Cover Page

Universiteit Leiden

The handle http://hdl.handle.net/1887/28967 holds various files of this Leiden University dissertation.

Author: Palm, Margaretha Maria (Margriet)
Title: High-throughput simulation studies of angiogenesis - Reverse engineering the role of tip cells and pericytes in vascular development
Issue Date: 2014-09-30

5

Large-scale parameter studies of cell-based models of tissue morphogenesis using *CompuCell3D* or *VirtualLeaf*

Sections 5.1 to 5.3 and 5.A will be published as:

Margriet M. Palm and Roeland M.H. Merks, *Large-scale parameter studies of cell-based models of tissue morphogenesis using* CompuCell3D *or* VirtualLeaf, Tissue Morphogenesis, Methods in Molecular Biology, 2014 (in press)

Section 5.4 is based on:

Floriane Lignet, Anna Emde, Margriet M. Palm, Yossi Yarden, Emmanuel Grenier, Roeland M.H. Merks and Benjamin Ribba, *Explaining morphogenesis of mammary epithelial spheroids overexpressing HER2: a cell-based computational model* (in preparation)

background graphic: collection of final morphologies of the epithelial spheroid formation model (section 5.4)

Abstract

Computational, cell-based models, such as the cellular Potts model have become a widely-used tool to study tissue formation. Most cell-based models mimic the physical properties of cells and their dynamic behavior, and generate images of the tissue that the cells form due to their collective behavior. Due to these intuitive parameters and output, cell-based models are often evaluated visually and the parameters are fine-tuned by hand. To get better insight into how in a cell-based model the microscopic scale (e.g., cell behavior, secreted molecular signals, and cell-ECM interactions) determines the macroscopic scale, we need to generate morphospaces and perform parameter sweeps, involving large numbers of individual simulations. This chapter describes a protocol and presents a set of scripts for automatically setting up, running and evaluating large-scale parameter sweeps of cell-based models. We demonstrate the use of the protocol using a recent cellular Potts model of blood vessel formation model implemented in *CompuCell3D*. We show the versatility of the protocol by adapting it to an alternative cell-based modeling framework, *VirtualLeaf*.

5.1 Introduction

To study the mechanisms of tissue morphogenesis, it is often useful to see a tissue as a swarm of interacting cells that follow a set of stereotypic or stochastic rules, which would be determined ultimately by their genome. The decisions of the cells are then guided by present and past interactions with adjacent cells and the micro-environment. In this view, tissue morphogenesis is a problem of collective cell behavior, in which tissues emerge, sometimes via non-intuitive mechanisms, from stereotypic or stochastic rules that the individual cells follow.

A useful computational tool for studying collective cell behavior is *cell-based modeling* [16, 186]. The inputs to a cell-based model are the behavioral rules that cells follow. The output of a cell-based model is the tissue morphogenesis that follows indirectly from the collective behavior of the individual cells. Cell-based models have been applied to a wide range of problems in developmental biology, including somitogenesis [87] , tumor development [187–190] liver regeneration [191], plant development [192, 193], epithelial branching [194], cystogenesis [195] and angiogenesis and vasculogenesis [25, 28, 33, 34, 40, 81]. In many cell-based models cell behavior is described at a phenomenological level, based on experimental observations. More recent approaches have introduced detailed models of genetic networks guiding cell behavior, see e.g. [87, 196]. These studies demonstrate the utility of cell-based modeling for elucidating the mechanisms of development.

Because most cell-based simulations cannot be solved analytically, insight into their behavior must be obtained using computer simulation. Individual simulations with visual output can give some initial intuition about the behavior and parameter sensitivity of the model. To obtain a more systematic overview of the range of behaviors the model can exhibit, and its sensitivity to parameters, it becomes necessary to rerun the simulation many times for different parameters, and, in case of stochastic models, to obtain statistical measures of the model results by rerunning many random instantiations of the model. If values for the model parameters cannot be determined experimentally, we must test the model for a range of experimentally plausible parameter values [28, 197]. And where parameter values are partly known, systematic parameter studies help predict the response of the system to pharmaceutical treatments [40, 81] or evaluate the behavior of a tissue. Thus, systematic parameter studies are a central tool for analyzing cell-based modeling.

As cell-based models become more complex and take longer to run, performing such parameter studies can become a challenging problem both in terms of computational power and in terms of data management. Here, we describe a protocol and release a set of Python scripts to automatically set up, run and analyze large parameter sweeps of cell-based models on desktop machines, computational clusters, or in the cloud. Although the protocol and parts of the scripts can be used with any kind of simulation tool that can be started from the command line, we illustrate the protocol in detail with a simulation of vasculogenesis (blood vessel formation; [25, 134]), developed using the cell-based simulation package *CompuCell3D* [178]. *CompuCell3D* is an implementation of the Cellular Potts Model (CPM) [84, 85], a widely used cell-based simulation method. The CPM is lattice-based technique that simulates the stochastic, amoeboid motility of biological cells in response to local cues from adjacent cells and diffusive signals, in this way making predictions on collective cell behavior. To illustrate that the parameter sweep can be applied to any kind of simulation tool with a command-line interface, we also show how to adapt the Python scripts to set up, run and analyze a parameter sweep for *VirtualLeaf* [192], which is an alternative cell-based modeling technique. Furthermore, we show how the protocol was adapted to set up parameter sweeps as part of a collaborative study of *in vitro* and *in silico* tumor growth.

5.2 Materials

The following materials and prior knowledge are required for using and extending the code provided in this protocol.

5.2.1 Python

To use *CompuCell3D*, and to run the *CompuCell3D* extensions and parameter sweep scripts presented in this chapter, you will use the programming language Python.

1. Download and install the latest version of the Python 2.x branch (see note 1 in section 5.A) from `http://www.Python.org/download/`. Alternatively, Linux users can install Python using their package manager.
2. Familiarize yourself with Python (see note 2 in section 5.A). We recommend `http://en.wikibooks.org/wiki/Non-Programmer's_Tutorial_for_Python_2.6` for readers with no programming experience and `http://docs.Python.org/2/tutorial` for readers with programming experience in other programming languages.

5.2.2 Cloud computing

Because cell-based simulation models typically take dozens of minutes to hours to complete, depending on the technique you use and the complexity of the model, we recommend using a computer cluster or a cloud computing service to run multiple simulations in parallel. You can acquire access to a computing cluster via your institute or use online services, like *Amazon Webservices* (`http://aws.amazon.com`). If you want to use a computer cluster, familiarize yourself with its usage.

5.2.3 *CompuCell3D*

CompuCell3D [178] offers an easy-to-use graphical user interface for setting up and running simulations. *CompuCell3D* is designed as a modular framework and can therefore easily be extended, either using Python or C++.

1. The Python scripts provided in this chapter require the most recent Numpy version. Download Numpy from `http://sourceforge.net/projects/numpy/files/`and install it (see note 3 in section 5.A) before installing *CompuCell3D*. This will prevent *CompuCell3D* from installing an older release of Numpy.
2. Download a suitable *CompuCell3D* installer from `http://www.compucell3d.org/SrcBin`. Currently installers are available for: Ubuntu Linux 64bit (10.04 and 12.04), OS X (10.5.8, 10.6 and 10.8) and Windows. If there is no installer for your operating system, build *CompuCell3D* from source: `http://www.compucell3d.org/CompilingCC3D` (see note 4 in section 5.A). When

you are installing *CompuCell3D* on a cluster, you may also need to compile *CompuCell3D* because of the absence of suitable installers or because you are not allowed to run an installer from your account. During the installation of *CompuCell3D* you must specify the installation directory, in the remainder of this chapter we will refer to this directory as CC3DPATH.

