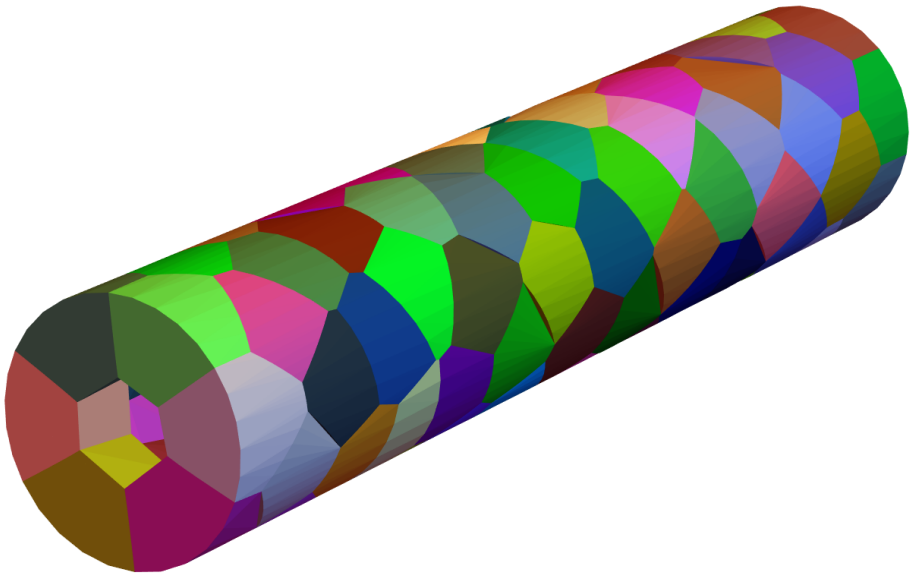
3DTubularVoronoi – User Manual V1

Léna Guitou, Eloy Tomás Serrano Andrés, Alberto Conejero and Javier Buceta

March 14, 2025



Abstract

Computational model for 3D tissue simulation modeling tubuluar epithelial tissues. Using Voronoi tessellations, a computational geometry tool that divides the plane into polygonal disjoint convex cells. We also take into account different cellular dynamics with the Metropolis algorithm. Through this approach, we have achieved a realistic model consistent with the most recent research in which a new shape, very famous in the recent literature, emerges naturally: the scutoid.

Contents

1 Modeling approach

Epithelial cells in a tubular tissue have their apical surface in contact with the inner cavity of the tube, called lumen (Figure 1A). We model tubular epithelia as a 3D solid cylinder with an inner cylindrical cavity which represents the lumen. Our approach consists in discretizing this solid into intermediate cylinders (called layers), the intern cylinder representing the apical layer and the external cylinder representing the basal layer (Figure 1B).

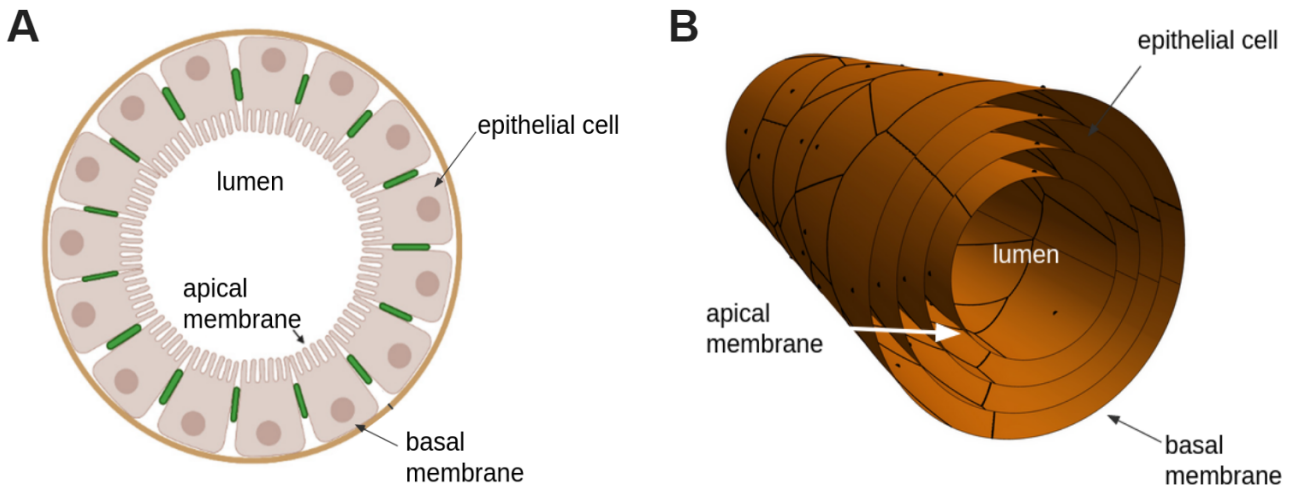


Figure 1: Modeling tubular epithelial tissue. A) Scheme of the front view of a tubular epithelia. Cells have a nucles (darker circle) and cell-cell junctions (in green). B) Modeling approach: 3D representation of a epithelia tube discretized into layers. Voronoi tesselations are defined by the orthogonal projections (black points).

