CompuCell3D Manual and Tutorial

Version 3.4.1

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

Biocomplexity Institute and Department of Physics, Indiana University, 727 East 3rd Street, Bloomington IN, 47405-7105, USA

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 don't 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.

I. 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.

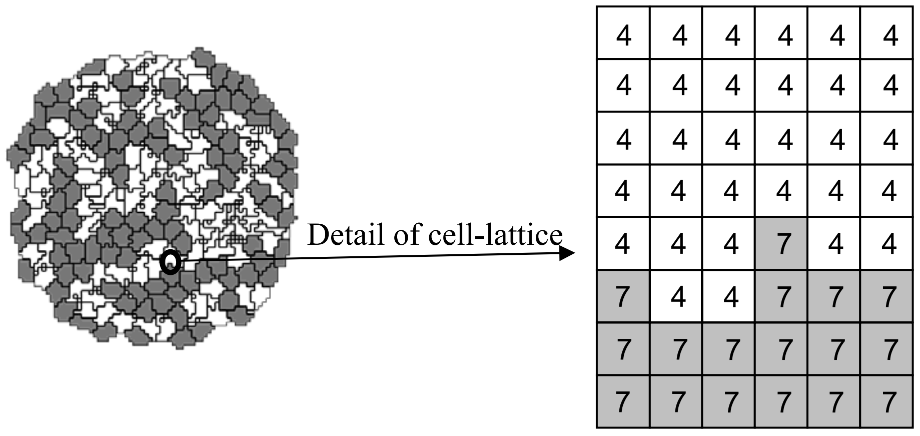


Figure 1. Detail of a typical two-dimensional GGH cell-lattice configuration. Each colored domain represents a single spatially-extended cell. The detail shows that each generalized cell is a set of cell-lattice sites (or *pixel*), $\vec{i}$, with a unique index, $\sigma\left(\vec{i}\right)$, here 4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

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, sub-compartments 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.

II. 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)***.

III. 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 (Figure 1), which reside on a single *cell lattice* and may correspond to biological cells, sub-compartments 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, $\vec{i}$, the *cell index* of the generalized cell occupying pixel $\vec{i}$ by $\sigma(\vec{i})$ and the *type* of the generalized cell $\sigma(\vec{i})$ by $\tau(\sigma(\vec{i}))$. 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 cell's 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 e*ffective energy*, which determines a generalized cell's 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 generalized 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 generalized-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* (*PDE*s). 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 generalized cells. The movement of cells and variations in local diffusion constants (or diffusion tensors in anisotropic ECM) mean 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.

III.A. Effective Energy

The core of GGH simulations is the *effective energy*, which describes cell behaviors 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 minimize the adhesion energy.

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

$$H_{\text{boundary}} = \sum_{\substack{\vec{i},\vec{j}\\ \text{neighbors}}} J\left(\tau\left(\sigma\left(\vec{i}\right)\right),\tau\left(\sigma\left(\vec{j}\right)\right)\right)\left(1-\delta\left(\sigma\left(\vec{i}\right),\sigma\left(\vec{j}\right)\right)\right), \tag{1}$$

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

$$J\left(\tau\left(\sigma\right),\tau\left(\sigma'\right)\right)=J\left(\tau\left(\sigma'\right),\tau\left(\sigma\right)\right). \tag{2}$$

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:

$$H_{\text{constraint}} = \lambda\left(value - target\_value\right)^2. \tag{3}$$

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-minimizing 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 generalized cells from their target volumes:

$$H_{\text{vol}} = \sum_{\sigma} \lambda_{\text{vol}}(\sigma)\left(v(\sigma) - V_{\text{t}}(\sigma)\right)^2 , \tag{4}$$

where for cell σ, $\lambda_{\text{vol}}(\sigma)$ denotes the *inverse compressibility* of the cell, $v(\sigma)$ is the number of pixels in the cell (its *volume*), and $V_{\text{t}}(\sigma)$ is the cell's *target volume*. This constraint defines $P \equiv -2\lambda\left(v(\sigma) - V_{\text{t}}(\sigma)\right)$ as the *pressure* inside the cell. A cell with $v < V_{\text{t}}$ has a positive internal pressure, while a cell with $v > V_{\text{t}}$ has a negative internal pressure.