3. Familiarize yourself with the CPM [84, 85] and with *CompuCell3D*. A step-by-step tutorial explaining the CPM and how to set up and run simulations with *CompuCell3D* can be found in [178] and an overview of the functions of *CompuCell3D* functions can be found in the reference manual [198]. The *CompuCell3D* installation includes a variety of example models, which can be found in the directory "Demos" located in the installation directory of *CompuCell3D*.

5.2.4 *CC3DSimUtils*

As supplementary material to this book chapter, we provide a Python module *CC3DSimUtils*. This module can be used to visualize and analyze simulation results, and to set up simulations with *CompuCell3D*.

1. Create a project directory at any location. In this directory we will store all code, simulation scripts and results. We will refer to the path of this directory as PROJECTPATH.
2. Create a directory named "src" in PROJECTPATH.
3. Download and extract CC3DSimUtils.zip to the "src" directory. Make sure that the directory containing *CC3DSimUtils* is named "CC3DSimUtils". In the subdirectory "doc" of "CC3DSimUtils" you will find the documentation of *CC3DSimUtils* ("html/CC3DSimUtils.html").
4. Tell Python about the location of *CC3DSimUtils*. For a Python script that will be executed from the root of PROJECTPATH, insert the following commands to the beginning of the Python script:

```
import sys
sys.path.append("src/")
```

5. Alternatively, experienced users can add the path to *CC3DSimUtils* to the system variable PYTHONPATH.
6. Install the following packages, which are required for *CC3DSimUtils* (see note 5 in section 5.A):
   - **Scipy**: `http://sourceforge.net/projects/scipy/files/` (see note 3 in section 5.A)
   - **Python imaging library (PIL)**: `www.pythonware.com/products/pil/`
   - **Mahotas**: `http://luispedro.org/software/mahotas` (see note 6 in section 5.A)
   - **Pymorph**: `http://luispedro.org/software/pymorph`.

Installation instructions for these packages can be found at their websites. Alternatively, use a Python package manager, such as setup_tools or pip. For example:

```
>> pip install pymorph
```

5.3 Methods

We illustrate the use of *CC3DSimUtils* using a model of vascular network formation based on the Cellular Potts model, which is implemented in *CompuCell3D*. The model is described in detail elsewhere [25, 134]. Briefly, the model captures the self-organization of endothelial cells into vascular network-like structures, based on the following assumptions: (a) endothelial cells have an elongated shape, they (b) adhere to one another, and (c) they move and rotate randomly [134]. In a variant of the model, chemotaxis speeds up network formation and increases the stability of the networks [25, 134].

Overall, the presented workflow is as follows. We first organize the project directory with several subdirectories that will hold all code, simulation scripts, simulation data, images and analysis results. We then run the model once and analyze the dynamics of network formation. Next, we use this model as the basis for a parameter study: We show how to set up, perform and evaluate a parameter study using *CompuCell3D* and *CC3DSimUtils*. Finally, we illustrate the versatility of the parameter sweep protocol by adapting the Python scripts to the alternative cell-based modeling framework *VirtualLeaf* [192].

5.3.1 Organize Project directory

To organize the simulations, we create a project directory in which you store simulation and analysis scripts, raw simulation data, simulation images and analysis results. The structure of this directory is based on the structure suggested by Noble [199].

1. Create a project directory at any location, if you have not yet done so in section 5.2.4. From now on we will refer to the path to the project directory with PROJECTPATH. This directory will be used for all examples in this section.
2. Create the following subdirectories:
   - "src": holds all non-executable code, such as the *CC3DSimUtils* module;
   - "bin": holds all executable code, such as analysis scripts;
   - "scripts": holds all simulation scripts that will be used with *CompuCell3D*;
   - "log": holds text files that list parameter values and random seeds for automatically generated simulations;
   - "data": holds all raw simulation data;

- "images": holds all images that show the configuration of cells resulting from a simulation;
- "results": holds all data files and images resulting from analysis methods.

5.3.2 Run the *CompuCell3D* model from the command line

We run a simulation of the blood vessel formation model using the command line interface of *CompuCell3D*. By running *CompuCell3D* from the command line we can bypass the graphical user interface, which reduces simulation time. Furthermore, using the command line enables us to use a computer cluster, because clusters are usually unable to run a graphical interface. All commands provided in this section are designed to be executed from the root of the PROJECTPATH. When we refer to directories, we refer to subdirectories of PROJECTPATH.

1. Download the file "steppables.zip" from the supplementary materials and extract it to the "src" directory.
2. Download "longcells_chem.zip" from the supplementary materials and extract it to the "scripts" directory. The zip-file contains three files: a CC3DML file (".xml"), and Python file (extension ".py") and a "*CompuCell3D*" file (extension ".cc3d"). Together, these three files specify a single *CompuCell3D* simulation. Change the variable projectpath on line 5 of "longcells_chem.py" (in subdirectory "longcells_chem" of the "scripts" directory) to your PROJECTPATH. Windows users can either use the slash (/) or two backslashes (\\) in path definitions.
3. Run the simulation by typing the following in a terminal emulator (Linux and OS X) or Command Prompt (windows) (see notes 7 to 9 in section 5.A):
4. Linux and OS X

```
>> CC3DPATH/runScript.sh --noOutput -i PROJECTPATH/
   scripts/longcells_chem.cc3d
```

5. Windows

```
>> CC3DPATH\runScript.bat --noOutput -i PROJECTPATH\
   scripts\longcells_chem.cc3d
```

6. This may take up to ∼45 minutes.
7. Next we plot the simulation results, and combine the images for a number of time steps in a single figure. Download "longcells_chem_draw.py" and "default.ctb" from the supplementary materials. Save "longcells_chem_draw.py" in the "bin" directory and "default.ctb" in your PROJECTPATH. Run "draw_longcells_chem.py" (see note 10 in section 5.A):

```
>> python bin/longcells_chem_draw.py
```

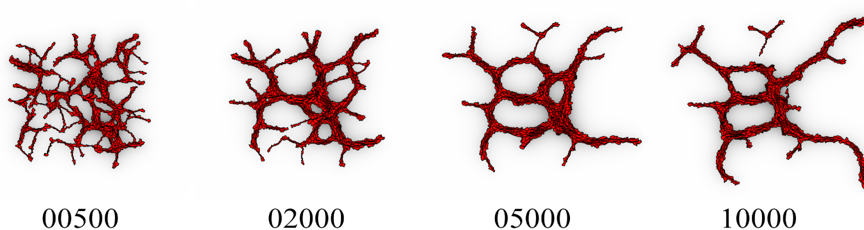

Figure 5.1: Morphologies at 500, 2000, 5000, 10 000 time steps, for a simulation with "longcells_chem".

The directory "longcells_chem" (subdirectory of "images") contains the morphologies of consecutive time steps, and the "results" directory contains a collage similar to Figure 5.1 ("longcells_chem.png") of the morphologies for 500, 2000, 5000 and 10 000 simulations steps combined. The morphologies are created with the function makeImages from *CC3DSimUtils*. This function draws images using the data files generated by the simulation. The function stackImages from *CC3DSimUtils* can be used to combine any set of images of the same size. See the *CC3DSimUtils* documentation for further details.