1.1 Voronoi tesselations

For the simulations, we considered a tube of length L composed of N layers, with an inner cylinder of radius R. In the code, the tube length, the radius of the cylinders and the number of layers can be changed. The radius of the external cylinder is calculated by setting the value of the ratio between the apical and basal radius.

The goal is to build Voronoi tesselations on each layer of the tube. However, computing the geodesic distance in 3D can be very time consuming and can suffer from a loss of precision. That is why we transferred the cylinder to the plane. The most intuitive way to treat the cylinder as a plane would be to "unroll" it, resulting in a flat rectangle, as shown in Figure 2. The cylinder and the plane are topologically distinct, so there is no global isometry between them, however they present local isometry. This property allows us to translate each point of the cylinder to the plane, preserving the distances between them since the geodesic distance is maintained in the environment.

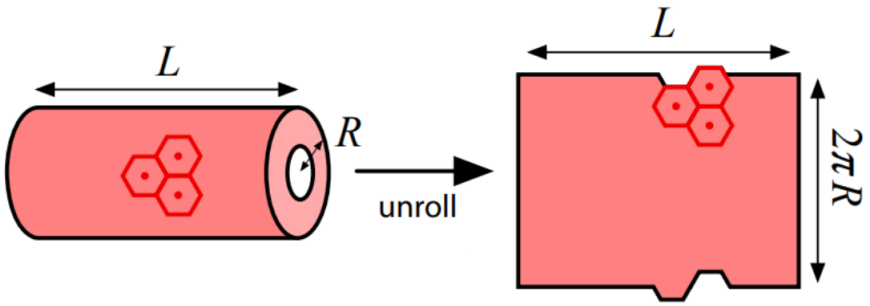


Figure 2: Cylinder unrolling process. Unrolling of the exterior surface of a tube of length L and radius R. The hexagons are given by the Voronoi cells and represent tissue cells. After the unrolling, we treat the cylindrical surface as a plane, and so we need boundary conditions since the cylinder and the plane are not topologically equal.

1.2 Parametrisation

To work on a surface instead of 3D cylinder, we need to define a parametrization allowing us to maintain local isometry. We can define the parametrization $f_0(u, v)$ which translates the local isometry of a plan to a cylinder. Let P be a plane of equation $z = 0$ and C_{R_a} the apical cylinder of length L and of equation $x^2 + y^2 = 1$. Then, the parametrization f_1 of the local isometry can be defined such as:

$$\begin{aligned} f_1 : P_{\text{rect}} &\longrightarrow C_{R_a} \\ (u, v) &\longmapsto \left(v, R_a \cos\left(\frac{u}{R_a}\right), R_a \sin\left(\frac{u}{R_a}\right)\right), \quad (u, v) \in [0, R_a] \times [0, L] \end{aligned}$$

To convert points from the plan to the cylinder and design for instance the graphical 3D tube, we will then use the parametrization f_1. Also, we define the following mapping to translate points from the apical cylinder to another cylinder k of radius R_k by scaling the u-coordinate while keeping the v-coordinate constant, ensuring a smooth, continous transition between the two surfaces.

$$\begin{aligned} f_2 : P_{rectap} &\longrightarrow P_{rectk} \\ (u, v) &\mapsto \left(\frac{R_k}{R} u, v \right). \end{aligned}$$

1.3 Boundary conditions

In the process of unrolling, there are inherent challenges because a plane and a cylinder are not topologically equivalent. Consequently, while we can locally transport points and paths, we can't do so globally. This distinction becomes evident when considering movement on the surface: on a cylinder, there's a direction where one can move and return to the starting point, which isn't possible on a bounded rectangle without specific boundary conditions. These conditions are crucial because in calculating Voronoi tessellations within a rectangle, points near opposite edges (which are close on the cylinder) become far apart, complicating the accurate calculation of the diagram's cells. This is why, we decided to use two replicas of the rectangle in order to overcome the issue of lateral border points (Figure 3)