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

$$H_{\text{surf}} = \sum_{\sigma} \lambda_{\text{surf}}(\sigma)\left(s(\sigma) - S_{\text{t}}(\sigma)\right)^2 , \tag{5}$$

where $s(\sigma)$ is the surface area of cell σ, S_{t} is its target surface area, and $\lambda_{\text{surf}}(\sigma)$ 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*:

$$\begin{aligned} H_{\text{GGH}} = &\sum_{\substack{\vec{i},\vec{j} \\ \text{neighbors}}} J\left(\tau\left(\sigma\left(\vec{i}\right)\right), \tau\left(\sigma\left(\vec{j}\right)\right)\right)\left(1 - \delta\left(\sigma\left(\vec{i}\right), \sigma\left(\vec{j}\right)\right)\right) \\ &+ \sum_{\sigma} \lambda_{\text{vol}}(\sigma)\left(v(\sigma) - V_{\text{t}}(\sigma)\right)^2 . \end{aligned} \tag{6}$$

III.B. 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, $\vec{i}$, randomly from the cell lattice, and then randomly select one of its neighboring pixels, $\vec{i}'$, as a *source* pixel. If they belong to the same generalized cell (*i.e.*, if $\sigma(\vec{i}) = \sigma(\vec{i}')$) we do not need copy index. Otherwise the cell containing the source pixel $\sigma(\vec{i}')$ 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.

[1] Because of lattice discretization 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 .

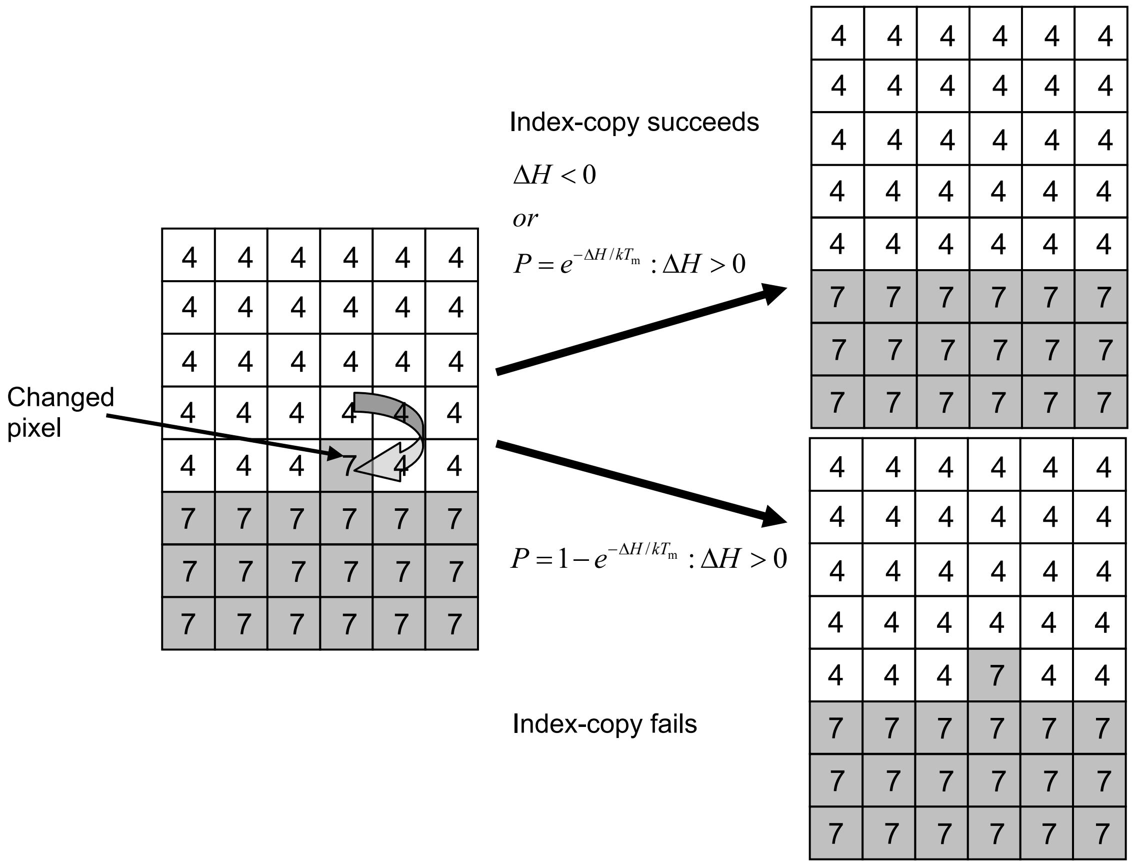

Index-copy succeeds
$\Delta H < 0$
or
$P = e^{-\Delta H/kT_m} : \Delta H > 0$
Changed pixel
$P = 1 - e^{-\Delta H/kT_m} : \Delta H > 0$
Index-copy fails

