9"
      - uses: pre-commit/action@v3.0.0
```

```
test-ubuntu:
  name: Python ${{ matrix.python }} on Ubuntu
  runs-on: [self-hosted, linux, x64]
  strategy:
    matrix:
      python: ["3.8", "3.9", "3.10"]
  defaults:
    run:
      shell: bash
  steps:
    - name: Cancel Previous Runs
      uses: styfle/cancel-workflow-action@0.11.0
      with:
        access_token: ${{ github.token }}

    - name: Checkout
      uses: actions/checkout@v2

    - name: Setup Python and cache deps
      uses: actions/setup-python@v4
      with:
        python-version: ${{ matrix.python }}
        cache: "pip"
        cache-dependency-path: requirements/CI.txt

    - name: Install pip deps
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements/CI.txt
    - name: Run tests
      run: |
        DISABLE_NUMBA=1 pytest -x ./tests

coverage:
  name: Coverage
```

```
runs-on: [self-hosted, linux, x64]
steps:
  - name: Cancel Previous Runs
    uses: styfle/cancel-workflow-action@0.11.0
    with:
      access_token: ${{ github.token }}

  - name: Checkout
    uses: actions/checkout@v2
  - name: Setup Python and cache deps
    uses: actions/setup-python@v4
    with:
      python-version: "3.9"
      cache: "pip"
      cache-dependency-path: requirements/CI.txt

  - name: Install pip deps
    run: |
      python -m pip install --upgrade pip
      pip install -r requirements/CI.txt

  - name: Generate coverage report
    run: |
      DISABLE_NUMBA=1 pytest --cov=ec_connectivity --cov-report=xml -x ./tests

  - name: Upload coverage reports to Codecov
    uses: codecov/codecov-action@v3
    with:
      token: ${{ secrets.CODECOV_TOKEN }}
      file: ./coverage.xml
      flags: unittests
      name: codecov-umbrella
      fail_ci_if_error: true
```