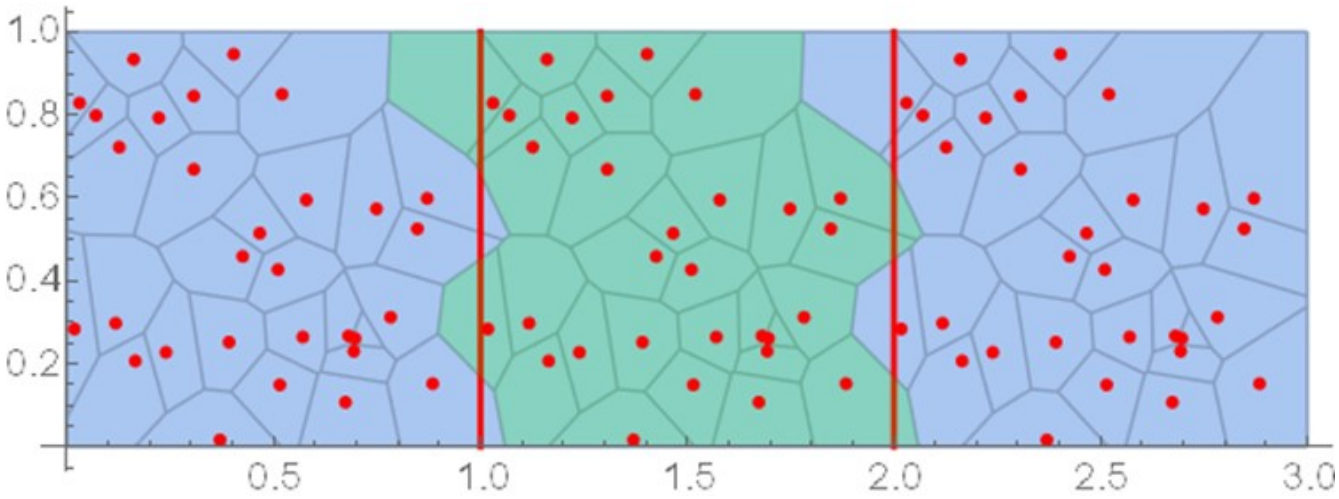


Figure 3: Replication of the unrolled cylinder to solved topological issues. We used 3 unrolled cylinders to compute the Voronoi tesselations. Image generated with the software Mathematica. Red points represent the generating seeds, red lines correspond to the unrolling axis, and green Voronoi cells are the actual geodesic Voronoi diagram in the cylinder after the unrolling.

1.4 Building layers

Once the apical layer is defined (the inner cylinder), we can build the layers. We execute the same process for the 3 cylinders. Let us consider the apical cylinder (the middle one in Figure 3) that we converted into a rectangle of length L and width $2\pi R_a$ (Figure 2). We set M random centroid points $C_i^a, i \in \{1, \ldots, M\}$ of coordinates (u, v) and define the apical tissue using the geodesic Voronoi tessellations. Next, we build the layers by computing the orthogonal projection of each apical centroid, with a projection length $\Delta R = (R_b - R_a)/(N-1)$ (Figure 4). Thus, the cylinders have a radius $R_j = R_a + j\Delta R,\ j \in \{0, \ldots, N-1\}$ with centroids defined such as $C_i^j(u,v) = C_i^a(u,v) + j\tau(u,v)$, with $\tau(u,v)$ the normal vector to the tangent plane of the apical surface at the point $C_i^a(u,v)$. Then, we compute the geodesic Voronoi tessellations on each surface so that each cell C_i is defined by the projections of all the surfaces i.e the cell C_i is defined by $(C_i^a, C_i^2, ..., C_i^N)$. As explained, we process the same way for the replicated rectangles.

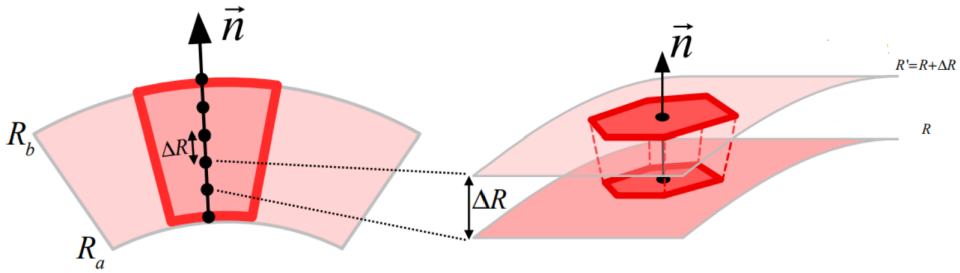


Figure 4: Orthogonal projection for layers building. The structure of a single cell in the model is defined in red bold. If the tube has an interior radius R_a (apical) and an exterior radius R_b (basal), discretizing into N layers we get $\Delta R = (R_b - R_a)/(N-1)$, and N cylinders with radius $R_j = R_a + j\Delta R,\ j \in \{0, \ldots, N-1\}$. To each tissue cell, we assign a generator seed in the interior layer of the tube (at radius R_a) and project it orthogonally into superior layers. Each cylindrical layer is unrolled as in Figure 2, and the seeds generate a Voronoi cell in each layer.

1.5 Vertex model

Now we can set the dynamics of the tissue that relies on its mechanical relaxation. The energy of the system is defined based on the vertex model considering the mechanical properties of each cell. In this case, we compute the energy E_i for each Voronoi tesselation (i.e. centroids of each cell). Thus, the energy E_i^j of the Voronoi centroid i in the layer j is defined such as:

$$E_i^j = \frac{K}{2}(A_i^j - A_0)^2 + \frac{\Gamma_j}{2}(P_i^j)^2 + \Lambda P_i^j \tag{1}$$

where K is the elastic constant, A_i^j the cell area of centroid i in layer j, A_0 the preferred cell area, Γ the contractility constant, P the cell perimeter and Λ the adhesion constant. We assume the elastic and adhesion constant homogenous along the radius and we assume that the contractility constant depends on the radius such as the contractility forces are higher on the apical layer. Thus, we define the ratio $s_j = \frac{R_j}{R_a}$ to have

$$\Gamma_j = \Gamma_0 e^{1-\frac{s_j}{s_0}}$$

with Γ_0 and s_0 constants.