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:

$$P\left(\sigma\left(\vec{i}\right) \rightarrow \sigma\left(\vec{i}'\right)\right) = \left\{\exp\left(-\mathit{\Delta} H / T_{\mathrm{m}} : \mathit{\Delta} H > 0;\ 1 : \mathit{\Delta} H \leq 0\right)\right\}, \tag{7}$$

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 $\Delta H / T_{\mathrm{m}}$. MCS and experimental time are proportional in biologically-meaningful situations ***(87-90)***.

III.C. 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 ($\vec{i}'$) and destination ($\vec{i}$) 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 pixel's neighbors ($\vec{i}''$). If the neighbor pixel belongs to a different generalized cell from the target pixel, *i.e.*, when $\sigma(\vec{i}'') \neq \sigma(\vec{i})$ (see equation (1)), we decrease ΔH by $J\left(\tau\left(\sigma\left(\vec{i}\right)\right), \tau\left(\sigma\left(\vec{i}''\right)\right)\right)$. If the neighbor belongs to a cell different from the source pixel ($\vec{i}'$) we increase ΔH by $J\left(\tau\left(\sigma\left(\vec{i}'\right)\right), \tau\left(\sigma\left(\vec{i}''\right)\right)\right)$.

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

$$\begin{aligned}
&\Delta H_{\text{vol}} = H_{\text{vol}}^{\text{new}} - H_{\text{vol}}^{\text{old}} = \\
&\lambda_{\text{vol}}\left[\left(v\left(\sigma\left(\vec{i}'\right)\right)+1-V_{\text{t}}\left(\sigma\left(\vec{i}'\right)\right)\right)^2 + \left(v\left(\sigma\left(\vec{i}\right)\right)-1-V_{\text{t}}\left(\sigma\left(\vec{i}\right)\right)\right)^2\right] \\
&-\lambda_{\text{vol}}\left[\left(v\left(\sigma\left(\vec{i}'\right)\right)-V_{\text{t}}\left(\sigma\left(\vec{i}'\right)\right)\right)^2 + \left(v\left(\sigma\left(\vec{i}\right)\right)-V_{\text{t}}\left(\sigma\left(\vec{i}\right)\right)\right)^2\right] \\
&= \lambda_{\text{vol}}\left[\left\{1+2\left(v\left(\sigma\left(\vec{i}'\right)\right)-V_{\text{t}}\left(\sigma\left(\vec{i}'\right)\right)\right)\right\} + \left\{1-2\left(v\left(\sigma\left(\vec{i}\right)\right)-V_{\text{t}}\left(\sigma\left(\vec{i}\right)\right)\right)\right\}\right],
\end{aligned} \tag{8}$$

where $v\left(\sigma\left(\vec{i}'\right)\right)$ and $v\left(\sigma\left(\vec{i}\right)\right)$ denote the volumes of the generalized cells containing the source and target 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 quasi-local, 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.

