CompuCell3D Manual and Tutorial
Version 3.6.0

Maciej H. Swat, Susan D. Hester, Randy W. Heiland, Benjamin L. Zaitlen,
James A. Glazier, Abbas Shirinifard

Contents

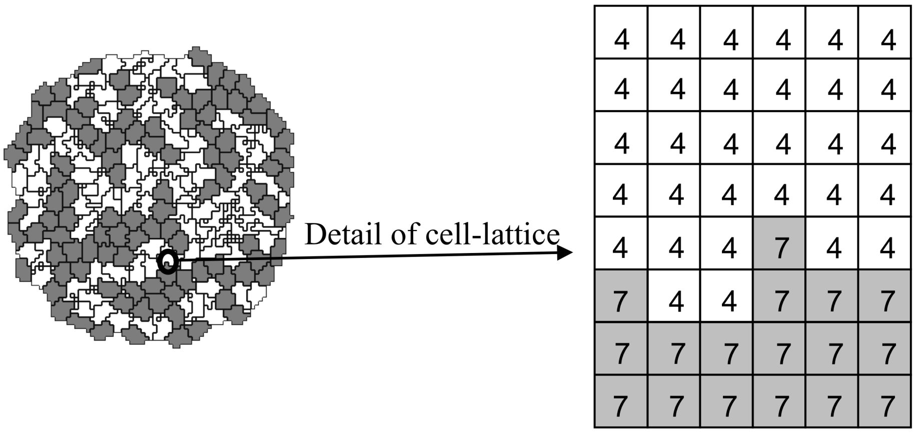
The goal of this manual is to teach biomodelers how to effectively use multi-scale, multi-cell simulation environment CompuCell3D to build, test, run and post-process simulations of biological phenomena occurring at single cell, tissue or even up to single organism levels. We first introduce basics of the Glazier-Graner-Hogeweg (GGH) model aka Cellular Potts Model (CPM) and then follow with essential information about how to use CompuCell3D and show simple examples of biological models implemented using CompuCell3D. Subsequently we will introduce more advanced simulation building techniques such as Python scripting and writing extension modules using C++. In everyday practice, however, the knowledge of C++ is not essential and C++ modules are usually developed by core CompuCell3D developers. However, to build sophisticated and biologically relevant models you will probably want to use Python scripting. Thus we strongly encourage readers to acquire at lease basic knowledge of Python. We dont want to endorse any particular book but to guide users we might suggests names of the authors of the most popular books on Python programming: David Beazley, Mark Lutz, Mark Summerfield, Michael Dawson, Magnus Lie Hetland.

1 Introduction

The last decade has seen fairly realistic simulations of single cells that can confirm or predict experimental findings. Because they are computationally expensive, they can simulate at most several cells at once. Even more detailed subcellular simulations can replicate some of the processes taking place inside individual cells. E.g., Virtual Cell (http://www.nrcam.uchc.edu) supports microscopic simulations of intracellular dynamics to produce detailed replicas of individual cells, but can only simulate single cells or small cell clusters.

Simulations of tissues, organs and organisms present a somewhat different challenge: how to simplify and adapt single cell simulations to apply them efficiently to study, *in silico*, ensembles of several million cells. To be useful, these simplified simulations should capture key cell-level behaviors, providing a phenomenological description of cell interactions without requiring prohibitively detailed molecular-level simulations of the internal state of each cell. While an understanding of cell biology, biochemistry, genetics, etc. is essential for building useful, predictive simulations, the hardest part of simulation building is identifying and quantitatively describing appropriate subsets of this knowledge. In the excitement of discovery, scientists often forget that modeling and simulation, by definition, require simplification of reality.

One choice is to ignore cells completely, *e.g.*, Physiome (1) models tissues as continua with bulk mechanical properties and detailed molecular reaction networks, which is computationally efficient for describing dense tissues and non-cellular materials like bone, extracellular matrix (ECM), fluids, and diffusing chemicals (2, 3), but not for situations where cells reorganize or migrate.



4	4	4	4	4	4
4	4	4	4	4	4
4	4	4	4	4	4
4	4	4	4	4	4
4	4	4	7	4	4
7	4	4	7	7	7
7	7	7	7	7	7
7	7	7	7	7	7

Figure 1: Detail of a typical two-dimensional GGH cell-lattice configuration. Each colored domain represents a single spatially-extended cell. The detail shows that each generalized cell is a set of cell-lattice sites (or *pixel*), $\mathbf{i}$, with a unique index, $\sigma(\mathbf{i})$, here 4 or 7. The color denotes the cell type, $\tau(\sigma(\mathbf{i}))$.

Multi-cell simulations are useful to interpolate between single-cell and continuum-tissue extremes because cells provide a natural level of abstraction for simulation of tissues, organs and organisms (4). Treating cells phenomenologically reduces the millions of interactions of gene products to several behaviors: most cells can move, divide, die, differentiate, change shape, exert forces, secrete and absorb chemicals and electrical charges, and change their distribution of surface properties. The Glazier-Graner-Hogeweg (GGH) approach facilitates multiscale simulations by defining spatially extended generalized cells, which can represent clusters of cells, single cells, subcompartments of single cells or small subdomains of non-cellular materials. This flexible definition allows tuning of the level of detail in a simulation from intracellular to continuum without switching simulation framework to examine the effect of changing the level of detail on a macroscopic outcome, e.g., by switching from a coupled ordinary differential- equation (ODE) Reaction-Kinetics (RK) model of gene regulation to a Boolean description or from a simulation that includes subcellular structures to one that neglects them.

2 GGH Applications

Because it uses a regular cell lattice and regular field lattices, GGH simulations are often faster than equivalent Finite Element (FE) simulations operating at the same spatial granularity and level of modeling detail, permitting simulation of tens to hundreds of thousands of cells on lattices of up to 1024^3 pixels on a single processor. This speed, combined with the ability to add biological mechanisms via terms in the effective energy, permit GGH modeling of a wide variety of situations, including: tumor growth (5-9), gastrulation (10-12), skin pigmentation (13-16), neurospheres (17), angiogenesis (18-23), the immune system (24, 25), yeast colony growth (26, 27), *myxobacteria* (28-31), stem cell differentiation (32, 33), *Dictyostelium discoideum* (34-37), simulated evolution (38-43), general developmental patterning (14, 44), convergent extension (45, 46), epidermal formation (47), hydra regeneration (48, 49), plant growth, retinal patterning (50, 51), wound healing (47, 52, 53), biofilms (54-57), and limb-bud development (58, 59).

3 GGH Simulation Overview

All GGH simulations include a list of objects, a description of their *interactions* and *dynamics* and appropriate *initial conditions.*

Objects in a GGH simulation are either generalized cells or *fields* in two dimensions (2D) or three dimensions (3D). Generalized cells are spatially-extended objects (Fig. 1), which reside on a single *cell lattice* and may correspond to biological cells, subcompartments of biological cells, or to portions of non-cellular materials, *e.g.*, ECM, fluids, solids, etc. (8, 48, 60-72). We denote a lattice site or *pixel* by a vector of integers, $\mathbf{i}$, the *cell index* of the generalized cell occupying pixel $\mathbf{i}$ by $\sigma(\mathbf{i})$ and the *type* of the generalized cell $\sigma(\mathbf{i})$ by $\tau(\sigma(\mathbf{i}))$. Each generalized cell has a unique cell index and contains many pixels. Many generalized cells may share the same cell type. Generalized cells permit coarsening or refinement of simulations, by increasing or decreasing the number of lattice sites per cell, grouping multiple cells into clusters or subdividing cells into variable numbers of *subcells* (subcellular compartments). Compartmental simulation permits detailed representation of phenomena like cell shape and polarity, force transduction, intracellular membranes and organelles and cell-shape changes. For details on the use of subcells, which we do not discuss in this chapter see (27, 31, 73, 74). Each generalized cell has an associated list of attributes, *e.g.*, *cell type, surface area* and *volume*, as well as more complex attributes describing a cells state, biochemical interaction networks, etc. *Fields* are continuously-variable concentrations, each of which resides on its own lattice. Fields can represent chemical diffusants, non-diffusing ECM, etc. Multiple fields can be combined to represent materials with textures, *e.g.*, fibers.

Interaction descriptions and *dynamics* define how GGH objects behave both biologically and physically. Generalized-cell behaviors and interactions are embodied primarily in the *effective energy*, which determines a generalized cells shape, motility, adhesion and response to extracellular signals. The effective energy mixes true energies, such as cell-cell adhesion with terms that mimic energies, *e.g.*, the response of a cell to a chemotactic gradient of a field (75). Adding *constraints* to the effective energy allows description of many other cell properties, including osmotic pressure, membrane area, etc. (76-83).

The cell lattice evolves through attempts by generalised cells to move their boundaries in a caricature of cytoskeletally-driven cell motility. These movements, called *index-copy attempts*, change the effective energy, and we accept or reject each attempt with a probability that depends on the resulting *change of the effective energy*, ΔH, according to an *acceptance function*. Nonequilibrium statistical physics then shows that the cell lattice evolves to locally minimize the total effective energy. The classical GGH implements a modified version of a classical stochastic Monte-Carlo pattern-evolution dynamics, called *Metropolis dynamics with Boltzmann acceptance* (84, 85). A *Monte Carlo Step (MCS)* consists of one index-copy attempt for each pixel in the cell lattice.

Auxiliary equations describe cells absorption and secretion of chemical diffusants and extracellular materials (*i.e.*, their interactions with fields), state changes within cells, mitosis, and cell death. These auxiliary equations can be complex, *e.g.*, detailed RK descriptions of complex regulatory pathways. Usually, state changes affect generalised-cell behaviors by changing parameters in the terms in the effective energy (e.g., cell target volume or type or the surface density of particular cell-adhesion molecules).

Fields also evolve due to secretion, absorption, diffusion, reaction and decay according to partial differential equations (PDEs). While complex coupled-PDE models are possible, most simulations require only secretion, absorption, diffusion and decay, with all reactions described by ODEs running inside individual generalised cells. The movement of cells and variations in local diffusion constants (or diffusion tensors in anisotropic ECM) means that diffusion occurs in an environment with moving boundary conditions and often with advection. These constraints rule out most sophisticated PDE solvers and have led to a general use of simple forward-Euler methods, which can tolerate them.

The *initial condition* specifies the initial configurations of the cell lattice, fields, a list of cells and their internal states related to auxiliary equations and any other information required to completely describe the simulation.

3.1 Effective Energy

The core of GGH simulations is the *effective energy*, which describes cell behaviours and interactions.

One of the most important effective-energy terms describes cell adhesion. If cells did not stick to each other and to extracellular materials, complex life would not exist (86). Adhesion provides a mechanism for building complex structures, as well as for holding them together once they have formed. The many families of adhesion molecules (CAMs, cadherins, etc.) allow embryos to control the relative adhesivities of their various cell types to each other and to the noncellular ECM surrounding them, and thus to define complex architectures in terms of the cell configurations which minimise the adhesion energy.

To represent variations in energy due to adhesion between cells of different types, we define a *boundary energy* that depends on $J\big[\tau(\sigma(\mathbf{i})),\tau(\sigma(\mathbf{j}))\big]$, *the boundary energy per unit area* between two cells $(\sigma(\mathbf{i}),\sigma(\mathbf{j}))$ of given types $(\tau(\sigma(\mathbf{i})),\tau(\sigma(\mathbf{j})))$ at a *link* (the interface between two neighboring pixels):

$$\mathcal{H}_{\text{boundary}} = \sum_{(\mathbf{i},\mathbf{j})\text{ neighbors}} J\big[\tau(\sigma(\mathbf{i})),\tau(\sigma(\mathbf{j}))\big]\big[1-\delta(\sigma(\mathbf{i}),\sigma(\mathbf{j}))\big]\,, \tag{1}$$

where the sum is over all neighboring pairs of lattice sites $\mathbf{i}$ and $\mathbf{j}$ (note that the neighbor range may be greater than one), and the boundary-energy coefficients are symmetric,

$$J\big[\tau(\sigma(\mathbf{i})),\tau(\sigma(\mathbf{j}))\big] = J\big[\tau(\sigma(\mathbf{j})),\tau(\sigma(\mathbf{i}))\big]\,. \tag{2}$$

In addition to boundary energy, most simulations include multiple constraints on cell behavior. The use of constraints to describe behaviors comes from the physics of classical mechanics. In the GGH context we write *constraint energies* in a general *elastic* form:

$$\mathcal{H}_{\text{constraint}} = \lambda(\text{value} - \text{target\_value})^2 \tag{3}$$

The constraint energy is zero if *value* = *target_value* (the constraint is satisfied) and grows as value diverges from *target_value*. The constraint is *elastic* because the exponent of 2 effectively creates an ideal spring pushing on the cells and driving them to satisfy the constraint. λ is the *spring constant* (a positive real number), which determines the *constraint strength*. Smaller values of λ allow the pattern to deviate more from the *equilibrium condition* (*i.e.*, the condition satisfying the constraint). Because the constraint energy decreases smoothly to a minimum when the constraint is satisfied, the energy minimising dynamics used in the GGH automatically drives any configuration towards one that satisfies the constraint. However, because of the stochastic simulation method, the cell lattice need not satisfy the constraint exactly at any given time, resulting in random fluctuations. In addition, multiple constraints may conflict, leading to configurations which only partially satisfy some constraints.

Because biological cells have a given volume at any time, most GGH simulations employ a *volume constraint*, which restricts volume variations of generalised cells from their target volumes:

$$\mathcal{H}_{\text{volume}} = \sum_{\sigma} \lambda_{\text{volume}}(\tau(\sigma)) \big[v(\tau(\sigma)) - V_t(\tau(\sigma))\big]^2 \tag{4}$$

where for cell σ, $\lambda_{\text{volume}}(\sigma)$ denotes the *inverse compressibility* of the cell, $v(\sigma)$ is the number of pixels in the cell (its *volume*), and $V_t(\sigma)$ is the cells *target volume.* This constraint defines $P \equiv -2\lambda(v(\sigma) - V_t(\sigma))$ as the *pressure* inside the cell. A cell with $v < V_t$ has a positive internal pressure, while a cell with $v > V_t$ has a negative internal pressure.

Since many cells have nearly fixed amounts of cell membrane, we often use a surface area constraint of form:

$$\mathcal{H}_{\text{surface}} = \sum_{\sigma} \lambda_{\text{surface}}(\tau(\sigma)) \big[s(\tau(\sigma)) - S_t(\tau(\sigma))\big]^2 \tag{5}$$

where $s(\sigma)$ is the surface area of cell σ, $S_t(\sigma)$ is its target surface area, and λ_{surface} is its *inverse membrane compressibility.*[1]

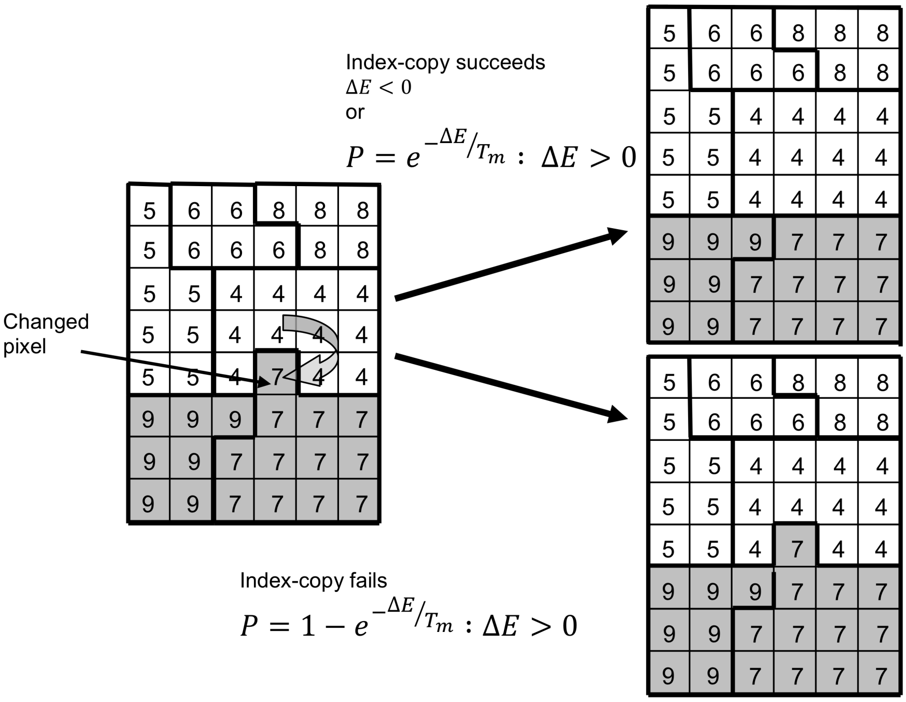
Adding the boundary energy and volume constraint terms together (Equations (1) and (4)), we obtain the basic *GGH effective energy*:

$$\begin{aligned} \mathcal{H}_{GGH} \quad = \quad & \sum_{(\mathbf{i},\mathbf{j})\text{ neighbors}} J\big[\tau(\sigma(\mathbf{i})), \tau(\sigma(\mathbf{j}))\big]\,[1 - \delta(\sigma(\mathbf{i}), \sigma(\mathbf{j}))] \\ & + \sum_{\sigma} \lambda_{\text{volume}}(\tau(\sigma)) \big[v(\tau(\sigma)) - V_t(\tau(\sigma))\big]^2 \\ & + \sum_{\sigma} \lambda_{\text{surface}}(\tau(\sigma)) \big[s(\tau(\sigma)) - S_t(\tau(\sigma))\big]^2 + E_{\text{chemotaxis}}\,. \end{aligned} \tag{6}$$

[1] Because of lattice discretisation and the option of defining long range neighborhoods, the surface area of a cell scales in a non-Euclidian, lattice-dependent manner with cell volume, *i.e.*, $s(v) \neq (4\pi)^{1/3}(3v)^{2/3}$, see (61) on bubble growth.

3.2 Dynamics

A GGH simulation consists of many attempts to copy cell indices between neighboring pixels. In CompuCell3D the default dynamical algorithm is *modified Metropolis dynamics*. During each index-copy attempt, we select a target pixel, **i**, randomly from the cell lattice, and then randomly select one of its neighboring pixels, **i**, as a source pixel. If they belong to the same generalised cell (*i.e.*, if $\sigma(\mathbf{i}) = \sigma(\mathbf{j})$) we do not need copy index. Otherwise, the cell containing the source pixel attempts to occupy the target pixel. Consequently, a successful index copy increases the volume of the source cell and decreases the volume of the target cell by one pixel.


Index-copy succeeds
$\Delta E < 0$
or
$P = e^{-\Delta E/T_m} : \Delta E > 0$
Changed pixel
Index-copy fails
$P = 1 - e^{-\Delta E/T_m} : \Delta E > 0$

5	6	6	8	8	8
5	6	6	6	8	8
5	5	4	4	4	4
5	5	4	4	4	4
5	5	4	7	4	4
9	9	9	7	7	7
9	9	7	7	7	7
9	9	7	7	7	7

5	6	6	8	8	8
5	6	6	6	8	8
5	5	4	4	4	4
5	5	4	4	4	4
5	5	4	4	4	4
9	9	9	7	7	7
9	9	7	7	7	7
9	9	7	7	7	7

5	6	6	8	8	8
5	6	6	6	8	8
5	5	4	4	4	4
5	5	4	4	4	4
5	5	4	7	4	4
9	9	9	7	7	7
9	9	7	7	7	7
9	9	7	7	7	7

Figure 2: GGH representation of an index-copy attempt for two cells on a 2D square lattice – The "white" pixel (source) attempts to replace the "grey" pixel (target). The probability of accepting the index copy is given by equation (7).

In the modified Metropolis algorithm we evaluate the change in the total effective energy due to the attempted index copy and accept the index-copy attempt with probability:

$$P(\sigma(\mathbf{i}) \to \sigma(\mathbf{j})) = \begin{cases} \exp(-\Delta H/T_m) & \text{if } \Delta H > 0 \\ 1 & \text{if } \Delta H \leq 0 \end{cases} \tag{7}$$

where T_m is a parameter representing the *effective cell motility* (we can think of T_m as the amplitude of cell-membrane fluctuations). Equation (7) is the *Boltzmann acceptance* *function*. Users can define other acceptance functions in CompuCell3D. The conversion between MCS and experimental time depends on the average values of $\Delta H/T_m$. MCS and experimental time are proportional in biologically-meaningful situations (87-90).

3.3 Algorithmic Implementation of Effective-Energy Calculations

Consider an effective energy consisting of boundary-energy and volume-constraint terms as in equation (6). After choosing the source ($\mathbf{i}$) and destination ($\mathbf{j}$) pixels (the cell index of the source will overwrite the target pixel if the index copy is accepted), we calculate the change in the effective energy that would result from the copy. We evaluate the change in the boundary energy and volume constraint as follows. First we visit the target pixels neighbors ($\mathbf{j}$). If the neighbor pixel belongs to a different generalised cell from the target pixel, *i.e.*, when $\sigma(\mathbf{j}) \neq \sigma(\mathbf{i})$ (see equation (1)), we decrease ΔH by $J\big[\tau(\sigma(\mathbf{i})), \tau(\sigma(\mathbf{j}))\big]$. If the neighbor belongs to a cell different from the source pixel ($\mathbf{j}$), we increase ΔH by $J\big[\tau(\sigma(\mathbf{i})), \tau(\sigma(\mathbf{j}))\big]$.

The change in volume-constraint energy is evaluated according to:

$$\begin{aligned}
\Delta H_{\text{volume}} &= \mathcal{H}^{\text{new}}_{\text{volume}} - \mathcal{H}^{\text{old}}_{\text{volume}} \\
&= \lambda_{\text{volume}}\Big[(v(\sigma(\mathbf{j})) + 1 - V_t(\sigma(\mathbf{j})))^2 + (v(\sigma(\mathbf{i})) - 1 - V_t(\sigma(\mathbf{i})))^2\Big] \\
&\quad -\lambda_{\text{volume}}\Big[(v(\sigma(\mathbf{j})) - V_t(\sigma(\mathbf{j})))^2 + (v(\sigma(\mathbf{i})) - V_t(\sigma(\mathbf{i})))^2\Big] \qquad (8) \\
&= \lambda_{\text{volume}}\Big[\{1 + 2(v(\sigma(\mathbf{j})) - V_t(\sigma(\mathbf{j})))\} + \{1 - 2(v(\sigma(\mathbf{i})) - V_t(\sigma(\mathbf{i})))\}\Big]
\end{aligned}$$

where $v(\sigma(\mathbf{j}))$ and $v(\sigma(\mathbf{i}))$ denote the volumes of the generalised cells containing the target and source pixels, respectively.

In this example, we could calculate the change in the effective energy locally, *i.e.*, by visiting the neighbors of the target of the index copy. Most effective energies are quasilocal, allowing calculations of ΔH similar to those presented above. The locality of the effective energy is crucial to the utility of the GGH approach. If we had to calculate the effective energy for the entire cell lattice for each index-copy attempt, the algorithm would be prohibitively slow.

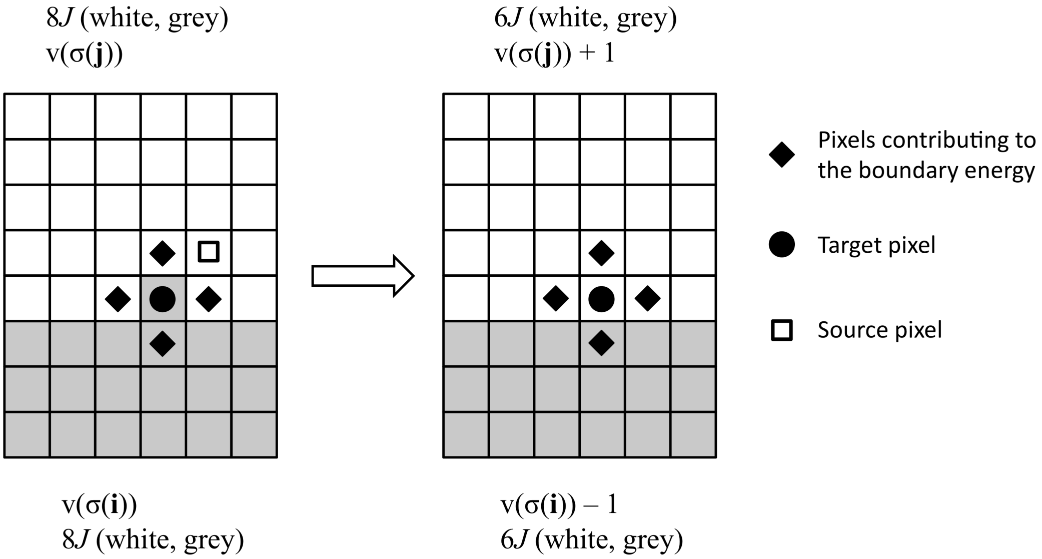


Figure 3: Calculating changes in the boundary energy and the volume-constraint energy on a nearest-neighbor square lattice.

For longer-range interactions we use the appropriate list of neighbors, as shown in Figure 4 and Table 1. Longer-range interactions are less anisotropic but result in slower simulations.

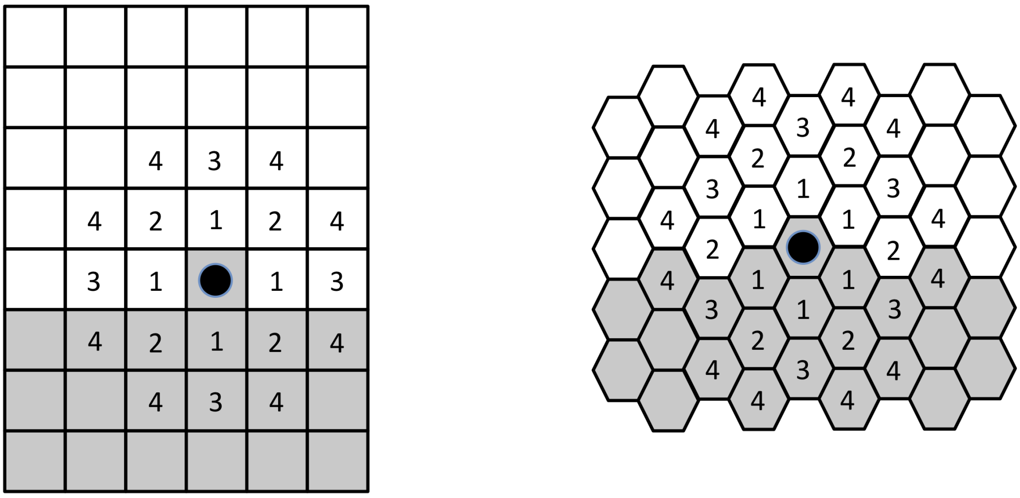


Figure 4: Locations of n^{th}-nearest neighbors on a 2D square lattice and a 2D hexagonal lattice.

	2D Square Lattice		2D Square Lattice	
neighbor order	Number of neighbors	Euclidian distance	Number of neighbors	Euclidian distance
1	4	1	6	$\sqrt{2/\sqrt{3}}$
2	4	$\sqrt{2}$	6	$\sqrt{6/\sqrt{3}}$
3	4	2	6	$\sqrt{8/\sqrt{3}}$
4	8	$\sqrt{5}$	12	$\sqrt{14/\sqrt{3}}$

Table 1: Multiplicity and Euclidian distances of n^{th}-nearest neighbors for 2D square and hexagonal lattices. The number of n^{th} neighbors and their distances from the central pixel differ in a 3D lattice. CompuCell3D calculates distance between neighbors by setting the volume of a single pixel (whether in 2D or 3D) to 1.

4 CompuCell3D

One advantage of the GGH model over alternative techniques is its mathematical simplicity. We can implement fairly easily a computer program that initialises the cell lattice and fields, performs index copies and displays the results. In the 15 years since the GGH model was developed, researchers have written numerous programs to run GGH simulations. Because all GGH implementations share the same logical structure and perform similar tasks, much of this coding effort has gone into rewriting code already developed by someone else. This redundancy leads to significant research overhead and unnecessary duplication of effort and makes model reproduction, sharing and validation needlessly cumbersome.

To overcome these problems, we developed CompuCell3D as a framework for GGH simulations (91, 92). Unlike specialised research code, CompuCell3D is a *simulation environment* that allows researchers to rapidly build and run shareable GGH-based simulations. It greatly reduces the need to develop custom code and its adherence to open-source standards ensures that any such code is shareable.

CompuCell3D supports non-programmers by providing visualisation tools, an *eXtensible Markup Language (XML)* interface for defining simulations, and the ability to extend the framework through specialised modules. The C++ computational kernel of CompuCell3D is also accessible using the open-source scripting language Python, allowing users to create complex simulations without programming in lower-level languages such as C or C++. Unlike typical research code, changing a simulation does not require recompiling CompuCell3D.

Users define simulations using *CompuCell3D XML (CC3DML) configuration files* and/or

Python scripts. CompuCell3D reads and parses the CC3DML configuration file and uses it to define the basic simulation structure, then initialises appropriate Python services (if they are specified) and finally executes the underlying simulation algorithm.

CompuCell3D is modular: each module carries out a defined task. CompuCell3D terminology calls modules associated with index copies or index-copy attempts plugins. Some plugins calculate changes in effective energy, while others (lattice monitors) react to accepted index copies, *e.g.*, by updating generalised cells' surface areas, volumes or lists of neighbors. Plugins may depend on other plugins. For example, the `Volume` plugin (which calculates the volume-energy constraint in equation (4)) depends on `VolumeTracker` (a lattice monitor), which, as its name suggests, tracks changes in generalised cells volumes. When implicit plugin dependencies exist, CompuCell3D automatically loads and initialises dependent plugins. In addition to plugins, CompuCell3D defines modules called *steppables* which run either repeatedly after a defined intervals of Monte Carlo Steps or once at the beginning or end of the simulation. Steppables typically define initial conditions, alter cell states, update fields or output intermediate results.

Figure 5 shows the relations among index-copy attempts, Monte Carlo Steps, steppables and plugins.

