Math 607E Project

Jupiter Algorta

December 2023

1 Introduction

Advancing our understanding of cellular behaviour is pivotal in various scientific domains, ranging from fundamental biology to applications in medicine and tissue engineering. At UBC, my research focuses on modelling cell behaviour using the established Cellular Potts Model (CPM) formalism. The Morpheus software group's efforts have been instrumental, enabling us to delve deeply into the biological intricacies of cellular behaviour, with our emphasis on the biological model itself rather than the framework.

This project aims to address a specific facet of cellular dynamics by proposing a novel approach—an individual cell model structured with springs and stimulated to protrude in a predefined direction guided by a Partial Differential Equation (PDE). Such a focus allows for a more nuanced exploration of the challenges inherent in creating computational models that mirror the complex behaviour of individual cells and discovering clever solutions using simple mathematical tools.

2 Development

Throughout all stages of this project, the foundation is built upon the formalism of the linear spring-mass model. Before delving into the development, we will discuss how I implemented the model and some of the characteristics of the parameters. The cell membrane will be represented through point-mass objects connected by springs, defined for points i and j as follows:

$$\begin{aligned}
\delta_{ij} &= \frac{(x_i - x_j)}{\|x_i - x_j\|} \cdot \left(\frac{dx_i}{dt} - \frac{dxj}{dt}\right) \\
m_i \frac{d^2 x_i}{dt^2} &= -\frac{(x_i - x_j)}{\|x_i - x_j\|}[(k(\|x_i - x_j\| - r_{ss}) + c\delta_{ij}]
\end{aligned}$$

Here, x_i and x_j are vectors in $\mathbf{R}^2$, m_i represents the mass of object i, k denotes the spring constant, c is the damping coefficient, and r_{ss} is the equilibrium length of the spring. To apply the numerical method solvers discussed in class, I converted this system into a first-order system:

$$\begin{aligned}
\delta_{ij} &= \frac{(x_i - x_j)}{\|x_i - x_j\|} \cdot (v_j - v_i) \\
\frac{dx_i}{dt} &= v_i \\
\frac{dv_i}{dt} &= \frac{-1}{m_i} \frac{(x_i - x_j)}{\|x_i - x_j\|}[k(\|x_i - x_j\| - r_{ss}) + c\delta_{ij}]
\end{aligned}$$

Another crucial addition to the basics of this model was the incorporation of a center point mass. The significance of this central point is twofold: it prevents the cell from collapsing its volume and becoming a straight line, while also introducing the concept of internal cell pressure. The mass of this central node is set to be the sum of all other nodes' masses. The springs connecting the center point to the outer ones have their own constants and equilibrium sizes, as they serve different functions than those mimicking the cell's membrane. The challenge of volume collapse leads us to one of the major problems I had to overcome with this model—tangling and collision of the springs.

2.1 Collision

The system begins to collapse in undesired and unrealistic ways when one of the point masses passes through a spring. Here is a brief schematic illustrating how this occurs:

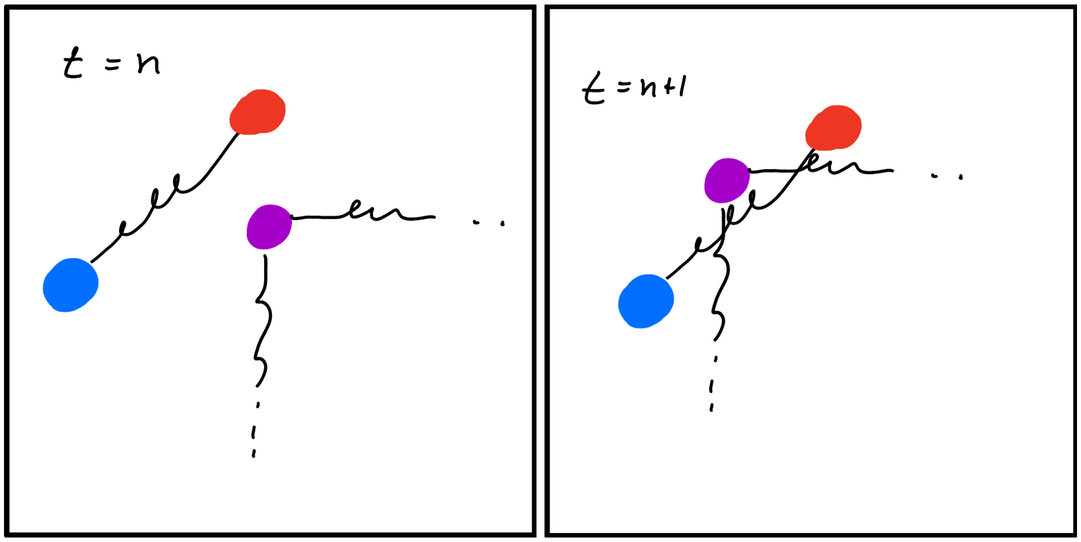


Figure 1: Here the purple point-mass crosses the spring connecting the red and blue dot

To address this issue, I need to develop a function that can perform two tasks: identify collisions and resolve them.