To reduce the number of unknown parameters, we adimensionalized the system, defining a characteristic length $l_c = \sqrt{A_0}$ so that the dimensionless area $\tilde{A_0} = 1$ and a characteristic energy $e_c = KA_0^2$. It is what is implemented in the code. Then, the dimensionless energy $\tilde{E_i^j}$ of centroid i in layer j is defined using Equation 1 such as:

$$\tilde{E_i^j} = \frac{\tilde{K}}{2}(\tilde{A_i^j} - 1)^2 + \frac{1}{2}\tilde{\Gamma_j} + (\tilde{P_i^j})^2 + \tilde{\Lambda}\tilde{P_i^j} \tag{2}$$

where $\tilde{K} = KA_0^2/e_c = 1$, $\tilde{\Gamma}_j = \Gamma_j/(KA_0)$ and $\tilde{\Lambda} = \Lambda/(KA^{3/2})$. The total energy E_{total} of the tissue can be computed as the sums of the energies of each cell defined by tesselations in each layer:

$$\tilde{E}_{\text{total}} = \sum_{i=1}^{M}\sum_{j=1}^{N}\tilde{E}_i^j$$

1.6 A_0 estimation

The preferred cell area $A0$ (cell area in a flat tissue curvature) is at this point an unknown parameter. To evaluate this constant, we estimated the average energy of each cell. In our cylindrical model, cells tend to have a larger area at the basal layer compared to the apical surface (see Figure 4). This discrepancy influences adhesion energy calculations, which relies on the assumption of cell volume conservation, given the constant liquid content within cells.

Since the area of each layer j is defined such as $A_j = 2\pi R_j L$ (see Figure 2), the average cell area in layer j can be defined as the layer area divided by the number of cells:

$$\langle A_j \rangle = \frac{2\pi R_j L}{M} \tag{3}$$

where R_j is the cylinder radius of layer j, L is the length tube and M the number of cells within the tissue. We defined the ratio $s_j = \frac{R_j}{R_a}$ in Equation 1 . If we estimate the average apical cell area in the similar way, then Equation 3 reads as follows:

$$\langle A_j \rangle = \langle A_a \rangle s_j \tag{4}$$

To estimate the value of the cell area A_0, we use the expression of the average elastic energy. The average elastic energy of the layer j of radius R_j is given by

$$\langle E_{\text{elast}j} \rangle = \frac{1}{2}(\langle A_a \rangle s_j - A_0)^2$$

Thus, the average energy function is a parabole that reaches a minimum at $s^* = \frac{A_0}{\langle A_a \rangle}$ where $\langle A_a \rangle$ is the average apical cell area. This means that the elastic energy reached its minimum on the layer k such as $s^* = R_k/R_a$. Now, we apply the volume conservation rule $\langle V_{flat} \rangle = \langle V_{curv} \rangle$ to evaluate A_0:

$$A_0(R_b - R_a) = \frac{\pi(R_b^2 - R_a^2)L}{M} \tag{5}$$

Therefore, from Equation 5, we can estimate that the cell area A_0 can be defined such as

$$A_0 = \frac{\pi L(R_b + R_a)}{M} \tag{6}$$

This reveals that the elastic energy is the lowest in the middle of the tissue $s^* = \frac{R_b + R_a}{2R_a}$.

1.7 Tesselations motion

In order to reach energy relaxation, we used statistical mechanics and an algorithm that allows an heuristic way to move the seeds (Figure 5). In short, successively cell centroids of a same layer are selected and moved following a 2-dimensional Gaussian distribution. At each iteration, the energy of each tesselation is computed, thus the energy of each cell. The evolution of the movement is given by the Markov chain. At each movement, Voronoi tesselations are computed and the total energy is calculated. If the energy decreases then the algorithm keeps going with the next cell. The movement is "accepted" and cell centroids are moved a distance r in a random direction q. This distance is defined such as:

$$r \approx \sqrt{A_j}/10$$

If the movement is not accepted (i.e. the computed energy does not decrease), the Metropolis algorith gives the approval of motion depending on the Boltzmann distribution with a probability defined such as

$$e^{-\beta(E_y - E_x)}$$

where β is the product of the Boltzmann constant and thermodynamic temperature). Once the M cells of the apical layer moved, it does the same to the following layers. We can repeat this process as much as wanted. The number of steps has to be enough large to reach the thermal equilibrium.

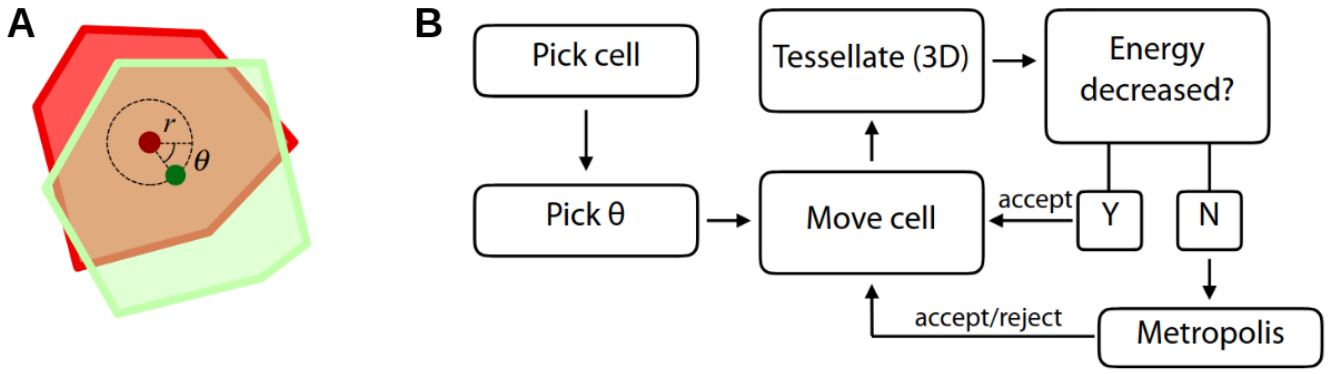


Figure 5: Algorithm overview. A/ Cell centroids are moved a distance $r \approx \sqrt{A_j}/10$ in a random direction q and redefine the Voronoi tessellation (from red to green) where A_j is the cell area on layer j of radius R_j. B/ The flow diagram summarizes one iteration of the the simulation scheme that combines computational geometry techniques, a vertex-like energetic estimation, and a Metropolis dynamics.