CompuCell3D includes a *Graphical User Interface (GUI)* and visualisation tool, *CompuCell Player* (also referred to as *Player*). From Player the user opens a CC3DML configuration file and/or Python file and hits the "Play" button to run the simulation. Player allows users to define multiple 2D or 3D visualisations of generalised cells, fields or various vector plots while the simulation is running and save them automatically for post-processing.

5 Building CC3DML-Based Simulations Using CompuCell3D

To show how to build simulations in CompuCell3D, the reminder of this chapter provides a series of examples of gradually increasing complexity. For each example we provide a brief explanation of the physical and/or biological background of the simulation and listings of the CC3DML configuration file and Python scripts, followed by a detailed explanation of their syntax and algorithms. We can specify many simulations using only a simple CC3DML configuration file. We begin with three examples using only CC3DML to define simulations.

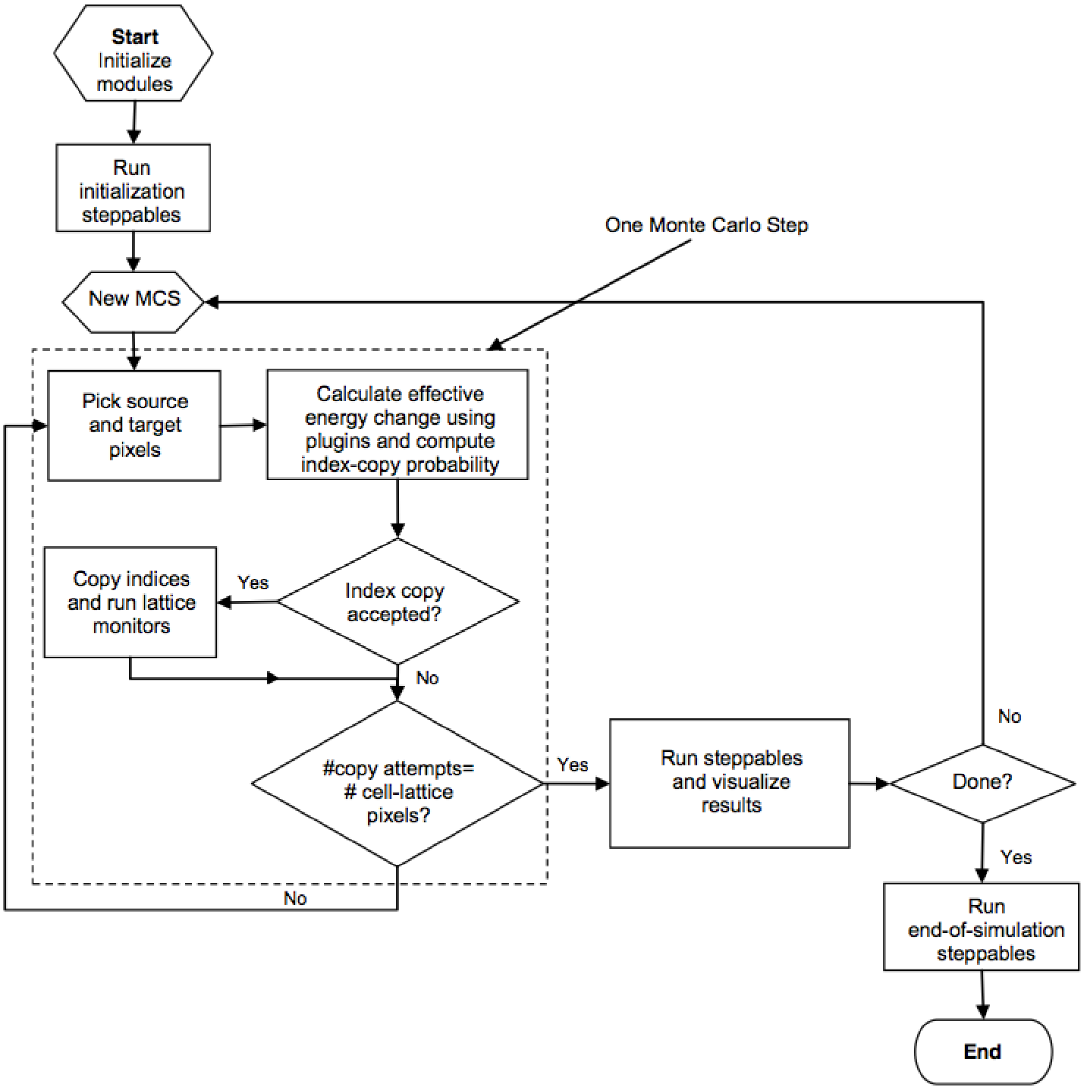

Start
Initialize modules
Run initialization steppables
One Monte Carlo Step
New MCS
Pick source and target pixels
Calculate effective energy change using plugins and compute index-copy probability
Copy indices and run lattice monitors
Yes
Index copy accepted?
No
#copy attempts= # cell-lattice pixels?
Yes
No
Run steppables and visualize results
Done?
No
Yes
Run end-of-simulation steppables
End

Figure 5: Flow chart of the GGH algorithm as implemented in CompuCell3D.

5.1 A Short Introduction to XML

XML is a text-based data-description language, which allows standardised representations of data. XML syntax consists of lists of elements, each either contained between opening (<Tag>) and closing (</Tag>) tags[2]:

```
<Tag Attribute1="text1">ElementText</Tag>
```

[2] In the text, we denote XML, CC3DML and Python code using the Courier font. In listings presenting syntax, user-supplied variables are given in *italics*. Broken-out listings are boxed. Punctuation at the end of boxes is implicit.

or of form:

```
<Tag Attribute1="attribute_text1" Attribute2="attribute_text2"/>
```

We will denote the `<Tag>` ⋯ `</Tag>` syntax as a `<Tag>` tag pair. The opening tag of an XML element may contain additional attributes characterising the element. The content of the XML element (`ElementText` in the above example) and the values of its attributes (`text1`, `attribute_text1`, `attribute_text2`) are strings of characters. Computer programs that read XML may interpret these strings as other data types such as integers, Booleans or floating point numbers. XML elements may be nested. The simple example below defines an element `Cell` with subelements (represented as nested XML elements) `Nucleus` and `Membrane` assigning the element `Nucleus` an attribute `Size` set to "10" and the element `Membrane` an attribute Area set to "20.5", and setting the value of the `Membrane` element to `Expanding`:

```
<Cell>
   <Nucleus Size="10"/>
   <Membrane Area="20.5">Expanding</Membrane>
</Cell>
```

Although XML parsers ignore indentation, all the listings presented in this chapter are block-indented for better readability.

5.2 Grain-Growth Simulation

One of the simplest CompuCell3D simulations mimics crystalline grain growth or annealing. Most simple metals are composed of microcrystals, or grains, each of which has a particular crystalline-lattice orientation. The atoms at the surfaces of these grains have a higher energy than those in the bulk because of their missing neighbors. We can characterise this excess energy as a boundary energy. Atoms in convex regions of a grain's surface have a higher energy than those in concave regions, in particular than those in the concave face of an adjoining grain, because they have more missing neighbors. For this reason, an atom at a convex curved boundary can reduce its energy by hopping across the grain boundary to the concave side (62). The movement of atoms moves the grain boundaries, lowering the net configuration energy through annealing or coarsening, with the net size of grains growing because of grain disappearance. Boundary motion may require thermal activation because atoms in the space between grains may have higher energy than atoms in grains. The effective energy driving grain growth is simply the boundary energy in equation (1).

In CompuCell3D, we can represent grains as generalized cells. CC3DML Listing 1 defines our grain-growth simulation.

Listing 1: CC3DML configuration file for 2D grain-growth simulation.

```
<CompuCell3D>

<Potts>
   <Dimensions x="100" y="100" z="1"/>
   <Steps>10000</Steps>
   <Temperature>5</Temperature>
   <Boundary_y>Periodic</Boundary_y>
   <Boundary_x>Periodic</Boundary_x>
   <NeighborOrder>3</NeighborOrder>
</Potts>

<Plugin Name="CellType">
   <CellType TypeName="Medium" TypeId="0"/>
   <CellType TypeName="Grain" TypeId="1"/>
</Plugin>

<Plugin Name="Contact">
   <Energy Type1="Medium" Type2="Grain">0</Energy>
   <Energy Type1="Grain" Type2="Grain">5</Energy>
   <Energy Type1="Medium" Type2="Medium">0</Energy>
   <NeighborOrder>3</NeighborOrder>
</Plugin>

<Steppable Type="UniformInitializer">
   <Region>
      <BoxMin x="0" y="0" z="0"/>
      <BoxMax x="100" y="100" z="1"/>
      <Gap>0</Gap>
      <Width>5</Width>
      <Types>Grain</Types>
   </Region>
</Steppable>

</CompuCell3D>
```

Each CC3DML configuration file begins with the `<CompuCell3D>` tag and ends with the `</CompuCell3D>` tag. A CC3DML configuration file contains three sections in the following sequence: the *lattice section* (contained within the `<Potts>` tag pair), the *plugins section*, and the *steppables section*. The lattice section defines global parameters for the simulation: cell-lattice and field-lattice dimensions (specified using the syntax `<Dimensions x="x_dim" y="y_dim" z="z_dim"/>`), the number of Monte Carlo Steps to run (defined within the `<Steps>` tag pair) the effective cell motility (defined within the `<Temperature>` tag pair) and boundary conditions. The default boundary conditions are no-flux. However, in this simulation, we have changed them to be periodic along the x and y axes by assigning the value `Periodic` to the `<Boundary_x>` and `<Boundary_y>` tag pairs. The value set by the `<NeighborOrder>` tag pair defines the range over which source pixels are selected for index-copy attempts (see Figure 4 and Table 1).

The plugins section lists the plugins the simulation will use. The syntax for all plugins which require parameter specification is:

```
<Plugin Name=”PluginName”>
   <ParameterSpecification/>
</Plugin>
```

The `CellType` plugin uses the parameter syntax

```
<CellType TypeName=”Name” TypeId=”IntegerNumber”/>
```

to map verbose generalised-cell-type names to numeric cell `TypeIds` for all generalised-cell types. It does not participate directly in index copies, but is used by other plugins for cell-type-to-cell-index mapping. Even though the grain-growth simulation fills the entire cell lattice with cells of type `Grain`, the current implementation of CompuCell3D requires that all simulations define the `Medium` cell type with `TypeId` 0. `Medium` is a special cell type with unconstrained volume and surface area that fills all cell-lattice pixels unoccupied by cells of other types[3]

The `Contact` plugin calculates changes in the boundary energy defined in equation (1) for each index-copy attempt. The parameter syntax for the `Contact` plugin is:

```
<Energy Type1=”TypeName1” Type2=”TypeName1”>EnergyValue</Energy>
```

where `TypeName1` and `TypeName2` are the names of the cell types and `EnergyValue` is the boundary-energy coefficient, `J(TypeName1,TypeName2)`, between cells of `TypeName1` and `TypeName2` (see equation (1)). The `<NeighborOrder>` tag pair specifies the interaction range of the boundary energy. The default value of this parameter is 1.

The steppables section includes only the `UniformInitializer` steppable. All steppables have the following syntax:

```
<Steppable Type=”SteppableName” Frequency=”FrequencyMCS”>
   <ParameterSpecification/>
</Steppable>
```

The `Frequency` attribute is optional and by default is 1 MCS.

CompuCell3D simulations require specification of initial condition. The simplest way to define the initial cell lattice is to use the built-in initializer steppables, which construct

[3] We highlight in yellow sections or text describing CompuCell3D behaviours which may be confusing or lead to hard-to-track errors.

simple regions filled with generalised cells.

The `UniformInitializer` steppable in the grain-growth simulation defines one or more rectangular (parallelepiped in 3D) regions filled with generalised cells of user selectable types and sizes. We enclose each region definition within a `<Region>` tag pair. We use the `<BoxMin>` and `<BoxMax>` tags to describe the boundaries of the region, The `<Width>` tag pair defines the size of the square (cubical in 3D) generalised cells and the `<Gap>` tag pair creates space between neighboring cells. The `<Types>` tag pair lists the types of generalised cells. The grain-growth simulation uses only one cell type, `Grain`, but we can also initialise cells using types randomly chosen from the list, as in Listing 2.

Listing 2: CC3DML code excerpt using the `UniformInitializer` steppable to initialize a rectangular region filled with 5×5 pixel generalised cells with randomly-assigned cell types (either `Condensing` or `NonCondensing`).

```
<Steppable Type="UniformInitializer">
    <Region>
        <BoxMin x="0" y="0" z="0"/>
        <BoxMax x="100" y="100" z="1"/>
        <Gap>0</Gap>
        <Width>5</Width>
        <Types>Grain</Types>
    </Region>
</Steppable>
```

The coordinate values in `BoxMax` element must be one more than the coordinates of the corresponding corner of the region to be filled. So to fill a square of side 10 beginning with pixel location $(5, 5)$ we use the following region-boundary specification:

```
<BoxMin x="5" y="5" z="0"/>
<BoxMax x="16" y="16" z="1"/>
```

Listing the same type multiple times results in a proportionally higher fraction of generalised cells of that type. For example,

```
<Types>Condensing,Condensing,NonCondensing</Types>
```

will allocate approximately 2/3 of the generalised cells to type `Condensing` and 1/3 to type `NonCondensing`. `UniformInitializer` allows specification of multiple regions. Each region is independent and can have its own cell sizes, types and cell spacing. If the regions overlap, later-specified regions overwrite earlier-specified ones. If region specification does not cover the entire lattice, uninitialised pixels have type `Medium`.

Figure 6 shows sample output generated by the grain-growth simulation.

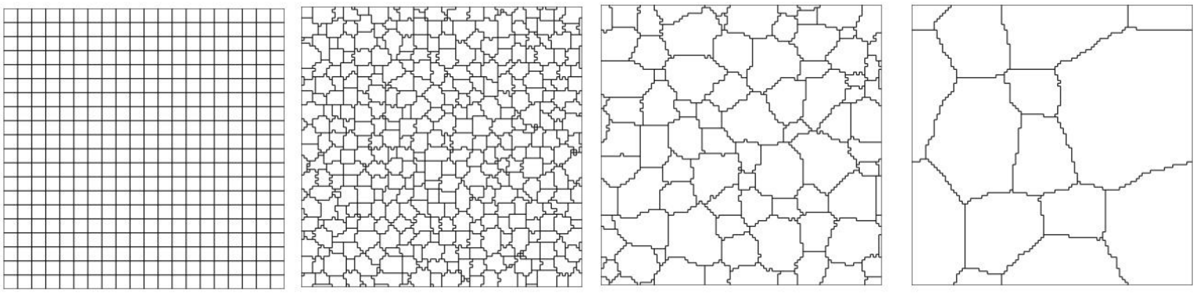


Figure 6: Snapshots of the cell-lattice configuration for the grain-growth simulation on a 100 × 100 pixel 3rd-neighbor square lattice, as specified in Listing 1. Boundary conditions are periodic in both directions.

One advantage of GGH simulations compared to FE simulations is that going from 2D to 3D is easy. To run a 3D grain-growth simulation on a 100 × 100 × 100 lattice we only need to make the following replacements in Listing 1:

For 2D

```
<Dimensions x=”100” y=”100” z=”1”/>
```

for 3D

```
<Dimensions x=”100” y=”100” z=”100”/>
```

and for 2D

```
<BoxMax x=”100” y=”100” z=”1”/>
```

for 3D

```
<BoxMax x=”100” y=”100” z=”100”/>
```

Grain growth simulations are particularly sensitive to lattice anisotropy, so running them on lower-anisotropy lattices is desirable. Longer-range lattices are less anisotropic but cause simulations to run slower. Fortunately a hexagonal lattice of a given range is less anisotropic than a square lattice of the same range. To run the grain-growth simulation on a hexagonal lattice, we add `<LatticeType>Hexagonal</LatticeType>` to the lattice section in Listing 1 and change the two occurrences of:

```
<NeighborOrder>3</NeighborOrder>
```

to

```
<NeighborOrder>1</NeighborOrder>
```

Figure 7 shows snapshots for this simulation.

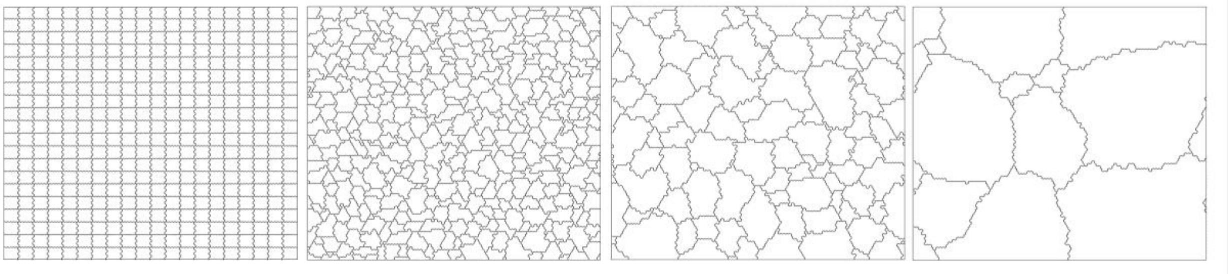


Figure 7: Snapshots of the cell-lattice configuration for the grain-growth simulation on a 100 × 100 pixel 1^{st}-neighbor hexagonal lattice as specified in Listing 1 with substitutions described in the text. The x and y length units in an hexagonal lattice differ, resulting in differing x and y dimensions for a cell lattice with an equal number of pixels in the x and y directions.

One inconvenience of the current implementation of CompuCell3D is that it does not automatically rescale parameter values when interaction range, lattice dimensionality or lattice type change. When changing these attributes, users must recalculate parameters to keep the underlying physics of the simulation the same.

CompuCell3D dramatically reduces the amount of code necessary to build and run a simulation. The grain-growth simulation took about 25 lines of CC3DML instead of 1000 lines of C, C++ or Fortran.

5.3 Cell-Sorting Simulation

Cell sorting is an experimentally-observed phenomenon in which cells with different adhesivities are randomly mixed and re-aggregated. They can spontaneously sort to reestablish coherent homogenous domains (93, 94). Sorting is a key mechanism in embryonic development.

The grain-growth simulation used only one type of generalised cell. Simulating sorting of two types of biological cell in an aggregate floating in solution is slightly more complex. Listing 3 shows a simple cell-sorting simulation. It is similar to Listing 1 with a few additional modules (shown in **bold**). The effective energy is that in equation (6).

Listing 3: CC3DML configuration file simulating cell sorting between `Condensing` and `NonCondensing` cell types. Highlighted text indicates modules absent in Listing 1. Notice how little modification of the grain-growth CC3DML configuration file this simulation requires.

```
<CompuCell3D>
```

```
<Potts>
   <Dimensions x="100" y="100" z="1"/>
   <Steps>10000</Steps>
   <Temperature>10</Temperature>
   <NeighborOrder>2</NeighborOrder>
</Potts>

<Plugin Name="Volume">
   <TargetVolume>25</TargetVolume>
   <LambdaVolume>2.0</LambdaVolume>
</Plugin>

<Plugin Name="CellType">
   <CellType TypeName="Medium" TypeId="0"/>
   <CellType TypeName="Condensing" TypeId="1"/>
   <CellType TypeName="NonCondensing" TypeId="2"/>
</Plugin>

<Plugin Name="Contact">
   <Energy Type1="Medium" Type2="Medium">0</Energy>
   <Energy Type1="NonCondensing" Type2="NonCondensing">16</Energy>
   <Energy Type1="Condensing" Type2="Condensing">2</Energy>
   <Energy Type1="NonCondensing" Type2="Condensing">11</Energy>
   <Energy Type1="NonCondensing" Type2="Medium">16</Energy>
   <Energy Type1="Condensing" Type2="Medium">16</Energy>
   <NeighborOrder>2</NeighborOrder>
</Plugin>

<Steppable Type="BlobInitializer">
   <Region>
      <Gap>0</Gap>
      <Width>5</Width>
      <Radius>40</Radius>
      <Center x="50" y="50" z="0"/>
      <Types>Condensing,NonCondensing</Types>
   </Region>
</Steppable>

</CompuCell3D>
```

The main change from Listing 1 to the lattice section is that we omit the boundary condition specification and use default no-flux boundary conditions.

In the `CellType` plugin we introduce the two cell types, `Condensing` and `NonCondensing`, in place of `Grain`. In addition we do not the fill lattice completely with `Condensing` and `NonCondensing` cells so the interactions with `Medium` become important. The boundary-energy matrix in the `Contact` plugin thus requires entries for the additional cell-type pairs.

The hierarchy of boundary energies listed results in cell sorting.

We also add the Volume plugin, which calculates the volume-constraint energy as given in equation (4). In this plugin the `<TargetVolume>` tag pair sets target volume $V_t = 25$ for both `Condensing` cells and `NonCondensing` and the `<LambdaVolume>` tag pair sets the constraint strength $\lambda_{\text{vol}} = 2.0$ for both cell types. We will see later how to define volume-constraint parameters for each cell type or each cell individually.

In the cell-sorting simulation we initialise the cell lattice using the `BlobInitializer` steppable which specifies circular (or spherical in 3D) regions filled with square (or cubical in 3D) cells of user-defined size and types. The syntax is very similar to that for `UniformInitializer`.

Looking in detail at the syntax of BlobInitializer in Listing 3, the `<Radius>` tag pair defines the radius of a circular (or spherical) domain of cells in pixels. The `<Center>` tag, with syntax `<Center x="x_position" y="y_position" z="z_position"/>`, defines the coordinates of the centre of the domain. The remaining tags are the same as for `UniformInitializer`. As with `UniformInitializer`, we can define multiple regions. We can use both `UniformInitializer` and `BlobInitializer` in the same simulation. In the case of overlap, later-specified regions overwrite earlier ones.

We show snapshots of the cell-sorting simulation in Figure 8. The less cohesive `NonCondensing` cells engulf the more cohesive `Condensing` cells, which cluster and form a single central domain. By changing the boundary energies we can produce other cell-sorting patterns (95, 96).

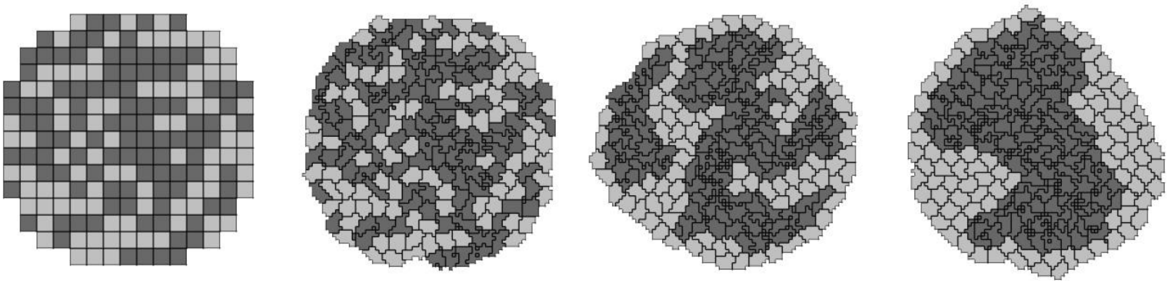


Figure 8: Snapshots of the cell-lattice configurations for the cell-sorting simulation in Listing 3. The boundary-energy hierarchy drives `NonCondensing` (light grey) cells to surround `Condensing` (dark grey) cells. The white background denotes surrounding `Medium`.

5.4 Bacterium-and-Macrophage Simulation

In the two simulations we have presented so far, the cellular pattern develops without fields. Often, however, biological patterning mechanisms require us to introduce and evolve chemical fields and to have cells behaviours depend on the fields. To illustrate the use of fields, we model the in vitro behaviour of bacteria and macrophages in blood. In the famous experimental movie taken in the 1950s by David Rogers at Vanderbilt University, the macrophage appears to chase the bacterium, which seems to run away from the macrophage. We can model both behaviours using cell secretion of diffusible chemical signals and movement of the cells in response to the chemical (*chemotaxis*): the bacterium secretes a signal (a *chemoattractant*) that attracts the macrophage and the macrophage secretes a signal (a *chemo-repellant*) which repels the bacterium (97).

Listing 4 shows the CC3DML configuration file for the bacterium-and-macrophage simulation.

Listing 4: CC3DML configuration file for the bacterium-and-macrophage simulation.

```
<CompuCell3D>

<Potts>
   <Dimensions x=”100” y=”100” z=”1”/>
   <Steps>100000</Steps>
   <Temperature>20</Temperature>
   <LatticeType>Hexagonal</LatticeType>
</Potts>

<Plugin Name=”CellType”>
   <CellType TypeName=”Medium” TypeId=”0”/>
   <CellType TypeName=”Bacterium” TypeId=”1” />
   <CellType TypeName=”Macrophage” TypeId=”2”/>
   <CellType TypeName=”Red” TypeId=”3”/>
   <CellType TypeName=”Wall” TypeId=”4” Freeze=””/>
</Plugin>

<Plugin Name=”VolumeFlex”>
   <VolumeEnergyParameters CellType=”Macrophage” TargetVolume=”150”
      LambdaVolume=”15”/>
   <VolumeEnergyParameters CellType=”Bacterium” TargetVolume=”10”
      LambdaVolume=”60”/>
   <VolumeEnergyParameters CellType=”Red” TargetVolume=”100”
      LambdaVolume=”30”/>
</Plugin>

<Plugin Name=”SurfaceFlex”>
   <SurfaceEnergyParameters CellType=”Macrophage” TargetSurface=”50”
      LambdaSurface=”30”/>
   <SurfaceEnergyParameters CellType=”Bacterium” TargetSurface=”10”
      LambdaSurface=”4”/>
   <SurfaceEnergyParameters CellType=”Red” TargetSurface=”40”
      LambdaSurface=”0”/>
```

```
</Plugin>

<Plugin Name=”Contact”>
  <Energy Type1=”Medium” Type2=”Medium”>0</Energy>
  <Energy Type1=”Macrophage” Type2=”Macrophage”>150</Energy>
  <Energy Type1=”Macrophage” Type2=”Medium”>8</Energy>
  <Energy Type1=”Bacterium” Type2=”Bacterium”>150</Energy>
  <Energy Type1=”Bacterium” Type2=”Macrophage”>15</Energy>
  <Energy Type1=”Bacterium” Type2=”Medium”>8</Energy>
  <Energy Type1=”Wall” Type2=”Wall”>0</Energy>
  <Energy Type1=”Wall” Type2=”Medium”>0</Energy>
  <Energy Type1=”Wall” Type2=”Bacterium”>150</Energy>
  <Energy Type1=”Wall” Type2=”Macrophage”>150</Energy>
  <Energy Type1=”Wall” Type2=”Red”>150</Energy>
  <Energy Type1=”Red” Type2=”Red”>150</Energy>
  <Energy Type1=”Red” Type2=”Medium”>30</Energy>
  <Energy Type1=”Red” Type2=”Bacterium”>150</Energy>
  <Energy Type1=”Red” Type2=”Macrophage”>150</Energy>
  <NeighborOrder>2</NeighborOrder>
</Plugin>

<Plugin Name=”Chemotaxis”>
  <ChemicalField Source=”FlexibleDiffusionSolverFE” Name=”ATTR”>
    <ChemotaxisByType Type=”Macrophage” Lambda=”1”/>
  </ChemicalField>

  <ChemicalField Source=”FlexibleDiffusionSolverFE” Name=”REP”>
    <ChemotaxisByType Type=”Bacterium” Lambda=”−0.1”/>
  </ChemicalField>
</Plugin>

<Steppable Type=”FlexibleDiffusionSolverFE”>
  <DiffusionField>
    <DiffusionData>
      <FieldName>ATTR</FieldName>
      <DiffusionConstant>0.10</DiffusionConstant>
      <DecayConstant>0.00005</DecayConstant>
      <DoNotDiffuseTo>Wall</DoNotDiffuseTo>
      <DoNotDiffuseTo>Red</DoNotDiffuseTo>
    </DiffusionData>

    <SecretionData>
      <Secretion Type=”Bacterium”>200</Secretion>
    </SecretionData>
  </DiffusionField>

  <DiffusionField>
    <DiffusionData>
      <FieldName>REP</FieldName>
```

```
         <DiffusionConstant>0.10</DiffusionConstant>
         <DecayConstant>0.001</DecayConstant>
         <DoNotDiffuseTo>Wall</DoNotDiffuseTo>
         <DoNotDiffuseTo>Red</DoNotDiffuseTo>
      </DiffusionData>

      <SecretionData>
         <Secretion Type="Macrophage">200</Secretion>
      </SecretionData>
   </DiffusionField>
</Steppable>

<Steppable Type="PIFInitializer">
   <PIFName>bacterium_macrophage_2D_wall_v3.pif</PIFName>
</Steppable>

</CompuCell3D>
```

The simulation has five generalised-cell types: `Medium`, `Bacterium`, `Macrophage`, `Red` blood cells and a surrounding `Wall`. It also has two diffusible fields, representing a chemoattractant, `ATTR`, and a chemorepellent, `REP`. Because the default boundary energy between any generalised-cell type and the edge of the cell lattice is zero, we define a surrounding wall to prevent cells from sticking to the cell-lattice boundary. As in our previous simulations, we assign cell types using the `CellType` plugin. Note the new syntax in the line specifying the cell type making up the walls:

```
<CellType TypeName="Wall" TypeId="4" Freeze=""/>
```

The `Freeze=""` attribute excludes generalised cells of type `Wall` from participating in index copies, which makes the walls immobile.

We replace the `Volume` plugin with `VolumeFlex` and add the plugin `SurfaceFlex`. These plugins allow independent assignment of target values and constraint strengths in the volume-constraint and surface-constraint energies (equations (4) and (5)). These plugins require a line for each generalised-cell type, specifying the type name and the target volume (or target surface area), and λ_{vol} (or λ_{surf}) for that generalised-cell type, *e.g.*:

```
<VolumeEnergyParameters CellType="Name" TargetVolume="Value"
LambdaVolume="Value"/>
```

We implement the actual bacterium-macrophage "chasing" mechanism using the `Chemotaxis` plugin, which specifies how a generalised cell of a given type responds to a field. The `Chemotaxis` plugin biases a cells motion up or down a field gradient by changing the calculated effective-energy change used in the acceptance function, equation (7). For a field

$c(\mathbf{i})$:

$$\Delta H_{\text{chem}} = -\lambda(c(\mathbf{i}) - c(\mathbf{j}))\,, \tag{9}$$

where $c(\mathbf{i})$ is the chemical field at the index-copy target pixel, $c(\mathbf{j})$ the field at the index-copy source pixel, λ_{chem} the strength and direction of chemotaxis. If $\lambda_{\text{chem}} > 0$ and $c(\mathbf{i}) > c(\mathbf{j})$, then ΔH_{chem} is negative, increasing the probability of accepting the index copy in equation (7). The net effect is that the cell moves up the field gradient with a velocity $\sim \lambda_{\text{chem}}\bar{\nabla}c$. If $\lambda < 0$ is negative, the opposite occurs, and the cell will move down the field gradient. Plugins with more sophisticated ΔH_{chem} calculations (*e.g.*, including response saturation) are available within CompuCell3D (see the *CompuCell3D User Guide*).