5.3.3 Analyzing a single *CompuCell3D* simulation

Now we have a set of simulation results and images. We next present a series of methods to quantify these simulation results.

1. Calculate the compactness of the vascular network simulations. The compactness is defined as: $\frac{A_{cells}}{A_{hull}}$; with A_{cells} the total area of the largest connected component and A_{hull} the area of the convex hull. The convex hull can be seen as the smallest "gift wrapping" around an object. Download "longcells_chem_compactness.py" from the supplementary materials, save it in the "bin" directory and run with:

```
>> python bin/longcells_chem_compactness.py
```

   The "results" directory will contain a tab-separated text file, "longcells_chem_compactness.data", which lists the compactness for every time step measured. A plot of this data should look similar to Figure 5.2.
2. Analyze where the elongated cells align with one another and where defects in alignment occur. To do so, we first quantify and visualize the relative orientations of the cells. Calculate an angle θ between the cell at a pixel $\vec{x}$ and the average orientation in the neighborhood of $\vec{x}$. The orientation of a cell, $\vec{v}$, is the orientation of the long axis of that cell. Assuming

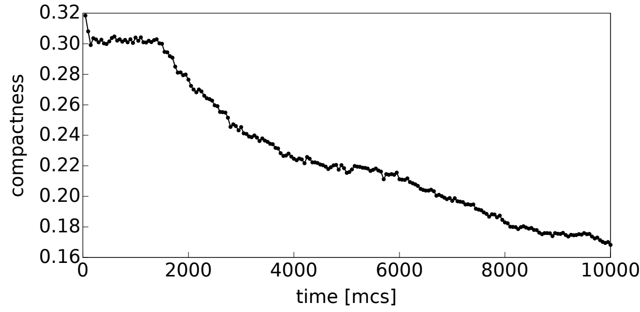

Figure 5.2: Time evolution of the compactness for a single simulation of "longcells_chem".

that cells are close to elliptic, we can approximate $\vec{v}$ by the orientation of the eigen-vector corresponding to the largest eigen-value of the cell's inertia tensor. For a cell C, defined as the set of pixels with coordinates $\vec{x} = (x_1, x_2)$ that the cell occupies, the inertia tensor is defined as:

$$I(C) = \begin{pmatrix} \sum_{\vec{x}\in C} x_2^2 & -\sum_{\vec{x}\in C} x_1 x_2 \\ -\sum_{\vec{x}\in C} x_1 x_2 & \sum_{\vec{x}\in C} x_1^2 \end{pmatrix}.$$

The average cell orientation within a disk of radius r centered on $\vec{x}$ is called the *director*:

$$\vec{n}(\vec{x}, r) = \langle \vec{v}(\sigma(\vec{y})) \rangle_{\{\vec{y}\in Z^2 : |\vec{x}-\vec{y}|<r\}}.$$

The angle θ between the cell orientation $\vec{v}$ and the director $\vec{n}$ is a measure for local cell alignment:

$$\theta(\vec{x}, r)_{raw} = \cos^{-1}\left(\frac{|\vec{n}(\vec{x}, r)\cdot \vec{v}(\sigma(\vec{x}))|}{|\vec{n}(\vec{x}, r)||\vec{v}(\sigma(\vec{x}))|}\right).$$

Download "longcells_chem_alignment.py" from the supplementary materials, save it to the "bin" directory, and then run with:

```
>> python bin/longcells_chem_alignment.py
```

This script calculates θ for $r = 3$ at each pixel and plots it on the morphology. The script stores the resulting images in the subdirectory "longcells_chem" of "images". It also creates a collage similar to Figure 5.3 ("longcell_chem_reldir_r=3.png" in "results"), which combines plots of θ for $r = 3$ at time steps 500, 2000, 5000 and 10 000.

3. The 2D nematic order parameter $S(r) = \langle \cos(2\theta(\vec{X}(\sigma), r)) \rangle_\sigma$, with $\vec{X}(\sigma)$ the center of mass of cell σ, quantifies the degree of local alignment in a morphology with a number between 0 and 1. $S(r) \to 1$ for cells aligning with

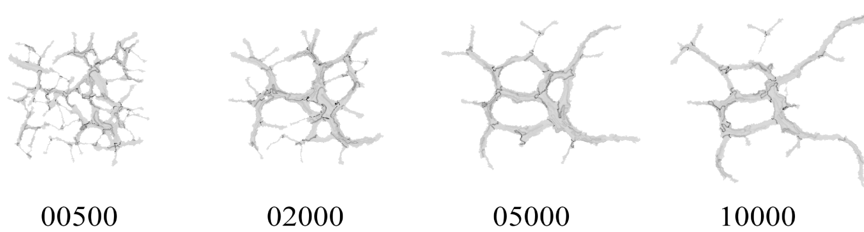

Figure 5.3: Angle θ between cells and the local director (for $r = 3$) mapped on the morphologies at 500, 2000, 5000, 10 000 time steps, for a simulation with "longcells_chem".

one other on overage over a distance r, and $S(r) \to 0$ for cells with random orientations. Download "longcells_chem_orderparameter.py" from the supplementary materials and save it in the "bin" directory. Run the script with:

```
>> python bin/longcells_chem_orderparameter.py
```

This script produces a file, named "longcells_chem_orderparameter.data", in the "results" directory that contains the order parameter for radii 20, 40 and 600 for every 250th time step. Note that radius $r = 600$ the disk covers the whole 400x400 simulation domain, so $S(600)$ becomes a *global order parameter*. Plotting the evolution of the order parameters should result in an image similar to Figure 5.4.

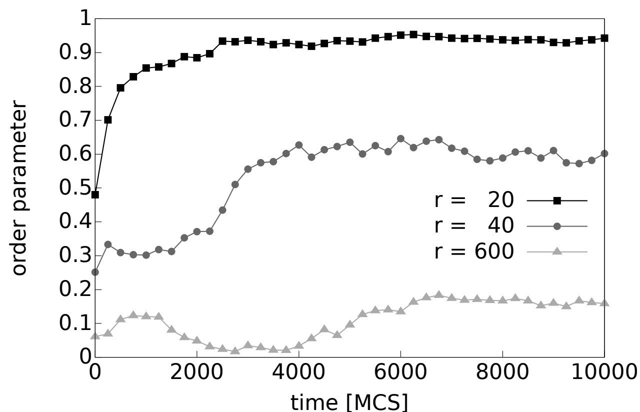

Figure 5.4: Time evolution of the order parameter for radii 20, 40 and 600 (global). for a single simulation of "longcells_chem".

5.3.4 Setting up and running a parameter sweep with CompuCell3D

We showed how to set up, run and analyze a single simulation using *CompuCell3D* and *CC3DSimUtils*. To gain insight into how specific parameters affect the model behavior, a model should be simulated repeatedly with different parameter values. In case of a stochastic model, such as the CPM, the simulation for each parameter value should be repeated multiple times to obtain good statistics. Here we show how to set up and run such a parameter sweep. First, we create a *driver* script that runs and analyzes a single simulation. Next, we show how to automatically create the simulation scripts for each parameter value and simulation repeat. In this example we vary the surface tension and turn chemotaxis on or off.