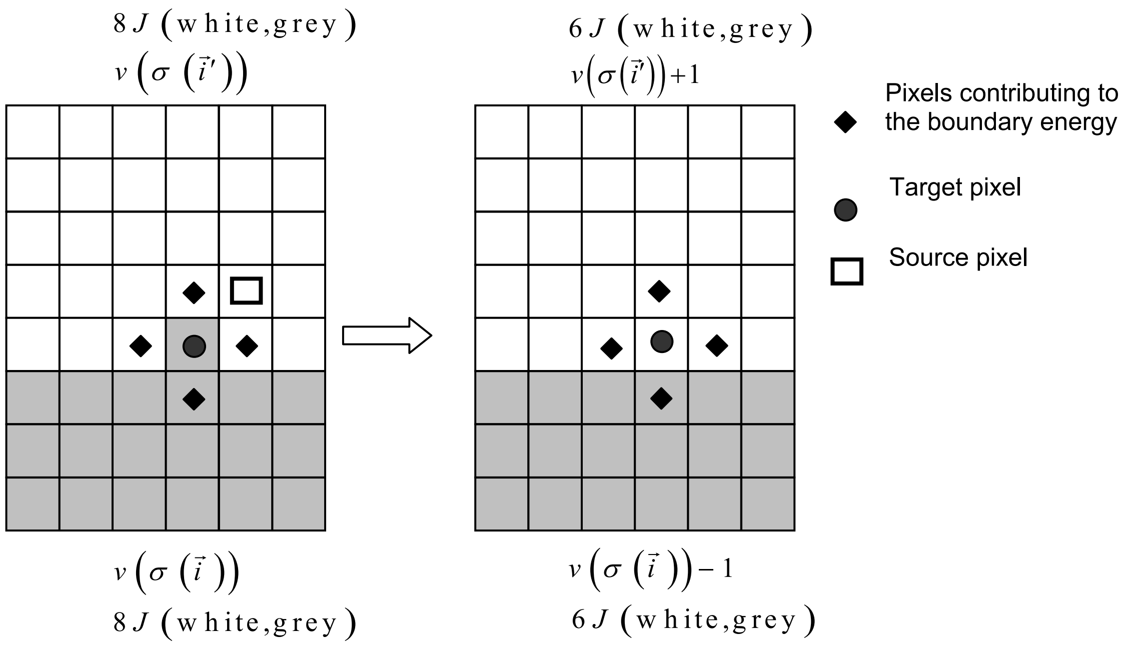


Figure 3. Calculating changes in the boundary energy and the volume-constraint energy on a nearest-neighbor square lattice.

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.

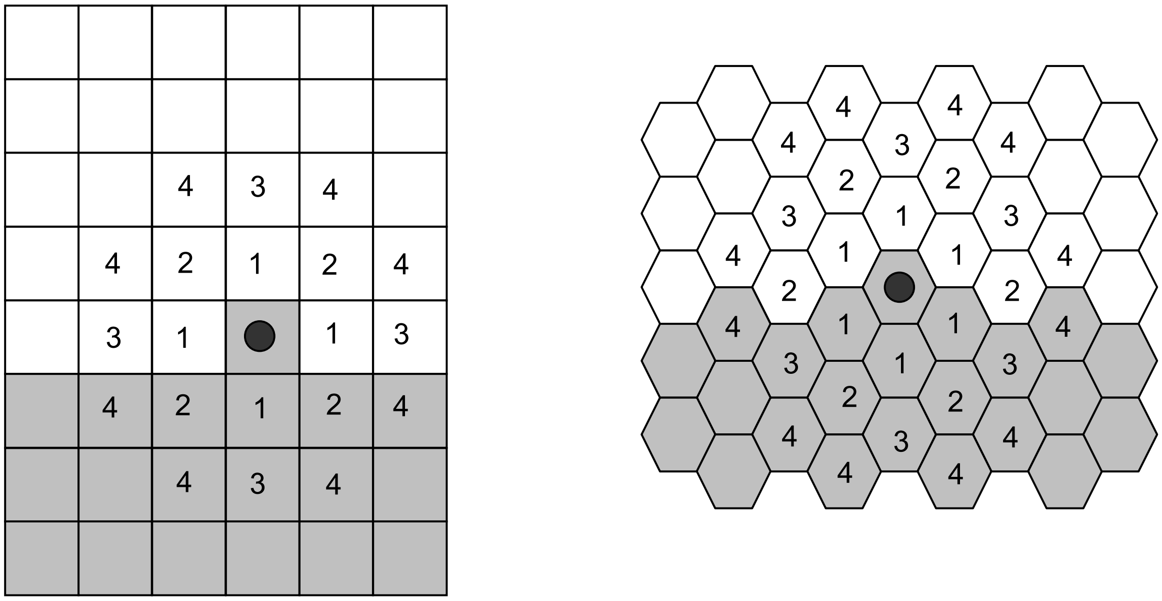


Figure 4. Locations of *n*th-nearest neighbors on a 2D square lattice and a 2D hexagonal lattice.