2.1.1 Potential Collisions

Firstly, the code must identify potential collisions before they occur, allowing us to compare frames and detect collisions. Computing all point masses against every outer spring can be computationally expensive, especially with a high number of point masses. To mitigate this, each point is checked to determine if it lies within the smallest rectangle that the spring occupies.

Consider a spring connecting points i and j with positions (x_i, y_i) and (x_j, y_j). Without loss of generality, let $x_i < x_j$ and $y_i < y_j$. Then, a point k is considered for collision checking with the spring connecting i and j if $x_i < x_k < x_j$ and $y_i < y_k < y_j$.

2.1.2 Identifying Collisions

After storing the points of potential collisions, a collision checker must traverse through two points in time (previous and current) and return a Boolean variable indicating whether a collision occurred. Since this problem is not continuous, we need an algorithm that can gather information from both frames and compare them. To achieve this, I will compute a coordinate change on both frames, bringing the x-axis to the spring (x'), and the y-axis will be the normal vector of the spring (y'). The matrix for such coordinate change in the frame number n is given as follows:

$$A_n = \begin{bmatrix} x_j - x_i & -(y_j - y_i) \\ y_j - y_i & x_j - x_i \end{bmatrix}$$

One may observe that this is essentially a rotation matrix. Although simplifications can be made to express the matrix in terms of the angle that the spring makes, I opted not to pursue this approach due to the additional computation it would entail.

It's important to note that as the spring itself is in motion, the rotation matrix will also vary from frame to frame. With this coordinate change setup, we can transform the coordinates of the potential collision. A collision is detected when the new y-coordinate changes sign.

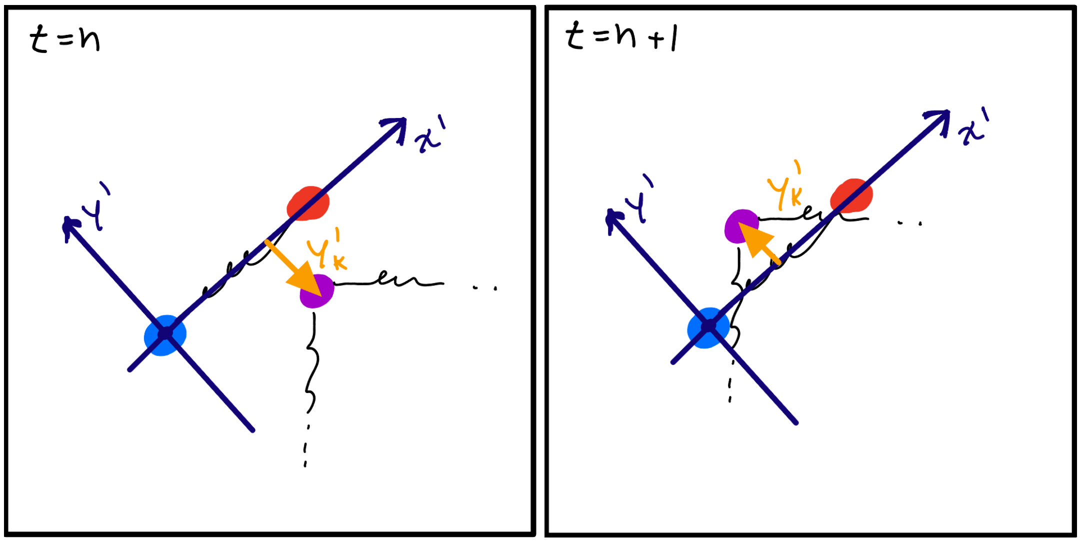


Figure 2: The same collision as discussed in fig. 1 is presented here, along with a representation of the new axis after the coordinate change. This visualization helps illustrate why a shift in the sign of the new y coordinate would signify a collision—indicating a crossing of the new x-axis, which corresponds to the spring.

2.1.3 Collision Solver

In this section, I will employ the concept of continuous collision to simulate a perfect elastic collision between the point mass and the spring. An assumption is made that the spring has infinite mass compared to the point mass, treating the spring as if it were a wall.