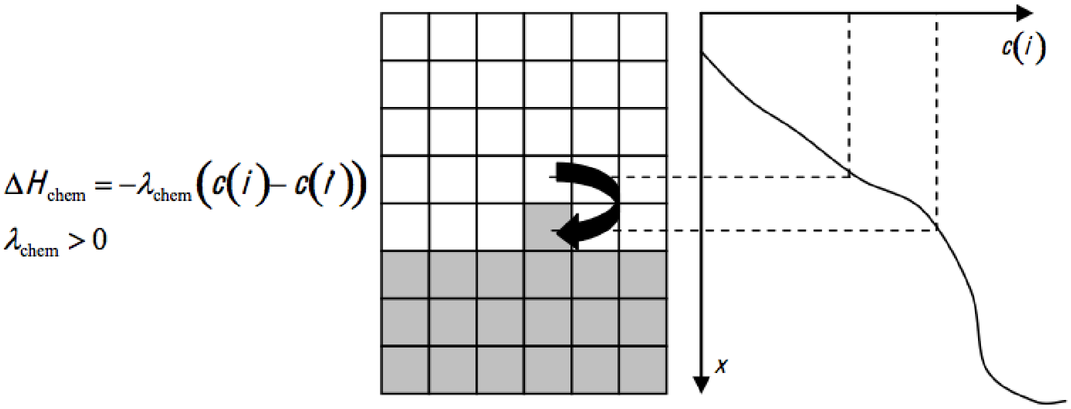

$\Delta H_{\text{chem}} = -\lambda_{\text{chem}}(c(i) - c(j))$
$\lambda_{\text{chem}} > 0$
$c(i)$
x

Figure 9: Connecting a field to GGH dynamics using a chemotaxis-energy term. The difference in the value of the field c at the source, $\mathbf{j}$, and target, $\mathbf{i}$, pixels changes the ΔH of the index-copy attempt. Here $c(\mathbf{i}) > c(\mathbf{j})$ and $\lambda > 0$, so $\Delta H_{\text{chem}} < 0$, increasing the probability of accepting the index-copy attempt in equation (7).

In the `Chemotaxis` plugin we must identify the names of the fields, where the field information is stored, the list of the generalised-cell types that will respond to the fields, and the strength and direction of the response (`Lambda` $= \lambda_{\text{chem}}$). The information for each field is specified using the syntax:

```
<ChemicalField Source="where_field_is_stored" Name="field_name">
  <ChemotaxisByType Type="cell_type1" Lambda="lambda1"/>
  <ChemotaxisByType Type="cell_type2" Lambda="lambda1"/>
</ChemicalField>
```

In our current example, the first field, named `ATTR`, is stored in `FlexibleDiffusionSolverFE`. `Macrophage` cells are attracted to `ATTR` with $\lambda_{\text{chem}} = 1$. None of the other cell types responds to `ATTR`. Similarly, `Bacterium` cells are repelled by `REP` with $\lambda_{\text{chem}} = -0.1$.

Keep in mind that fields are not created within the `Chemotaxis` plugin, which only specifies how different cell types respond to the fields. We define and store the fields elsewhere. Here, we use the `FlexibeDiffusionSolverFE` steppable as the source of our fields. The

`FlexibleDiffusionSolverFE` steppable is the main CompuCell3D tool for defining diffusing fields, which evolve according to the diffusion equation:

$$\frac{\partial c(\mathbf{i})}{\partial t} = D(\mathbf{i})\nabla^2 c(\mathbf{i}) - k(\mathbf{i})c(\mathbf{i}) + s(\mathbf{i})\,, \tag{10}$$

where $c(\mathbf{i})$ is the field concentration and $D(\mathbf{i}), k(\mathbf{i})$, and $s(\mathbf{i})$ denote the diffusion constant (in m^2/s), decay constant (in s^{-1}) and secretion rates (in concentration/s) of the field, respectively. $D(\mathbf{i}), k(\mathbf{i})$, and $s(\mathbf{i})$ may vary with position and cell-lattice configuration.

As in the `Chemotaxis` plugin, we may define the behaviours of multiple fields, enclosing each one within `<DiffusionField>` tag pairs. For each field defined within a `<DiffusionData>` tag pair, users provide values for the name of the field (using the `<FieldName>` tag pair), the diffusion constant (using the `<DiffusionConstant>` tag pair) , and the decay constant (using the `<DiffusionConstant>` tag pair). Forward-Euler methods are numerically unstable for large diffusion constants, limiting the maximum nominal diffusion constant allowed in CompuCell3D simulations. However, by increasing the PDE-solver calling frequency, which reduces the effective time step, CompuCell3D can simulate arbitrarily large diffusion constants. For more information, see the *CompuCell3D User Guide*.

Each optional `<DoNotDiffuseTo>` tag pair, with syntax:

```
<DoNotDiffuseTo>cell_type</DoNotDiffuseTo>
```

prevents the field from diffusing into field-lattice pixels where the corresponding cell-lattice pixel, $\mathbf{i}$, is occupied by a cell, $\sigma(\mathbf{i})$, of the specified type. In our case, chemical fields do not diffuse into the pixels occupied by `Wall` or `Red` cells. The optional `<SecretionData>` tag pair defines a subsection which identifies cells types that secrete or absorb the field and the rates of secretion:

```
<SecretionData>
   <Secretion Type="cell_type1">real_rate1</Secretion>
   <Secretion Type="cell_type2">real_rate2</Secretion>
<SecretionData>
```

A negative rate simulates absorption. In the bacterium and macrophage simulation, `Bacterium` cells secrete `ATTR` and `Macrophage` cells secrete `REP`.

We load the initial configuration for the bacterium-and-macrophage simulation using the `PIFInitializer` steppable. Many simulations require initial generalised-cell configurations that we cannot easily construct from primitive regions filled with cells using `BlobInitializer` and `UniformInitializer`. To allow maximum flexibility, CompuCell3D can read the initial cell-lattice configuration from *Pixel Initialization Files* (PIFs). A PIF is a text file that allows users to assign multiple rectangular (parallelepiped in 3D) pixel regions or single pixels to particular cells.

Each line in a PIF has the syntax:

```
Cell_id Cell_type x_low x_high y_low y_high z_low z_high
```

where `Cell_id` is a unique cell index. A PIF may have multiple, possibly non-adjacent, lines starting with the same `Cell_id`; all lines with the same `Cell_id` define pixels of the same generalised cell. The values `x_low, x_high, y_low, y_high, z_low` and `z_high` define rectangles (parallelepipeds in 3D) of pixels belonging to the cell. In the case of overlapping pixels, a later line overwrites pixels defined by earlier lines. The following line describes a 6 × 6-pixel square cell with cell index 0 and type `Amoeba`:

```
0 Amoeba 10 15 10 15 0 0
```

If we save this line to the file 'amoebae.pif', we can load it into a simulation using the following syntax:

```
<Steppable Type="PIFInitializer">
   <PIFName>amoebae.pif</PIFName>
</Steppable>
```

Listing 5 illustrates how to construct arbitrary shapes using a PIF. Here we define two cells with indices 0 and 1, and cell types `Amoeba` and `Bacterium`, respectively. The main body of each cell is a 6 × 6 square to which we attach additional pixels.

Listing 5: Simple PIF initialising two cells.

```
0 Amoeba 10 15 10 15 0 0
1 Bacterium 25 30 25 30 0 0
0 Amoeba 16 16 15 15 0 0
1 Bacterium 25 27 31 35 0 0
```

All lines with the same cell index (first column) define a single cell.

Figure 10 shows the initial cell-lattice configuration specified in Listing 5:

In practice, because constructing complex PIFs by hand is cumbersome, we generally use custom-written scripts to generate the file directly, or convert images stored in graphical formats (*e.g.*, gif, jpeg, png) from experiments or other programs.

Listing 6 shows the PIF for the bacterium-and-macrophage simulation.

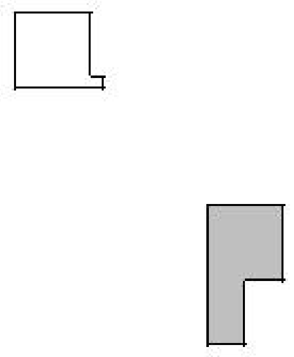


Figure 10: Initial configuration of the cell lattice based on the PIF in Listing 5.

Listing 6: PIF defining the initial cell-lattice configuration for the bacterium-and-macrophage simulation. The file is stored as 'bacterium_macrophage_2D_wall_v3.pif'.

```
0 Red 10 20 10 20 0 0
1 Red 10 20 40 50 0 0
2 Red 10 20 70 80 0 0
3 Red 40 50 0 10 0 0
4 Red 40 50 30 40 0 0
5 Red 40 50 60 70 0 0
6 Red 40 50 90 95 0 0
7 Red 70 80 10 20 0 0
8 Red 70 80 40 50 0 0
9 Red 70 80 70 80 0 0
10 Wall 0 99 0 1 0 0
10 Wall 98 99 0 99 0 0
10 Wall 0 99 98 99 0 0
10 Wall 0 1 0 99 0 0
11 Bacterium 5 5 5 5 0 0
12 Macrophage 35 35 35 35 0 0
13 Bacterium 65 65 65 65 0 0
14 Bacterium 65 65 5 5 0 0
15 Bacterium 5 5 65 65 0 0
16 Macrophage 75 75 95 95 0 0
17 Red 24 28 10 20 0 0
18 Red 24 28 40 50 0 0
19 Red 24 28 70 80 0 0
20 Red 40 50 14 20 0 0
21 Red 40 50 44 50 0 0
22 Red 40 50 74 80 0 0
23 Red 54 59 90 95 0 0
24 Red 70 80 24 28 0 0
```

```
25  Red  70  80  54  59  0  0
26  Red  70  80  84  90  0  0
27  Macrophage  10  10  95  95  0  0
```

In Listing 4 we read the cell lattice configuration from the file 'bacterium_macrophage_2D_wall_v3.pif' using the lines:

```
<Steppable Type="PIFInitializer">
   <PIFName>bacterium_macrophage_2D_wall_v3.pif</PIFName>
</Steppable>
```

Figure 11 shows snapshots of the bacterium-and-macrophage simulation. By adjusting the properties and number of bacteria, macrophages and red blood cells and the diffusion properties of the chemical fields, we can build a surprisingly good reproduction of the experiment.

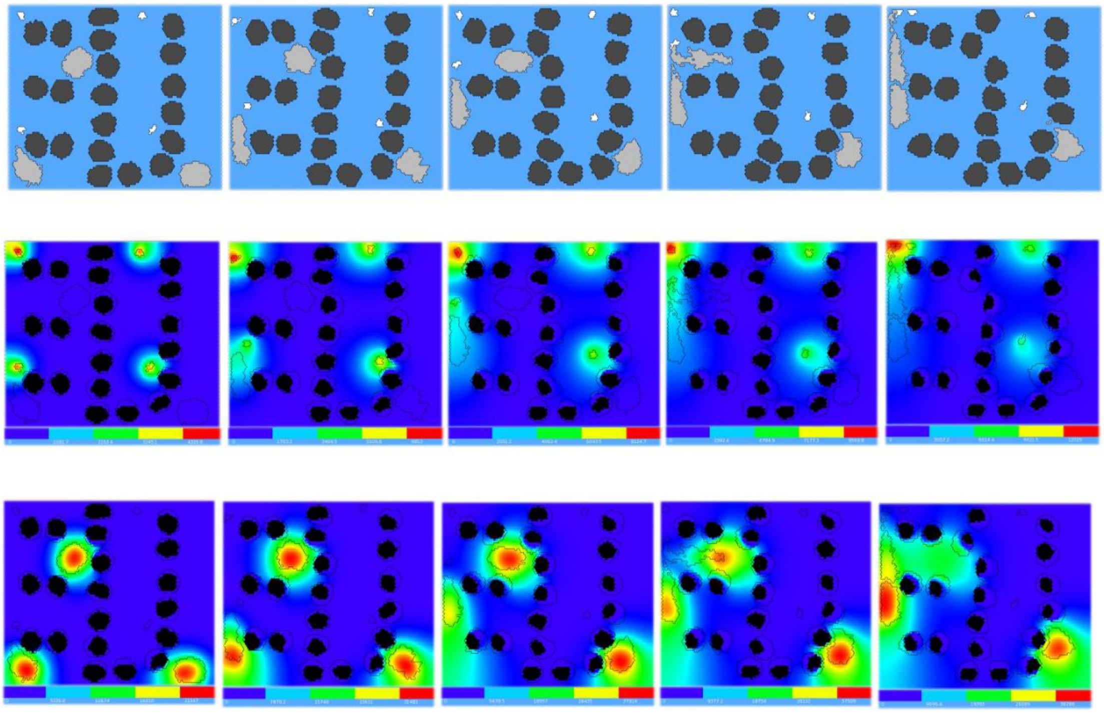


Figure 11: Snapshots of the bacterium-and-macrophage simulation from Listing 4 and the PIF in Listing 6 saved in the file 'bacterium_macrophage_2D_wall_v3.pif'. The upper row shows the cell-lattice configuration with the `Macrophages` in grey, `Bacteria` in white, red blood cells in dark grey and Medium in blue. Middle row shows the concentration of the chemoattractant ATTR secreted by the `Bacteria`. The bottom row shows the concentration of the chemorepellant REPL secreted by the `Macrophages`. The bars at the bottom of the field images show the concentration scales (blue, low concentration, red, high concentration).

6 Python Scripting

CC3DML is convenient for building simple simulations such as those we presented above. To describe more complex simulations, CompuCell3D allows users to write specialized, shareable modules in C/C++ (through the *CompuCell3D Application Programming Interface*, or CC3D API) or Python (through a Python-scripting interface). C and C++ modules have the advantage that they run at native speed. However, developing them requires knowledge of both C/C++ and the CC3D API, and their integration with CompuCell3D requires recompilation of the source code. Python module development is less complicated, since Python has simpler syntax than C/C++ and users can modify and extend a library of Python-module templates included with CompuCell3D. Moreover, Python modules do not require recompilation.

Tasks performed by CompuCell3D modules either relate to index-copy attempts (plugins) or run either once, at the beginning or end of a simulation, or once every several MCS (steppables). Tasks run every index-copy attempt, like effective-energy-term calculations, are the most frequently-called tasks in a GGH simulation and writing them in Python may slow simulations. However, steppables and lattice monitors are good candidates for Python implementation and cause negligible performance degradation. Python implementations are suitable for most cell-parameter adjustments that depend on the state of the simulation, *e.g.*, simulating cell growth in response to a chemical, cell-type differentiation and changes in cell-cell adhesion.

6.1 A Short Introduction to Python

Python is an object-oriented scripting language with all the syntactic constructs present in any modern programming language. Python supports popular flow-control statements such as `if-elif-else` conditional instructions and `for` and `while` loops. Unlike C/C++, Python does not use ’ ’ to end lines or ’{’ and ’}’ to define code blocks. Instead, Python relies on indentation to define blocks of code. `if` statements, `for` or `while` loops and their subsections are created by a ’:’ and increasing the level of indentation. The end of a block is indicated by a decrease in the level of indentation. Python uses the ’=’ operator for assignments and ’==’ for checking equality between objects. For example, in the following code:

```
b=2
if b==2:
    a=10
    for c in range(0,a):
        b=a+c
        print b
```

we indent the body of the `if` statement and the body of the inner `for` loop. The `for` loop is executed inside the `if` statement. `a=0` assigns the variable `a` a value of 10, while `b==2` is true if `b` has a value of 2. The `for` loop assigns the variable `c` values 0 through `a-1` and executes instructions inside the loop body.

As an object-oriented language, Python supports *classes, inheritance* and *polymorphism*. Accessing *members* of *objects* uses the '.' operator. For example, to access the real part of a complex number, we use the following code:

```
a=complex(2,3)
a=1.5+0.5j
print a.real
```

Here, `real` is a member of the Python class `complex`, which represents complex numbers. If the object has composite subobjects, we use the '.' operator recursively:

```
object.subobject.member_of_subobject
```

Users may define Python objects without declaring their type. A single data structure such as a list or dictionary can store objects of multiple types. Python provides automatic memory management, which frees users from remembering to deallocate memory for objects that are no longer used.

Long source code lines can be carried over to the following line using the '\' character:

```
very_long_variable_name = \
very_long_variable_name * very_important_constant
```

Note that double underscore '__' has a reserved meaning in Python and should not be confused with a single underscore '_'.

We will present additional Python features in the subsequent sections and explain step-by-step some basic concepts of Python programming (for more on Python, see *Learning Python*, by Mark Lutz (98)). For more information on Python scripting in CompuCell3D, see our *Python Tutorials* and *CompuCell3D User Guide* (available from the CompuCell3D website, www.compucell3d.org).

6.2 Building Python-Based CompuCell3D Simulations

Python scripting allows users to augment their CC3DML configuration files with Python scripts or to code their entire simulations in Python (in which case the Python script looks very similar to the CC3DML script it replaces). Listing 7 shows the standard block of template code for running a Python script in conjunction with a CC3DML configuration file.

Listing 7: Basic Python template to run a CompuCell3D simulation through a Python interpreter. Later examples will be based on this script.

```
import sys
from os import environ
from os import getcwd
import string
sys.path.append(environ[”PYTHON_MODULE_PATH”])
import CompuCellSetup

sim,simthread = CompuCellSetup.getCoreSimulationObjects()

#Create extra player fields here or add attributes
CompuCellSetup.initializeSimulationObjects(sim,simthread)

#Add Python steppables here
steppableRegistry=CompuCellSetup.getSteppableRegistry()
CompuCellSetup.mainLoop(sim,simthread,steppableRegistry)
```

The `import sys` line provides access to standard functions and variables needed to manipulate the Python runtime environment. The next two lines,

```
from os import environ
from os import getcwd
```

import `environ` and `getcwd` housekeeping functions into the current *namespace* (*i.e.*, current script) and are included in all our Python programs. In the next three lines,

```
import string
sys.path.append(environ[”PYTHON_MODULE_PATH”])
import CompuCellSetup
```

we import the `string` module, which contains convenience functions for performing operations on strings of characters, set the search path for Python modules and import the `CompuCellSetup` module, which provides a set of convenience functions that simplify initialisation of CompuCell3D simulations.

Next, we create and initialize the core CompuCell3D modules:

```
sim,simthread = CompuCellSetup.getCoreSimulationObjects()
CompuCellSetup.initializeSimulationObjects(sim,simthread)
```

We then create a steppable registry (a Python *container* that stores steppables, *i.e.*, a list of all steppables that the Python code can access) and pass it to the function that runs the simulation:

```
steppableRegistry=CompuCellSetup.getSteppableRegistry()
CompuCellSetup.mainLoop(sim,simthread,steppableRegistry)
```

In the next section, we extend this template to build a simple simulation.

6.3 Cell-Type-Oscillator Simulation

Suppose that we would like to add a caricature of oscillatory gene expression to our cell-sorting simulation (Listing 3) so that cells exchange types every 100 MCS. We will implement the changes of cell types using a Python steppable, since it occurs at intervals of 100 MCS.

Listing 8 shows the changes to the Python template in Listing 7 that are necessary to create the desired type switching (changes are shown in **bold**).

Listing 8: Python script expanding the template code in Listing 7 into a simple `TypeSwitcherSteppable` steppable. The code illustrates dynamic modification of cell parameters using a Python script. Lines added to Listing 7 are shown in bold.

```
import sys
from os import environ
from os import getcwd
import string

sys.path.append(environ["PYTHON_MODULE_PATH"])

import CompuCellSetup
sim,simthread = CompuCellSetup.getCoreSimulationObjects()

from PySteppables import *

class TypeSwitcherSteppable(SteppablePy):
    def __init__(self,_simulator,_frequency=100):
        SteppablePy.__init__(self,_frequency)
        self.simulator=_simulator
        self.inventory=self.simulator.getPotts().getCellInventory()
        self.cellList=CellList(self.inventory)

    def step(self,mcs):
        for cell in self.cellList:
            if cell.type==1:
                cell.type=2
            elif (cell.type==2):
                cell.type=1
            else:
                print "Unknown type. In cellsort simulation\
                there should only be two types 1 and 2"

#Create extra player fields here or add attributes
CompuCellSetup.initializeSimulationObjects(sim,simthread)

#Add Python steppables here
steppableRegistry=CompuCellSetup.getSteppableRegistry()
typeSwitcherSteppable=TypeSwitcherSteppable(sim,100);
steppableRegistry.registerSteppable(typeSwitcherSteppable)

CompuCellSetup.mainLoop(sim,simthread,steppableRegistry)
```

A CompuCell3D steppable is a class (a type of object) that holds the parameters and functions necessary for carrying out a task. Every steppable defines at least 4 functions: `__init__(self, _simulator, _frequency)`, `start(self)`, `step(self, mcs)` and `finish(self)`.

CompuCell3D calls the `start(self)` function once at the beginning of the simulation before any index-copy attempts. It calls the `step(self, mcs)` function periodically after every `_frequency` MCS. It calls the `finish(self)` function once at the end of the simulation. Listing 8 does not have explicit `start(self)` or `finish(self)` functions. Instead, the class definition :

```
class TypeSwitcherSteppable(SteppablePy):
```

causes the `TypeSwitcherSteppable` to inherit components of the `SteppablePy` class. `SteppablePy` contains default definitions of the `start(self)`, `step(self,mcs)` and `finish(self)` functions. Inheritance reduces the length of the user-written Python code and ensures that the `TypeSwitcherSteppable` object has all needed components. The line:

```
from PySteppables import *
```

makes the content of 'PySteppables.py' file (or module) available in the current namespace. The `PySteppables` module includes the `SteppablePy` base class.

The `__init__` function is a constructor that accepts user-defined parameters and initialises a steppable object. Consider the `__init__` function of the `TypeSwitcherSteppable`:

```
def __init__(self,_simulator,_frequency=100):
    SteppablePy.__init__(self,_frequency)
    self.simulator=_simulator
    self.inventory=self.simulator.getPotts().getCellInventory()
    self.cellList=CellList(self.inventory)
```

In the def line, we pass the necessary parameters: self (which is used in Python to access class variables from within the class), _simulator (the main CompuCell3D kernel object which runs the simulation), and _frequency (which tells steppableRegistry how often to run the steppable, here, every 100 MCS). Next we call the constructor for the inheritance class, SteppablePy, as required by Python. The following statement:

```
self.simulator=_simulator
```

assigns to the class variable self.simulator a reference to _simulator object, passed from the main script. We can think about Python reference as a pointer variable that stores the address of the object but not a copy of the object itself. The last two lines construct a list of all generalised cells in the simulation, a *cell inventory*, which allows us to visit all the cells with a simple for loop to perform various tasks. The cell inventory is a dynamic structure, *i.e.*, it updates automatically when cells are created or destroyed during a simulation.

The section of the TypeSwitcherSteppable steppable which implements the cell-type switching is found in the step(self, mcs) function:

```
def step(self,mcs):
    for cell in self.cellList:
        if cell.type==1:
            cell.type=2
        elif (cell.type==2):
            cell.type=1
        else:
            print "Unknown type"
```

Here we use the cell inventory to iterate over all cells in the simulation and reassign their cell types between cell.type 1 and cell.type 2. If we encounter a cell.type that is neither 1 nor 2 (which we should not), we print an error message.

Once we have created a steppable (*i.e.*, created an object of class TypeSwitcherSteppable) we must register it using registerSteppable function from steppableRegistry object:

```
typeSwitcherSteppable=TypeSwitcherSteppable(sim,100);
steppableRegistry.registerSteppable(typeSwitcherSteppable)
```

CompuCell3D will not run unregistered steppables. As we will see, much of the script is not specific to this example. We will recycle it with slight changes in later examples.

Figure 12 shows snapshots of the cell-type-oscillator simulation.

We mentioned earlier that users can run simulations without a CC3DML configuration file. Listing 9 shows the cell-type-oscillator simulation written entirely in Python, with changes to Listing 8 shown in **bold**.

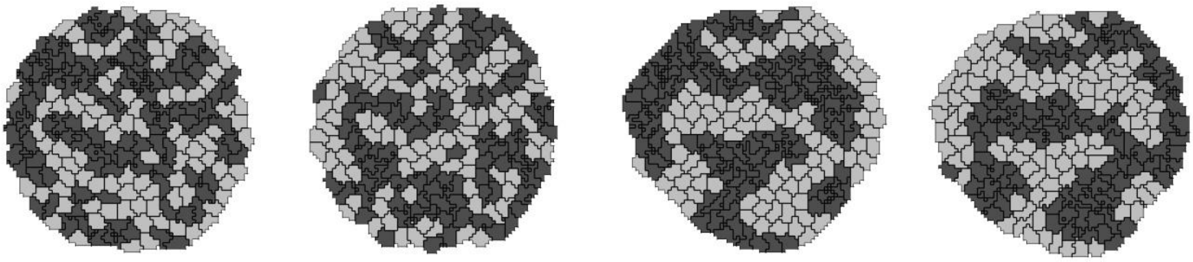


Figure 12: Results of the Python cell-type-oscillator simulation using the `TypeSwitcherSteppable` steppable implemented in Listing 8 in conjunction with the CC3DML cell-sorting simulation in Listing 3. Cells exchange types and corresponding adhesivities and colors every 100 MCS; *i.e.*, between $t = 90$ MCS and $t = 110$ MCS and between $t = 1490$ MCS and $t = 1510$ MCS.

Listing 9: Stand-alone Python cell-type-oscillator script containing an initial section that replaces the CC3DML configuration file from Listing 3. Lines added to Listing 8 are shown in bold.

```
def configureSimulation(sim):
    import CompuCell
    import CompuCellSetup

    ppd=CompuCell.PottsParseData()
    ppd.Steps(20000)
    ppd.Temperature(5)
    ppd.NeighborOrder(2)
    ppd.Dimensions(CompuCell.Dim3D(100,100,1))

    ctpd=CompuCell.CellTypeParseData()
    ctpd.CellType("Medium",0)
    ctpd.CellType("Condensing",1)
    ctpd.CellType("NonCondensing",2)
    cpd=CompuCell.ContactParseData()
    cpd.Energy("Medium","Medium",0)
    cpd.Energy("NonCondensing","NonCondensing",16)
    cpd.Energy("Condensing","Condensing",2)
    cpd.Energy("NonCondensing","Condensing",11)
    cpd.Energy("NonCondensing","Medium",16)
    cpd.Energy("Condensing","Medium",16)

    vpd=CompuCell.VolumeParseData()
    vpd.LambdaVolume(1.0)
    vpd.TargetVolume(25.0)

    bipd=CompuCell.BlobInitializerParseData()
    region=bipd.Region()
    region.Center(CompuCell.Point3D(50,50,0))
    region.Radius(40)
    region.Types(”Condensing”)
    region.Types(”NonCondensing”)
    region.Width(5)

    CompuCellSetup.registerPotts(sim,ppd)
    CompuCellSetup.registerPlugin(sim,ctpd)
    CompuCellSetup.registerPlugin(sim,cpd)
    CompuCellSetup.registerPlugin(sim,vpd)
    CompuCellSetup.registerSteppable(sim,bipd)

import sys
from os import environ
from os import getcwd
import string

sys.path.append(environ[”PYTHON_MODULE_PATH”])

import CompuCellSetup
sim,simthread = CompuCellSetup.getCoreSimulationObjects()

configureSimulation(sim)

from PySteppables import *

class TypeSwitcherSteppable(SteppablePy):
    def __init__(self,_simulator,_frequency=100):
        SteppablePy.__init__(self,_frequency)
        self.simulator=_simulator
        self.inventory=self.simulator.getPotts().getCellInventory()
        self.cellList=CellList(self.inventory)

    def step(self,mcs):
        for cell in self.cellList:
            if cell.type==1:
                cell.type=2
            elif (cell.type==2):
                cell.type=1
            else:
                print ”Unknown␣type.␣In␣cellsort␣simulation␣there␣should
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣only␣be␣two␣types␣1␣and␣2”


#Create extra player fields here or add attributes
CompuCellSetup.initializeSimulationObjects(sim,simthread)

#Add Python steppables here
steppableRegistry=CompuCellSetup.getSteppableRegistry()
```

```
typeSwitcherSteppable=TypeSwitcherSteppable(sim,100);
steppableRegistry.registerSteppable(typeSwitcherSteppable)

CompuCellSetup.mainLoop(sim,simthread,steppableRegistry)

CompuCellSetup.mainLoop(sim,simthread,steppableRegistry)
```

The `configureSimulation` function replaces the CC3DML file from Listing 3. After importing `CompuCell` and `CompuCellSetup`, we have access to functions and modules that provide all the functionality necessary to code a simulation in Python. The general syntax for the opening lines of each block is:

```
snpd=CompuCell.SectionNameParseData()
```

where `SectionName` refers to the name of the section in a CC3DML configuration file and `snpd` is the name of the object of type `SectionNameParseData`. The rest of the block usually follows the syntax:

```
snpd.TagName(values)
```

where `TagName` corresponds to the name of the tag pair used to assign a value to a parameter in a CC3DML file. For values within subsections, the syntax is:

```
snpd.SubsectionName().TagName(values)
```

To input dimensions, we use the syntax:

```
snpd.TagName(CompuCell.Dim3D(x_dim,y_dim,z_dim))
```

where `x_dim, y_dim`, and `z_dim` are the x, y and z dimensions. To input a set of (x, y, z) coordinates, we use the syntax:

```
snpd.TagName(CompuCell.Point3D(x_coord,y_coord,z_coord))
```

where `x_coord, y_coord`, and `z_coord` are the x, y, and z coordinates.