1. Download "driver.py" from the supplementary materials and save it a subdirectory of PROJECTPATH called "bin". The driver script runs a simulation (section 5.3.2) and analyzes the simulation results (section 5.3.3). Change the variables `projectpath` and `cc3dpath` in "driver.py" such that `projectpath` points to your PROJECTPATH and `cc3dpath` points to your CC3DPATH. For example, to run the driver script for "longcells_chem.cc3d" run the driver script with:

```
>> python bin/driver.py longcells_chem
```

   The concept of collecting all operations concerning a single simulation in one driver script can be applied to any modeling method that can be invoked from the command line.

2. (for cluster users) When a driver script is used on a cluster, you may also include commands to compress and pack the data to facilitate data transfer to your desktop machine. The command system in the Python module "os" can call the compression utilities from your driver script. To create a compressed archive containing all files starting with "longcells_chem_001-1" and ending with ".data", append the following line to the driver script:

```
os.system("tar -czf data_longcells_chem_001-1.tar.gz
    longcells_chem_001-1*.data")
```

3. To automatically set up the simulation, we use template simulation scripts. Download "templates.zip" from the supplementary materials and extract it in the root of PROJECTPATH. This will create a new folder named "templates" and in it you find four files: "longcells_chem.py", "longcells_chem.xml", "longcells_nochem.py" and "longcells_nochem.xml". The first two files serve as templates for the simulations with chemotaxis and the second two files serve as templates for the simulations without chemotaxis.
4. Automatically generate the scripts needed to run a *CompuCell3D* simulation. Download "preprocess.py" from the supplementary materials, save it

in the "bin" folder and run with:

```
>> python bin/preprocess.py
```

For each parameter value specified in 'preprocess.py' this script creates for each repeat: a *CompuCell3D* script, and a directory containing a CC3DML script and Python script in the "scripts" directory. For each simulation repeat a unique random seed is generated to ensure that each simulation is different (see note 11 in section 5.A). Each simulation is identified by an automatically generated simulation name, constructed as: [description]_ [number]-[repeat]. We use the [description] to differentiate between simulations with and without chemotaxis: "longcells_chem" and "longcells_ nochem". The three-digit simulation [number] is used to link a simulation to a parameter value. The [repeat] is a number that is used to set apart the simulation repeats. Besides the scripts for *CompuCell3D,* "preprocess.py" also generates log files (in the directory "log") that store the parameter values ("longcells_1-10.sim") and the random seeds ("longcells_1-10_10x .seed"). "preprocess.py" is specific for changing the surface tension in a set of templates. For other *CompuCell3D* models and/or other parameter sets, adapt "preprocess.py" using the functionality in the *Experiment* class of *CC3DSimUtils*. See the *CC3DSimUtils* documentation for more details.

5. (for cluster users) The simulations become faster if you save the simulation results on a section of the file system local to the node you are running on (often called *scratch space*, and move the data to your home directory when the simulation is finished. Point the variable datapath in "preprocess.py" to the scratch space, and add commands to the driver to copy the data back to your home directory. For this we recommend using the Python standard library modules *os* and *shutil*.
6. (for cluster users). The script "preprocess_cluster.py" automatically generates the job scripts needed to schedule the simulations on cluster using PBS [200]. Download "preprocess_cluster.py" from the supplementary materials to the "bin" directory, create a directory "clusterscripts" in PROJECTPATH, and run "preprocess_cluster.py":

```
>> python bin/preprocess_cluster.py
```

After running the script, there will be a number of PBS scripts in the "clusterscripts" directory and should look like:

```
#PBS -S /bin/bash
#PBS -lnodes=1:cores8:ppn=8
#PBS -lwalltime=8:00:00
cd \$HOME
python driver.py longcells_chem_001-1 > log/
   longcells_chem_001-1.out 2> log/longcells_chem_001
```

```
    -1.err &
...
python driver.py longcells_chem_001-8 > log/
    longcells_chem_001-8.out 2> log/longcells_chem_001
    -8.err &
wait
```

Submit the job script with "qsub" to add the run to the queue on the cluster. Each PBS script contains 8 jobs and requests an 8-core node (see Note 12 in section 5.A) . To change these parameters, change the variables cores and ppn in "preprocess_cluster.py". You may also need to modify the function createPBS in "CC3DPipeline.py" in *CC3DSimUtils* to fit the hardware and scheduling software of the cluster you are using.

7. When all simulations are finished we have a collection of raw data files, data analysis results and images. For each simulation all data files should be located in the "data" directory, in a subdirectory with the simulation name, for example: "PROJECTPATH/data/longcells_chem_001-1/". Similarly, the images are expected to be in a subdirectory with the simulation name in the directory "images": "PROJECTPATH/images/longcells_chem_001-1/". If this is not the case, move your data files and/or images to these locations.

5.3.5 Analyzing a *CompuCell3D* parameter sweep

After running the parameter sweep we have raw data, data analysis results and images for each simulation. Here we show how to collect and present this data.

1. Create a *morphospace*, a collage of simulated morphologies as a function of one or two simulation parameters. Download "postprocess_morphospace.py" from the supplementary materials, save it to the "bin" directory, and run it with:

```
>> python bin/postprocess_morphospace.py
```

   Now, you should find an image named "longcells_1-10_morphospace_100000.png" in the "results" folder, which should look similar to Figure 5.5. The morphospace is created with the function morphImages, from *CC3DSimUtils*. See the *CC3DSimUtils* documentation for more details.

2. Calculate the compactness as function of the surface tension. Download "postprocess_compactness.py" from the supplementary materials and save it in the "bin" directory. Run it with:

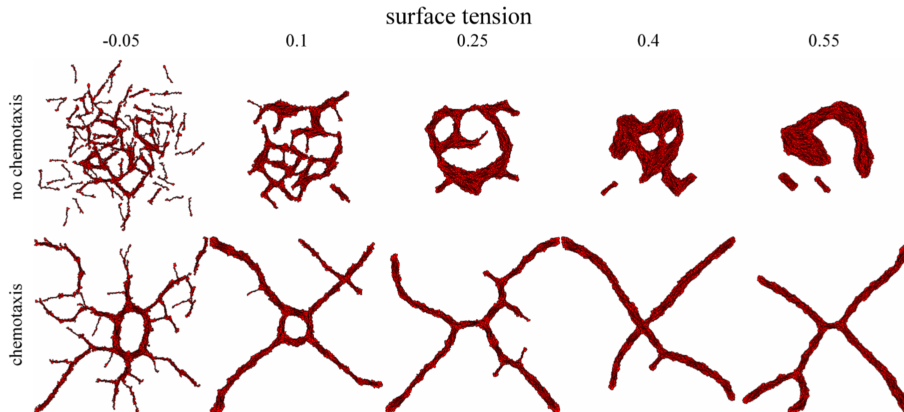

Figure 5.5: Morphospace showing the effects of varying surface tensions with and without chemotaxis.

```
>> python bin/postprocess_compactness.py
```

The script collects the compactness at the last time step of each simulation repeat for each tested parameter value and it calculates the mean and standard deviation over the simulation repeats. The results can be found in "longcells_chem_1-10_10x_compactness.data" (simulations with chemotaxis) and "longcells_nochem_1-10_10x_compactness.data" (simulations without chemotaxis) in the "results" directory. Plotting this data should result in a plot similar to Figure 5.6.