The underlying idea of this algorithm is that upon confirming a collision between frames n and $n+1$, we can generate a parameterized curve from the position of point k in frame n—denoted as $p_k(n)$—to its position in the subsequent frame. This curve is expressed in another variable s, where $s \in [0, 1]$, defined as follows:

$$u(s) = [p_k(n+1)]s + [p_k(n)](1 - s)$$

Now, considering that the point will rebound from the spring, in the spring coordinate system, this corresponds to a change of sign for the new y-axis. Since there is also a shift in velocity, we will take the change in velocity into

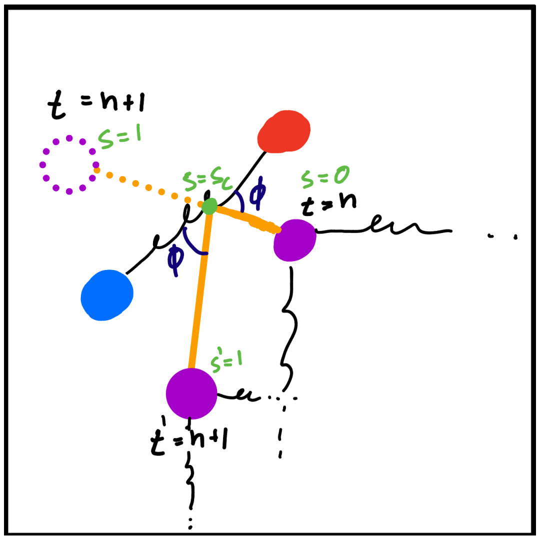


Figure 3: Continuous collision

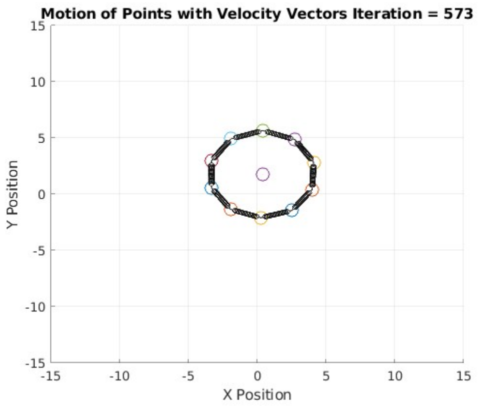
account as well, denoted as $v_k(n)$. In the spring coordinate system, $p_k(n+1)$ becomes:

$$(A_{n+1})^{-1}p_k(n+1) = \begin{bmatrix} -\frac{(-x_j+x_i)x_k(n+1)}{x_i^2-2x_jx_i+x_j^2+y_i^2-2y_jy_i+y_j^2} - \frac{(-y_j+y_i)y_k(n+1)}{x_i^2-2x_jx_i+x_j^2+y_i^2-2y_jy_i+y_j^2} \\ \frac{(-y_j+y_i)x_k(n+1)}{x_i^2-2x_jx_i+x_j^2+y_i^2-2y_jy_i+y_j^2} - \frac{(-x_j+x_i)y_k(n+1)}{x_i^2-2x_jx_i+x_j^2+y_i^2-2y_jy_i+y_j^2} \end{bmatrix}$$

Therefore, to compute the new position – $p'_k(n+1)$ – we will do:

$$p'_k(n+1) = (A_{n+1}) \begin{bmatrix} -\frac{(-x_j+x_i)x_k(n+1)}{x_i^2-2x_jx_i+x_j^2+y_i^2-2y_jy_i+y_j^2} - \frac{(-y_j+y_i)y_k(n+1)}{x_i^2-2x_jx_i+x_j^2+y_i^2-2y_jy_i+y_j^2} \\ -\frac{(-y_j+y_i)x_k(n+1)}{x_i^2-2x_jx_i+x_j^2+y_i^2-2y_jy_i+y_j^2} + \frac{(-x_j+x_i)y_k(n+1)}{x_i^2-2x_jx_i+x_j^2+y_i^2-2y_jy_i+y_j^2} \end{bmatrix}$$

Use the above as the new position and velocity for the point k and continue with the simulation.