In the first block (`PottsParseData`), we input the cell-lattice parameter values using the syntax:

```
ppd.ParameterName(value)
```

where `ParameterName` matches a parameter name used in the CC3DML lattice section.

Next we define the cell types using the syntax:

```
ctpd.CellType("cell_type",cell_id)
```

The next section assigns boundary energies between the cell types:

```
cpd.Energy("cell_type_1","cell_type_2",contact_energy)
```

We specify the rest of the parameter values in a similar fashion, following the general syntax described above.

The examples in Listing 8 and Listing 9 define the `TypeSwitcherSteppable` class within the main Python script. However, separating extension modules from the main script and using an `import` statement to refer to modules stored in external files is more practical. Using separate files ensures that each module can be used in multiple simulations without duplicating source code, and makes scripts more readable and editable. We will follow this convention in our remaining examples.

6.4 Two-Dimensional Foam-Flow Simulation

CompuCell3D can simulate simple physical experiments with foams. Indeed, GGH techniques grew out of foam-simulation techniques (73). Our next example shows how to use CC3DML and Python scripts to simulate quasi-two-dimensional foam flow.

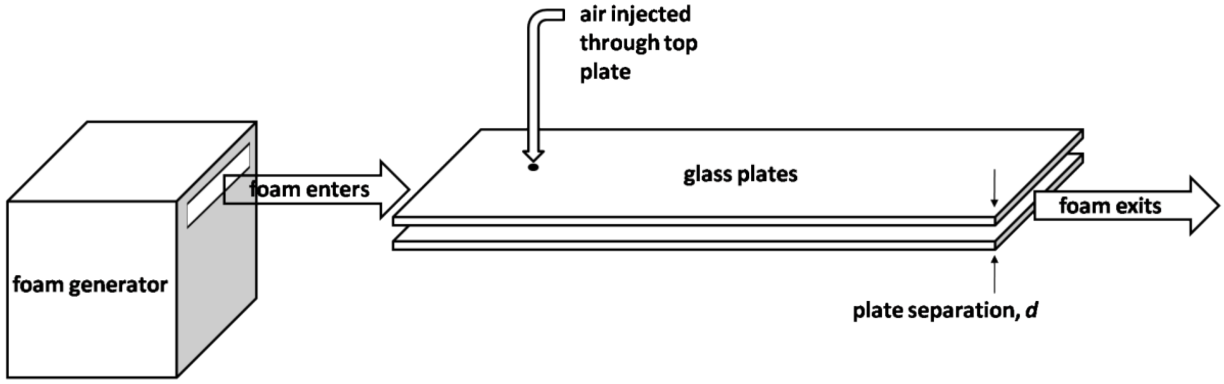


Figure 13: Schematic of experiment for studying quasi-2D foam flow.

The experimental apparatus (Figure 13) consists of a channel created by two parallel rectangular glass plates separated by 5 mm, with the gap between their long sides sealed and that between their short sides open. A foam generator injects small, uniform size bubbles at one short end, pushing older bubbles towards the open end of the channel, creating a foam flow. The top glass plate has a hole through which we inject air. Bubbles passing under this point grow because of the air injected into them, forming characteristic patterns (Figure 14) (99).

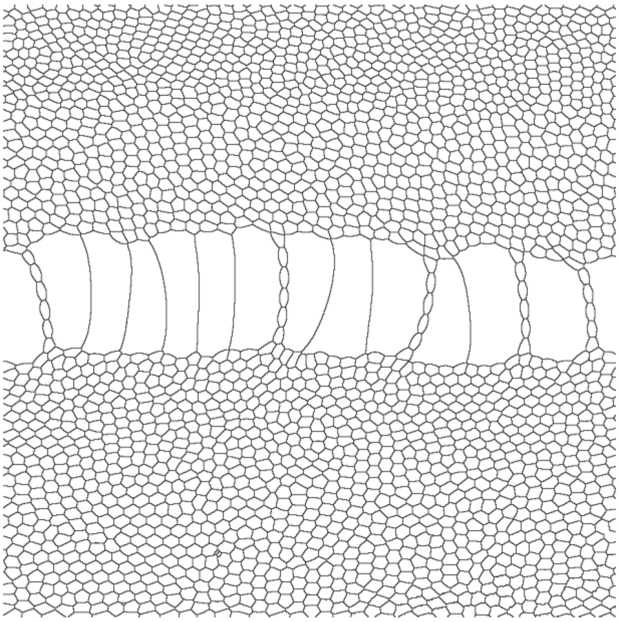


Figure 14: Detail of processed experimental image of flowing quasi-2D bubbles. Image size is 15 × 15 cm.

Generalised cells will represent bubbles in this simulation. To simulate this experiment in CompuCell3D we need to write Python steppables that 1) create bubbles at one end of the channel, 2) inject air into the bubble which includes a given location (the identity of this bubble will change in time due to the flow), 3) remove bubbles at the open end of the channel. We will store the source code in a file called **foamairSteppables.py**. We will also need a main Python script to call these steppables appropriately.

We simulate bubble injection by creating generalised cells (bubbles) along the lattice edge corresponding to the left end of the channel (small-x values of the cell lattice). We simulate air injection into a bubble at the injection point, by identifying the bubble currently at the injection point and increasing its target volume at a fixed rate. Removing a bubble from the simulation simply requires assigning it a target volume of zero once it comes close to the right end of the channel (large-x values of the cell lattice).

We first define a CC3DML configuration file for the foam-flow simulation (Listing 10).

Listing 10: CC3DML configuration file for the foam-flow simulation. This file initialises needed plugins but all of the interesting work is done in Python.

```
<CompuCell3D>

<Potts>
    <Dimensions x="200" y="50" z="1"/>
    <Steps>10000</Steps>
    <Temperature>5</Temperature>
    <LatticeType>Hexagonal</LatticeType>
</Potts>

<Plugin Name="VolumeLocalFlex"/>

<Plugin Name="CellType">
    <CellType TypeName="Medium" TypeId="0"/>
    <CellType TypeName="Foam" TypeId="1"/>
</Plugin>

<Plugin Name="Contact">
    <Energy Type1="Medium" Type2="Medium">5</Energy>
    <Energy Type1="Foam" Type2="Foam">5</Energy>
    <Energy Type1="Foam" Type2="Medium">5</Energy>
    <NeighborOrder>3</NeighborOrder>
</Plugin>

<Plugin Name="CenterOfMass"/>

</CompuCell3D>
```

The CC3DML configuration file is simple: it initialises the `VolumeLocalFlex`, `CellType`, `Contact` and `CenterOfMass` plugins. We do not use a cell-lattice-initializer steppable, because all bubbles are created as the simulation runs. We use `VolumeLocalFlex` because individual bubbles will change their target volumes during the simulation. We also include the `CenterOfMass` plugin to track the changing centroids of each bubble. The `CenterOf Mass` plugin in CompuCell3D actually calculates $\mathbf{x}_\sigma^C$, the centroid of the generalised cell multiplied by volume of the cell:

$$\mathbf{x}_\sigma^C = \sum_{\mathbf{i}} \mathbf{i}\,\delta(\sigma(\mathbf{j}), \sigma(\mathbf{i}))\,, \tag{11}$$

so the actual centroid of the bubble is:

$$\mathbf{x}_\sigma = \frac{\mathbf{x}_\sigma^C}{v(\sigma)}\,. \tag{12}$$

The ability to track a generalised-cells centroid is useful if we need to pick a single reference point in the cell. In this example we will remove bubbles whose centroids have x-coordinate greater than a cutoff value.

We will implement the Python script in four sections: 1) a main script (Listing 11), which runs every MCS and calls the steppables to (2) create bubbles at the left end of the cell lattice (`BubbleNucleator`, Listing 12), (3) enlarge the target volume of the bubble at the injector site (`AirInjector`, Listing 13), and (4) set the target volume of bubbles at the right end of the cell lattice to zero (`BubbleCellRemover`, Listing 14). We store classes (2-4) in a separate file called **foamairSteppables.py**.

Listing 11: Main Python Script for foam-flow simulation. Changes to the template (Listing 7) are shown in bold.

```
import sys
from os import environ
import string
sys.path.append(environ[”PYTHON_MODULE_PATH”])

import CompuCellSetup

sim,simthread = CompuCellSetup.getCoreSimulationObjects()

#Create extra player fields here
CompuCellSetup.initializeSimulationObjects(sim,simthread)

#Add Python steppables here
steppableRegistry=CompuCellSetup.getSteppableRegistry()

from foamairSteppables import BubbleNucleator
bubbleNucleator=BubbleNucleator(sim,20)
bubbleNucleator.setNumberOfNewBubbles(1)
bubbleNucleator.setInitialTargetVolume(25)
bubbleNucleator.setInitialLambdaVolume(2.0)
bubbleNucleator.setInitialCellType(1)
steppableRegistry.registerSteppable(bubbleNucleator)

from foamairSteppables import AirInjector
airInjector=AirInjector(sim,40)
airInjector.setVolumeIncrement(25)
airInjector.setInjectionPoint(50,25,0)
steppableRegistry.registerSteppable(airInjector)

from foamairSteppables import BubbleCellRemover
bubbleCellRemover=BubbleCellRemover(sim)
bubbleCellRemover.setCutoffValue(170)
steppableRegistry.registerSteppable(bubbleCellRemover)

CompuCellSetup.mainLoop(sim,simthread,steppableRegistry)
```

The main script in Listing 11 builds on the template Python code in Listing 7; we show changes in bold. The line:

```
from foamairSteppables import BubbleNucleator
```

tells Python to look for the `BubbleNucleator` class in the file named **foamairSteppables.py**.

```
bubbleNucleator=BubbleNucleator(sim, 20)
```

creates the steppable `BubbleNucleator` that will run every 20 MCS. The next few lines in this section pass the number of bubbles to create, which in our case is one:

```
bubbleNucleator.setNumberOfNewBubbles(1)
```

the initial V_t for the new bubble, which is 25 pixels:

```
bubbleNucleator.setInitialTargetVolume(25)
```

the initial λ_{vol} for the bubble:

```
bubbleNucleator.setInitialLambdaVolume(2.0)
```

and the bubbles `type.id`:

```
bubbleNucleator.setInitialCellType(1)
```

Finally, we register the steppable:

```
steppableRegistry.registerSteppable(bubbleNucleator)
```

The next group of lines repeats the process for the `AirInjector` steppable, reading it from the file **foamairSteppables.py**:

```
from foamairSteppables import AirInjector
```

`AirInjector` is run every 40 MCS:

```
airInjector=AirInjector(sim, 40)
```

and increases V_t by 25:

```
airInjector.setVolumeIncrement(25)
```

for the bubble occupying the pixel at the point $(50, 25, 0)$ on the cell lattice:

```
airInjector.setInjectionPoint(50,25,0)
```

As before, the final line registers the steppable:

```
steppableRegistry.registerSteppable(airInjector)
```

The final new section reads the `BubbleCellRemover` steppable from the file **foamairSteppables.py**:

```
from foamairSteppables import BubbleCellRemover
```

and invokes the steppable, telling it to run every MCS; note that we have omitted the number after `sim`:

```
bubbleCellRemover=BubbleCellRemover(sim)
```

Next we set 170 as the x-coordinate at which we will destroy bubbles:

```
bubbleCellRemover.setCutoffValue(170)
```

and, finally, register `BubbleCellRemover`

```
steppableRegistry.registerSteppable(bubbleCellRemover)
```

We must also write Python code to define the three steppables `BubbleCellRemover`, `AirInjector`, and `BubbleCellRemover` and save them in the file **foamairSteppables.py**.

Listing 12 shows the code for the BubbleNucleator steppable.

Listing 12: Python code for the BubbleNucleator steppable. This module creates bubbles at points with random y coordinates and x coordinate of 3.

```
from CompuCell import Point3D
from random import randint

class BubbleNucleator(SteppablePy):
    def __init__(self,_simulator,_frequency=1):
        SteppablePy.__init__(self,_frequency)
        self.simulator=_simulator

    def start(self):
        self.Potts=self.simulator.getPotts()
        self.dim=self.Potts.getCellFieldG().getDim()

    def setNumberOfNewBubbles(self,_numNewBubbles):
        self.numNewBubbles=int(_numNewBubbles)

    def setInitialTargetVolume(self,_initTargetVolume):
        self.initTargetVolume=_initTargetVolume
    def setInitialLambdaVolume(self,_initLambdaVolume):
        self.initLambdaVolume=_initLambdaVolume

    def setInitialCellType(self,_initCellType):
        self.initCellType=_initCellType

    def createNewCell(self,pt):
        print "Nucleated␣bubble␣at␣",pt
        cell=self.Potts.createCellG(pt)
        cell.targetVolume=self.initTargetVolume
        cell.type=self.initCellType
        cell.lambdaVolume=self.initLambdaVolume

    def nucleateBubble(self):
        pt=Point3D(0,0,0)
        pt.y=randint(0,self.dim.y-1)
        pt.x=3
        self.createNewCell(pt)

    def step(self,mcs):
        for i in xrange(self.numNewBubbles):
            self.nucleateBubble()
```

The first two lines import necessary modules, where the line:

```
from CompuCell import Point3D
```

allows us to access points on the simulation cell lattice, and the line:

```
from random import randint
```

allows us to generate random integers.

In the constructor of the `BubbleNucleator` steppable class we assign to the variable `self.simulator` a reference to the `simulator` object from the CompuCell3D kernel. In the `start(self)` function, we assign a reference to the `Potts` object from the CompuCell3D kernel to the variable `self.Potts`:

```
self.Potts=self.simulator.getPotts()
```

and assign the dimensions of the cell lattice to self.dim:

```
self.dim=self.Potts.getCellFieldG().getDim()
```

In addition to the four essential steppable member functions (`__init__(self, _simulator, _frequency)`, `start(self)`, `step(self, mcs)` and `finish(self)`), `BubbleNucleator` includes several functions, some of which set parameters and some of which perform necessary tasks. The functions setNumberOfNewBubbles, setInitialTargetVolume and setInitialLambdaVolume accept the values passed from the main Python script in Listing 11.

The CreateNewCell function requires that we pass the coordinates of the point, pt, at which to create a new bubble:

```
def CreateNewCell (self,pt):
```

Then we use a built-in CompuCell3D function to add a new bubble at that location:

```
cell=self.Potts.createCellG(pt)
```

assigning the new cell a target volume $V_t = targetVolume$:

```
cell.targetVolume=self.initTargetVolume
```

type, $\tau = type$:

```
cell.type=self.initCellType
```

and compressibility $\lambda_{\text{vol}} = lambdaVolume$:

```
cell.lambdaVolume=initLambdaVolume
```

based on the values passed to the BubbleNucleator steppable from the main script.

The first three lines of the nucleateBubble function create a reference to a point on the cell lattice (pt=Point3D(0,0,0)), assign it a random y-coordinate between 0 and y_dim-1:

```
pt.y=randint(0,self.dim.y-1)
```

and an x-coordinate of 3:

```
pt.x=3
```

The line calls the createNewCell function and passes it the point (pt) at which to create the new bubble:

```
self.createNewCell(pt)
```

Finally, the step(self,mcs) function calls the nucleateBubble function self.numNewBubbles times per MCS.

Listing 13 shows the code for the AirInjector steppable.

Listing 13: Python code for the AirInjector steppable which stimulates air injection into the bubble currently occupying the cell-lattice pixel. Air injection begins after 5000 MCS to allow the channel to partially fill with bubbles.

```python
class AirInjector(SteppablePy):
    def __init__(self,_simulator,_frequency=1):
        SteppablePy.__init__(self,_frequency)
        self.simulator=_simulator
        self.Potts=self.simulator.getPotts()
        self.cellField=self.Potts.getCellFieldG()

    def start(self): pass

    def setInjectionPoint(self,_x,_y,_z):
        self.injectionPoint=CompuCell.Point3D(int(_x),int(_y),int(_z))

    def setVolumeIncrement(self,_increment):
        self.volumeIncrement=_increment

    def step(self,mcs):
        if mcs <5000:
            return
        cell=self.cellField.get(self.injectionPoint)
        if cell:
            cell.targetVolume+=self.volumeIncrement
```

The first three lines of the `__init__(self,_simulator,_frequency)` function are identical to the same lines in the `BubbleNucleator` steppable (Listing 12). The final line of the function:

```python
self.cellField=self.Potts.getCellFieldG()
```

loads the cell-lattice parameters. The `start(self)` function in this steppable does not do anything:

```python
def start(self): pass
```

The next two functions read the `injectionPoint` and `volumeIncrement` passed to the `AirInjector` steppable by the main Python script (Listing 11). The `step` function uses these values to identify the bubble at the injection site, `self.injectionPoint`:

```python
cell=self.cellField.get(self.injectionPoint)
```

and then increment that bubbles target volume V_t by `self.volumeIncrement`:

```
if cell:
    cell.targetVolume+=self.volumeIncrement
```

Note the syntax:

```
if cell:
```

which we use to test whether a cell is `Medium` or not. `Medium` in CompuCell3D is assigned a `NULL` pointer, which, in Python, becomes a `None` object. Python evaluates the `None` object as `False` and other objects (in our case, bubbles) as `True`, so the task is only carried out on bubbles, not `Medium`.

In the first two lines of the `step(self,mcs)` function, we tell the function not to perform its task until 5000 MCS have elapsed:

```
if mcs <5000:
    return
```

The 5000 MCS delay allows the simulation to establish a uniform flow of small bubbles throughout a large portion of the cell lattice.

Finally, we define the `BubbleCellRemover` steppable (Listing 14) which we save in the file **foamairSteppables.py**.

Listing 14: Python code for the BubbleCellRemover steppable. This module removes cells once the x-coordinates of their centroids > cutoffValue by setting their target volumes to zero and increasing their λ_{vol} to 10000.

```
class BubbleCellRemover(SteppablePy):
    def __init__(self,_simulator,_frequency=1):
        SteppablePy.__init__(self,_frequency)
        self.simulator=_simulator
        self.inventory=self.simulator.getPotts().getCellInventory()
        self.cellList=CellList(self.inventory)

    def start(self):
        self.Potts=self.simulator.getPotts()
        self.dim=self.Potts.getCellFieldG().getDim()

    def setCutoffValue(self,_cutoffValue):
        self.cutoffValue=_cutoffValue

    def step(self,mcs):
        for cell in self.cellList:
            if cell:
                    if int(cell.xCM/float(cell.volume))>self.cutoffValue:
                        cell.targetVolume=0
                        cell.lambdaVolume=10000
```

At each MCS we scan the cell inventory looking for cells whose centroid has an x-coordinate close to the right end of the lattice and remove these cells from the simulation by setting their target volumes to zero and increasing λ_{vol} to 10000.

The first two lines of the `__init__ (self,_simulator,_frequency)` function are identical to the corresponding lines in the `BubbleNucleator` and `AirInjector` steppables (Listing 12 and Listing 13). In the third line of the function, we gain access to the generalised-cell inventory:

```
self.inventory=self.simulator.getPotts().getCellInventory()
```

and in the fourth line we make a list containing all of the generalised cells in the simulation:

```
self.cellList=CellList(self.inventory)
```

The `start(self)` function is identical to that of the `BubbleNucleator` steppable (Listing 12), and performs the same function.

The next function:

```
setCutoffValue(self, _cutoffValue)
```

reads the `cutoffValue` for the x-coordinate that we passed to `BubbleCellRemover` from the main Python script (Listing 11). Finally, the `step(self, mcs)` function iterates through the cell inventory. We first check to make sure that the cell is not `Medium`:

```
if cell:
```

For each non-`Medium` cell we test whether the x-coordinate of the cells centroid is greater than the `cutoffValue`:

```
if int(cell.xCM/float(cell.volume))>self.cutoffValue:
```

and, if it is, set that cells `targetVolume`, V_t, to zero:

```
cell.targetVolume=0
```

and its $\lambda_{\text{vol}} = 10000$:

```
cell.lambdaVolume=10000
```

Running the CC3DML file from Listing 10 and the main Python script from Listing 11 (which loads the steppables in Listing 12, Listing 13 and Listing 14 from the file **foamairSteppables.py**) produces the snapshots shown in Figure 15.

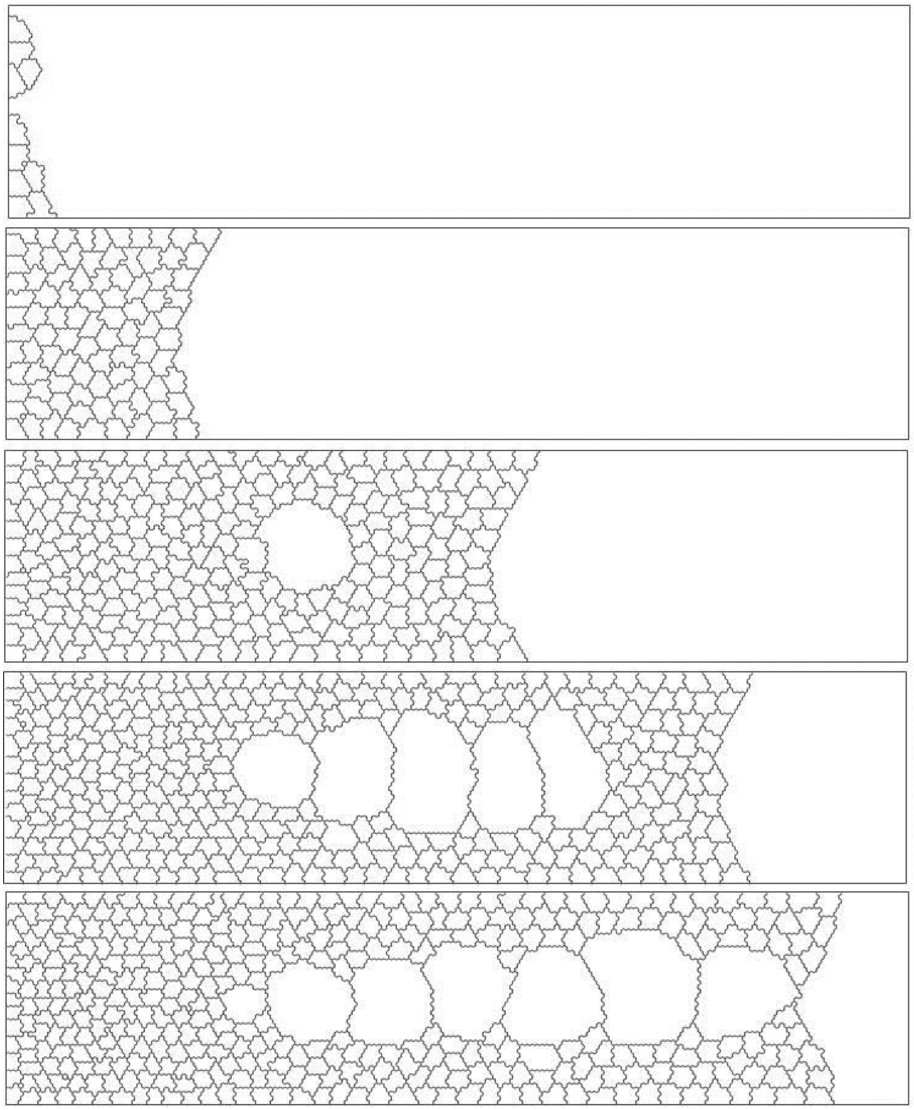


Figure 15: Results of the foam-flow simulation on a 2D 3rd-neighbor hexagonal lattice at t = 200 MCS, t = 2000 MCS, t = 5500 MCS, t = 7500 MCS, and t = 9880 MCS. Simulation code is given in Listing 10,Listing 11, Listing 12, Listing 13, and Listing 14.

6.5 Diffusing-Field-Based Cell-Growth Simulation

One of the most frequent uses of Python scripting in CompuCell3D simulations is to modify cell behavior based on local field concentrations. To demonstrate this use, we incorporate stem-cell-like behavior into the cell-sorting simulation from Listing 1. This extension requires including relatively sophisticated interactions between cells and diffusing chemical, *FGF* (100).

We simulate a situation where `NonCondensing` cells secrete FGF, which diffuses freely through the cell lattice and obeys:

$$\frac{\partial[FGF](\vec{i})}{\partial t} = 0.10\nabla^2[FGF](\vec{i}) + 0.05(\tau(\sigma(\vec{i})), \texttt{NonCondensing}), \tag{13}$$

where [FGF] denotes the FGF concentration and `Condensing` cells respond to the field by growing at a constant rate proportional to the FGF concentration at their centroids:

$$\frac{dV_t(\sigma)}{dt} = 0.01[FGF](\vec{x}_\sigma) \tag{14}$$

When they reach a threshold volume, the `Condensing` cells undergo mitosis. One of the resulting daughter cells remains a `Condensing` cell, while the other daughter cell has an equal probability of becoming either another `Condensing` cell or a `DifferentiatedCondensing` cell. `DifferentiatedCondensing` cells do not divide.

Each generalised cell in CompuCell3D has a default list of attributes, *e.g.*, type, volume, surface (area), target volume, etc. However, CompuCell3D allows users to add cell attributes during execution of simulations. For example, in the current simulation, we will record data on each cell division in a list attached to each cell. Generalised cell attributes can be added using either C++ or Python. However, attributes added using Python are not accessible from C++ modules.

As in the foam-flow simulation, we divide the necessary simulation tasks among different Python modules (or classes) which we save in a file **cellsort_2D_field_modules.py** and call from the main Python script. We reuse elements of the CC3DML files we presented earlier to construct the CC3DML configuration file, presented in Listing 15.

Listing 15: CC3DML code for the diffusing-field-based cell-growth simulation.

```
<CompuCell3D>

<Potts>
    <Dimensions x="200" y="200" z="1"/>
    <Steps>10000</Steps>
    <Temperature>10</Temperature>
    <NeighborOrder>2</NeighborOrder>
</Potts>
```

```
<Plugin Name=”VolumeLocalFlex”/>

<Plugin Name=”CellType”>
    <CellType TypeName=”Medium” TypeId=”0”/>
    <CellType TypeName=”Condensing” TypeId=”1”/>
    <CellType TypeName=”NonCondensing” TypeId=”2”/>
    <CellType TypeName=”CondensingDifferentiated” TypeId=”3”/>
</Plugin>

<Plugin Name=”Contact”>
    <Energy Type1=”Medium” Type2=”Medium”>0</Energy>
    <Energy Type1=”NonCondensing” Type2=”NonCondensing”>16</Energy>
    <Energy Type1=”Condensing” Type2=”Condensing”>2</Energy>
    <Energy Type1=”NonCondensing” Type2=”Condensing”>11</Energy>
    <Energy Type1=”NonCondensing” Type2=”Medium”>16</Energy>
    <Energy Type1=”Condensing” Type2=”Medium”>16</Energy>
    <Energy Type1=”CondensingDifferentiated”
        Type2=”CondensingDifferentiated”>2</Energy>
    <Energy Type1=”CondensingDifferentiated” Type2=”Condensing”>2</Energy>
    <Energy Type1=”CondensingDifferentiated”
        Type2=”NonCondensing”>11</Energy>
    <Energy Type1=”CondensingDifferentiated” Type2=”Medium”>16</Energy>
    <NeighborOrder>2</NeighborOrder>
</Plugin>

<Plugin Name=”CenterOfMass”/>

<Steppable Type=”FlexibleDiffusionSolverFE”>
    <DiffusionField>
        <DiffusionData>
            <FieldName>FGF</FieldName>
            <DiffusionConstant>0.10</DiffusionConstant>
            <DecayConstant>0.00005</DecayConstant>
        </DiffusionData>
        <SecretionData>
            <Secretion Type=”NonCondensing”>0.05</Secretion>
        </SecretionData>
    </DiffusionField>
</Steppable>

<Steppable Type=”BlobInitializer”>
    <Region>
        <Gap>0</Gap>
        <Width>5</Width>
        <Radius>40</Radius>
        <Center x=”100” y=”100” z=”0”/>
        <Types>Condensing,NonCondensing</Types>
    </Region>
</Steppable>
```

```
</CompuCell3D>
```

The CC3DML code is a slightly extended version of the cell-sorting code in Listing 3 plus the `FlexibleDiffusionSolverFE` discussed in the bacterium-and-macrophage simulation (see Listing 4). The initial cell-lattice does not contain any `CondensingDifferentiated` cells. These cells appear only as the result of mitosis. We use the `VolumeLocalFlex` plugin to allow the target volume to vary individually for each cell, allowing cell growth as discussed in the foam-flow simulation. We manage the volume-constraint parameters using a Python script. The `CenterOfMass` plugin provides a reference point in each cell at which we measure the FGF concentration. We then adjust the cell's target volume accordingly.

To build this simulation in CompuCell3D we need to write several Python routines. We need:

1. A steppable, `VolumeConstraintSteppable` to initialize the volume-constraint parameters for each cell and to simulate cell growth by periodically increasing `Condensing` cells target volumes in proportion to the FGF concentration at their centroids.

2. A plugin, `CellsortMitosis`, that runs the CompuCell3D mitosis algorithm when any cell reaches a threshold volume and then adjusts the parameters of the resulting parent and daughter cells. This plugin also appends information about the time and type of cell division to a list attached to each cell.

3. A steppable, `MitosisDataPrinterSteppable`, that prints the cell-division information from the lists attached to each cell.

4. A class, `MitosisData`, which `MitosisDataPrinterSteppable` uses to extract and format the data it prints.

5. A main Python script to call the steppables and the `CellsortMitosis` plugin appropriately.

We store the source code for routines 1)-4) in a separate file called **cellsort_2D_field_modules.py**.

Listing 16 shows the main Python script for the diffusing-field-based cell-growth simulation, with changes to the template (Listing 7) shown in blue.