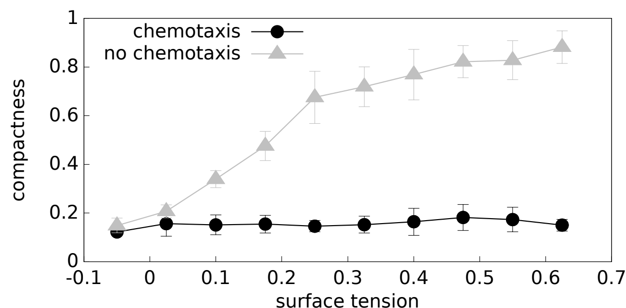

Figure 5.6: Compactness for simulations with and without chemotaxis, plotted against the surface tension. The compactness was calculated at 100 000 time steps, and for each parameter the simulation was repeated 10 times (error bars represent standard deviation).

3. Quantify the degree of cell alignment by calculating the mean and standard deviation of the order parameter. Download "postprocess_orderparameter.py" from the supplementary materials, save it into the "bin" directory and run the script with:

```
>> python bin/postprocess_orderparameter.py
```

This script calculates the mean and standard deviation the order parameter $S(r, t)$ as a function of time for radii $r = 20$ and $r = 40$, and the global order parameter ($r = 600$) for all simulation repeats of one surface tensions. The results can be found in "longcells_chem_003_10x_order parameter.data" and "longcells_nochem_003_10x_orderparameter.data", in the "results" directory. With the data in these files we generated the plot in Figure 5.7 for simulations of 1 000 000 time steps. Note that to reduce simulation time the scripts presented in this chapter produce only 100 000 time steps.

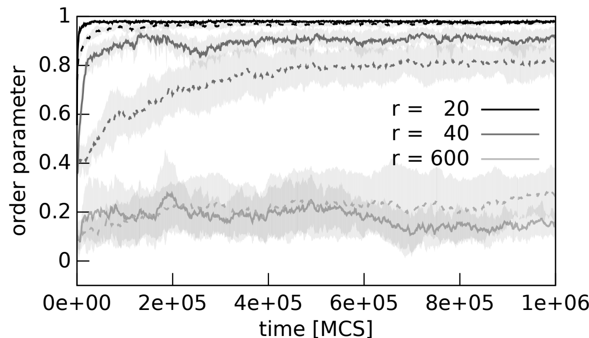

Figure 5.7: Time evolution of the order parameter for radii 20, 40 and 600 (global) for simulations with (solid) and without (dashed) chemotaxis. Each line represents the average order parameter of 10 simulations and the gray areas represent the standard deviation.

4. To relate cell motility to the degree of cell alignment, we detect clusters of aligned cells in the morphology. We loosely define a "cluster" as a set of cells aligned with the local director, $\theta(\vec{x}, r) < \theta$, with θ a threshold value. Clusters are separated from other clusters by regions with values of $\theta(\vec{x}, r) > \theta$ (dark gray regions in Figure 5.3). More formally, clusters are detected as follows:
   a) Define a binary matrix B of dimensions equal to the simulation domain.
   b) Assign a value of $B(\vec{x}) = 1$ to all cell pixels $\vec{x}$ for which $\theta(\vec{x}, r) < \theta$, with θ a threshold value and $B(\vec{x}) = 0$ for all other pixels.
   c) In B detect all connected components larger than 50 pixels. A set of cells forms a cluster if each cell overlaps with the same connected component in B for 50% of its area or more.

5. To study how aggregation of aligned cells in clusters affects cell behavior we measured the translation and rotation of the cells as a function of cluster size. The translational diffusion coefficient D_t quantifies the translational motility of cells. It is derived from the *mean squared displacement* (MSD) of a cell: $\langle|\vec{X}(\sigma, t) - \vec{X}(\sigma, 0)|^2\rangle_\sigma = 4D_t t$. The rotational diffusion coefficient D_r is derived from the *mean squared rotation* (MSR) of a cell: $\langle(\alpha(\sigma, t) - \alpha(\sigma, 0))^2\rangle_\sigma = 2D_r t$.
6. Calculate the translational and rotational diffusion coefficients. Download "postprocess_diffusion.py" from the supplementary materials, save it in the "bin" directory and run the script with:

```
>> python bin/postprocess_diffusion.py
```

This script creates time series of the MSD and the MSR of each cell as a function of the cluster size and uses those time series to calculate the translational and rotational diffusion coefficients. First, cells are binned according to cluster size for each time step, with a bin size of 5 cells and the first bin running from 2 to 5 cells. Then, the MSD and MSR of each cell are split into chunks of 10 consecutive time steps, during which that cell belonged to the same cluster size bin. Using these binned chunks the translational and rotational diffusion coefficients are calculated with a least square fit of respectively the MSD and MSR. The diffusion coefficients, together with the standard error of the estimate of the fit, are stored in "longcells_nochem_003_10x _diffusion.data" in the "results" directory. In Figure 5.8 we plot the translational and rotational diffusion coefficients calculated using data from time step 500 to 250 000 (similar to our previous work [134]). As mentioned before, the scripts presented in this chapter only produce 100 000 time steps in order to reduce simulation time.

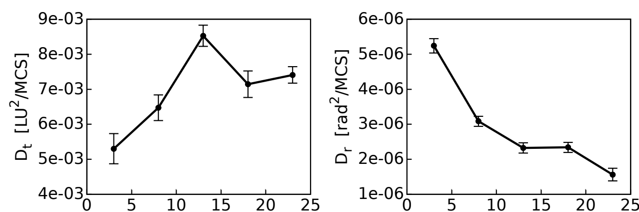

Figure 5.8: Diffusion coefficients as a function of the cluster size. **A.** translation diffusion coefficient and **B.** rotational diffusion coefficient. These diffusion coefficients were calculated from 10 simulations of 250 000 time steps. The error bars represent the standard error of the estimates of the least square fit.

5.3.6 Adapting the protocol to *VirtualLeaf*

The scripts described in sections 5.3.4 and 5.3.5 can be adapted to any simulation package that is 1) invoked from the command line and 2) for which model parameters are specified in a text file. As an example, we show how to use the scripts used in section 5.3.4 and 5.3.5 to set up, run and analyze for the cell-based, vertex-based modeling framework *VirtualLeaf* [192, 201]. A model in the *VirtualLeaf* is defined by a so-called *plugin* and the model parameters are defined in a so-called *leaf* file. With the *leaf* file and the *plugin*, *VirtualLeaf* can be invoked from the command line. Thus, *VirtualLeaf* meets both of the requirements of the parameter sweep protocol.