1.8 Real-Time estimation

Last but not least, the data generated by the code are given per iteration. At each iteration, the energy of each tesselation is computed. However, it is unknown at this point what is the equivalency in real time unit. To connect the algorithm iterations to real time, we based our approach on the method presented by Farhadifar *et al.* [1]. Vertex displacement over time was modeled using the following equation:

$$y(t) = d_0 + d_1 \left(1 - e^{-\frac{t}{\tau}}\right), \tag{7}$$

where τ represents the relaxation time for the system to reach equilibrium, ranging between $13\,\mathrm{s}$ and $94\,\mathrm{s}$. Similarly, we modeled energy relaxation using the equation:

$$y(x) = a + b \left(1 - e^{-\frac{x}{n_0}}\right), \tag{8}$$

where n_0 is a parameter that links iterations x to real-time.

To fit this model, we performed a nonlinear regression using the R-package *nls.multstart*. This package perform a non-linear least squares regression with the Levenberg-Marquardt algorithm using multiple starting values for increasing the chance that the minimum found is the global minimum. Thus for 100 simulations, we fitted the function to the energy relaxation curve for each simulation, averaged the results, and obtained the set of parameters (a, b, n_0) for Equation 8. For a single simulation, we found

$$a = 277.62346, b = -86.38992, n_0 = 14.00112$$

and for an average of 100 simulations, we found

$$a = 289.90758, b = -97.33677, n_0 = 15.80831.$$

Using the second set of parameters we established the relationship between iterations and real-time as follows:

$$\frac{t}{\tau} = \frac{x}{n_0} \quad \Rightarrow \quad t = \frac{x \cdot \tau}{n_0}, \tag{9}$$

where τ is the relaxation time in seconds, x is the iteration, and n_0 is the scaling parameter. The simulations show a relaxation time τ thar varries between 16 s and 93 s. Thus, if we estimate that the time relaxation $\tau = 16$ s, then the real time of one iteration ($x = 1$):

$$t = \frac{1 \cdot 16}{15.81} \approx 1.01 \text{ s} \tag{10}$$

In the same way, if we estimate that the time relaxation $\tau = 93$ s, then the real time of one iteration is:

$$t = \frac{1 \cdot 93}{15.81} \approx 5.88 \text{ s} \tag{11}$$

2 Installation and Execution

2.1 Installation

3DTubularVoronoi is open and available at `https://github.com/lenaguitou/3DTubularVoronoi.git`. Download the whole repository using **Code/Download ZIP** and extract it. It is very recommended to conserve the folder strcuture, otherwise paths need to be changed inside the main code *use_of_code.Rmd*. You need to install R and RStudio to execute the code and Mathematica for tubes visualization (if you have a license). If you dont, download Wolfram Player and be sure that the Mathematica script *TubularGraphicsMain.nb* is in the main folder.

The folder **tubular_simulations_source** contains all the source codes, that you should not modify. The folder **results** contains the data and figures generated. The file *use_of_code.html* is read-only visual version of the code to see what the by-default code generates.

In Windows and MacOs, no issues of installation had been reported. In Linux, you might face an issue with some packages, that can be simply solved using the following commands: ***sudo apt-get install gdal-bin proj-bin libgdal-dev libproj-dev libfontconfig1-dev libfreetype6-dev –reinstall libharfbuzz-dev libfribidi-dev pkg-config***

2.2 Execution

To start open the file *use_of_code.Rmd* with RStudio. Within this file you can see several sections delimited by gray boxes called chunks. Each chunk can be executed individually with the green triangle on the right top. When a chunk is running, the green triangle is replaced by a red square until the execution is over. If you are not familiar with R and Studio, it is recommended to use the "Visual" mode, accesible on the left top next to "Source". It is recommended to read the comments within the code and not execute all the code at once. The necessary packages (*devtools, dplyr, deldir, ggplot2, plotly, foreach, doParallel, viridis, svglite* and *nls.multstart)* can be installed executing the first chunk. Assure that all the packages have been well installed by looking at the terminal for eventual errors.