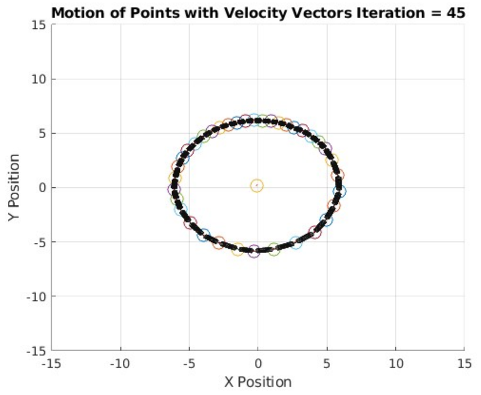


Figure 4: Results of the implementation are presented above. In these simulations, I balanced the perimeter and volume of the cell, creating configurations resembling a circle with radii of 4 (left) and 6 (right). As the cell lacks protrusions, it maintains a circular shape and remains relatively stationary. The visualization of the springs is courtesy of Xue-She Wang (2023), and the code can be found at `https://github.com/wangxueshe/Plot-2D-Spring-in-Matlab`.

3 Rac-Based Protrusion

The model central to my research revolves around the Rac-based protrusion of white blood cells. Rac is a protein that functions akin to a switch inside the cell. When in the "on" state, it is activated and binds to the cell's membrane, exhibiting a low diffusion rate. In its inactive state, Rac freely moves within the cell's cytoplasm, suggesting a high diffusion rate in its availability around the cell's membrane. The accumulation of active Rac in local high-density regions is sustained through a positive feedback loop. Simultaneously, it produces a local inhibitor, which operates on a much slower time scale and diffusion rate.

Under normal circumstances, Rac activation is facilitated by a chemical gradient left by bacteria. Upon activation and binding to the cell membrane, Rac promotes cellular protrusion, aiding white blood cells in moving toward bacteria. However, quantifying these experiments in a laboratory setting poses challenges, prompting the collaborative efforts of Orion Weiner and Jason Town from UCSF, now partners in our research group. They successfully modified white blood cells to respond to light stimulation, allowing them to gather quantitative data on Rac production.

The partial differential equation (PDE) model that encapsulates their experiments is defined in periodic boundaries around the cell membrane in 1D:

$$\begin{aligned}
\frac{\partial A}{\partial t} &= B\left(k_0 + 0.2U + \gamma\frac{A^n}{K^n + A^n}\right) - (\delta + 0.4H)\,A + D_A\frac{\partial^2 A}{\partial x^2}\\
\frac{\partial B}{\partial t} &= -B\left(k_0 + 0.2U + \gamma\frac{A^n}{K^n + A^n}\right) + (\delta + 0.4H)\,A + D_A\frac{\partial^2 A}{\partial x^2}\\
\frac{\partial H}{\partial t} &= \epsilon\,(\alpha A - \delta_h H) + D_H\frac{\partial^2 H}{\partial x^2}
\end{aligned}$$

Variable - A: Active form of the protein around the membrane. Initialized with random values. Its diffusion rate is low to promote wave-pinning behaviour. Initiated as 1.5+ rand_uni$(-0.1, 0.1)$ across space.

Variable - B: Inactive form of the protein around the membrane. Has a higher diffusion rate than A. Acts as a global inhibitor. Initiated as 0.8 across space.

Variable - U**:** The light stimulus around the cell. Initialized at 0, and it changes to 0.05 in specified regions of the 1-d space.
Variable - H**:** Local Rac inhibitor, slow time scale and low diffusion rate. Initiated as 0
Constant - k_0 : Basal activation of the protein. Value = 0.067.
Constant - γ : Rate of positive feedback of active protein. Value = 1.
Constant - ϵ : Time scale ratio between Rac's dynamic and the local Rac inhibitor. Value = 0.05.
Constant - δ : Basal inactivation of the protein. Value = 1.
Constant - δ_H : Basal decay of the local Rac inhibitor. Value = 0.4.
Constant - α : Activation of local inhibitor promoted by active protein. Value = 0.1.
Constant - K : Threshold for positive feedback for wave pinning. Value = 1.
Constant - n : Hill function coefficient. Value = 2.
Constant - D_A : Diffusion rate of active protein. Value $= 0.05 * \left(\frac{3}{2\pi}\right)^2$.
Constant - D_B : Diffusion rate of inactive protein. Value $= 10 * \left(\frac{3}{2\pi}\right)^2$.
Constant - D_H : Diffusion rate of local inhibitor. Value $= 0.001 * \left(\frac{3}{2\pi}\right)^2$.

With periodic boundaries, the anticipated outcome is the generation of travelling wave solutions. However, the introduction of a local light stimulus allows us to trap that moving wave. The term $\left(\frac{3}{2\pi}\right)^2$ serves as a space-scaling factor to align with the parameter regime I work with, yielding comparable results.