1. Create a project directory for your *VirtualLeaf* project (as described in section 5.3.1).
2. Adapt "driver.py" to run and analyze *VirtualLeaf* simulations.
   a) To run the simulation, "driver.py" uses the function os.system (on line 26), which attempts to execute its argument on the command line. For example, for a *VirtualLeaf* model defined in "plugin.cpp" and the parameters defined in "leaf.xml", this argument must be (see note 13 in section 5.A):

   ```
   /path/to/VirtualLeaf/bin/VirtualLeaf -b -l leaf.xml
       -m libplugin
   ```

   Assign the path to the VirtualLeaf executable to execpath (line 7) and assign the executable name, *i.e.*, "VirtualLeaf" to executable (lines 10-11) . Next, define a new variable named plugin, before line 26, that points to the plugin in which your model is defined (*e.g.*, 'libplugin'). Now, change line 26 to:

   ```
   os.system('execpath'+'/'+executable+' -b -l '+
      projectpath+'/scripts/'+id+'.xml -m '+plugin)
   ```

   b) Remove the commands on line 28 and further, and replace them with calls to your own analysis functions.
3. Create a template *leaf* file for your model that contains the default parameter values for your model.
4. Create a new Python script to automatically generate *leaf* files, based on "preprocess.py".
   a) Copy line 1 of "preprocess.py" to import the necessary Python libraries.
   b) Create a function `buildLeafFile` to change specific parameter values in a template *leaf* file. Because *leaf* files are based on XML, you can use "Experiment.py" as an example on how to adapt an XML file using Python. Besides changing specific parameter values, `buildLeafFile` also assigns a random seed, sets the intervals at which the simulation generates graphical and numerical output, and it sets the file names

and location of the output. As with the *CompuCell3D* simulations, model output files should be identified by the simulation description, simulation number and repeat number: description_number-repeat, and be stored in a directory with the same name in the "data" directory of PROJECTPATH.

c) Define a variable projectpath (see line 47 of "preprocess.py"):

```
projectpath = [PROJECTPATH]
```

replacing [PROJECTPATH] for your PROJECTPATH.

d) Define the parameters of the parameter sweep. These are simname for the simulation description, `offset` for the first simulation number, `repeats` for the number of repeats, and rep0 for the first repeat number (see lines 51-58 of "preprocess.py"). For example:

```
simname = 'leaf'
offset = 1
repeats = 10
rep0 = 1
```

e) Set the simulation time (simtime) and the frequency at which output is generated (savefreq) (see lines 59-62 of "preprocess.py"). For example:

```
simtime = 1000
savefreq = 25
```

f) Create a list of parameter values, named par, that will be tested in the parameter sweep (see line 64 of "preprocess.py").

g) Create output files for simulation settings and seeds, and write the file headers (see lines 70-76 of "preprocess.py"):

```
runid = simname+'_'+str(offset)+'-'+str(offset+len(
    par)-1)
# open log file for parameter values
out = open('log/'+runid+'.sim','w')
out.write('\#id\tPARAMETERNAME')
# open log file for random seeds
sout = open('log/'+runid+'_'+str(repeats)+'x''.seed
    ','w')
sout.write('\#id\tseed')
```

h) Iterate over the parameters and the simulation repeats (see lines 79-101 of "preprocess.py"). The outer loop is used to write the tested parameter values to the log files. In the inner loop the random seed is generated and the *leaf* file is created:

```
seeds = []
for i,p in enumerate(par):
  out.write('\n'+name+'_'+string.zfill(i+offset,3)+
      '\t'+str(p)+'\n')
  for n in range(rep0,repeats+rep0):
    simid = name+'_'+string.zfill(i+offset,3)+'-'+
        str(n)
    seed = random.randint(1,10**9)
    # check if seed is unique
    while seed in seeds:
      seed = random.randint(1,10**9)
    seeds.append(seed)
    sout.write('\n'+simid+'\t'+str(seed))
    #--- Create leaf file ---\#
    buildLeafFile(...)
```

5. Adapt "preprocess_cluster.py" to create a set of scripts for the PBS job scheduler.
   a) Change `numlist` (line 7) such that it represents the ranges from the lowest simulation number to the highest simulation number.
   b) Change `replist` (line 9) such that it represents the range from the lowest repeat number to the highest repeat number.
   c) Replace `basename` (line 13) with your simulation description.
   d) Change `cores` (line 17), `ppn` (line 19) and `runtime` (line 21) to fit the type of node, number of processors per node and simulation time you will request on the cluster.
   e) Replace line 25 with:

```
joblist = [name+'_'+string.zfill(num,3)+'-'+str(n)
           for name in simnames
           for num in numlist for n in replist]
```

6. Depending on the analysis performed in the driver script, create your own set of post-process scripts. For this, you can use post-process scripts from section 5.3.5 as examples:
   - "postprocess_morphospace.py" can be used as an example for creating morphospaces;
   - "postprocess_compactness.py" can be used as an example to collect values for one time point per simulation, and mapping that data on parameter values;
   - "postprocess_orderparameter.py" can be used as an example on how to collect time course data for multiple simulations.

5.4 Case study - Mammary epithelial spheroid morphogenesis

We applied the protocol described in this chapter in a collaborative, computational study on the growth of epithelial spheroids *in vitro*. These spheroids develop from mammary epithelial cells and are used as a model for breast cancer development. Wild-type mammary epithelial cells (WT) form a small spheroid by dividing randomly (Figure 5.9**A**-**B**) [202]. The cells on the inside of the spheroid stop dividing, while the cells on the outside keep on dividing (Figure 5.9**C**). The outside cells may become polarized cells that only divide perpendicular to the spheroid surface, while the other cells on the spheroid surface keep on dividing in random randomly (Figure 5.9**D**) [203]. Eventually, the cells on the inside die, resulting in a lumen (Figure 5.9**E**-**F**) [203], and the polarized cells stop dividing, stabilizing the spheroid [204]. When mammary epithelial cells overexpress human epidermal growth factor receptor 2 (HER-2), large spheroids with a minimal lumen develop [202–204]. The aim of the study is to find how the probabilities of proliferation, polarization and apoptosis is changed in HER-2 cells in comparison to wild-type cells. To this end we developed a cell-based model that reproduces epithelial spheroid formation with WT epithelial cells. Then, try to identify the differences between WT and HER-2 cells by varying the proliferation, polarization and apoptosis probabilities in the model.

In this section we describe the model and quantification methods. Then, we show how we adapted our protocol to set up a parameter sweep that is used to setup the spheroid formation model for WT cells. An extensive discussion of the results and description of the further steps in this study are outside the scope of this section.

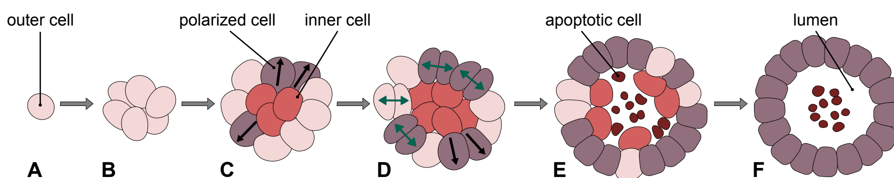

Figure 5.9: Development of an epithelial spheroid. **A**-**B** a single outer cell (pink) divides to form a cluster of outer cells. **C** outer cells that lose contact with the ECM become inner cells (red) and outer cells may become polarized cells (purple-gray). **D** outer cells divide in random directions, while the polarized cells divide perpendicular to the spheroid surface. **E**-**F** inner cells may enter apoptosis (burgundy cells) resulting in the formation of a lumen, and polarized cells become stabilized cells that do not divide. (This image is adapted from an image that was kindly provided by Floriane Lignet.)