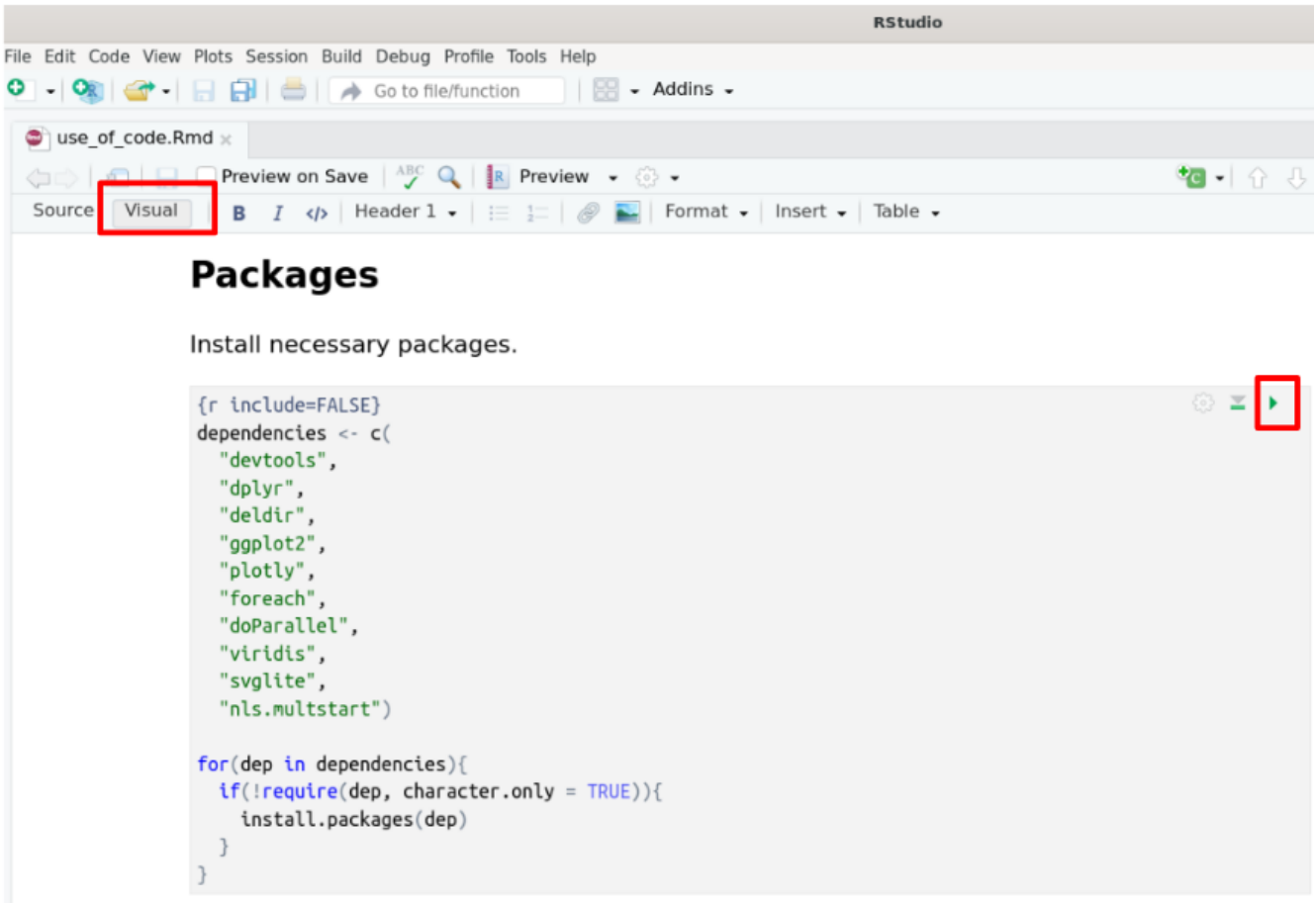


Figure 6: All the necessary packages can be installed executing the first chunk. A chunk can be executed with the green triangle, highlighted in red. Visual mode (higlighted in red) is recommended for users who are not familiar with R.

2.3 Single simulation

To launch one single simulation, refer to the section **1/ Single Simulation**. Here is the description of each chunk:

2.3.1 Parameters

Parameters (dimensionless) for a single simulation can be edited in the list called *params* defined such as follows:

1. *seed*: generating pseudo-random numbers
2. *n_steps*: simulation steps
3. *n_cells*: number of cells (M)
4. *n_layers*: numbers of layers (N)
5. *apical_rad*: apical radius ($\tilde{R}_a$)

6. *ratio_rad*: ratio between apical and basal radius $\frac{R_a}{R_b}$
7. *cyl_length*: cylinder length ($\tilde{L}$)
8. *kappa*: cell membrane elasticity (always 1 because of adimentionality of the model)
9. *gamma*: cell membrane contractility ($\tilde{\Gamma}_0$)
10. *lambda*: cell-cell adhesion constant ($\tilde{\Lambda}$)
11. $beta$: constant involved in jumps because of perturbations
12. *s0_ratio*: characteristic ratio between the actual radius and apical radius

```{r}
params = list(
  seed = 1, # randomness
  n_steps = 50, # simulation steps
  n_cells = 100, # number of cells
  n_layers = 3, # numbers of layers (min 3)
  apical_rad = 5/(2*pi), # apical radius
  ratio_rad = 2.5, # ratio between apical and basal radius
  cyl_length = 20, # cylinder length
  gamma = 0.15, #3  contractility
  lambda = 0.04, # adhesion
  #kappa = 1 cause adimentionality (elasticity)
  beta = 100, #constant involved in jumps because of perturbations
  s0_ratio = 1 # characteristic ratio between the acztual radius and apical radius
  )
```

Figure 7: Chunk where parameters for the model are defined. Any parameter can be modified unless the dimensionless elastic constant κ that is always equal to 1.