Listing 16: Main Python script for the diffusing-field-based cell-growth simulation. Changes to the template code (Listing 7) shown in blue.

```
import sys
from os import environ
from os import getcwd
import string

sys.path.append(environ[”PYTHON_MODULE_PATH”])

import CompuCellSetup

sim,simthread = CompuCellSetup.getCoreSimulationObjects()

#add additional attributes
pyAttributeAdder,listAdder=CompuCellSetup.attachListToCells(sim)

CompuCellSetup.initializeSimulationObjects(sim,simthread)

#notice importing CompuCell to main script has to be
#done after call to getCoreSimulationObjects()
import CompuCell
changeWatcherRegistry=CompuCellSetup.getChangeWatcherRegistry(sim)
stepperRegistry=CompuCellSetup.getStepperRegistry(sim)

from cellsort_2D_field_modules import CellsortMitosis
cellsortMitosis=CellsortMitosis(sim,changeWatcherRegistry,\
stepperRegistry)
cellsortMitosis.setDoublingVolume(50)

#Add Python steppables here
steppableRegistry=CompuCellSetup.getSteppableRegistry()

from cellsort_2D_field_modules import VolumeConstraintSteppable
volumeConstraint=VolumeConstraintSteppable(sim)
steppableRegistry.registerSteppable(volumeConstraint)

from cellsort_2D_field_modules import MitosisDataPrinterSteppable
mitosisDataPrinterSteppable=MitosisDataPrinterSteppable(sim)
steppableRegistry.registerSteppable(mitosisDataPrinterSteppable)

CompuCellSetup.mainLoop(sim,simthread,steppableRegistry)
```

The first change to the template code (Listing 7) is:

```
pyAttributeAdder,listAdder=CompuCellSetup.attachListToCells(sim)
```

which instructs the CompuCell3D kernel to attach a Python-defined list to each cell when it creates it. This list serves as a generic container which can store any set of Python objects and hence any set of generalized-cell properties. In the current simulation, we use the list to store objects of the class `MitosisData`, which records the Monte Carlo Step at which each cell division involving the current cell or its parent, happened, as well as, the cell index and cell type of the parent and daughter cells.

Because one of our Python modules is a lattice monitor rather than a steppable, we need to create `stepperRegistry` and `changeWatcherRegistry` objects, which store the two types of lattice monitors:

```
changeWatcherRegistry=CompuCellSetup.getChangeWatcherRegistry(sim)
stepperRegistry=CompuCellSetup.getStepperRegistry(sim)
```

The `CellsortMitosis` plugin is a lattice monitor because it acts in response to certain index-copy events; it is invoked whenever a cell's volume reaches the threshold volume for mitosis. The following lines create the `CellsortMitosis` lattice monitor and register it with the `stepperRegistry` and `changeWatcherRegistry`:

```
from cellsort_2D_field_modules import CellsortMitosis
cellsortMitosis = CellsortMitosis(sim,changeWatcherRegistry,\
stepperRegistry)
```

Because the base class inherited by `CellsortMitosis`, unlike our steppables, handles registration internally, we do not have to register `CellsortMitosis` explicitly. We can now set the threshold volume at which `Condensing` cells divide:

```
cellsortMitosis.setDoublingVolume(50)
```

Next we import the `VolumeConstraintSteppable` steppable, which initializes cells target volumes and compressibilities at the beginning of the simulation and also implements chemical-dependent cell growth for `Condensing` cells, and register it:

```
from cellsort_2D_field_modules import VolumeConstraintSteppable
volumeConstraint=VolumeConstraintSteppable(sim)
steppableRegistry.registerSteppable(volumeConstraint)
```

Finally, we import, create and register the `MitosisDataPrinterSteppable` steppable, which prints the content of `MitosisData` objects for cells that have divided:

```
from cellsort_2D_field_modules import MitosisDataPrinterSteppable
mitosisDataPrinterSteppable=MitosisDataPrinterSteppable(sim)
steppableRegistry.registerSteppable(mitosisDataPrinterSteppable)
```

The number of `MitosisData` objects stored in each cell at any given Monte Carlo Step depends on cell type (`NonCondensing` cells do not divide, whereas `Condensing` cells can divide multiple times), and how often a given cell has divided.

Moving on to the Python modules, we consider the `VolumeConstraintSteppable` steppable shown in Listing 17.

Listing 17: Python code for the `VolumeConstraintSteppable` written in the file **cellsort_2D_field_modules.py** for the diffusing-field-based cell-growth simulation.

```
import sys
from os import environ
from os import getcwd
import string

class VolumeConstraintSteppable(SteppablePy):
    def __init__(self,_simulator,_frequency=1):
        SteppablePy.__init__(self,_frequency)
        self.simulator=_simulator
        self.inventory=self.simulator.getPotts().getCellInventory()
        self.cellList=CellList(self.inventory)

    def start(self):
        for cell in self.cellList:
            cell.targetVolume=25
            cell.lambdaVolume=2.0

    def step(self,mcs):
        field=CompuCell.getConcentrationField(self.simulator,”FGF”)
        comPt=CompuCell.Point3D()
        for cell in self.cellList:
            if cell.type==1: #Condensing cell
                comPt.x=int(round(cell.xCM/float(cell.volume)))
                comPt.y=int(round(cell.yCM/float(cell.volume)))
                comPt.z=int(round(cell.zCM/float(cell.volume)))
                concentration=field.get(comPt) #get concentration at comPt
                #and increase cell's target volume
                cell.targetVolume+=0.1*concentration
```

The `__init__` constructor looks very similar to the one in Listing 14, with the difference that we pass `_frequency=1` to update the cell volumes once per MCS. We also request the field-lattice dimensions and values from CompuCell3D:

```
self.dim=self.simulator.getPotts().getCellFieldG().getDim()
```

and specify that we will work with a field named FGF:

```
self.fieldName=”FGF”
```

The script contains two functions: one that initializes the cells volume-constraint parameters (`start(self)`) and one that updates them (`step(self, mcs)`). The `start(self)` function executes only once, at the beginning of the simulation. It iterates over each cell (`for cell in self.cellList:`) and assigns the initial cells `targetVolume` ($V_t(\sigma) = 25$ pixels) and `lambdaVolume` ($\lambda_{\mathrm{vol}}(\sigma) = 2.0$) parameters as the `VolumeLocalFlex` plugin requires.

The first line of the `step(self, mcs)` function extracts a reference to the FGF concentration field defined using the `FlexibleDiffusionSolverFE` steppable in the CC3DML file (each field created in a CompuCell3D simulation is registered and accessible by both C++ and Python). The function then iterates over every cell in the simulation. If a cell is of `cell.type` 1 (`Condensing` - see the CC3DML configuration file, Listing 15), we calculate its centroid:

```
comPt.x=int(round(cell.xCM/float(cell.volume)))
comPt.y=int(round(cell.yCM/float(cell.volume)))
comPt.z=int(round(cell.zCM/float(cell.volume)))
```

and retrieve the FGF concentration at that point:

```
concentration=field.get(comPt)
```

We then increase the target volume of the cell by 0.01 times that concentration:

```
cell.targetVolume+=0.1*concentration
```

We must include the `CenterOfMass` plugin in the CC3DML code. Otherwise the centroid (`cell.xCM, cell.yCM, cell.zCM`) will have the default value (0,0,0).

Listing 18 shows the code for the `CellsortMitosis` plugin. The plugin divides the mitotic cell into two cells and adjusts both cells' attributes. It also initializes and appends `MitosisData` objects to the original cell's (`self.parentCell`) and daughter cell's (`self.childCell`) attribute lists.

Listing 18: Python code for the `CellsortMitosis` written in the file **cellsort_2D_field_modules.py**. The plugin handles division of cells when they reach a threshold volume.

```
from random import random
from PyPluginsExamples import MitosisPyPluginBase

class CellsortMitosis(MitosisPyPluginBase):
    def __init__(self,_simulator,_changeWatcherRegistry, \
    _stepperRegistry):
        MitosisPyPluginBase.__init__(self,_simulator,\
        _changeWatcherRegistry,_stepperRegistry)

    def updateAttributes(self):
        self.parentCell.targetVolume=self.parentCell.volume/2.0
        self.childCell.targetVolume=self.parentCell.targetVolume
        self.childCell.lambdaVolume=self.parentCell.lambdaVolume

        if (random()<0.5):
            self.childCell.type=self.parentCell.type
        else:
            self.childCell.type=3

        ##record mitosis data in parent and daughter cells
        mcs=self.simulator.getStep()
        mitData=MitosisData(mcs,self.parentCell.id,self.parentCell.type,\
        self.childCell.id,self.childCell.type)

        #get a reference to lists storing Mitosis data
        parentCellList=CompuCell.getPyAttrib(self.parentCell)
        childCellList=CompuCell.getPyAttrib(self.childCell)

        parentCellList.append(mitData)
        childCellList.append(mitData)
```

The second line of Listing 18:

```
from PyPluginsExamples import MitosisPyPluginBase
```

lets us access the CompuCell3D base class `MitosisPyPluginBase`.

`CellsortMitosis` inherits the content of the `MitosisPyPluginBase` class. `MitosisPyPluginBase` internally accesses the CompuCell3D-provided `Mitosis` plugin, which is written in C++, and handles all the technicalities of plugin initialization behind the scenes. The `MitosisPyPluginBase` class provides a simple-to-use interface to this plugin. To create a customized version of `MitosisPyPluginBase`, `CellsortMitosis`, we must call the constructor of `MitosisPyPluginBase` from the `CellsortMitosis` constructor:

```
MitosisPyPluginBase.__init__(self,_simulator, _changeWatcherRegistry, \
_stepperRegistry)
```

We also need to reimplement the function `updateAttributes(self)`, which is called by `MitosisPyPluginBase` after mitosis takes place, to define the post-division cells parameters. The objects `self.childCell` and `self.parentCell` that appear in the function are initialized and managed by `MitosisPyPluginBase`. In the current simulation, after division we set V_t for the parent and daughter cells to half of the V_t of the parent just prior to cell division. λ_{vol} is left unchanged for the parent cell and the same value is assigned to the daughter cell:

```
self.parentCell.targetVolume=self.parentCell.volume/2.0
self.childCell.targetVolume=self.parentCell.targetVolume
self.childCell.lambdaVolume=self.parentCell.lambdaVolume
```

The cell type of one of the two daughter cells (`childCell`) is randomly chosen to be either `Condensing` (*i.e.*, the same as the parent type) or `CondensingDifferentiated`, which we have defined to be `cell.type` 3 (Listing 15):

```
if (random()<0.5):
    self.childCell.type=self.parentCell.type
else:
    self.childCell.type=3
```

The parent cell remains `Condensing`. We now add a description of this cell division to the lists attached to each cell. First we collect the data in a list called `mitData`:

```
mcs=self.simulator.getStep()
mitData=MitosisData(mcs,self.parentCell.id,self.parentCell.type, \
self.childCell.id,self.childCell.type)
```

then we access the lists attached to the two cells:

```
parentCellList=CompuCell.getPyAttrib(self.parentCell)
childCellList=CompuCell.getPyAttrib(self.childCell)
```

and append the new mitosis data to these lists:

```
parentCellList.append(mitData)
childCellList.append(mitData)
```

Listing 19 shows the Python code for the `MitosisData` class, which stores the data on the cell division that we append to the cells attribute lists after each cell division.

Listing 19: Python code for the `MitosisData` written in the file **cellsort_2D_field_modules.py**. `MitosisData` objects store information about cell divisions involving the parent and daughter cells.

```
from random import random
from PyPluginsExamples import MitosisPyPluginBase

class MitosisData:
    def __init__(self,_MCS,_parentId,_parentType,_offspringId, \
    _offspringType):
        self.MCS=_MCS
        self.parentId=_parentId
        self.parentType=_parentType
        self.offspringId=_offspringId
        self.offspringType=_offspringType

    def __str__(self):
        return "Mitosis␣time="+str(self.MCS)+ \
        "parentId="+str(self.parentId)+ \
        "offspringId="+str(self.offspringId)
```

In the constructor of `MitosisData`, we read in the time (in MCS) of the division, along with the parent and daughter cell indices and types. The `__str__(self)` convenience function returns an ASCII string representation of the time and cell indices only, to allow the Python `print` command to print out this information.

Listing 20 shows the Python code for the `MitosisDataPrinterSteppable` steppable, which prints the mitosis data to the user's screen.

Listing 20: Python code for the `MitosisDataPrinter` steppable written in the file **cellsort_2D_field_modules.py**. The steppable prints the cell-division history for dividing cells (see Figure 18).

```
class MitosisDataPrinterSteppable(SteppablePy):
    def __init__(self,_simulator,_frequency=100):
        SteppablePy.__init__(self,_frequency)
        self.simulator=_simulator
        self.inventory=self.simulator.getPotts().getCellInventory()
        self.cellList=CellList(self.inventory)

    def step(self,mcs):
        for cell in self.cellList:
            mitDataList=CompuCell.getPyAttrib(cell)
            if len(mitDataList) > 0:
                print "MITOSIS_DATA_FOR_CELL_ID",cell.id
                for mitData in mitDataList:
                    print mitData
```

The constructor is identical to that for the `VolumeConstraintSteppable` steppable (Listing 17). Within the `step(self,mcs)` function, we iterate over each cell (`for cell in self.cellList:`) and access the Python list attached to the cell (mitDataList=CompuCell.getPyAttrib(cell)). If a given cell has undergone mitosis, then the list will have entries, and thus a nonzero length. If so, we print the MitosisData objects stored in the list:

```
if len(mitDataList) > 0:
    print "MITOSIS_DATA_FOR_CELL_ID", cell.id
    for mitData in mitDataList:
        print mitData
```

Figure 16 and Figure 17 show snapshots of the diffusing-field-based cell-growth simulation. Figure 18 shows a sample screen output of the cell-division history.

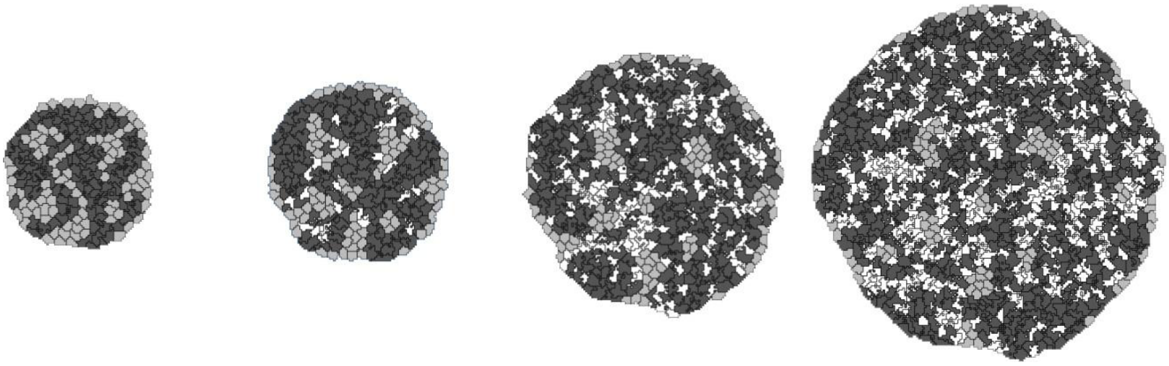


Figure 16: Snapshots of the diffusing-field-based cell-growth simulation obtained by running the CC3DML file in Listing 15 in conjunction with the Python file in Listing 16. As the simulation progresses, NonCondensing cells (light gray) secrete diffusing chemical, FGF, which causes Condensing (dark gray) cells to proliferate. Some Condensing cells differentiate to CondensingDifferentiated (white) cells.

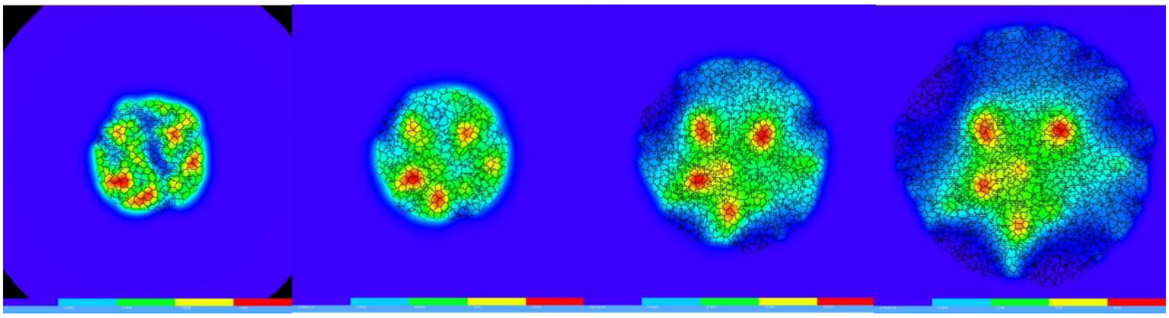
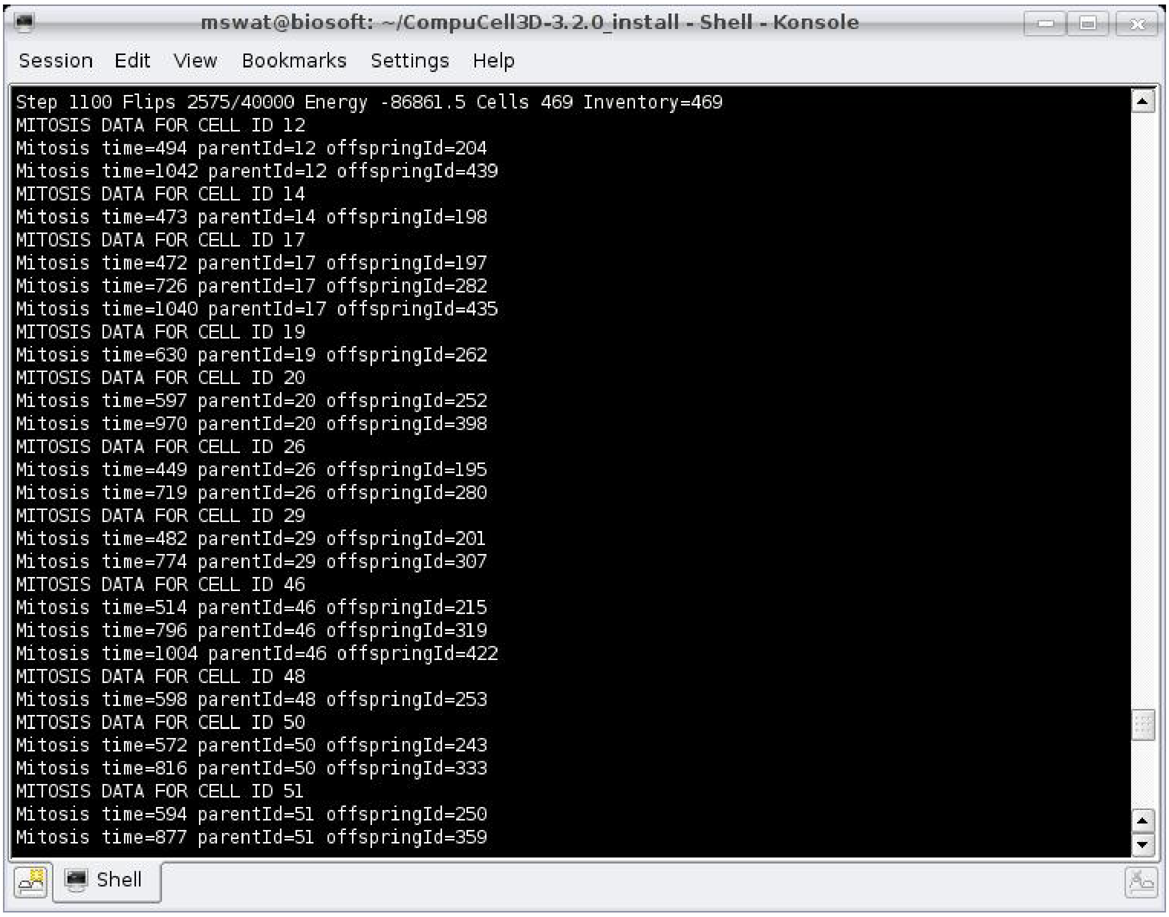


Figure 17: Snapshots of FGF concentration in the diffusing-field-based cell-growth simulation obtained by running the CC3DML file in Listing 15 in conjunction with the Python files in Listing 16, Listing 17, Listing 18, Listing 19, Listing 20. The bars at the bottom of the field images show the concentration scales (blue, low concentration; red, high concentration).

The diffusing-field-based cell-growth simulation includes concepts that extend easily to simulate biological phenomena that involve diffusants, cell growth and mitosis, *e.g.*, limb-bud development (58, 59), tumor growth (5-9) and *Drosophila* imaginal-disc development.


mswat@biosoft: ~/CompuCell3D-3.2.0_install - Shell - Konsole
Session
Edit
View
Bookmarks
Settings
Help
Shell

```
Step 1100 Flips 2575/40000 Energy -86861.5 Cells 469 Inventory=469
MITOSIS DATA FOR CELL ID 12
Mitosis time=494 parentId=12 offspringId=204
Mitosis time=1042 parentId=12 offspringId=439
MITOSIS DATA FOR CELL ID 14
Mitosis time=473 parentId=14 offspringId=198
MITOSIS DATA FOR CELL ID 17
Mitosis time=472 parentId=17 offspringId=197
Mitosis time=726 parentId=17 offspringId=282
Mitosis time=1040 parentId=17 offspringId=435
MITOSIS DATA FOR CELL ID 19
Mitosis time=630 parentId=19 offspringId=262
MITOSIS DATA FOR CELL ID 20
Mitosis time=597 parentId=20 offspringId=252
Mitosis time=970 parentId=20 offspringId=398
MITOSIS DATA FOR CELL ID 26
Mitosis time=449 parentId=26 offspringId=195
Mitosis time=719 parentId=26 offspringId=280
MITOSIS DATA FOR CELL ID 29
Mitosis time=482 parentId=29 offspringId=201
Mitosis time=774 parentId=29 offspringId=307
MITOSIS DATA FOR CELL ID 46
Mitosis time=514 parentId=46 offspringId=215
Mitosis time=796 parentId=46 offspringId=319
Mitosis time=1004 parentId=46 offspringId=422
MITOSIS DATA FOR CELL ID 48
Mitosis time=598 parentId=48 offspringId=253
MITOSIS DATA FOR CELL ID 50
Mitosis time=572 parentId=50 offspringId=243
Mitosis time=816 parentId=50 offspringId=333
MITOSIS DATA FOR CELL ID 51
Mitosis time=594 parentId=51 offspringId=250
Mitosis time=877 parentId=51 offspringId=359
```

Figure 18: Sample output from the MitosisDataPrinterSteppable steppable in Listing 20.

7 Conclusion

In most cases, building a complex CompuCell3D simulation requires writing Python modules, a main Python script and a CC3DML configuration file. While the effort to write this code can be substantial, it is much less than that required to develop custom simulations in lower-level languages. Working from the substantial base of Python templates provided by CompuCell3D further streamlines simulation development. Python programs are fairly short, so simulations can be published in journal articles, greatly facilitating simulation validation, reuse and adaptation. Finally, CompuCell3Ds modular structure allows new Python modules to be reused from simulation to simulation. The CompuCell3D website, www.compucell3d.org, allows users to archive their modules and make them accessible to other users.

We hope the examples we have shown will convince readers to evaluate the suitability of GGH simulations using CompuCell3D for their research. All the code examples presented in this chapter are available from www.compucell3d.org. They will be curated to ensure their correctness and compatibility with future versions of CompuCell3D.

8 Acknowledgements

We gratefully acknowledge support from the National Institutes of Health, National Institute of General Medical Sciences, grants 1R01 GM077138-01A1 and 1R01 GM076692-01, and the Office of Vice President for Research, the College of Arts and Sciences, the Pervasive Technologies Laboratories and the Biocomplexity Institute at Indiana University. Indiana Universitys University Information Technology Services provided time on their BigRed clusters for simulation execution. Early versions of CompuCell and CompuCell3D were developed at the University of Notre Dame by J.A.G., Dr. Mark Alber and Dr. Jesus Izaguirre and collaborators with the support of National Science Foundation, Division of Integrative Biology, grant IBN-00836563. Since the primary home of CompuCell3D moved to Indiana University in 2004, the Notre Dame team have continued to provide important support for its development.

9 XML Syntax of CompuCell3D modules

9.1 Potts Section

The first section of the .xml file defines the global parameters of the lattice and the simulation.

```
<Potts>
    <Dimensions x=”101” y=”101” z=”1”/>
    <Anneal>0</Anneal>
    <Steps>1000</Steps>
    <FluctuationAmplitude>5</FluctuationAmplitude>
    <Flip2DimRatio>1</Flip2DimRatio>
    <Boundary_y>Periodic</Boundary_y>
    <Boundary_x>Periodic</Boundary_x>
    <NeighborOrder>2</NeighborOrder>
    <DebugOutputFrequency>20</DebugOutputFrequency>
    <RandomSeed>167473</RandomSeed>
        <EnergyFunctionCalculator Type=”Statistics”>
            <OutputFileName Frequency=”10”>statData.txt</OutputFileName>
            <OutputCoreFileNameSpinFlips Frequency=”1” GatherResults=””
                OutputAccepted=”” OutputRejected=”” OutputTotal=””>
                statDataSingleFlip
            </OutputCoreFileNameSpinFlips>
        </EnergyFunctionCalculator>
```

```
</Potts>
```

This section appears at the beginning of the configuration file. Line `<Dimensions x="101" y="101" z="1"/>` declares the dimensions of the lattice to be 101 x 101 x 1, *i.e.*, the lattice is two-dimensional and extends in the xy plane. The basis of the lattice is 0 in each direction, so the 101 lattice sites in the x and y directions have indices ranging from 0 to 100. `<Steps>1000</Steps>` tells CompuCell how long the simulation lasts in MCS. After executing this number of steps, CompuCell can run simulation at zero temperature for an additional period. In our case it will run for `<Anneal>10</Anneal>` extra steps. FluctuationAmplitude parameter determines intrinsic fluctuation or motility of cell membrane. Fluctuation amplitude is a temperature parameter in classical GGH model formulation. We have decided to use `FluctuationAmplitude` term instead of temperature because using word “temperature” to describe intrinsic motility of cell membrane was quite confusing.

In the above example, fluctuation amplitude applies to all cells in the simulation. To define fluctuation amplitude separately for each cell type we use the following syntax:

```
<FluctuationAmplitude>
    <FluctuationAmplitudeParameters CellType="Condensing" \
        FluctuationAmplitude="10"/>
    <FluctuationAmplitudeParameters CellType="NonCondensing" \
        FluctuationAmplitude="5"/>
</FluctuationAmplitude>
```

When CompuCell3D encounters expanded definition of FluctuationAmplitude it will use it in place of a global definition - `<FluctuationAmplitude>5</FluctuationAmplitude>`

To complete the picture CompUCell3D allows users to set fluctuation amplitude individually for each cell. Using Python scripting we write:

```
for cell in self.cellList:
    if cell.type==1:
        cell.fluctAmpl=20
```

When determining which value of fluctuation amplitude to use, CompuCell first checks if `fluctAmpl` is non-negative. If this is the case it will use this value as fluctuation amplitude. Otherwise it will check if users defined fluctuation amplitude for cell types using expanded XML definition and if so it will use those values as fluctuation amplitudes. Lastly it will resort to globally defined fluctuation amplitude (Temperature). Thus, it is perfectly fine to use FluctuationAmplitude XML tags and set `fluctAmpl` for certain cells. In such a case CompuCell3D will use `fluctAmpl` for cells for which users defined it and for all other cells it will use values defined in the XML.

In GGH model, the fluctuation amplitude is determined taking into account fluctuation amplitude of “source” (expanding) cell and destination (being overwritten) cell. Currently

CompuCell3D supports 3 type functions used to calculate resultant fluctuation amplitude (those functions take as argument fluctuation amplitude of “source” and “destination” cells and return fluctuation amplitude that is used in calculation of pixel-copy acceptance). The 3functions are `Min`, `Max`, and `ArithmeticAverage` and we can set them using the following option of the Potts section:

```
<Potts>
   <FluctuationAmplitudeFunctionName>
      Min
   </FluctuationAmplitudeFunctionName>
   ...
</Potts>
```

By default we use `Min` function. Notice that if you use global fluctuation amplitude definition (Temperature) it does not really matter which function you use. The differences arise when “source” and “destination” cells have different fluctuation amplitudes.

The above concepts are best illustrated by the following example:

```
<PythonScript>Demos/FluctuationAmplitude/FluctuationAmplitude.py\
</PythonScript>

<Potts>
   <Dimensions x="100" y="100" z="1"/>
   <Steps>10000</Steps>
   <FluctuationAmplitude>5</FluctuationAmplitude>
   <FluctuationAmplitudeFunctionName>ArithmeticAverage\
   </FluctuationAmplitudeFunctionName>
   <NeighborOrder>2</NeighborOrder>
</Potts>
```

Where in the XML section we define global fluctuation amplitude and we also use ArithmeticAverage function to determine resultant fluctuation amplitude for the pixel copy.

In python script we will periodically set higher fluctuation amplitude for lattice quadrants so that when running the simulation we can see that cells belonging to different lattice quadrants have different membrane fluctuations:

```
class FluctuationAmplitude(SteppableBasePy):
    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator,_frequency)
        self.quarters=[[0,0,50,50],[0,50,50,100],\
        [50,50,100,100],[50,0,100,50]]
        self.steppableCallCounter=0

    def step(self, mcs):
        quarterIndex=self.steppableCallCounter % 4
        quarter=self.quarters[quarterIndex]
        for cell in self.cellList:
            if cell.xCOM>=quarter[0] and cell.yCOM>=quarter[1] and\
            cell.xCOM<quarter[2] and cell.yCOM<quarter[3]:
                cell.fluctAmpl=50
            else:
            #this means CompuCell3D will use globally
            #defined FluctuationAmplitude
                cell.fluctAmpl=-1

        self.steppableCallCounter+=1
```

Assigning negative fluctuationAmplitude `cell.fluctAmpl=-1` is interpreted by CompuCell3D as a hint to use fluctuation amplitude defined in the XML.