5.4.1 Model of epithelial spheroid formation

The development of epithelial cell spheroids is modeled using the cellular Potts method [84, 85]. In this method the different cellular phenotypes that manifest during spheroid development are modeled explicitly, as well as transitions between the types and proliferation. The cell types involved in the spheroid development are: *outer, inner, polarized, stabilized* and *apoptotic*. Besides these biological cell types, two extra tissue types are added to represent *ECM* and *lumen*. All cells, except stabilized cells, can transition into another cell type, or proliferate (Figure 5.10). Outer cells can become inner cells or proliferate. Polarized cells can also become inner cells and proliferate, and they can become stabilized cells. Inner cells can become apoptotic cells and apoptotic cells die and become lumen. Both the type transitions and proliferation occur with a probability p, but only if the cells' environment fits a set of conditions. For this the environment of a cell, identified by σ, is characterized by the fraction of cell membrane M the cell shares with certain cell types:

$$\mathrm{M}(\sigma, \mathrm{types}) = \sum_{\tau \in \mathrm{types}} \frac{P_{\sigma \cup \tau}}{P(\sigma)}$$

with $P_{\sigma \cup \tau}$ is the length of the perimeter that cell σ shares with cells of type τ and $P(\sigma)$ is the total perimeter of cell σ. In the model there are three of these fractions defined: the fraction of the membrane shared with ECM ($\mathrm{M}(\sigma, \{\mathrm{ECM}\})$), the fraction of the membrane shared with lumen ($\mathrm{M}(\sigma, \{\mathrm{lumen}\})$), and the fraction of membrane shared with cells ($\mathrm{M}(\sigma, \mathrm{cells})$ with cells=$\{$outer, inner, polarized, stabilized, apoptotic$\}$). Figure 5.10 shows the criteria for $\mathrm{M}(\sigma, \{\mathrm{ECM}\})$, $\mathrm{M}(\sigma, \{\mathrm{lumen}\})$, and $\mathrm{M}(\sigma, \mathrm{cells})$ for each transition and proliferation.

5.4.2 Quantification of spheroid morphology

To quantitatively compare the morphologies formed by the model with *in vitro* spheroids we compute several *shape indexes*: solidity, cell to hull ratio, and core factor. The solidity s measures how similar the shape of the spheroid is to a circle. For this we divide the radius (r_A) of a circle with the area of the spheroid (A) by the radius (r_P) of a circle with the perimeter of the spheroid (P):

$$s = \frac{r_A}{r_P} \quad \text{with} \quad r_A = \frac{A}{\pi} \quad \text{and} \quad r_P = \frac{P}{2\pi}.$$

The cell to hull ratio (CTH) measures the convexity of the spheroids; the higher the CTH the more convex the spheroid. For this we compute the ratio of the spheroid area (A) and the area of the convex hull around the spheroid

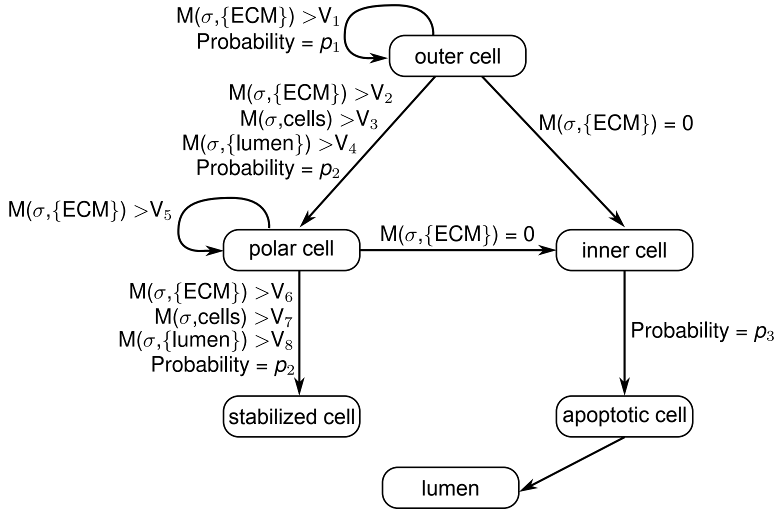

Figure 5.10: Flowchart of the cell type transitions.

(A_{hull}).

$$c = \frac{A}{A_{\text{hull}}}$$

The core factor is used to quantify the relative size of the lumen by computing the ratio of the lumen area (A_{lumen}) and the spheroid area (A).

$$f = \frac{A_{\text{lumen}}}{A}$$

All of these metrics were computed with Matlab and the Image Processing Toolbox.

5.4.3 Parameter identification

To identify the transition conditions for which the model reproduces experimental observations we performed a parameter sweep. There are 4 transition rules with eight transition parameters: $V = \{V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8\}$. Because each V represents a fraction of the cell membrane, the sum of V's in one transition condition (Figure 5.10) can never be larger than one. From these we derive the following limitations for V:

$$\begin{aligned}
V_1 &\leq 1 \\
V_2 + V_3 + V_4 &\leq 1 \\
V_5 &\leq 1 \\
V_6 + V_7 + V_8 &\leq 1
\end{aligned}$$

For the parameter sweep we generate 10 000 sets V. Each value was randomly drawn from a uniform distribution on the interval $[0, 1]$, and sets that did not obey Equation 5.4.3 were disregarded and replaced by a newly generated set. For each set of V ten simulations are preformed with different random seeds.

Because the model is implemented in *CompuCell3D*, with *Python* steppables for the type transitions, proliferation and apoptosis, we can easily adapt the scripts used in section 5.3.4. In our previous work the number of simulations was limited to a couple of hundreds and therefore creating simulation files for each simulations was not an issue. In contrast, here we have 100 000 simulations and therefore we chose to adapt the protocol to generate simulation scripts on the fly. For this we automatically generate a list of 10 000 sets V and use each line as an input for the driver script. The driver script then finds and replaces the parameter values in the simulations scripts. Besides generating simulation scripts on the fly, the protocol for setting up and running the simulations was similar to those previously described in section 5.3.4. For the morphological analysis we did not use *CC3DSimUtils*, but instead we used a Matlab script. Because it is quite complicated to run Matlab code on the cluster we used, the analysis was done for all simulations at once when all data was retrieved from the cluster.

Next, we visualize the results of the parameter sweep using scatter plots, which is typical first step in parameter identification [205]. Figure 5.11 shows these scatter plots in which each point represents the mean of the ten simulation repeats. Based on these graphs the effects of each parameter on the shape of the spheroid and the size of the lumen can be assessed. For example, V_5 seems to have the strongest influence on the lumen area: the higher V_5, the less lumen forms. More complex sensitivity analysis techniques [205] may be used to further analyze the results and find the best values for V. Then, this process can be repeated for the transition probabilities p_1, p_2, and p_3, to find how these parameters should be changed such that the model reproduces the HER2 spheroids.

Altogether, with this case study we showed that the protocol described in this chapter can easily be adapted for setting up a parameter identification study of a cell model. Finding the best values for V may require the use of more complex sensitivity analysis techniques [205]. When the values for V are identified, a similar parameter sweep can be performed for probabilities p_1, p_2, and p_3 to identify the differences between WT and HER-2 cells.