2.3.2 Execution and data saving

To execute the code, run the chunk without editing anything. While running, a message should display with the index of the current iteration. Once it is over, a message should display about the right data storage in the **results** folder and the execution time of the code.

```r
{r}
PATH_MODEL = "tubular_simulations_source/model_execution.R"
source(PATH_MODEL)
start.time <- Sys.time()
simulation <- perform_simulations(algorithm="static",parameters=params)
end.time <- Sys.time()
time.taken <- end.time - start.time
print(paste("Execution Time:", round(as.numeric(time.taken, units="mins"), 2), "mins"))
# Results from the algorithm
result_alg <- simulation$results
parameters <- simulation$parameters
algorithm <- simulation$algorithm
hist_points = result_alg$points_evolution
hist_energy = result_alg$energy_evolution
```

```
[1] "step 1 done"
[1] "step 2 done"
[1] "step 3 done"
[1] "step 4 done"
[1] "step 5 done"
[1] "step 6 done"
[1] "step 7 done"
[1] "step 8 done"
[1] "step 9 done"
```

Figure 8: Chunk to execute the code. Messages should display to validate that the algorithm is executing correctly. In this chunk, nothing has to be edited.

This chunks returns a data frame called *simulation* containing 3 elements: a data frame called *result_alg*, a list called *parameters* and a string called *algorithm*. The data frame *result_alg* contains the history of the points coordinates (*points_evolution*) and the history of the energies (*energy_evolution*):

- *points_evolution* contains $(n\_steps + 1) \times n\_cells$ rows. Each row represents a point, the columns "x" and "y" represent the x and y coordinates of the point, and the column "Iteration" represents the step of the algorithm.
- *energy_evolution* contains $(n\_steps + 1)$ rows and 5 columns, one indicating the iteration and the other one indicating the corresponding total,elastic,contractile and adhesion energy of the tube.

For simplicity we store the two lists of *results_alg* in the variables (*hist_points* and *hist_energy*). The list *parameters* allows to access any parameter value using the symbol "$". For instance, the number of layers of the simulation can be called by "parameters$n_layers".

```
[1] "Results saved successfully in path: results/static/results_simulation_25_01_31.RData"
[1] "Execution Time: 0.19 mins"
```

Figure 9: Messages that should display once the code finished running and that the data had been saved properly at the path indicated. Execution time is also provided.

Automatically, the data from *result_alg* is stored in the folder **results/static**.

2.3.3 Loading results

To access data from previous simulations, indicate the path of the file you want to import in *fileName*. Results are stored in the same variable names. If the chunk run properly then a message should be displayed.

```
{r}
fileName <- "results/static/results_simulation_25_03_05_1.RData"

# DO NOT MODIFY
PATH_FUNCTIONS = "tubular_simulations_source/save_simulation_results.R"
source(PATH_FUNCTIONS)
simulation <- load_results(fileName)
result_alg <- simulation$results
parameters <- simulation$parameters
algorithm <- simulation$algorithm
hist_points = result_alg$points_evolution
hist_energy = result_alg$energy_evolution
```

```
[1] "Results loaded successfully from path: results/static/results_simulation_25_03_05_1.RData"
```

Figure 10: Chunk to load previous results. A message should display with the path of the file imported.

2.4 Tube visualization

To visualize the tube, you first need to save the tesselations data from the algorithm. The chunk generates a *.csv* file that is saved in the folder **saved_tesselation** and the file is named "saved_tesselation". A message should display to indicate that the file had been properly saved.

```
{r}
# DO NOT MODIFY
PATH_FUNCTIONS = "tubular_simulations_source/save_utils.R"
source(PATH_FUNCTIONS)
pts_info = save_tessellation_layers(result_alg[[1]], n = parameters$n_cells, RadiusA
=parameters$apical_rad, Ratio = parameters$ratio_rad,cyl_length = parameters$cyl_length, Layers =
parameters$n_layers,it=parameters$n_steps)
```

```
[1] "Results saved successfully in path: results/saved_tessellation.csv"
```

Figure 11: Chunk to save the tesselations data to visualize the 3D tube. A message should display with the path where the data is saved. Nothing to edit in this chunk.

This *.csv* file contains the cell id (*id_cell*), the number of layers (*Layer*), the cell radius (*Radius*), the coordinates of the cell centroids (*Centroidx* and *Centroidy*), the number of vertices of the cell (*n_vertices*), the coordinates of the vertices i with 11 maximum (*vertix,vertiy*). Thus, it follows this format:

id_cell,layer,radius,centroidx,centroidy,n_vertices,vert1x,vert2x,....vert11x, verty1,verty2,.....vert11y

Then, you can open the Mathematica script called *TubularGraphicsMain.nb*, placed in the main folder **3DVoronoiTubular**. You do not need to edit the script and can execute the whole script. To do this, you can simply select "Evaluation" and then "Evaluate notebook".