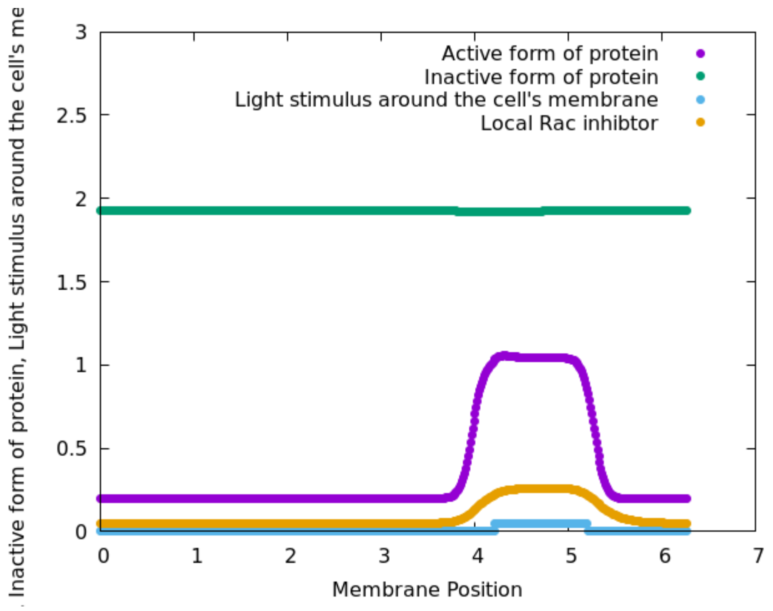


Figure 5: Example of simulation ran in Morpheus inside a CPM. $x \in [0, 2\pi]$ represents the space around the cell's membrane

4 Implementation of PDE

The concept is to run the partial differential equation (PDE) in a unit circle centred at the cell. I define the center as the average of all outer point positions and then project the PDE values onto the outer point-mass objects. If a certain threshold is met, that node protrudes outward in proportion to the active protein. Since, in this model, the cell's state (size, shape, velocity) does not affect the protein dynamics, I precompute the PDE results and use them while solving the spring-mass ordinary differential equation (ODE) system.

4.1 Method of Lines

To rephrase this problem in a format convenient for the methods learned or mentioned, I discretize space into equispaced points ($N = 100$) and maintain all variables as a single vector named R. This yields the following method of lines:

$$\frac{d}{dt}R = f(R) + \frac{D}{h^2}R$$

$$R = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_N \\ B_1 \\ B_2 \\ \vdots \\ B_N \\ H_1 \\ H_2 \\ \vdots \\ H_N \end{bmatrix}, f(R) = \begin{bmatrix} B_1\left(k_0 + 0.2U_1 + \gamma\frac{A_1^n}{K^n + A_1^n}\right) - (\delta + 0.4H_1)A_1 \\ B_2\left(k_0 + 0.2U_2 + \gamma\frac{A_2^n}{K^n + A_2^n}\right) - (\delta + 0.4H_2)A_2 \\ \vdots \\ B_N\left(k_0 + 0.2U_N + \gamma\frac{A_N^n}{K^n + A_N^n}\right) - (\delta + 0.4H_N)A_N \\ -B_1\left(k_0 + 0.2U_1 + \gamma\frac{A_1^n}{K^n + A_1^n}\right) + (\delta + 0.4H_1)A_1 \\ -B_2\left(k_0 + 0.2U_2 + \gamma\frac{A_2^n}{K^n + A_2^n}\right) + (\delta + 0.4H_2)A_2 \\ \vdots \\ -B_N\left(k_0 + 0.2U_N + \gamma\frac{A_N^n}{K^n + A_N^n}\right) + (\delta + 0.4H_N)A_N \\ \epsilon(\alpha A_1 - \delta_h H_1) \\ \epsilon(\alpha A_2 - \delta_h H_2) \\ \vdots \\ \epsilon(\alpha A_N - \delta_h H_N) \end{bmatrix}$$

$$L = \begin{bmatrix} -2 & 1 & 0 & \dots & 1 \\ 1 & -2 & 1 & \dots & 0 \\ 0 & 1 & -2 & \dots & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 1 & 0 & \dots & 1 & -2 \end{bmatrix}, D = \begin{bmatrix} D_A L & & 0 \\ & D_B L & \\ 0 & & D_H L \end{bmatrix}$$

4.2 SBDF-2

To obtain an approximate solution $R(t)$, we will employ the SBDF-2 method, as it has been recommended for reaction-diffusion models, and this problem bears resemblance to such models. For a single time step, we get:

$$R^{n+1} = \frac{4}{3}R^n - \frac{1}{3}R^{n-1} + \frac{2\Delta t}{3}\left(\frac{D}{h^2}\right)R^{n+1} + \frac{4\Delta t}{3}f(R^n) - \frac{2\Delta t}{3}f(R^{n-1})$$