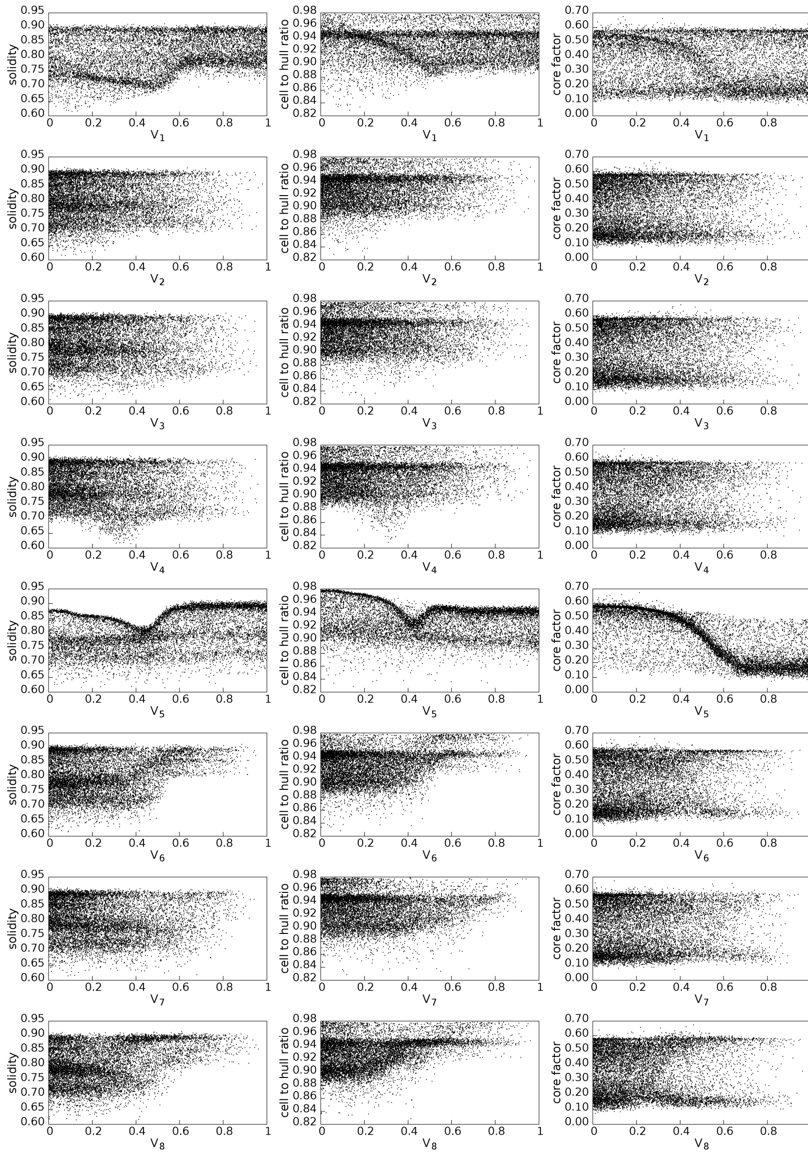

Figure 5.11: Effects of varying V_1-V_8 on the spheroid solidity, cell to hull ratio and compactness. Each point represents the mean of 10 simulations with identical values for V_1-V_8.

5.A Notes

note 1 There are two Python branches: Python 2.x and Python 3.x, which are not fully compatible. All code supplied with this chapter and *CompuCell3D* is compatible with Python 2.6 and new versions of the Python 2.x branch.

note 2 Python uses indentation to delimit blocks of codes. In code that is copied from different sources, indentation may be broken due to different indentation lengths or mixing of tabs and spaces. See `http://www.Python.org/dev/peps/pep-0008/#indentation` and `http://www.Python.org/dev/peps/pep-0008/#tabs-or-spaces` for more information on how to correctly indent your code.

note 3 Windows users should install the Numpy or Scipy version that fits with your Python version. First check your Python version:

```
>> python -V
```

Note the first two digits of the Python version, e.g. 2.7. Now go to the download page of Numpy or Scipy and select the latest version. Here you should find an installer that ends with your python version, e.g. "numpy-x.y.z-win32-superpack-python27.exe".

note 4 When you compile *CompuCell3D*, always check the *CompuCell3D* website for the most recent instructions and dependencies. Here we list some extra instructions for the compilation of *CompuCell3D*.

- The *CompuCell3D* developers recommend to use "cmake-gui", for systems without a graphical user interface the "cmake curses gui", also known as "ccmake", can be used as an alternative.
- Ensure that you compile *CompuCell3D* with the "release" flag because omitting this flag significantly increases simulation time. The "release" flag can be set with the "cmake-gui" or "ccmake".

note 5 *CC3DSimUtils* needs freetype fonts for the labels on images. You may need to install freetype (http://www.freetype.org/download.html) or change the variable fontpath in the function definitions of makeImage, stackImages and morphImages (all in "ImageUtils.py" in *CC3DSimUtils*).

note 6 For windows users we strongly recommend to download the installer for Mahotas at: `http://www.lfd.uci.edu/~gohlke/pythonlibs/`. Building the source of Mahotas, for example using pip, is not recommended.

note 7 *CompuCell3D* interprets all paths relative to its own path. Therefore, when running a simulation using runScript.sh you should specify the full path to the simulation file, for example

- Windows:
  `C:\Users\username\project_name\scripts\script.cc3d`
- Linux: `/home/username/project_name/scripts/script.cc3d`
- OS X: `/Users/username/project_name/scripts/script.cc3d`

note 8 On windows, running runScript.bat changes the working directory to the CC3DPATH. Make sure to change it back to the PROJECTPATH afterwards.

note 9 In this model we use a connectivity constraint to ensure that each cell consists of single connected component. Calculating the connected components is computationally expensive, therefore *CompuCell3D* only checks for local connectivity by checking if a cell is a single connected component within a small neighborhood. This can cause pixels to become *frozen*, because any change in their neighborhood breaks local connectivity. We fixed this by adding an extra test to the connectivity constraint for pixels that fit the pattern of a frozen pixel. We used this fixed connectivity constraint for all our simulation. This plugin ("ConnectivityFroNo.zip") can be downloaded from the supplementary materials and compiled as a part of the *CompuCell3D* developer zone, see the *CompuCell3D* developers' manual [206] for instructions.

note 10 On Windows the Python installation directory may not be in the $PATH, this results in an error like:

```
'Python' is not recognized as an internal or
   external command, operable program or batch file
   .
```

Adding the installation directory of Python to your $PATH system variable should solve this problem.

note 11 Random seeds are used to initialize a random generator. Every time a random generator is initialized with the same results, it returns the same sequence of pseudo-random numbers. Thus, if we run a *CompuCell3D* simulation twice with identical seeds, we get identical results. If no random seed is provided, *CompuCell3D* uses the current time to generate a random seed. When multiple simulations are started at the same moment, for example on a computer cluster, they will get the same seed. Thus, predefining random seeds has two advantages: 1) the results are reproducible; 2) the random seeds in parameter sweep are unique.

note 12 It is often more efficient to leave one core of a node idle. This core is then reserved for system processes while the remaining nodes are reserved for user processes, *i.e.* the simulations. To do so, set ppn

(processes per node) in "preprocess_cluster.py" to one less than there are cores (number of cores of the requested node).

note 13 Due to a bug in the current version of *VirtualLeaf* (1.0.1), if *VirtualLeaf* is invoked with both a *leaf* file and a *plugin* in which another *leaf* file is defined, the *leaf* file defined in the *plugin* will be used. To correct this problem, add the following code to "virtualleaf.cpp' after the line with "model_catalogue.InstallFirstModel();", and recompile:

```
if (leaffile){
  main_window->Init(leaffile);
}
```

5.B Supplementary materials

All the supplementary materials can be found at
http://persistent-identifier.org/?identifier=urn:nbn:nl:ui:18-22500.