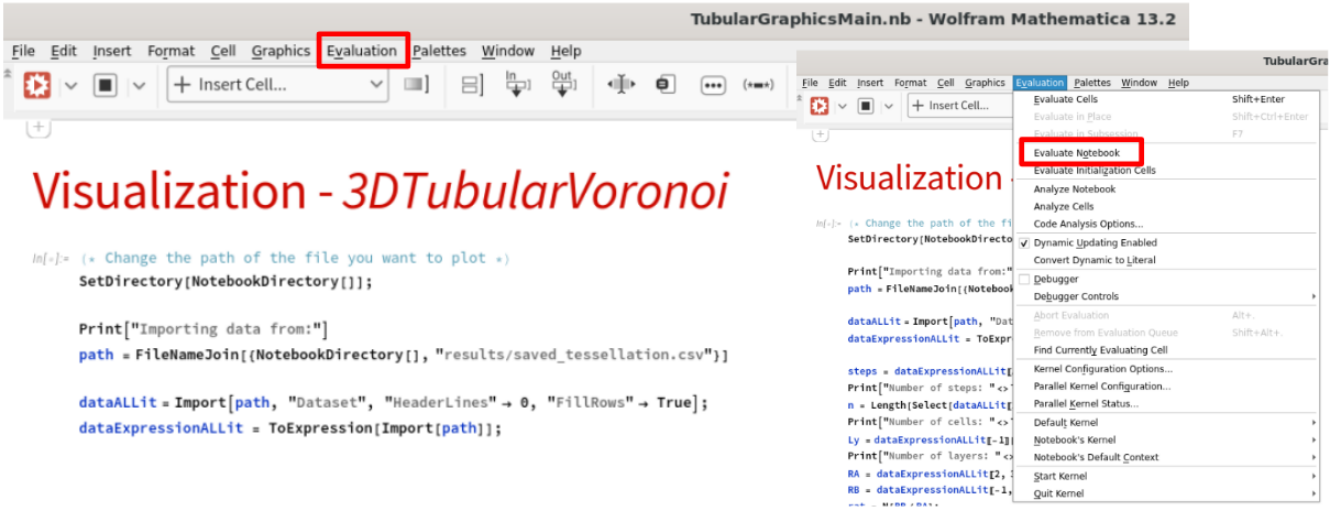


Figure 12: Mathematica script to generate the 3D graphical tube. To execute the script, select "Evaluation" and then "Evaluate notebook".

A message of the right importation of the tesselations data should be display, as well as the properties of the simulation: the number of steps, cells and layers. The script generates and saves the figure of the tube at first and last frame. It also displays an animation of the tubes over the iterations.

2.5 Parallelization

To launch several simulations simultaneously with different seeds (random initial conditions), refer to the section **2/ Parallelization**. Choose the number of simulations using the variable *N_SIM* and edit the list of parameters.

```r
{r include=FALSE}
N_SIM = 10 # number of simulations

library(foreach)
library(doParallel)
PATH_MODEL = "tubular_simulations_source/model_execution.R"
source(PATH_MODEL)
N_CLUSTERS <- detectCores() - 1  # Leave one core free for other processes
cl <- makeCluster(N_CLUSTERS)
registerDoParallel(cl)
start.time <- Sys.time()
cat("Simulations start")
par_results <- foreach(i= 1:N_SIM) .combine = rbind, .packages = "deldir") %dopar% {
 params = list(
  seed = i, # randomness
  n_steps = 50, # simulation steps
  n_cells = 100, # number of cells
  n_layers = 3, # numbers of layers (min 3)
  apical_rad = 5/(2*pi)  # apical radius
```

Figure 13: This chunk launches several simulations (here *N_SIM = 10*) in parallel to minimize execution time. Each simulation has the same list of parameters but with a random initialization of the points.

The results generated are stored in the dataframe *par_results* contains *N_SIM* dataframes, each one composed as described in 2.3.2. As for the single simulation, the results are stored in the folder **results/parallel**.

```
> par_results
      results parameters algorithm
 [1,] list,2  list,11    "static"
 [2,] list,2  list,11    "static"
 [3,] list,2  list,11    "static"
 [4,] list,2  list,11    "static"
 [5,] list,2  list,11    "static"
 [6,] list,2  list,11    "static"
 [7,] list,2  list,11    "static"
 [8,] list,2  list,11    "static"
 [9,] list,2  list,11    "static"
[10,] list,2  list,11    "static"
```

Figure 14: Example of the data frame *par_result* with *N_SIM = 10*

As for the single simulation, you can load previous results choosing the *fileName* and the number of simulations *N_SIM* you want to import.

```r
{r}
fileName <- "results/parallel/results_simulation_25_02_03_"
N_SIM = 10

# DO NOT MODIFY
PATH_FUNCTIONS = "tubular_simulations_source/save_simulation_results.R"
source(PATH_FUNCTIONS)
par_results <- NULL
for(i in 1:N_SIM){
  filename <- paste0(fileName,i,".RData")
  # Extract
  simulation_i <- load_results(filename)
  results_i <- simulation_i$results
  parameters_i <- simulation_i$parameters
  algorithm_i <- simulation_i$algorithm
  # Build the array
  par_results <- rbind(par_results, list(results = results_i, parameters = parameters_i, algorithm =
algorithm_i))}
par_parameters <- par_results[[1,2]]
```

Figure 15: This chunks allows to load previous simulations executed in parallel. Here we load *N_SIM = 10* simulations from the path provided in *fileName*.

You also can save the tesselation data of a selected simulation.

References

[1] R. Farhadifar, J.-C. Röper, B. Aigouy, S. Eaton, and F. Jülicher. The influence of cell mechanics, cell-cell interactions, and proliferation on epithelial packing. *Current Biology*, 17:2095–2104, 2007.