The below section describes Temperature and CellMotility tags which are being deprecated (however for compatibility reasons we still support those):

The first section of the .xml file defines the global parameters of the lattice and the simulation.

```
<Potts>
   <Dimensions x="101" y="101" z="1"/>
   <Anneal>0</Anneal>
   <Steps>1000</Steps>
   <Temperature>5</Temperature>
   <Flip2DimRatio>1</Flip2DimRatio>
   <Boundary_y>Periodic</Boundary_y>
   <Boundary_x>Periodic</Boundary_x>
   <NeighborOrder>2</NeighborOrder>
   <DebugOutputFrequency>20</DebugOutputFrequency>
   <RandomSeed>167473</RandomSeed>
   <EnergyFunctionCalculator Type="Statistics">
      <OutputFileName Frequency="10">statData.txt</OutputFileName>
      <OutputCoreFileNameSpinFlips Frequency="1" GatherResults=""
      OutputAccepted="" OutputRejected="" OutputTotal="">statDataSingleFlip
      </OutputCoreFileNameSpinFlips>
   </EnergyFunctionCalculator>
</Potts>
```

This section appears at the beginning of the configuration file. Line `<Dimensions x="101" y="101" z="1"/>` declares the dimensions of the lattice to be 101 × 101 × 1, *i.e.*, the lattice is two-dimensional and extends in the xy plane. The basis of the lattice is 0 in each direction, so the 101 lattice sites in the x and y directions have indices ranging from 0 to 100. `<Steps>1000</Steps>` tells CompuCell how long the simulation lasts in MCS. After executing this number of steps, CompuCell can run simulation at zero temperature for an additional period. In our case it will run for `<Anneal>10</Anneal>` extra steps. Setting the temperature is as easy as writing `<Temperature>5</Temperature>`.

We can also set temperature (or in other words cell motility) individually for each cell type. The syntax to do this is following:

```
<CellMotility>
   <MotilityParameters  CellType="Condensing"  Motility="10"/>
   <MotilityParameters  CellType="NonCondensing"  Motility="5"/>
</CellMotility>
```

You may use it in the Potts section in place of `<Temperature>`.

Now, as you remember from the discussion about the difference between spin-flip attempts and MCS we can specify how many spin flips should be attempted in every MCS. We specify this number indirectly by specifying the `Flip2DimRatio` - `<Flip2DimRatio>1</Flip2DimRatio>`, which tells CompuCell that it should make 1× number of lattice sites attempts per MCS in our case one MCS is $101 \times 101 \times 1$ spin-flip attempts. To set $2.5 \times 101 \times 101 \times 1$ spin flip attempts per MCS you would write `<Flip2DimRatio>2.5</Flip2DimRatio>`.

The next line specifies the neighbor order. The higher neighbor order the longer the Euclidian distance from a given pixel. In previous versions of CompuCell3D we have been using `<FlipNeighborMaxDistance>` or `<Depth>` (in Contact energy plugins) flag to accomplish same task. Since now CompuCell3D supports two kinds of latices it would be inconvenient to change distances. It is much easier to think in terms n-th nearest neighbors. For the backwards compatibility we still support old flags but we discourage its use, especially that in the future we might support more than just two lattice types. Using nearest neighbor interactions may cause artefacts due to lattice anisotropy. The longer the interaction range, the more isotropic the simulation and the slower it runs. In addition, if the interaction range is comparable to the cell size, you may generate unexpected effects, since non-adjacent cells will contact each other.

On hex lattice those problems seem to be less severe and there 1st or 2nd nearest neighbor usually are sufficient.

The Potts section also contains tags called `<Boundary_y>` and `<Boundary_x>`.These tags impose boundary conditions on the lattice. In this case the x and y axes are periodic (`<Boundary_x>Periodic</Boundary_x>`) so that *e.g.* the pixel with $x = 0, y = 1, z = 1$ will neighbor the pixel with $x = 100, y = 1, z = 1$. If you do not specify boundary conditions CompuCell will assume them to be of type no-flux, *i.e.* lattice will not be extended. The conditions are independent in each direction, so you can specify any combination of boundary conditions you like.

`DebugOutputFrequency` is used to tell CompuCell3D how often it should output text information about the status of the simulation. This tag is optional.

`RandomSeed` is used to initialise random number generator. If you do not do this all simulations will use same sequence of random numbers. Something you may want to avoid in the real simulations but is very useful while debugging your models.

`EnergyFunctionCalculator` is another option of Potts object that allows users to output statistical data from the simulation for further analysis. The OutputFileName tag is used to specify the name of the file to which CompuCell3D will write average changes in energies returned by each plugins with corresponding standard deviations for those MCS whose values are divisible by the Frequency argument. Here it will write these data every 10 MCS.

A second line with `OutputCoreFileNameSpinFlips` tag is used to tell CompuCell3D to output energy change for every plugin, every spin flip for MCS' divisible by the frequency. Option `GatherResults=` will ensure that there is only one file written for accepted (`OutputAccepted`), rejected (`OutputRejected`)and accepted and rejected (`OutputTotal`) spin flips. If you will not specify `GatherResults` CompuCell3D will output separate files for different MCS's and depending on the `Frequency` you may end up with many files in your directory.

One option of the Potts section that we have not used here is the ability to customise acceptance function for Metropolis algorithm:

```
<Offset>-0.1</Offset>
<KBoltzman>1.2</KBoltzman>
```

This ensures that spin flips attempts that increase the energy of the system are accepted with probability $P = \exp^{-(\Delta E - \delta)/kT}$ where δ and k are specified by `Offset` and `KBoltzman` tags respectively. By default $\delta = 0$ and $k = 1$.

As an alternative to exponential acceptance function you may use a simplified version which is essentially 1 order expansion of the exponential:

$$P = 1 - \frac{E - \delta}{kT}$$

To be able to use this function all you need to do is to add the following line in the Potts section:

```
<AcceptanceFunctionName>FirstOrderExpansion</AcceptanceFunctionName>
```

9.1.1 Lattice Type

Early versions of CompuCell3D allowed users to use only square lattice. Most recent versions however, allow the simulation to be run on hexagonal lattice as well. To enable hexagonal lattice you need to put

```
<LatticeType>Hexagonal</LatticeType>
```

in the Potts section of the XML configuration file.

There are few things to be aware of. When using hexagonal lattice. Obviously your pixels are hexagons (2D) or rhombic dodecahedrons (3D) but what is more important is that surface or perimeter of the pixel (depending whether in 2D or 3D) is different than in the case of square pixel. The way CompuCell3D hex lattice implementation was done was that the volume of the pixel was constrained to be 1 regardless of the lattice type. Second, there is one to one correspondence between pixels of the square lattice and pixels of the hex lattice. Consequently we can come up with transformation equations which give positions of hex pixels as a function of square lattice pixel position:

$$\begin{aligned}
(x,y,z)_{\text{hex}} &= \left(x, \frac{\sqrt{3}}{2}y + \frac{\sqrt{3}}{3}, \frac{\sqrt{6}}{3}z\right) \text{ for } y \text{ odd} \wedge z \text{ odd} \\
(x,y,z)_{\text{hex}} &= \left(x + \frac{1}{2}, \frac{\sqrt{3}}{2}y + \frac{\sqrt{3}}{3}, \frac{\sqrt{6}}{3}z\right) \text{ for } y \text{ even} \wedge z \text{ odd} \\
(x,y,z)_{\text{hex}} &= \left(x, \frac{\sqrt{3}}{2}y, \frac{\sqrt{6}}{3}z\right) \text{ for } y \text{ odd} \wedge z \text{ even} \\
(x,y,z)_{\text{hex}} &= \left(x + \frac{1}{2}, \frac{\sqrt{3}}{2}y, \frac{\sqrt{6}}{3}z\right) \text{ for } y \text{ even} \wedge z \text{ even}
\end{aligned}$$

Based on the above facts one can work out how unit length and unit surface transform to the hex lattice. The conversion factors are given below:

For the 2D case, assuming that each pixel has unit volume, we get:

$$S_{\text{hex-unit}} = \sqrt{\frac{2}{3\sqrt{3}}} \approx 0.6204$$

$$L_{\text{hex-unit}} = \sqrt{\frac{2}{\sqrt{3}}} \approx 1.075$$

where $S_{\text{hex-unit}}$ denotes length of the hexagon and $L_{\text{hex-unit}}$ denotes a distance between centres of the hexagons. Notice that unit surface in 2D is simply a length of the hexagon side and surface area of the hexagon with side 'a' is:

$$S = 6\frac{\sqrt{3}}{4}a^2$$

In 3D we can derive the corresponding unit quantities starting with the formulae for Volume and surface of rhombic dodecahedron (12 hedra)

$$V = \frac{16}{9}\sqrt{3}a^3$$

$$S = 8\sqrt{2}a^2$$

where ’a’ denotes length of dodecahedron edge.

Constraining the volume to be one we get

$$a = \sqrt[3]{\frac{9V}{16\sqrt{3}}}$$

and thus unit surface is given by:

$$S_{\text{unit-hex}} = \frac{S}{12} = \frac{8\sqrt{2}}{12}\sqrt[3]{\frac{9V}{16\sqrt{3}}} \approx 0.445$$

and unit length by:

$$L_{\text{unit-hex}} = 2\frac{\sqrt{2}}{\sqrt{3}}a = 2\frac{\sqrt{2}}{\sqrt{3}}\sqrt[3]{\frac{9V}{16\sqrt{3}}} \approx 1.122$$

9.2 Plugins Section

In this section we overview XML syntax for all the plugins available in CompuCell3D. Plugins are either energy functions, lattice monitors or store user assigned data that CompuCell3D uses internally to configure simulation before it is run.

9.2.1 CellType Plugin

An example of the plugin that stores user assigned data that is used to configure simulation before it is run is a `CellType` Plugin. This plugin is responsible for defining cell types and storing cell type information. It is a basic plugin used by virtually every CompuCell simulation. The syntax is straight forward as can be seen in the example below:

```
<Plugin Name=”CellType”>
    <CellType TypeName=”Medium” TypeId=”0”/>
    <CellType TypeName=”Fluid” TypeId=”1”/>
    <CellType TypeName=”Wall” TypeId=”2” Freeze=””/>
</Plugin>
```

Here we have defined three cell types that will be present in the simulation: `Medium,Fluid,Wall`. Notice that we assign a number (`TypeId`) to every cell type. It is strongly recommended that `TypeIds` are consecutive positive integers (*e.g.*, 0,1,2,3...). `Medium` is traditionally given `TypeId`=0 but this is not a requirement. However every CC3D simulation must define `CellType` Plugin and include at least `Medium` specification.

Notice that in the example above cell type "Wall" has extra attribute `Freeze=""`. This attribute tells CompuCell that cells of "frozen" type will not be altered by spin flips. Freezing certain cell types is a very useful technique in constructing different geometries for simulations or for restricting ways in which cells can move. In the example below we have frozen cell types wall to create tube geometry for fluid flow studies.

9.2.2 Simple Volume and Surface Constraints

One of the most commonly used energy term in the GGH Hamiltonian is a term that restricts variation of single cell volume. Its simplest form can be coded as show below:

```
<Plugin Name=”Volume”>
    <TargetVolume>25</TargetVolume>
    <LambdaVolume>2.0</LambdaVolume>
</Plugin>
```

By analogy we may define a term which will put similar constraint regarding the surface of the cell:

```
<Plugin Name=”Surface”>
    <TargetSurface>20</TargetSurface>
    <LambdaSurface>1.5</LambdaSurface>
</Plugin>
```

These two plugins inform CompuCell that the Hamiltonian will have two additional terms associated with volume and surface conservation. That is when spin flip is attempted one cell will increase its volume and another cell will decrease. Thus overall energy of the system may or will change. Volume constraint essentially ensures that cells maintain the volume which close (this depends on thermal fluctuations) to target volume. The role of surface plugin is analogous to volume, that is to "preserve" surface. Note that surface plugin is commented out in the example above.

Energy terms for volume and surface constraints have the form:

$$\begin{aligned} E_{\text{volume}} &= \lambda_{\text{volume}}(v_{\text{cell}} - V_{\text{target}})^2 \\ E_{\text{surface}} &= \lambda_{\text{surface}}(s_{\text{cell}} - S_{\text{target}})^2 \end{aligned}$$

Remark:
Notice that flipping a single spin may cause surface change in more that two cells - this is especially true in 3D.

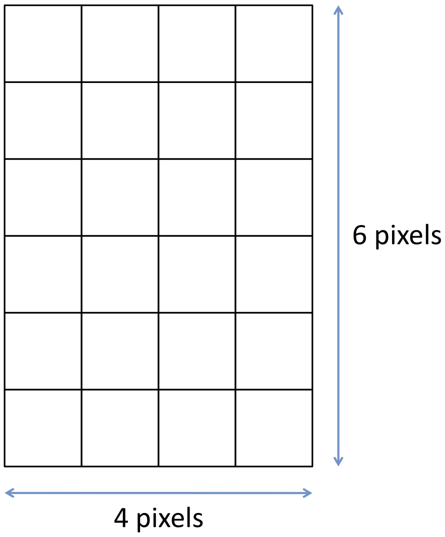


Figure 19: How to set target volume and target surface.

VAdd: Target volume is the total number of the pixels inside of the cell, that is a multiplication of the length and width. For example, in Fig. 19, the target volume of the cell size 4×6 pixels is 24. And the target surface is $4 + 4 + 6 + 6 = 20$.

9.2.3 VolumeTracker and SurfaceTracker plugins

These two plugins monitor lattice and update volume and surface of the cells once spin flip occurs. In most cases users will not call those plugins directly. They will be called automatically when either `Volume` (calls `VolumeTracker`) or `Surface` (calls `SurfaceTracker`) or `CenterOfMass` (calls `VolumeTracker`) plugins are requested. However one should be aware that in some situations, for example when doing foam coarsening simulation as presented in the introduction, when neither `Volume` or `Surface` plugins are called, one may still want to track changes ion surface or volume of cells . In such situations one can explicitly invoke `VolumeTracker` or `SurfaceTracker` plugin with the following syntax:

```
<Plugin Name="VolumeTracker"/>
```

```
<Plugin Name="SurfaceTracker"/>
```

9.2.4 VolumeFlex Plugin

VolumeFlex plugin is more sophisticated version of Volume plugin. While Volume plugin treats all cell types the same, *i.e.*, they all have the same target volume and lambda coefficient, VolumeFlex plugin allows you to assign different lambda and different target volume to different cell types. The syntax for this plugin is straightforward and essentially mimics the example below.

```
<Plugin Name="VolumeFlex">
    <VolumeEnergyParameters CellType="Prestalk" TargetVolume="68"
        LambdaVolume="15"/>
    <VolumeEnergyParameters CellType="Prespore" TargetVolume="69"
        LambdaVolume="12"/>
    <VolumeEnergyParameters CellType="Autocycling" TargetVolume="80"
        LambdaVolume="10"/>
    <VolumeEnergyParameters CellType="Ground" TargetVolume="0"
        LambdaVolume="0"/>
    <VolumeEnergyParameters CellType="Wall" TargetVolume="0"
        LambdaVolume="0"/>
</Plugin>
```

Notice that in the example above cell types Wall and Ground have target volume and coefficient lambda set to 0 - very unusual. That's because in this particular those cells are were frozen so the parameters specified for these cells do not matter. In fact it is safe to remove specifications for these cell types, but just for the illustration purposes we left them.

Using VolumeFlex plugin you can effectively freeze certain cell types. All you need to do is to put very high lambda coefficient for the cell type you wish to freeze. You have to be careful though , because if initial volume of the cell of a given type is different from target volume for this cell type the cells will either shrink or expand to match target volume (this is out of control and you should avoid it), and only after this initial volume adjustment will they remain frozen. That is provided LambdaVolume is high enough. In any case, we do not recommend this way of freezing cells because it is difficult to use, and also not efficient in terms of speed of simulation run.

9.2.5 SurfaceFlex Plugin

SurfaceFlex plugin is more sophisticated version of Surface plugin. Everything that was said with respect to VolumeFlex plugin applies to SurfaceFlex. For syntax see example below:

```
<Plugin Name="SurfaceFlex">
    <SurfaceEnergyParameters CellType="Prestalk" TargetSurface="90"
        LambdaSurface="0.15"/>
    <SurfaceEnergyParameters CellType="Prespore" TargetSurface="98"
        LambdaSurface="0.15"/>
    <SurfaceEnergyParameters CellType="Autocycling" TargetSurface="92"
        LambdaSurface="0.1"/>
    <SurfaceEnergyParameters CellType="Ground" TargetSurface="0"
        LambdaSurface="0"/>
    <SurfaceEnergyParameters CellType="Wall" TargetSurface="0"
        LambdaSurface="0"/>
</Plugin>
```

9.2.6 VolumeLocalFlex Plugin

`VolumeLocalFlex` plugin is very similar to `Volume` plugin. You specify both lambda coefficient and target volume, but as opposed to `Volume` Plugin the energy is calculated using target volume and lambda volume that are specified individually for each cell. In the course of simulation you can change this target volume depending on *e.g.*, concentration of FGF in the particular cell. This way you can specify which cells grow faster, which slower based on a state of the simulation. This plugin requires you to develop a module (plugin or steppable) which will alter target volume for each cell. You can do it either in C++ or even better in Python.

Example syntax:

```
<Plugin Name="VolumeLocalFlex"/>
```

9.2.7 SurfaceLocalFlex Plugin

This plugin is analogous to `VolumeLocalFlex` but operates on cell surface.

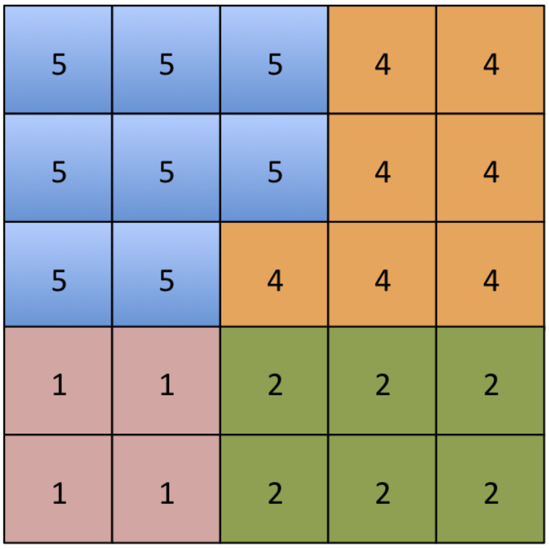
Example syntax:

```
<Plugin Name="SurfaceLocalFlex"/>
```

9.2.8 NeighborTracker Plugin

This plugin, as its name suggests, tracks neighbors of every cell. In addition it calculates common contact area between cell and its neighbors. We consider a neighbor this cell that has at least one common pixel side with a given cell. This means that cells that touch each other either “by edge” or by “corner” are not considered neighbors. See Fig 20.

Example syntax:



5	5	5	4	4
5	5	5	4	4
5	5	4	4	4
1	1	2	2	2
1	1	2	2	2

Figure 20: Cells 5, 4, 1 are considered neighbors as they have non-zero common surface area. Same applies to pair of cells 4, 2 and to 1 and 2. However, cells 2 and 5 are not neighbors because they touch each other "by corner". Notice that cell 5 has 8 pixels, cell 4 has 7 pixels, cell 1 has 4 pixels, and cell 2 has 6 pixels.

```
<Plugin Name=”NeighborTracker”/>
```

This plugin is used as a helper module by other plugins and steppables *e.g.*, `Elasticity` and `AdvectionDiffusionSolver` use NeighborTracker plugin.

9.2.9 Chemotaxis

Chemotaxis plugin , as its name suggests is used to simulate chemotaxis of cells. For every spin flip this plugin calculates change of energy associated with pixel move. There are several methods to define a change in energy due to chemotaxis. By default we define a chemotaxis using the following formula:

$$\Delta E_{\text{chem}} = \lambda(c(\vec{x}_{\text{neighbor}}) - c(\vec{x}))$$

where $c(\vec{x}_{\text{neighbor}})$, $c(\vec{x})$ denote chemical concentration at the spin-flip-source and spin-flip destination pixel. respectively.

We also support a slight modification of the above formula in the `Chemotaxis` plugin where ΔE is non-zero only if the cell located at $\vec{x}$ after the spin flip is non-medium to enable such mode users need to include `<Algorithm="Regular"/>` tag in the body of XML plugin.

Let's look at the syntax by studying the example usage of the `Chemotaxis` plugin:

```
<Plugin Name=”Chemotaxis”>
    <ChemicalField Source=”FlexibleDiffusionSolverFE” Name=”FGF”>
        <ChemotaxisByType Type=”Amoeba” Lambda=”300”/>
        <ChemotaxisByType Type=”Bacteria” Lambda=”200”/>
    </ChemicalField>
</Plugin>
```

The body of the chemotaxis plugin description contains sections called `ChemicalField`. In this section you tell CompuCell3D which module contains chemical field that you wish to use for chemotaxis. In our case it is `FlexibleDiffusionSolverFE`. Next, you need to specify the name of the field - `FGF` in our case. Next, you specify lambda for each cell type so that cells of different type may respond differently to a given chemical. In particular types not listed will not respond to chemotaxis at all. Older versions of CompuCell3D allowed for different syntaxes as well. Despite the fact that those syntaxes are still supported for backward compatibility reasons, we discourage their use, because, they are somewhat confusing.

Occasionally you may want to use different formula for the chemotaxis than the one presented above. Current CompCell3D allows you to use the following definitions of change in chemotaxis energy (`Saturation` and `SaturationLinear` respectively):

$$\Delta E_{\text{chem}} = \lambda \left(\frac{c(\vec{x}_{\text{neighbor}})}{s + c(\vec{x}_{\text{neighbor}})} - \frac{c(\vec{x})}{s + c(\vec{x})} \right)$$

or

$$\Delta E_{\text{chem}} = \lambda \left(\frac{c(\vec{x}_{\text{neighbor}})}{(s \cdot c(\vec{x}_{\text{neighbor}}) + 1)} - \frac{c(\vec{x})}{(s \cdot c(\vec{x}) + 1)} \right)$$

where ’s’ denotes saturation constant. To use first of the above formulas all you need to do is to let CompuCell3D know the value of the saturation coefficient:

```
<Plugin Name=”Chemotaxis”>
    <ChemicalField Source=”FlexibleDiffusionSolverFE” Name=”FGF”>
        <ChemotaxisByType Type=”Amoeba” Lambda=”0”/>
        <ChemotaxisByType Type=”Bacteria” Lambda=”2000000”
            SaturationCoef=”1”/>
    </ChemicalField>
</Plugin>
```

Notice that this only requires small change in line where you previously specified only lambda.

```
<ChemotaxisByType Type=”Bacteria” Lambda=”2000000” SaturationCoef=”1”/>
```

To use second of the above formulas use `SaturationLinearCoef` instead of `SaturationCoef`:

```
<Plugin Name=”Chemotaxis”>
```

```
    <ChemicalField Source="FlexibleDiffusionSolverFE" Name="FGF">
        <ChemotaxisByType Type="Amoeba" Lambda="0"/>
        <ChemotaxisByType Type="Bacteria" Lambda="2000000"
            SaturationLinearCoef="1"/>
    </ChemicalField>
</Plugin>
```

Sometimes it is desirable to have chemotaxis between only certain types of cells and not between other pairs of types. To deal with this situation it is enough to augment `ChemotaxisByType` element with the following attribute:

```
<ChemotaxisByType Type="Amoeba" Lambda="100" ChemotactTowards="Medium"/>
```

This will cause that the change in chemotaxis energy will be non-zero only for those spin flip attempts that will try to slip `Amoeba` and `Medium` pixels.

The definitions of chemotaxis presented so far do not allow specification of chemotaxis parameters individually for each cell. To do this we will use Python scripting. We still need to specify in the XML which fields are important from chemotaxis stand point. Only fields listed in the XML will be used to calculate chemotaxis energy:

```
<Plugin Name="CellType">
    <CellType TypeName="Medium" TypeId="0"/>
    <CellType TypeName="Bacterium" TypeId="1" />
    <CellType TypeName="Macrophage" TypeId="2"/>
    <CellType TypeName="Wall" TypeId="3" Freeze=""/>
</Plugin>

<Plugin Name="Chemotaxis">
    <ChemicalField Source="FlexibleDiffusionSolverFE" Name="ATTR">
        <ChemotaxisByType Type="Macrophage" Lambda="20"/>
    </ChemicalField>
</Plugin>
```

In the above excerpt from the XML configuration file we see that cells of type `Macrophage` will chemotax in response to `ATTR` gradient.

Using Python scripting we can modify chemotaxis properties of individual cells as follows:

```
class ChemotaxisSteering(SteppableBasePy):
    def __init__(self,_simulator,_frequency=100):
        SteppableBasePy.__init__(self,_simulator,_frequency)

    def start(self):
        for cell in self.cellList:
            if cell.type==2:
                cd=self.chemotaxisPlugin.addChemotaxisData(cell,"ATTR")
                cd.setLambda(20.0)
                #cd.initializeChemotactTowardsVectorTypes("Bacterium,Medium")
                cd.assignChemotactTowardsVectorTypes([0,1])
                break

    def step(self,mcs):
        for cell in self.cellList:
            if cell.type==2:
                cd=self.chemotaxisPlugin.getChemotaxisData(cell,"ATTR")
                if cd:
                    l=cd.getLambda()-3
                    cd.setLambda(l)
                break
```

In the `start` function for first encountered cell of type Macrophage (`type==2`) we insert `ChemotaxisData` object (it determines chemotaxis properties) and initialize λ parameter to 20. We also initialize vector of cell types towards which Macrophage cell will chemotax (it will chemotax towards Medium and Bacterium cells). Notice the break statement inside the `if` statement, inside the loop. It ensures that only first encountered Macrophage cell will have chemotaxing properties altered. In the step function we decrease lambda chemotaxis by 3 units every 100 MCS. In effect we turn a cell from chemotaxing up ATTR gradient to being chemorepelled.

In the above example we have more than one macrophage but only one of them has altered chemotaxis properties. The other macrophages have chemotaxis properties set in the XML section. CompuCell3D first checks if local definitions of chemotaxis are available (*i.e.*, for individual cells) and if so it uses those. Otherwise it will use definitions from from the XML.

The ChemotaxisData structure has additional functions which allow to set chemotaxis formula used. For example we may type:

```
def start(self):
    for cell in self.cellList:
        if cell.type==2:
            cd=self.chemotaxisPlugin.addChemotaxisData(cell,"ATTR")
            cd.setLambda(20.0)
            cd.setSaturationCoef(200.0)
            #cd.initializeChemotactTowardsVectorTypes("Bacterium,Medium")
            cd.assignChemotactTowardsVectorTypes([0,1])
            break
```

to activate Saturation formula. To activate SaturationLinear formula we would use:

```
cd.setSaturationLinearCoef(2.0)
```

CAUTION: when you use chemotaxis plugin you have to make sure that fields that you refer to and module that contains this fields are declared in the xml file. Otherwise you will most likely cause either program crash (which is not as bad as it sounds) or unpredicted behavior (much worse scenario, although unlikely as we made sure that in the case of undefined symbols, CompuCell3D exits)

9.2.10 ExternalPotential plugin

Chemotaxis plugin is used to cause directional cell movement. Another way to achieve directional movement is to use ExternalPotential plugin. This plugin is responsible for imposing a directed pressure (or rather force) on cells. It is used mainly in fluid flow studies with periodic boundary conditions along these coordinates along which force acts. If `NoFlux` boundary conditions are set instead , the cells will be squeezed.

This is the example usage of this plugin:

```
<Plugin Name="ExternalPotential">
    <Lambda x="-0.5" y="0.0" z="0.0"/>
</Plugin>
```

`Lambda` is a vector quantity and determines components of force along three axes. In this case we apply force along x.

We can also apply external potential to specific cell types:

```
<Plugin Name="ExternalPotential">
    <ExternalPotentialParameters CellType="Body1" x="-10" y="0" z="0"/>
    <ExternalPotentialParameters CellType="Body2" x="0" y="0" z="0"/>
    <ExternalPotentialParameters CellType="Body3" x="0" y="0" z="0"/>
</Plugin>
```

Where in `ExternalPotentialParameters` we specify which cell type is subject to external potential (`Lambda` is specified using `x,y,z` attributes).

We can also apply external potential to individual cells. In that case, in the XML section we only need to specify:

```
<Plugin Name="ExternalPotential"/>
```

and in the Python file we change `lambdaVecX, lambdaVecY, lambdaVecZ`, which are properties of cell. For example in Python we could write:

```
cell.lambdaVecX=-10
```

Calculations done by ExternalPotential Plugin are by default based on direction of pixel copy (similarly as in chemotaxis plugin). One can however force CC3D to do calculations based on movement of center of mass of cell. To use algorithm based on center of mass movement we use the following XML syntax:

```
<Plugin Name=”ExternalPotential”>
    <Algorithm>CenterOfMassBased</Algorithm>
     ...
</Plugin>
```

Remark: Note that in the pixel-based algorithm the typical value of pixel displacement used in calculations is of the order of 1 (pixel) whereas typical displacement of center of mass of cell due to single pixel copy is of the order of 1/cell volume (pixels) - $\sim$ 0.1 pixel. This implies that to achieve compatible behavior of cells when using center of mass algorithm we need to multiply lambdas by appropriate factor, typically of the order of 10.

9.2.11 CellOrientation plugin

Similarly as ExternalPotential plugin this plugin gives preference to those pixel copies whose direction aligns with polarization vector (which is a property of each cell):

$$\Delta E = -\lambda(\sigma(i)) * \vec{p}(\sigma(i)) \cdot \vec{c},$$

where $\sigma(i)$ denotes cell at site i, $\vec{p}$ is polarization vector for cell at site i and $\vec{c}$ pixel copy vector. Because two cell participate in the pixel copy process the net energy change is simply a sum of above expressions: one for growing cell and one for shrinking cell. To set lambda we have two options: use global setting in the XML:

```
<Plugin Name=”CellOrientation”>
    <LambdaCellOrientation>0.5</LambdaCellOrientation>
</Plugin>
```

Or set λ individually for each cell and manage values of λ from Python. In this case we use the following XML syntax:

```
<Plugin Name=”CellOrientation”>
    <LambdaFlex/>
</Plugin>
```

or equivalently the shorter version:

```
<Plugin Name=”CellOrientation”/>
```

If we manage λ values in Python we would use the following syntax to access and modify values of lambda:

```
self.cellOrientationPlugin.getLambdaCellOrientation(cell)
```

```
self.cellOrientationPlugin.setLambdaCellOrientation(cell ,0.5)
```

Calculations done by CellOrientation Plugin are by default based on direction of pixel copy (similarly as in chemotaxis plugin). One can however force CC3D to do calculations based on movement of center of mass of cell. To use algorithm based on center of mass movement we use the following XML syntax:

```
<Plugin Name=”CellOrientation”>
    <Algorithm>CenterOfMassBased</Algorithm>
    ...
</Plugin>
```

See remark in External potential description about rescaling of parameters when changing algorithm to Center Of Massbased.