Which simplifies to:

$$R^{n+1} = \left(I_{3N} - \frac{2\Delta t}{3}\left(\frac{D}{h^2}\right)\right)^{-1}\left[\frac{4}{3}R^n - \frac{1}{3}R^{n-1} + \frac{4\Delta t}{3}f(R^n) - \frac{2\Delta t}{3}f(R^{n-1})\right]$$

Since this problem relies on having an initial condition and a step after it, I will use forward Euler to get this extra step.

$$R^1 = R^0 + \Delta t\left(f(R^0) + \frac{D}{h^2}R^0\right)$$

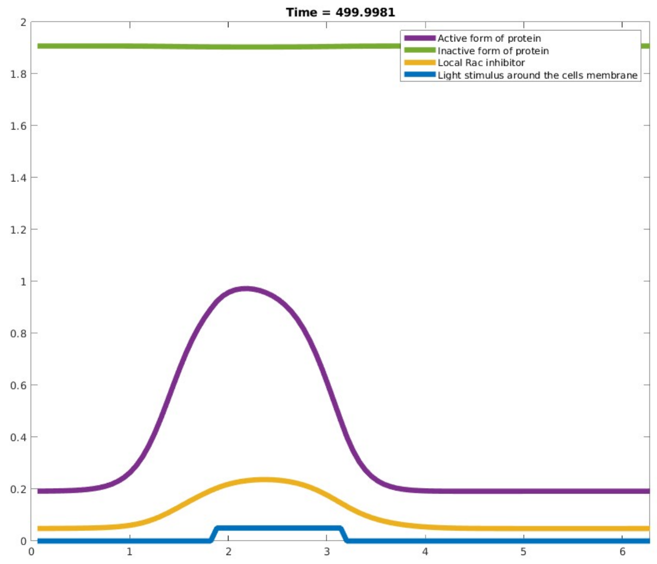


Figure 6: Results of the above implementation in Matlab with $h = \frac{2\pi}{N}$ and $\Delta t = h^2$ for $t \in [0, 500]$ which will be still scaled to match the time-scale of the cell model.

Now, armed with data on how the partial differential equation (PDE) dynamics evolve, we need to derive an algorithm to assign the PDE values to the nodes at the cell's membrane and the center node. This implementation will be integrated into the code governing the spring-mass system, where Rac promotes protrusions, acting as a velocity vector.

4.3 Time Scaling and Projection

Given that the PDE operates on a different time scale than the spring-mass system, a uniform mapping was performed to align the end of the simulation with the end of the Rac data.

Within the spring-mass system, we assume that Rac influences the rate of change of position, essentially acting as a velocity vector. To map the locations of the outer nodes to the PDE space, we first determine the center of the cell by averaging the locations of all nodes (excluding the center node) and denote them as x_c and y_c. After shifting the center to the origin, we compute the θ value of each node's position.

To map their θ to the PDE space x, we calculate what index of x produces the $min(x - \theta)$. Subtracting θ element-wise from x, the minimum value serves as the best approximation for θ in x, similar to how real numbers are mapped to floating-point representations. Now we can extract the relevant PDE data for each node.

4.4 Protrusion Vector Field

The core concept behind the protrusion is to function like air pressure in a balloon—an outward force radiating from the center towards the periphery. Unlike air pressure, this force is more robust, or active, in regions with high Rac activation. The vector field is expressed as follows:

$$P = \begin{bmatrix} -1.5R(\theta) * (R(\theta) > 0.4) * \frac{x - x_c}{\sqrt{(x-x_c)^2 + (y-y_c)^2}} \\ -1.5R(\theta) * (R(\theta) > 0.4) * \frac{y - y_c}{\sqrt{(x-x_c)^2 + (y-y_c)^2}} \end{bmatrix}$$

Here, we leverage the Boolean function ($R(\theta) > 0.4$) to construct a piece-wise vector field. This velocity is then integrated into the rate of change of the position of the nodes, inducing protrusion. The center node moves slightly differently to prevent constant changes if the same method is applied.

The center node moves in the direction of the outer node with the highest active Rac value, proportionally to the Rac activation in that area.