9.2.12 PolarizationVector plugin

PolarizationVector plugin is a simple plugin whose only task is to ensure that each cell in CompuCell3D simulation has as its attribute 3-component vector of floating point numbers. This plugin is normally used in together with CellOrientation but it also can be reused in other applications, assuming that we do not use CellOrientation plugin at the same time. The XML syntax is very simple:

```
<Plugin Name=”PolarizationVector”/>
```

To access or modify polarization vector requires use of Python scripting:

```
self.polarizationVectorPlugin.getPolarizationVector(cell)
```

or to change values of the polarization vector:

```
self.polarizationVectorPlugin.getPolarizationVector(cell,0.1,0.2,0.3)
```

9.2.13 CenterOfMass plugin

This plugin monitors changes n the lattice and updates centroids of the cell:

$$x_{CM} = \sum_i x_i\,, \quad y_{CM} = \sum_i y_i\,, \quad z_{CM} = \sum_i z_i$$

where i denotes pixels belonging to a given cell. To obtain coordinates of a center of mass f a given cell you need to divide centroids by cell volume:

$$X_{CM} = \frac{x_{CM}}{V}\,, \quad Y_{CM} = \frac{y_{CM}}{V}\,, \quad Z_{CM} = \frac{z_{CM}}{V}\,.$$

This plugin is aware of boundary conditions and centroids are calculated properly regardless which boundary conditions are used. The XML syntax is very simple:

```
<Plugin Name=”CenterOfMass”/>
```

9.2.14 Contact Energy plugin

Energy calculations for the foam simulation are based on the boundary or contact energy between cells (or surface tension, if you prefer). Together with volume constraint contact energy is one of the most commonly used energy terms in the GGH Hamiltonian. In essence it describes how cells “stick” to each other.

The explicit formula for the energy is:

$$E_{\text{adhesion}} = \sum_{\text{i,j neighbors}} J(\tau(\sigma(i)), \tau(\sigma(j)))(1 - \delta(\sigma(i), \sigma(j)))$$

where i and j label two neighboring lattice sites , σ's denote cell Ids, τ's denote cell types. In the case of foam simulation the total energy of the foam is simply the total boundary length times the surface tension (here defined to be $2J$).

Once again, in the above formula, you need to differentiate between cell types and cell Ids. This formula shows that cell types and cell Ids are not the same. The Contact plugin in the .xml file, defines the energy per unit area of contact between cells of different types $J(\tau(\sigma(i)), \tau(\sigma(j)))$ and the interaction range (`NeighborOrder`) of the contact:

```
<Plugin Name=”Contact”>
    <Energy Type1=”Foam” Type2=”Foam”>3</Energy>
    <Energy Type1=”Medium” Type2=”Medium”>0</Energy>
    <Energy Type1=”Medium” Type2=”Foam”>0</Energy>
    <NeighborOrder>2</NeighborOrder>
</Plugin>
```

In this case, the interaction range is 2, thus only up to second nearest neighbor pixels of a pixel undergoing a change or closer will be used to calculate contact energy change. Foam cells have contact energy per unit area of 3 and Foam and Medium as well as Medium and Medium have contact energy of 0 per unit area.

9.2.15 ContactLocalProduct plugin

This plugin calculates contact energy based on local (*i.e.*, per cell) cadherin expression levels. This plugin has to be used in conjunction with a steppable that assigns cadherin expression levels to the cell. Such steppables are usually written in Python - see ContactLocalProductExample in Demos directory.

We use the following formulas to calculate energy for this plugin:

$$E = \sum_{\text{i,j neighbors}} (E_{\text{offset}} - k_{\sigma(i),\sigma(j)} f(N(i),N(j))) \text{ if } \sigma(i) \text{ and } \sigma(j) \neq \text{medium}$$

$$E = \sum_{\text{i,j neighbors}} (E_{\text{offset}} - k_{\sigma(i),\sigma(j)}) \text{ if } \sigma(i) \text{ or } \sigma(j) = \text{medium}$$

By default, $E_{\text{offset}} = 0$. The $f(N(i),N(j))$ is a function of cadherins and can be either a simple product $N(i)N(j)$, a product of squared expression levels $N(i)^2N(j)^2$ or a $\min(N(i),N(j))$.

In the case of the second formula $E_{\text{offset}} - k_{\sigma(i),\sigma(j)}$ plays the role of “regular” contact energy between cell and medium.

The syntax of this plugin is as follows:

```
<Plugin Name="ContactLocalProduct">
  <ContactSpecificity Type1="Medium" Type2="Medium">0</ContactSpecificity>
  <ContactSpecificity Type1="Medium" Type2="CadExpLevel1">-16</ContactSpecificity>
  <ContactSpecificity Type1="Medium" Type2="CadExpLevel2">-16</ContactSpecificity>
  <ContactSpecificity Type1="CadExpLevel1" Type2="CadExpLevel1">-2</ContactSpecificity>
  <ContactSpecificity Type1="CadExpLevel1" Type2="CadExpLevel2">2.75</ContactSpecificity>
  <ContactSpecificity Type1="CadExpLevel2" Type2="CadExpLevel2">-1</ContactSpecificity>
  <ContactFunctionType>Quadratic</ContactFunctionType>
  <EnergyOffset>0.0</EnergyOffset>
  <NeighborOrder>2</NeighborOrder>
</Plugin>
```

Users need to specify ContactSpecificity ($k_{\sigma(i),\sigma(j)}$) between different cell types `ContactFunctionType` (by default it is set to `Linear` - $N(i)N(j)$ but other allowed key words are `Quadratic` - $N(i)^2N(j)^2$ and `Min` - $\min(N(i)N(j)))$. `EnergyOffset` can be set to user specified value using above syntax. `NeighborOrder` has the same meaning as for “regular” Contact plugin.

Alternatively one can write customized function of the two cadherins and use it instead of the 3 choices given above. To do this, simply use the following syntax:

```
<Plugin Name="ContactLocalProduct">
  <ContactSpecificity Type1="Medium" Type2="Medium">0</ContactSpecificity>
  <ContactSpecificity Type1="Medium" Type2="CadExpLevel1">-16</ContactSpecificity>
  <ContactSpecificity Type1="Medium" Type2="CadExpLevel2">-16</ContactSpecificity>
  <ContactSpecificity Type1="CadExpLevel1" Type2="CadExpLevel1">-2</ContactSpecificity>
  <ContactSpecificity Type1="CadExpLevel1" Type2="CadExpLevel2">2.75</ContactSpecificity>
  <ContactSpecificity Type1="CadExpLevel2" Type2="CadExpLevel2">-1</ContactSpecificity>
  <ContactFunctionType>Quadratic</ContactFunctionType>
  <EnergyOffset>0.0</EnergyOffset>
  <NeighborOrder>2</NeighborOrder>
  <CustomFunction>
    <Variable>J1</Variable>
    <Variable>J2</Variable>
    <Expression>sin(J1*J2)</Expression>
  </CustomFunction>
</Plugin>
```

Here we define variable names for cadherins in interacting cells ($J1$ denotes cadherin for one of the cells and $J2$ denotes cadherin for another cell). Then in the `Expression` tag we give mathematical expression involving the two cadherin levels. The expression syntax has to follow syntax of the muParser http://muparser.sourceforge.net/mup_features.html#idDef2.

9.2.16 AdhesionFlex plugin

Adhesion Flex is a generalization of ContactLocalProduct plugin. It allows setting individual adhesivity properties for each cell. Users can use either XML syntax or Python scripting to initialize adhesion molecule density for each cell. In addition, Medium can also carry its own adhesion molecules. We use the following formula to calculate Contact energy in AdhesionFlex plugin:

$$E = \sum_{i,j \text{ neighbors}} \left(- \sum_{m,n} k_{m,n} F(N_m(i), N_n(j)) \right) (1 - \delta_{\sigma(i),\sigma(j)})$$

where indexes i, j label pixels, $J(\sigma(i), \sigma(j))$ denotes contact energy between cell types $\sigma(i)$ and $\sigma(j)$, exactly as in “regular” contact plugin and indexes m, n label cadherins in cells composed of pixels i and j respectively. F denotes user-defined function of N_m and N_n. Although this may look a bit complex, the basic idea is simple: each cell has certain number of cadherins on its surface. When cell touch each other the resultant energy is simply a “product” - $k_{m,n} F(N_m(i), N_n(j))$ - of every cadherin from one cell with every cadherin from another cell.The XML syntax for this plugin is given below:

```
<Plugin Name="AdhesionFlex">
    <AdhesionMolecule Molecule="NCad"/>
    <AdhesionMolecule Molecule="NCam"/>
    <AdhesionMolecule Molecule="Int"/>
    <AdhesionMoleculeDensity CellType="Cell1" Molecule="NCad" Density="6.1"/>
    <AdhesionMoleculeDensity CellType="Cell1" Molecule="NCam" Density="4.1"/>
    <AdhesionMoleculeDensity CellType="Cell1" Molecule="Int" Density="8.1"/>
    <AdhesionMoleculeDensity CellType="Medium" Molecule="Int" Density="3.1"/>
    <AdhesionMoleculeDensity CellType="Cell2" Molecule="NCad" Density="2.1"/>
    <AdhesionMoleculeDensity CellType="Cell2" Molecule="NCam" Density="3.1"/>

    <BindingFormula Name="Binary">
        <Formula> min(Molecule1,Molecule2)</Formula>
        <Variables>
            <AdhesionInteractionMatrix>
                <BindingParameter Molecule1="NCad" Molecule2="NCad" > -1.0</BindingParameter>
                <BindingParameter Molecule1="NCam" Molecule2="NCam"> 2.0</BindingParameter>
                <BindingParameter Molecule1="NCad" Molecule2="NCam" > -10.0</BindingParameter>
                <BindingParameter Molecule1="Int" Molecule2="Int" > -10.0</BindingParameter>
            </AdhesionInteractionMatrix>
        </Variables>
    </BindingFormula>
    <NeighborOrder>2</NeighborOrder>
</Plugin>
```

k_{mn} matrix is specified within the `AdhesionInteractionMatrix` tag - the elements are listed using BindingParameter tags. The `AdhesionMoleculeDensity` tag specifies initial concentration of adhesion molecules. Even if you are going to modify those from Python (in the start function of the steppable) you are still required to specify the names of adhesion molecules and associate them with appropriate cell types. Failure to do so may result in simulation crash or undefined behaviors. The user-defined function F is specified using `Formula` tag where the arguments of the function are called `Molecule1` and `Molecule2`. The syntax has to follow syntax of the muParser -
http://muparser.sourceforge.net/mup_features.html#idDef2.

CompuCell3D example - *Demos/AdhesionFlex* - demonstrates how to manipulate concentration of adhesion molecules:

```
self.adhesionFlexPlugin.getAdhesionMoleculeDensity(cell,”NCad”)
```

allows to access adhesion molecule concentration using its name (as given in the XML above using `AdhesionMoleculeDensity` tag).

```
self.adhesionFlexPlugin.getAdhesionMoleculeDensityByIndex(cell,1)
```

allows to access adhesion molecule concentration using its index in the adhesion molecule density vector. The order of the adhesion molecule densities in the vector is the same as the order in which they were declared in the XML above - `AdhesionMoleculeDensity` tags.

```
self.adhesionFlexPlugin.getAdhesionMoleculeDensityVector(cell)
```

allows access to entire adhesion molecule density vector.

Each of these functions has its corresponding function which operates on `Medium`. In this case we do not give `cell` as first argument:

```
self.adhesionFlexPlugin.getMediumAdhesionMoleculeDensity(”Int”)
```

```
self.adhesionFlexPlugin.getMediumAdhesionMoleculeDensityByIndex(0)
```

```
self.adhesionFlexPlugin.getMediumAdhesionMoleculeDensityVector(cell)
```

To change the value of the adhesion molecule density we use `set` functions:

```
self.adhesionFlexPlugin.setAdhesionMoleculeDensity(cell,”NCad”,0.1)
self.adhesionFlexPlugin.setAdhesionMoleculeDensityByIndex(cell,1,1.02)
```

```
self.adhesionFlexPlugin.setAdhesionMoleculeDensityVector(cell, \
[3.4,2.1,12.1])
```

Notice that in this las function we passed entire Python list as the argument. CC3D will check if the number of entries in this vector is the same as the number of entries in the currently used vector. If so the values from the passed vector will be copied, otherwise they will be ignored.

IMPORTANT: during mitosis we create new cell (`childCell`) and the adhesion molecule vector of this cell will have no components. However in order for simulation to continue we have to initialize this vector with number of cadherins appropriate to childCell type. We know that `setAdhesionMoleculeDensityVector` is not appropriate for this task so we have to use:

```
self.adhesionFlexPlugin.assignNewAdhesionMoleculeDensityVector(cell, \
[3.4,2.1,12.1])
```

which will ensure that the content of passed vector is copied entirely into cells vector (making size adjustments as necessary).

IMPORTANT: You have to make sure that the number of newly assigned adhesion molecules is exactly the same as the number of adhesion molecules declared for the cell of this particular type.

All of `get` functions has corresponding `set` function which operates on `Medium`:

```
self.adhesionFlexPlugin.setMediumAdhesionMoleculeDensity(”NCam”,2.8)
```

```
self.adhesionFlexPlugin.setMediumAdhesionMoleculeDensityByIndex(2,16.8)
```

```
self.adhesionFlexPlugin.setMediumAdhesionMoleculeDensityVector(\
[1.4,3.1,18.1])
```

```
self.adhesionFlexPlugin.assignNewMediumAdhesionMoleculeDensityVector(\
[1.4,3.1,18.1])
```

9.2.17 ContactMultiCad plugin

ContactMultiCad plugin is a modified version of ContactLocalProduct plugin. In this case users can use several cadherins and describe how they translate into contact energy. The energy formula used by this plugin is given below:

$$E = \sum_{i,j \text{ neighbors}} \left(E_{\text{offset}} + J(\sigma(i), \sigma(j)) - \sum_{m,n} k_{mn} N_m(i) N_n(j) \right)$$

where indexes i, j label pixels, $J(\sigma(i), \sigma(j))$ denotes contact energy between cell types $\sigma(i)$ and $\sigma(j)$, exactly as in “regular” contact plugin and indexes m, n label cadherins in cells composed of pixels i and j respectively.

The syntax for this plugin is as follows:

```
<Plugin Name=”ContactMultiCad”>
   <Energy Type1=”Medium” Type2=”CadExpLevel1”>0</Energy>
   <Energy Type1=”Medium” Type2=”CadExpLevel2”>0</Energy>
   <Energy Type1=”CadExpLevel1” Type2=”CadExpLevel1”>0</Energy>
   <Energy Type1=”CadExpLevel1” Type2=”CadExpLevel2”>0</Energy>
   <Energy Type1=”CadExpLevel2” Type2=”CadExpLevel2”>0</Energy>

   <SpecificityCadherin>
      <Specificity Cadherin1=”NCad1” Cadherin2=”NCad1”>-10</Specificity>
      <Specificity Cadherin1=”NCad0” Cadherin2=”NCad0”>-12</Specificity>
      <Specificity Cadherin1=”NCad1” Cadherin2=”NCad0”>-1</Specificity>
   </SpecificityCadherin>

   <EnergyOffset>0.0</EnergyOffset>
   <NeighborOrder>2</NeighborOrder>
</Plugin>
```

Entries of the type `<Energy Type1="Medium" Type2="CadExpLevel1">0</Energy>` have the same meaning as in “regular” contact energy. Specificity parameters specification k_{mn} are enclosed between tags `<SpecificityCadherin>` and `<SpecificityCadherin>`. The names `NCad0` and `Ncad1` are arbitrary. However the matrix k_{mn} will be ordered according to lexiographic order of Cadherin names. For that reason we recommend that you name cadherins in such a way that makes it easy what the order will be. As in the example above using NameNumber (*e.g.*, NCad0, NCad1) makes it easy to figure out what the order will be (NCad0 will get index 0 and NCad1 will get index 1). This is important because cadherins will be set in Python and if you won’t keep track of the ordering of the specificity you might wrongly assign cadherins in Python and get unexpected results. In the example the order of cadherins is clear based on the definition of cadherin specificity parameters.

9.2.18 MolecularContact

This plugin is analogous to ContactLocalProduct and allows users to specify functional form of adhesion molecules interactions using Python syntax. It is in beta state and for this reason we are not discussing it in more detail and currently suggest to use Either AdhesionFlex or ContactLocal product plugins.

9.2.19 ContactCompartment

This plugin is a generalization of the contact energy plugin for the case of compartmental cell models.

$$E_{\text{ContactCompartment}} = \sum_{i,j \text{ neighbors}} J\big[\sigma(\mu_i, \nu_i), \sigma(\mu_j, \nu_j)\big]$$

where i and j denote pixels, $\sigma(\mu, \nu)$ denotes, as before, a cell type of a cell with μ cluster id and ν cell id. In compartmental cell models, a cell is a collection of subcells. Each subcell has a unique id (cell id). In addition to that, each subcell will have additional attribute, a cluster id that determines to which cluster of subcells a given subcell belongs (think of a cluster as a cell with nonhomogenous cytoskeleton). The idea here is to have different contact energies between subcells belonging to the same cluster and different energies for cells belonging to different clusters. Technically, subcells of a cluster are “regular” CompuCell3D cells. By giving them an extra attribute cluster id we can introduce a concept of compartmental cells. In our convention, $\sigma(0, 0)$ denotes medium

Introduction of cluster id and cell id are essential for the definition of $J\big[\sigma(\mu_i, \nu_i), \sigma(\mu_j, \nu_j)\big]$.

$$J\big[\sigma(\mu_i, \nu_i), \sigma(\mu_j, \nu_j)\big] = \begin{cases} J^{\text{external}}\big[\sigma(\mu_i, \nu_i), \sigma(\mu_j, \nu_j)\big] & \text{if } \ \mu_i \neq \mu_j \\ \\ J^{\text{internal}}\big[\sigma(\mu_i, \nu_i), \sigma(\mu_j, \nu_j)\big] & \text{if } \ \mu_i = \mu_j \end{cases}$$

As you can see from above there are two hierarchies of contact energies: **external** and **internal**. The energies depend on cell types as in the case “regular” Contact plugin. Now, however, depending whether pixels for which we calculate contact energies belong to the same cluster or not, we will use internal or external contact energies respectively.

9.2.20 LengthConstraint plugin

This plugin imposes elongation constraint on the cell. Effectively it “measures” a cell along its “axis of elongation” and ensures that cell length along the elongation axis is close to target length. For detailed description of this algorithm in 2D see Roeland Merks’ paper “Cell elongation is a key to in silico replication of in vitro vasculogenesis and subsequent remodeling” *Developmental Biology* **289** (2006) 44-54. This plugin is usually used in conjunction with Connectivity Plugin. The syntax is as follows:

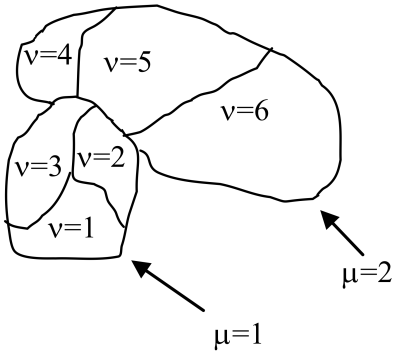


Figure 21: Two compartmental cells (cluster id $\mu = 1$ and cluster id $\mu = 2$) Compartmentalized cell $\mu = 1$ consists of subcells with cell id $\nu = 1, 2, 3$ and compartmentalized cell $\mu = 2$ consists of subcells with cell id $\nu = 4, 5, 6$.

```
<Plugin Name="LengthConstraint">
    <LengthEnergyParameters CellType="Body1" TargetLength="30"
    LambdaLength="5"/>
</Plugin>
```

LambdaLength determines the degree of cell length oscillation around `TargetLength` parameter. The higher `LambdaLength`, the less freedom a cell will have to deviate from `TargetLength`. In the 3D case we use the following syntax:

```
<Plugin Name="LengthConstraint">
    <LengthEnergyParameters CellType="Body1" TargetLength="20"
    MinorTargetLength="5" LambdaLength="100" />
</Plugin>
```

Notice new attribute called `MinorTargetLength`. In 3D it is not sufficient to constrain the “length” of the cell, you also need to constrain “width” of the cell along axis perpendicular to the major axis of the cell. This “width” is referred to as `MinorTargetLength`.

For 2D simulations we have also an option to use `LengthConstraintLocalFlex` plugin which calculate elongation constraints based on local parameters (*i.e.*, on a per cell basis). The syntax is as follows:

```
<Plugin Name="LengthConstraintLocalFlex"/>
```

The parameters are assigned using Python – see **Demos/elongationLocalFlexTest** example.

Remark: For 3D simulations we can only define elongation parameters on a per cell type basis. We will fix this limitation in the next release.

Remark: When using target length plugins (either global , as shown here, or local as we will show in the subsequent subsection) it is important to use connectivity constraint. This constrain will check if a given pixel copy can break cell connectivity. If so, it will add large energy penalty (defined by a user) to change of energy effectively prohibiting such pixel copy. In the case of 2D on square lattice checking cell connectivity can be done locally and thus is very fast. Unfortunately on hex lattice and in 3D on either lattice we dont have an algorithm of performing such check locally and therefore we do it globally using breadth first search algorithm and comparing volumes of cells calculated this way with actual volume of the cell. If they agree we conclude that cell connectivity is preserved. However the computational cost of running such algorithm, can be quite high. Therefore if one does need extremely elongated cells (it is when connectivity algorithm has to do a lot of work) one may neglect connectivity constraint and use Length constrain only. For slight cells elongations the connectivity should be preserved however, occasionally cells may fragment.

9.2.21 Connectivity plugin

The basic Connectivity plugin works **only in 2D and only on square lattice** and is used to ensure that cells are connected or in other words to prevent separation of the cell into pieces. The detailed algorithm for this plugin is described in Roeland Merks' paper "Cell elongation is a key to in-silico replication of in vitro vasculogenesis and subsequent remodeling" *Developmental Biology* **289** (2006) 44-54. There was one modification of the algorithm as compared to the paper. Namely, to ensure proper connectivity we had to reject all spin flips that resulted in more that two collisions. (see the paper for detailed explanation what this means).

The syntax of the plugin is straightforward:

```
<Plugin Name="Connectivity">
   <Penalty>100000</Penalty>
</Plugin>
```

`Penalty` denotes energy that will be added to overall change of energy if attempted spin flip would violate connectivity constraints. If the penalty is positive and much larger than the absolute value of other energy changes in the simulation this has the effect of preventing a spin flip from occurring.

A more general type of connectivity constraint is implemented in ConnectivityGlobal plugin. In this case we calculate volume of a cell using breadth first search algorithm and compare it with actual volume of the cell. If they agree we conclude that cell connectivity is preserved. This plugin works both in 2D and 3D and on either type of lattice. However, the computational cost of running such algorithm, can be quite high so it is best to limit this plugin to cell types for which connectivity of cell is really essential:

```
<Plugin Name="ConnectivityGlobal">
   <Penalty Type="Body1">1000000000</Penalty>
</Plugin>
```

In certain types of simulation it may happen that at some point cells change cell types. If a cell that was not subject to connectivity constraint, changes type to the cell that is constrained by global connectivity and this cell is fragmented before type change this situation normally would result in simulation freeze. However CompuCell3D, first before applying constraint it will check if the cell is fragmented. If it is, there is no constraint. Global connectivity constraint is only applied when cell is non-fragmented. The numerical value of `Penalty` in the XML syntax above does not really matter as long as it is greater than 0. CompuCell3D guarantees that cells for which penalty is greater than 0 will remain connected. We will modify global connectivity plugin to allow application of connectivity constraints to individual cells.

Quite often in the simulation we don't need to impose connectivity constraint on all cells or on all cells of given type. Usually only select cell types or select cells are elongated and therefore need connectivity constraint. In such a case we use ConnectivityLocalFlex plugin and assign connectivity constraints to particular cells in Python.

In XML we only declare:

```
<Plugin Name="ConnectivityLocalFlex"/>
```

In Python we manipulate/access connectivity parameters for individual cells using the following syntax:

```
self.connectivityLocalFlexPlugin.setConnectivityStrength(cell,20.7)
```

```
self.connectivityLocalFlexPlugin.getConnectivityStrength(cell)
```

See also example in **Demos/elongationLocalFlexTest**.

9.2.22 Mitosis plugin

Mitosis plugin carries out cell division into two cells once the parent cell reaches critical volume (`DoublingVolume`). The two cells after mitosis will have approximately the same volume although it cannot be guaranteed in general case if the parent cell is fragmented. One major problem with Mitosis plugin is that after mitosis the attributes of the offspring cell might not be initialized properly. By default cell type of the offspring cell will be the same as cell type of parent and they will also share target volume. All other parameters for the new cell remain uninitialized.

Remark: For this reason we stringly recommend using Mitosis plugin through Python interface as there users can quite easily customize what happens to parent and offspring cells after mitosis. An example of the use of Mitosis plugin through Python scripting is provided in CompuCell3Ds Python Scripting Manual. The syntax of the “standard” mitosis plugin is the following:

```
<Plugin Name="Mitosis">
   <DoublingVolume>50</DoublingVolume>
</Plugin>
```

Every time a cell reaches `DoublingVolume` it will undergo the mitosis and the offspring cell will inherit type and target volume of the parent. If this simple behavior is unsatisfactory consider use Python scripting to implement proper mitotic divisions of cells.

9.2.23 Secretion plugin

In earlier version os of CC3D secretion was part of PDE solvers. We still support this mode of model description however, starting in 3.5.0 we developed separate plugin which handles secretion only. Via secretion plugin we can simulate cellular secretion of various chemicals. The secretion plugin allows users to specify various secretion modes in the XML file - XML syntax is practically identical to the SecretionData syntax of PDE solvers. In addition to this Secretion plugin allows users to manipulate secretion properties of individual cells from Python level. To account for possibility of PDE solver being called multiple times during each MCS, the Secretion plugin can be called multiple times in each MCS as well. We leave it up to user the rescaling of secretion constants when using multiple secretion calls in each MCS. **Note**: Secretion for individual cells invoked via Python will be called only once per MCS.

Typical XML syntax for Secretion plugin is presented below:

```
<Plugin Name="Secretion">
   <Field Name="ATTR" ExtraTimesPerMC= 2 >
      <Secretion Type="Bacterium">200</Secretion>
      <SecretionOnContact Type="Medium"
         SecreteOnContactWith="B">300</SecretionOnContact>
      <ConstantConcentration Type="Bacterium">500</ConstantConcentration>
   </Field>
</Plugin>
```

By default `ExtraTimesPerMC` is set to 0 - meaning no extra calls to Secretion plugin per MCS.

Typical use of secretion from Python is demonstrated best in the example below:

```
class SecretionSteppable(SecretionBasePy):
    def __init__(self,_simulator,_frequency=1):
        SecretionBasePy.__init__(self,_simulator, _frequency)

    def step(self,mcs):
        attrSecretor=self.getFieldSecretor(”ATTR”)
        for cell in self.cellList:
            if cell.type==3:
                attrSecretor.secreteInsideCell(cell,300)
                attrSecretor.secreteInsideCellAtBoundary(cell,300)
                attrSecretor.secreteOutsideCellAtBoundary(cell,500)
                attrSecretor.secreteInsideCellAtCOM(cell,300)
```

Remark: Instead of using `SteppableBasePy` class we are using `SecretionBasePy` class. The reason for this is that in order for secretion plugin with secretion modes accessible from Python to behave exactly as previous versions of PDE solvers (where secretion was done first followed by "diffusion" step) we have to ensure that secretion steppable implemented in Python is called before each Monte Carlo Step, which implies that it will be also called before "diffusing" function of the PDE solvers. `SecretionBasePy` sets extra flag which ensures that steppable which inherits from `SecretionBasePy` is called before MCS (and before all "regular" Python steppables). There is no magic to `SecretionBasePy` - if you still want to use `SteppableBasePy` as a base class for secretion (or for that matter `SteppablePy`) do so, but remember that you need to set flag:

```
self.runBeforeMCS=1
```

to ensure that your new stoppable will run before each MCS. See example below for alternative implementation of `SecretionSteppable` using `SteppableBasePy` as a base class:

```
class SecretionSteppable(SteppableBasePy):
    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator, _frequency)
        self.runBeforeMCS=1

    def step(self,mcs):
        attrSecretor=self.getFieldSecretor(”ATTR”)
        for cell in self.cellList:
            if cell.type==3:
                attrSecretor.secreteInsideCell(cell,300)
                attrSecretor.secreteInsideCellAtBoundary(cell,300)
                attrSecretor.secreteOutsideCellAtBoundary(cell,500)
                attrSecretor.secreteInsideCellAtCOM(cell,300)
```

The secretion of individual cells is handled through Field Secretor objects. Field Secretor concenpt is quite convenient because the amoun of Python coding is quite small. To secrete chemical (this is now done for individual cell) we first create field secretor object, `attrSecretor=self.getFieldSecretor("ATTR")`, which allows us to secrete into field called ATTR.

Remark: Make sure that fields into which you will be secreting chemicals exist. They are usually fields defined in PDE solvers. When using secretion plugin you do not need to specify SecretionData section for the PDE solvers Then we pick a cell and using field secretor we simulate secretion of chemical ATTR by a cell:

```
attrSecretor.secreteInsideCell(cell,300)
```

Currently we support 4 secretion modes for individual cells:

1. `secreteInsideCell` this is equivalent to secretion in every pixel belonging to a cell
2. `secreteInsideCellAtBoundary` secretion takes place in the pixels belonging to the cell boundary
3. `secreteInsideCellAtBoundary` secretion takes place in pixels which are outside the cell but in contact with cell boundary pixels
4. `secreteInsideCellAtCOM` secretion at the center of mass of the cell

As you may infer from above modes 1, 2, and 3 require tracking of pixels belonging to cell and pixels belonging to cell boundary. If you are not using modes 1−3 you may disable pixel tracking by including `<DisablePixelTracker/>` and/or `<DisableBoundaryPixelTracker/>` tags as shown in the example below:

```
<Plugin Name=”Secretion”>
   <DisablePixelTracker/>
   <DisableBoundaryPixelTracker/>
   <Field Name=”ATTR” ExtraTimesPerMC= 2 >
      <Secretion Type=”Bacterium”>200</Secretion>
      <SecretionOnContact Type=”Medium” SecreteOnContactWith=”B”>300
      </SecretionOnContact>
      <ConstantConcentration Type=”Bacterium”>500</ConstantConcentration>
   </Field>
</Plugin>
```

9.2.24 PDESolverCaller plugin

PDE solvers in CompuCell3D are implemented as steppables . This means that by default they are called every MCS. In many cases this is insufficient. For example if diffusion constant is large, then explicit finite difference method will become unstable and the numerical solution will have no sense. To fix this problem one could call PDE solver many times during single MCS. This is precisely the task taken care of by PDESolverCaller plugin. The syntax is straightforward:

```
<Plugin Name="PDESolverCaller">
   <CallPDE PDESolverName="FlexibleDiffusionSolverFE"ExtraTimesPerMC="8"/>
</Plugin>
```

All you need to do is to give the name of the steppable that implements a given PDE solver and pass let CompCell3D know how many extra times per MCS this solver is to be called (here `FlexibleDiffusionSolverFE` was 8 extra times per MCS).

9.2.25 Elasticity and ElasticityTracker plugins

This plugin is responsible for handling the following energy term:

$$E = \sum_{\text{i,j-cellneighbors}} \lambda_{ij}(l_{ij} - L_{ij})^2$$

where l_{ij} is a distance between center of masses of cells i and j and L_{ij} is a target length corresponding to l_{ij}.

The syntax of this plugin is the following

```
<Plugin Name="ElasticityEnergy">
   <LambdaElasticity>200.0</LambdaElasticity>
   <TargetLengthElasticity>6</TargetLengthElasticity>
</Plugin>
```

In this case λ_{ij} and L_{ij} are the same for all participating cells types.

By adding extra attribute `<Local/>` to the above plugin:

```
<Plugin Name="ElasticityEnergy">
   <Local />
   <LambdaElasticity>200.0</LambdaElasticity>
   <TargetLengthElasticity>6</TargetLengthElasticity>
</Plugin>
```

we tell CompuCell3D to use λ_{ij} and L_{ij} defined on per pair of cells basis. The initialization of λ_{ij} and L_{ij} usually takes place in Python script and users must make sure that $l_{ij} = l_{ji}$ and $\lambda_{ij} = \lambda_{ji}$ or else one can get unexpected results. We provide example python and xml files that demo the use of plasticity plugin.

Users have to specify which cell types participate in the plasticity calculations. This is done by including ElasticityTracker plugin before Elasticity plugin in the xml file. The syntax is very clear:

```
<Plugin Name="ElasticityTracker">
```

```
    <IncludeType>Body1</IncludeType>
    <IncludeType>Body2</IncludeType>
    <IncludeType>Body3</IncludeType>
</Plugin>
```

All is required is a list of participating cell types. Here cells of type Body1, Body2, and Body3 will be taken into account for elasticity energy calculation purposes. The way in which CompuCell3D determines which cells are to be included in the elasticity energy calculations is by examining which cells are in contact with each other before simulation begins.

If the types of cells touching each other are listed in the list of IncludeTypes of ElasticityTracker then such cells are being taken into account when calculating elastic constraint. Cells which initially are not touching will not participate in calculations even if their type is included in the list of "ElasticityTracker". However, in some cases it is desirable to add elasticity pair even for cells that do not touch each other or do it once simulation has started. To do this ElasticityTracker plugin defines two function :

```
assignElasticityPair(_cell1 , _cell2)
```

```
removeElasticityPair(_cell1 , _cell2)
```

where _cell1 and _cell2 denote pointers to cell objects. These functions add or remove two cell links to or from elastic constraint. Typically they are called from Python level.

9.2.26 FocalPointPlasticity plugin

Similarly as Elasticity plugin, FocalPointPlasticity put constrains the distance between cells center of masses. The main difference is that the list of "focal point plasticity neighbors" can change as the simulation goes and user specifies the maximum number of "focal point plasticity neighbors" a given cell can have. Lets look at relatively simple XML syntax of FocalPointPlasticity plugin (see **Demos/FocalPointPlasticity** example and we will show more complex examples later):

```
<Plugin Name="FocalPointPlasticity">
    <Parameters Type1="Condensing" Type2="NonCondensing">
        <Lambda>10.0</Lambda>
        <ActivationEnergy>-50.0</ActivationEnergy>
        <TargetDistance>7</TargetDistance>
        <MaxDistance>20.0</MaxDistance>
        <MaxNumberOfJunctions>2</MaxNumberOfJunctions>
    </Parameters>

    <Parameters Type1="Condensing" Type2="Condensing">
        <Lambda>10.0</Lambda>
        <ActivationEnergy>-50.0</ActivationEnergy>
        <TargetDistance>7</TargetDistance>
        <MaxDistance>20.0</MaxDistance>
        <MaxNumberOfJunctions>2</MaxNumberOfJunctions>
    </Parameters>
    <NeighborOrder>1</NeighborOrder>
</Plugin>
```

`Parameters` section describes properties of links between cells. `MaxNumberOfJunctions`, `ActivationEnergy`, `MaxDistance` and `NeighborOrder` are responsible for establishing connections between cells. CC3D constantly monitors pixel copies and during pixel copy between two neighboring cells/subcells it checks if those cells are already participating in focal point plasticity constraint. If they are not, CC3D will check if connection can be made (*e.g.*, `Condensing` cells can have up to two connections with `Condensing` cells and up to 2 connections with `NonCondensing` cells - see first line of `Parameters` section and `MaxNumberOfJunctions` tag). The `NeighborOrder` parameter determines the pixel vicinity of the pixel that is about to be overwritten which CC3D will scan in search of the new link between cells. `NeighborOrder` 1 (which is default value if you do not specify this parameter) means that only nearest pixel neighbors will be visited. The `ActivationEnergy` parameter is added to overall energy in order to increase the odds of pixel copy which would lead to new connection.

Once cells are linked the energy calculation is carried out in a very similar way as for the Elasticity plugin:

$$E = \sum_{i,j \text{ cell neighbors}} \lambda_{ij}(l_{ij} - L_{ij})^2$$

where l_{ij} is a distance between center of masses of cells i and j and L_{ij} is a target length corresponding to l_{ij}.

λ_{ij} and L_{ij} between different cell types are determined using `Lambda` and `TargetDistance` tags. The `MaxDistance` determines the distance between cells center of masses when the link between those cells break. When the link breaks, then in order for the two cells to reconnect they would need to come in contact (in order to reconnect). However, it is usually more likely that there will be other cells in the vicinity of separated cells so it is more likely to establish new link than restore the broken one.

The above example was one of the simplest examples of use of FocalPointPlasticity. A more complicated one involves compartmental cells. In this case each cell has separate “internal” list of links between cells belonging to the same cluster and another list between cells belonging to different clusters. The energy contributions from both lists are summed up and everything that we have said when discussing example above applies to compartmental cells. Sample syntax of the FocalPointPlasticity plugin which includes compartmental cells is shown below. We use `InternalParameters` tag/section to describe links between cells of the same cluster (see **Demos/FocalPointPlasticity** example):

```
<Plugin Name="FocalPointPlasticity">

   <Parameters Type1="Top" Type2="Top">
      <Lambda>10.0</Lambda>
      <ActivationEnergy>-50.0</ActivationEnergy>
      <TargetDistance>7</TargetDistance>
      <MaxDistance>20.0</MaxDistance>
      <MaxNumberOfJunctions NeighborOrder="1">1</MaxNumberOfJunctions>
   </Parameters>

   <Parameters Type1="Bottom" Type2="Bottom">
      <Lambda>10.0</Lambda>
      <ActivationEnergy>-50.0</ActivationEnergy>
      <TargetDistance>7</TargetDistance>
      <MaxDistance>20.0</MaxDistance>
      <MaxNumberOfJunctions NeighborOrder="1">1</MaxNumberOfJunctions>
   </Parameters>

   <InternalParameters Type1="Top" Type2="Center">
      <Lambda>10.0</Lambda>
      <ActivationEnergy>-50.0</ActivationEnergy>
      <TargetDistance>7</TargetDistance>
      <MaxDistance>20.0</MaxDistance>
      <MaxNumberOfJunctions>1</MaxNumberOfJunctions>
   </InternalParameters>

   <InternalParameters Type1="Bottom" Type2="Center">
      <Lambda>10.0</Lambda>
      <ActivationEnergy>-50.0</ActivationEnergy>
      <TargetDistance>7</TargetDistance>
      <MaxDistance>20.0</MaxDistance>
      <MaxNumberOfJunctions>1</MaxNumberOfJunctions>
   </InternalParameters>

   <NeighborOrder>1</NeighborOrder>
</Plugin>
```

Sometimes it is necessary to modify link parameters individually for every cell pair. In this case we would manipulate FocalPointPlasticity links using Python scripting. Example **Demos/FocalPointPlasticityCompartments** demonstrates exactly this situation. Still, you need to include XML section as the one shown above for compartmental cells, because we need to tell CC3D how to link cells. The only notable difference is that in the XML we have to include `<Local/>` tag to signal that we will set link parameters (`Lambda, TargetDistance, MaxDistance`) individually for each cell pair:

```
<Plugin Name="FocalPointPlasticity">
   <Local/>
   <Parameters Type1="Top" Type2="Top">
      <Lambda>10.0</Lambda>
```

```
        <ActivationEnergy>-50.0</ActivationEnergy>
        <TargetDistance>7</TargetDistance>
        <MaxDistance>20.0</MaxDistance>
        <MaxNumberOfJunctions NeighborOrder="1">1</MaxNumberOfJunctions>
    </Parameters>
...
</Plugin>
```

Python steppable where we manipulate cell-cell focal point plasticity link properties is shown below:

```
class FocalPointPlasticityCompartmentsParams(SteppablePy):
    def __init__(self,_simulator,_frequency=10):
        SteppablePy.__init__(self,_frequency)
        self.simulator=_simulator
        self.focalPointPlasticityPlugin=CompuCell. \
        getFocalPointPlasticityPlugin()
        self.inventory=self.simulator.getPotts().getCellInventory()
        self.cellList=CellList(self.inventory)

    def step(self,mcs):
        for cell in self.cellList:
            for fppd in InternalFocalPointPlasticityDataList
            \ (self.focalPointPlasticityPlugin,cell):
                    self.focalPointPlasticityPlugin. \
                    setInternalFocalPointPlasticityParameters \
                    (cell,fppd.neighborAddress,0.0,0.0,0.0)
```

The syntax to change focal point plasticity parameters (or as here internal parameters) is as follows:

```
setFocalPointPlasticityParameters(cell1, cell2, lambda, \
targetDistance, maxDistance)
```

```
setFocalPointPlasticityParameters(cell1, cell2, lambda, \
targetDistance, maxDistance)
```

Similarly to inspect current values of the focal point plasticity parameters we would use the following Python construct:

```
for cell in self.cellList:
    for fppd in InternalFocalPointPlasticityDataList\
    (self.focalPointPlasticityPlugin,cell):
        print "fppd.neighborId",fppd.neighborAddress.id
        "␣lambda=",fppd.lambdaDistance
```

For non-internal parameters we simply use `FocalPointPlasticityDataList` instead of `InternalFocalPointPlasticityDataList`.

Examples **Demos/FocalPointPlasticity** show in relatively simple way how to use FocalPointPlasticity plugin. Those examples also contain useful comments.

When using FocalPointPlasticity Plugin from mitosis module one might need to break or create focal point plasticity links. To do so FocalPointPlasticity Plugin provides 4 convenience functions which can be invoked from the Python level:

```
deleteFocalPointPlasticityLink(cell1,cell2)

deleteInternalFocalPointPlasticityLink(cell1,cell2)

createFocalPointPlasticityLink(\
cell1,cell2,lambda,targetDistance,maxDistance)

createInternalFocalPointPlasticityLink(\
cell1,cell2,lambda,targetDistance,maxDistance)
```

VAdd: An example of the use of FocalPointPlasticity plugin with mitosis is as follows. In an XML file, which we name for example **mitosis_fpp.xml** we set the following:

```
<CompuCell3D>

<!-- Basic properties of CPM (GGH) algorithm -->
<Potts>
   <Dimensions x="100" y="100" z="1"/>
   <Steps>1200</Steps>
   <Temperature>10.0</Temperature>
   <NeighborOrder>1</NeighborOrder>
</Potts>

<!-- Listing all cell types in the simulation -->
<Plugin Name="CellType">
   <CellType TypeId="0" TypeName="Medium"/>
   <CellType TypeId="1" TypeName="type1"/>
</Plugin>

<!-- Constraint on cell volume. Each cell has different constraint -
constraints have to be initialized and managed in Python -->
<Plugin Name="VolumeLocalFlex"/>

<!-- Module tracking center of mass of each cell -->
<Plugin Name="CenterOfMass"/>

<!-- Specification of adhesion energies -->
<Plugin Name="Contact">
   <Energy Type1="Medium" Type2="Medium">10</Energy>
   <Energy Type1="Medium" Type2="type1">10</Energy>
   <Energy Type1="type1" Type2="type1">10</Energy>
   <NeighborOrder>2</NeighborOrder>
</Plugin>
```

```
<!-- Specification of focal point junctions -->
<!-- We separetely specify links between members of same cluster -
InternalParameters and members of different clusters Parameters.
When not using compartmental cells comment out InternalParameters
specification -->
<Plugin Name="FocalPointPlasticity">
   <!-- To modify FPP links individually for each cell pair uncomment
   line below -->
   <!-- <Local/> -->
   <!-- Note that even though you may manipulate lambdaDistance,
   targetDistance and maxDistance using Python you still need to set
   activation energy from XML level -->
   <!-- See CC3D manual for details on FPP plugin  -->

   <Parameters Type1="type1" Type2="type1">
      <Lambda>10</Lambda>
      <ActivationEnergy>-50</ActivationEnergy>
      <TargetDistance>5</TargetDistance>
      <MaxDistance>10</MaxDistance>
      <MaxNumberOfJunctions NeighborOrder="1">6</MaxNumberOfJunctions>
   </Parameters>

   <NeighborOrder>1</NeighborOrder>
</Plugin>

<!-- Initial layout of cells in the form of rectangular slab -->
   <Steppable Type="UniformInitializer">
      <Region>
         <BoxMin x="40" y="40" z="0"/>
         <BoxMax x="60" y="60" z="1"/>
         <Gap>0</Gap>
         <Width>5</Width>
         <Types>type1</Types>
      </Region>
   </Steppable>
</CompuCell3D>
```

In the main function of python file, we name it **mitosis_fpp_main.py** we set

```
import sys
from os import environ
from os import getcwd
import string

sys.path.append(environ["PYTHON_MODULE_PATH"])

import CompuCellSetup

sim,simthread = CompuCellSetup.getCoreSimulationObjects()

# add extra attributes here
```

```
CompuCellSetup.initializeSimulationObjects(sim,simthread)
# Definitions of additional Python-managed fields go here

#Add Python steppables here
steppableRegistry=CompuCellSetup.getSteppableRegistry()

from mitosis_fppSteppables import ConstraintInitializerSteppable
ConstraintInitializerSteppableInstance=\
ConstraintInitializerSteppable(sim,_frequency=40)
steppableRegistry.registerSteppable(ConstraintInitializerSteppableInstance)

from mitosis_fppSteppables import GrowthSteppable
GrowthSteppableInstance=GrowthSteppable(sim,_frequency=40)
steppableRegistry.registerSteppable(GrowthSteppableInstance)

from mitosis_fppSteppables import MitosisSteppable
MitosisSteppableInstance=MitosisSteppable(sim,_frequency=40)
steppableRegistry.registerSteppable(MitosisSteppableInstance)

CompuCellSetup.mainLoop(sim,simthread,steppableRegistry)
```

and in the steppable file, we name it **mitosis_fppSteppables.py**:

```
from PySteppables import *
import CompuCell
import sys

from PySteppablesExamples import MitosisSteppableBase


class ConstraintInitializerSteppable(SteppableBasePy):
    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator,_frequency)
    def start(self):
        for cell in self.cellList:
            cell.targetVolume=25
            cell.lambdaVolume=2.0


class GrowthSteppable(SteppableBasePy):
    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator,_frequency)

    def step(self,mcs):
        for cell in self.cellList:
            cell.targetVolume+=1
    # alternatively if you want to make growth a function of chemical
    # concentration uncomment lines below and comment lines above
        # field=CompuCell.getConcentrationField(self.simulator,\
        #"PUT_NAME_OF_CHEMICAL_FIELD_HERE")
        # pt=CompuCell.Point3D()
        # for cell in self.cellList:
            # pt.x=int(cell.xCOM)
            # pt.y=int(cell.yCOM)
            # pt.z=int(cell.zCOM)
            # concentrationAtCOM=field.get(pt)
            # cell.targetVolume+=0.01*concentrationAtCOM
            # you can use here any fcn of concentrationAtCOM


class MitosisSteppable(MitosisSteppableBase):
    def __init__(self,_simulator,_frequency=1):
        MitosisSteppableBase.__init__(self,_simulator, _frequency)

    def step(self,mcs):
        # print "INSIDE MITOSIS STEPPABLE"
        cells_to_divide=[]
        for cell in self.cellList:
            if cell.volume>50:
                cells_to_divide.append(cell)
                # Here is where we delete the links
                for fppd in self.getFocalPointPlasticityDataList(cell):
                  self.focalPointPlasticityPlugin.\
                  deleteFocalPointPlasticityLink(cell,fppd.neighborAddress)

        for cell in cells_to_divide:
            # to change mitosis mode leave one of the below lines uncommented
            self.divideCellRandomOrientation(cell)
            # self.divideCellOrientationVectorBased(cell,1,0,0)
            # self.divideCellAlongMajorAxis(cell)
            # self.divideCellAlongMinorAxis(cell)

    def updateAttributes(self):
        parentCell=self.mitosisSteppable.parentCell
        childCell=self.mitosisSteppable.childCell

        childCell.targetVolume=parentCell.targetVolume
        childCell.lambdaVolume=parentCell.lambdaVolume
        childCell.type = parentCell.type
        # Here is where we reform the links
        if parentCell:
            for neighbordata in self.getFocalPointPlasticityDataList(parentCell):
                self.focalPointPlasticityPlugin.\
                createFocalPointPlasticityLink(parentCell,\
                neighbordata.neighborAddress,10,5,20)
        if childCell:
            for neighbordata in self.getFocalPointPlasticityDataList(childCell):
                self.focalPointPlasticityPlugin.createFocalPointPlasticityLink\
                (childCell,neighbordata.neighborAddress,10,5,20)
```

9.2.27 Curvature Plugin

This plugin implements energy term for compartmental cells. It is based on “A New Mechanism for Collective Migration in Myxococcus xanthus”, J. Starruß, Th. Bley, L. Søgaard-Andersen and A. Deutsch, Journal of Statistical Physics, DOI: 10.1007/s10955-007-9298-9, (2007). For a “long” compartmental cell composed of many subcells it imposes constraint on curvature of cells. The syntax is slightly complex:

```
<Plugin Name=”Curvature”>
   <InternalParameters Type1=”Top” Type2=”Center”>
      <Lambda>100.0</Lambda>
      <ActivationEnergy>−50.0</ActivationEnergy>
   </InternalParameters>

   <InternalParameters Type1=”Center” Type2=”Center”>
      <Lambda>100.0</Lambda>
      <ActivationEnergy>−50.0</ActivationEnergy>
   </InternalParameters>

   <InternalParameters Type1=”Bottom” Type2=”Center”>
      <Lambda>100.0</Lambda>
      <ActivationEnergy>−50.0</ActivationEnergy>
   </InternalParameters>

   <InternalTypeSpecificParameters>
      <Parameters TypeName=”Top” MaxNumberOfJunctions=”1”
        NeighborOrder=”1”/>
      <Parameters TypeName=”Center” MaxNumberOfJunctions=”2”
        NeighborOrder=”1”/>
      <Parameters TypeName=”Bottom” MaxNumberOfJunctions=”1”
        NeighborOrder=”1”/>
</InternalTypeSpecificParameters>
</Plugin>
```

The `InternalTypeSpecificParameter` tells Curvature Plugin how many neighbors a cell of given type will have. In this case, numbers which make sense are 1 and 2. The middle segment will have 2 connection and head and tail segments will have only one connection with neighboring segments (subcells). The connections are established dynamically. The way it happens is that during simulation CC3D constantly monitors pixel copies and during pixel copy between two neighboring cells/subcells it checks if those cells are already “connected” using curvature constraint. If they are not, CC3D will check if connection can be made (*e.g.*, `Center` cells can have up to two connections and `Top` and `Bottom` only one connection). Usually establishing connections takes place at the beginning if the simulation and often happens within first Monte Carlo Step (depending on actual initial configuration, of course, but if segments touch each other connections are established almost immediately). The `ActivationEnergy` parameter is added to overall energy in order to increase the odds of pixel copy which would lead to new connection. `Lambda` tag/parameter determines “the strength” of curvature constraint. The higher the `Lambda` the more “stiff” cells will be, *i.e.*, they will tend to align along straight line.

9.2.28 PlayerSettings Plugin

This plugin allows users to specify or configure Player settings directly from XML, without s single click. Some users might prefer this way of setting configuring Player. In addition to this if users want to run two different simulations at the same time on the same machine but with different , say, cell colors, then doing it with “regular” Player configuration file might be tricky. The solution is to use PlayerSetting Plugin. The syntax of this plugin is as follows:

```
<Plugin Name="PlayerSettings">
   <Project2D XZProj="50"/>
   <Concentration LegendEnable="true" NumberOfLegendBoxes="3"/>
   <VisualControl ScreenshotFrequency="200" ScreenUpdateFrequency="10"
     NoOutput="true"   ClosePlayerAfterSimulationDone="true" />
   <Border BorderColor="red" BorderOn="false"/>
   <TypesInvisibleIn3D Types="0,2,4,5"/>
   <Cell Type="1" Color="red"/>
   <Cell Type="2" Color="yellow"/>
   <!-- Note: SaveSettings flag is unimportant for the new Player because
     whenever settings are changed from XML script they are written by default
     to disk. This seems to be default behavior of most modern applications.
     We may implement this feature later
   <Settings SaveSettings="false"/> -->
</Plugin>
```

9.2.29 BoundaryPixelTracker Plugin

9.2.30 GlobalBoundaryPixelTracker

9.2.31 PixelTracker Plugin

9.2.32 MomentOfInertia plugin

9.2.33 SimpleClock plugin

9.2.34 ConvergentExtension plugin

9.3 Steppable Section

Steppables are CompuCell modules that are called every Monte Carlo Step (MCS). More precisely, they are called after all the spin attempts in a given MCS have been carried out. Steppables may have various functions like for example solving PDE's, checking if critical concentration threshold have been met, updating target volume or target surface given the concentration of come growth factor, initializing cell field, writing numerical results to a file etc. In summary Steppables perform all functions that need to be done every MCS. In the reminder of this section we will present steppables currently available in the CompuCell and describe their usage.

9.3.1 UniformInitializer steppable

This steppable lays out pattern of cells on the lattice. It allows users to specify rectangular regions of field with square (or cube in 3D) cells of user defined types (or random types). Cells can be touching each other or can be separated by a gap.

The syntax of the plugin is as follows:

9.3.2 BlobInitializer steppable

9.3.3 PIFInitializer steppable

9.3.4 PIFDumper steppable

9.3.5 Mitosis steppable

This steppable is described in great detail in Python tutorial but because of its importance we are including a copy of that description here.

In developmental simulations we often need to simulate cells which grow and divide. In earlier versions of CompuCell3D we had to write quite complicated plugin to do that which was quite cumbersome and unintuitive (see example 9). The only advantage of the plugin was that exactly after the pixel copy which had triggered mitosis condition CompuCell3D called cell division function immediately. This guaranteed that any cell which was supposed divide at any instance in the simulation, actually did. However, because state of the simulation is normally observed after completion of full a Monte Carlo Step, and not in the middle of MCS it makes actually more sense to implement Mitosis as a steppable. Let us examine the simplest simulation which involves mitosis. We start with a single cell and grow it. When cell reaches critical (doubling) volume it undergoes Mitosis. We check if the cell has reached doubling volume at the end of each MCS. The folder containing this simulation is **examples_PythonTutorial/steppableBasedMitosis**. The mitosis algorithm is implemented in
examples_PythonTutorial/steppableBasedMitosis/steppableBasedMitosisSteppables.py.

```
from PySteppables import *
from PySteppablesExamples import MitosisSteppableBase
import CompuCell
import sys

class VolumeParamSteppable(SteppablePy):
    def __init__(self,_simulator,_frequency=1):
        SteppablePy.__init__(self,_frequency)
        self.simulator=_simulator
        self.inventory=self.simulator.getPotts().getCellInventory()
        self.cellList=CellList(self.inventory)

    def start(self):
        for cell in self.cellList:
            cell.targetVolume=25
            cell.lambdaVolume=2.0

    def step(self,mcs):
        for cell in self.cellList:
            cell.targetVolume+=1
```

```
class MitosisSteppable(MitosisSteppableBase):
    def __init__(self,_simulator,_frequency=1):
        MitosisSteppableBase.__init__(self,_simulator, _frequency)

    def step(self,mcs):
        cells_to_divide=[]
        for cell in self.cellList:
            if cell.volume>50: #mitosis condition
                cells_to_divide.append(cell)

        for cell in cells_to_divide:
            self.divideCellRandomOrientation(cell)

    def updateAttributes(self):
        parentCell=self.mitosisSteppable.parentCell
        childCell=self.mitosisSteppable.childCell
        childCell.targetVolume=parentCell.targetVolume
        childCell.lambdaVolume=parentCell.lambdaVolume

        if parentCell.type==1:
            childCell.type=2
        else:
            childCell.type=1
```

Two steppables: `VolumeParamSteppable` and `MitosisSteppable` are the essence of the above simulation. The first steppable initializes volume constraint for all the cells present at $T = 0$ MCS (only one cell) and then every 10 MCS (see the frequency with which `VolumeParamSteppable` is initialized to run - **examples_PythonTutorial/steppableBasedMitosis/steppableBasedMitosis.py**) it increases target volume of cells, effectively causing cells to grow.

The second steppable checks every 10 MCS (we can, of course, run it every MCS) if cell has reached doubling volume of 50. If so, such cell is added to the list `cells_to_divide` which subsequently is iterated and all the cells in it divide.

Remark: It is important to divide cells outside the loop where we iterate over entire cell inventory. If we keep dividing cells in this loop we are adding elements to the list over which we iterate over and this might have unwanted side effects. The solution is to use use list of cells to divide as we did in the example.

Notice that we call `self.divideCellRandomOrientation(cell)` function to divide cells. Other modes of division are available as well and they are shown in **examples_PythonTutorial/steppableBasedMitosis/steppableBasedMitosisSteppables.py** as commented line with appropriate explanation.

Notice MitosisSteppable inherits MitosisSteppableBase class (defined in **PySteppable-**

sExamples.py).It is the base class which ensures that after we call any of the cell dividing function (*e.g.*, `divideCellRandomOrientation`) CompuCell3D will automatically call `updateAttributes` function as well. `updateAttributes` function is very important and we must call it in order to ensure integrity and sanity of the simulation. During mitosis new cell is created (accessed in Python as `childCell` - defined in `MitosisSteppableBase` - `self.mitosisSteppable.childCell`) and as such this cell is uninitialized. It does have default attributes of a cell such as volume, surface (if we decide to use surface constraint or SurfaceTracker plugin) but all other parameters of such cell are set to default values. In our simulation we have been setting `targetVolume` and `lambdaVolume` individually for each cell. After mitosis `childCell` will need those parameters to be set as well. To make things more interesting, in our simulation we decided to change type of cell to be different than type of parent cell.

In more complex simulations where cells have more attributes which are used in the simulation, we have to make sure that in the `updateAttributes` function `childCell` and its attributes get properly initialized. It is also very common practice to change attributes of `parentCell` after mitosis as well to account for the fact that `parentCell` is not the original `parentCell` from before the mitosis.

Important: If you specify orientation vector for the mitosis the actual division will take place along the line/plane perpendicular to this vector.

Important: the name of the function where we update attributes after mitosis has to be exactly `updateAtttributes`. If it is called differently CC3D will not call it automatically. We can of course call such function by hand, immediately we do the mitosis but this is not a very elegant solution.

Now we will discuss how to use PDE solvers in CompuCell3D. Most of the PDE solvers solve PDE with diffusive terms. Let's take a look at them.

References

[1] M.A.J. Chaplain and G. Lolas, Mathematical Modelling of Cancer Cell Invasion of Tissue: The Role of The Urokinase Plasminogen Activation System, *Mathematical Models and Methods in Applied Sciences*, 15 (2005) 1685 - 1734.

[2] M.A.J. Chaplain, A. Gerisch, G. Lolas, A Mathematical and Computational Analysis of a Model of Cancer Cell Invasion of Tissue, *Journal of Mathematical Biology*

[3] A. Gerisch and M.A.J. Chaplain, Mathematical Modelling of Cancer Cell Invasion of Tissue: Local and non-Local Models and the Effect of Adhesion, *J. Theor. Biol.* 2007.

[4] Kevin J. Painter and Thomas Hillen, Volume-Filling and Quorum-Sensing in Models for Chemosensitive Movement, *Canadian Applied Mathematics Quarterly*, Vol. 10, No. 4, (2002).

[5] Global Existence for a Parabolic Chemotaxis Model with Prevention of Overcrowding, *Advances in Applied Mathematics*, 26 (2001), pp. 280 – 301.

[6] J.D. Murray, Mathematical Biology I An Introduction, Springer, 2002.