5 Final Results

Now, everything is constructed. We have a spring-mass system ensuring connectivity of the cell parts, a collision detector and solver preventing spring entanglement, and finally, a partial differential equation describing the internal mechanisms causing cell protrusion. Putting it all together, we obtain the following:

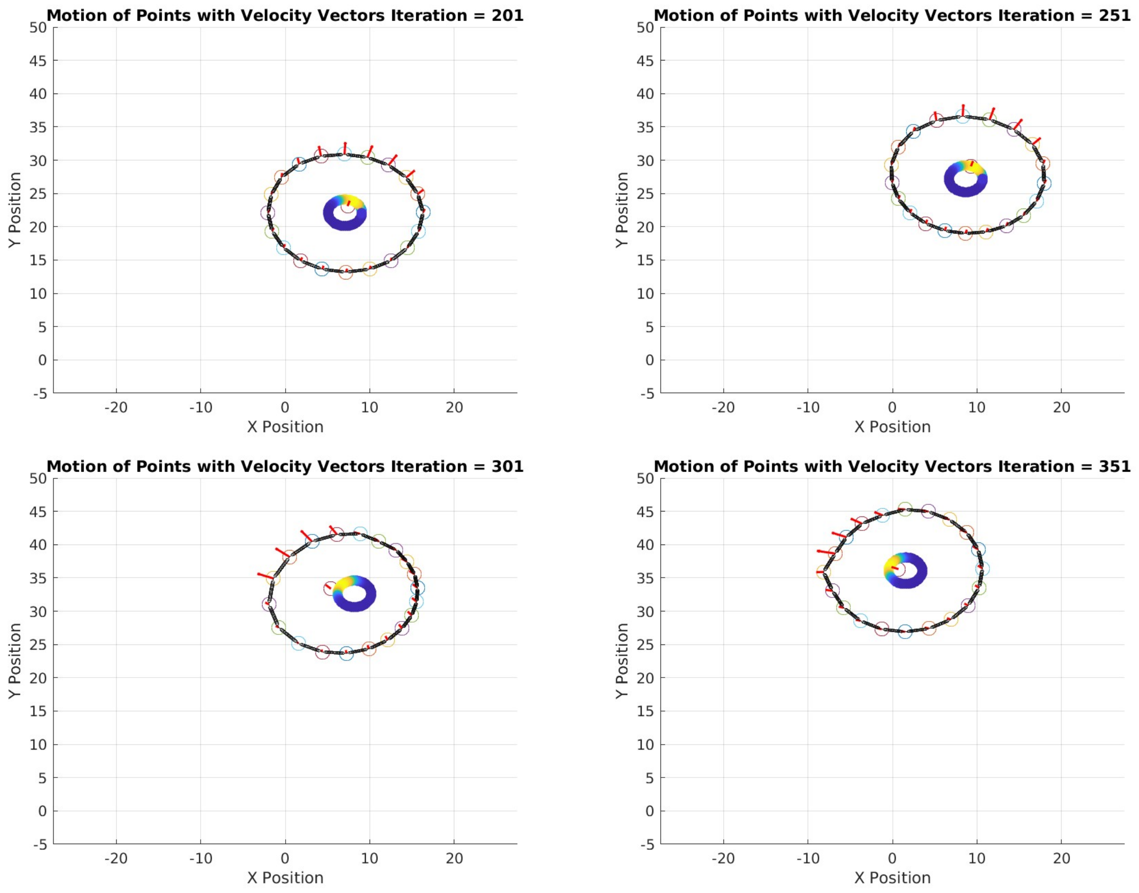


Figure 7: Apologies for the colloquialism, but this is incredibly exciting! The inner circle, displayed in yellow and blue tones, represents the active Rac data plotted in a circle centred at the cell's center (x_c, y_c). At iteration 250, the light stimuli shift from being around $\pi/4$ to $3\pi/4$, altering the cell's movement direction. Fantastic videos illustrating this can be found in the Appendix link, along with all the code.

6 Discussion

When comparing these results to my simulations in CPM, we observe qualitatively similar behaviours in terms of protrusion. However, the simplicity of the spring-mass model consistently yields a circular cell shape, regardless of protrusion. I've explored adjusting the spring parameters to achieve what we refer to as "asphericity." The asphericity of a given shape quantifies how close it is to a perfect circle by comparing the perimeter to the volume. More malleable cells exhibit "a lot of perimeters for not so much volume," giving them a more liquid-like behaviour. Attempting the same adjustments here results in undesired entanglement, revealing a limitation of the collision solver, particularly when dealing with multiple collisions simultaneously. Perhaps a valuable improvement would be to assign a radius of collision instead of treating nodes as mere points with no area. One of these attempts is depicted below:

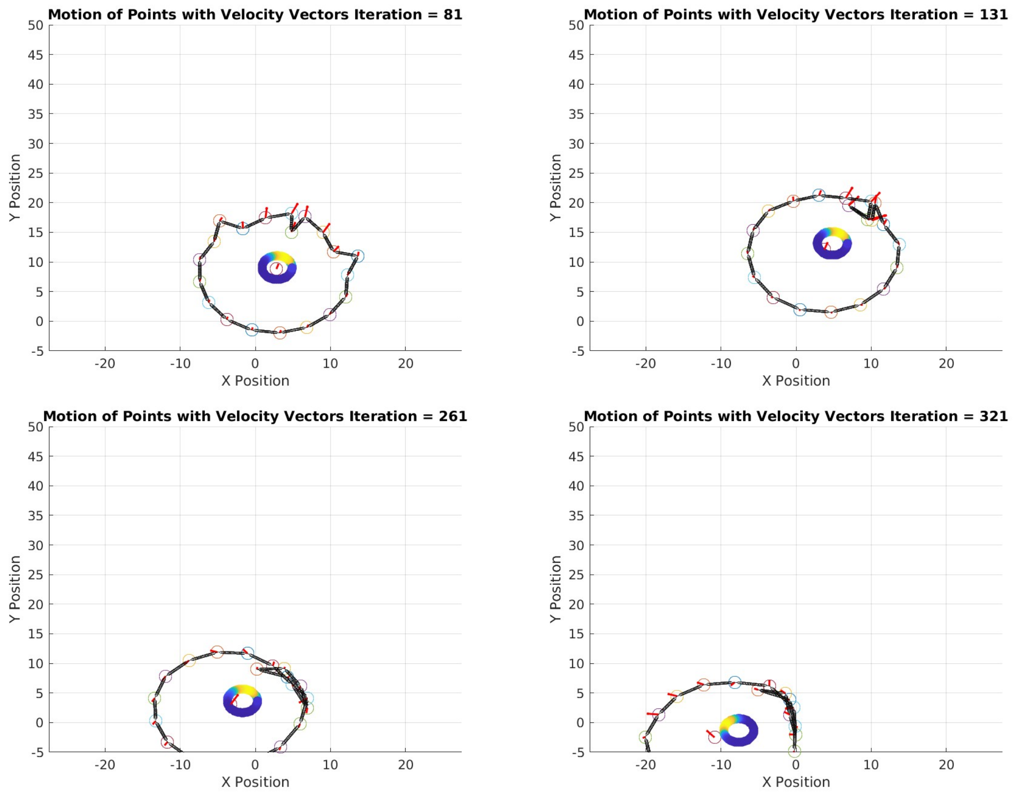


Figure 8: Here I used the same parameters as in Figure 7, but I doubled the rest size of the springs in the cell's membrane

7 Conclusion

As mentioned in the introduction, the primary objective of this project was not to develop a replacement for the formalisms used in my research. Instead, it aimed to illustrate how we continually stand on the shoulders of giants in scientific endeavours. Working with this simplified version has deepened my appreciation for the contributions of the Morpheus development team and enhanced my understanding of the complexities involved in creating a multi-scale model. The code, well-documented and accessible, is provided in the appendix link. Everything presented here was implemented in Matlab; I also tested it in Linux Octave, with the only exception being the video writer feature, which may not work.

Happy Holidays!

Hi Colin,

I wanted to express my gratitude for this incredible course here since we did not have a chance to present the projects. Although I may have been a bit MIA towards the end of the term due to significant opportunities in my research and poor time management on my part, I thoroughly enjoyed and learned a lot from the lectures.

Thank you for your guidance and for delivering such an engaging course. I hope you and your family have a wonderful end of the year and a delightful holiday!

Appendix

The repository for this code can be found at: `https://gitlab.math.ubc.ca/jupitera/jupiter-math607e-project`
Here is a brief description of what each script does:

- *runProject_GoodFormat.m* - Main "hub" of the simulation, where all the other functions are called, and parameters for springs and the number of nodes are set.
- *ode_rk4_center.m* - Where the derivatives relate all nodes' velocity and position. This is the $f(u)$ function of the RK4 method.
- *getLines.m* - Gets information from the position and indexation of the nodes and creates objects that contain information about each spring.
- *collision.m* - Where collisions between nodes and springs are detected and solved.
- *plot_cell.m* - Plots the simulation and saves a *.mp4* animation.
- *plot_cell_control.m* - Used to plot frames while doing debugging and testing. Works similarly to the above, but it has fewer features and is less automated.
- *Spring.m* - Code developed Xue-She Wang (2023), I found it online through the Matlab website.
- *optorac.m* - Runs Rac simulation and saves the data. Cannot run this together with the rest without running out of memory.