	2D Square Lattice		2D Hexagonal 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.

IV. CompuCell3D

One advantage of the GGH model over alternative techniques is its mathematical simplicity. We can implement fairly easily a computer program that initializes 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.

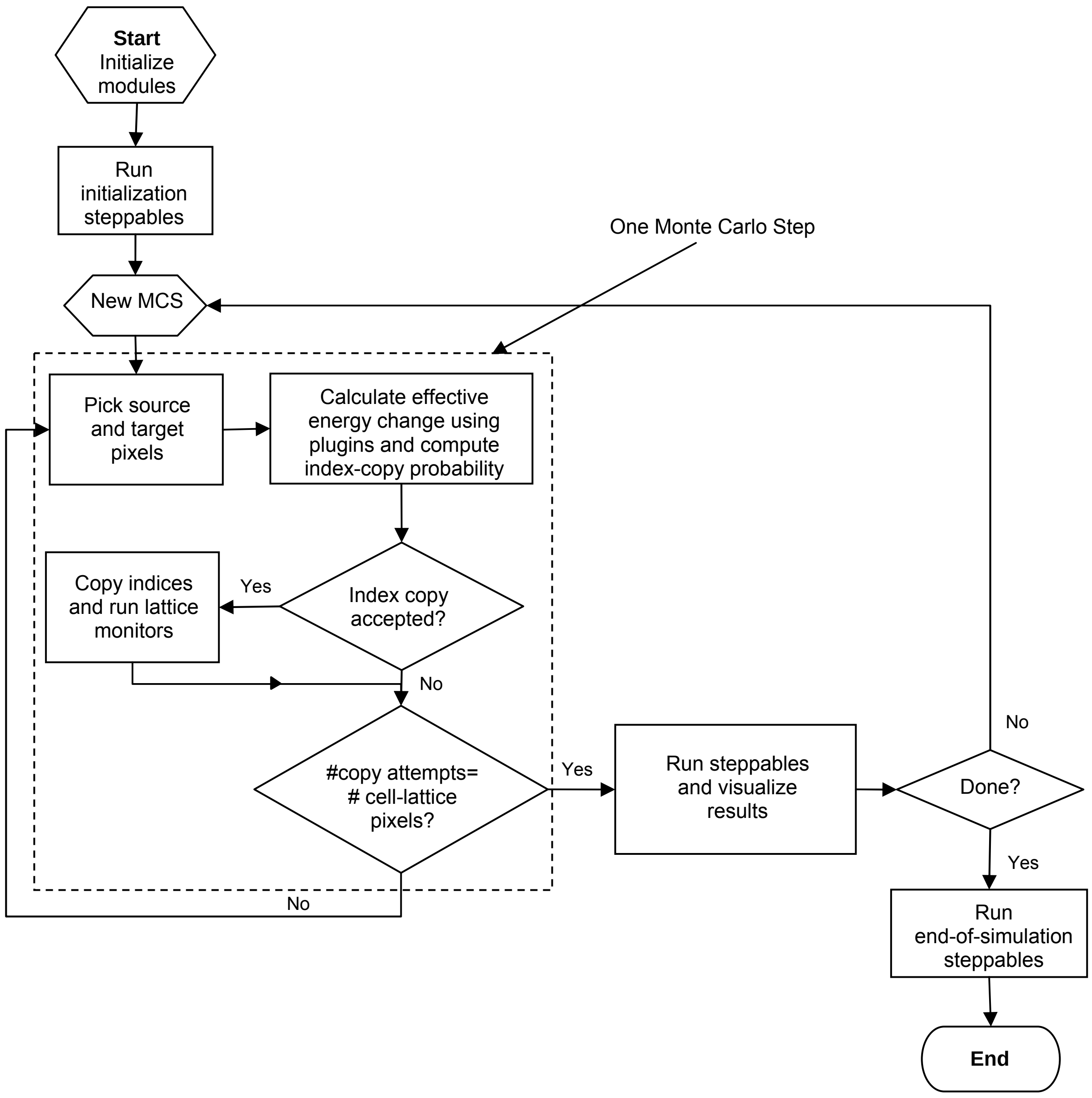
To overcome these problems, we developed CompuCell3D as a framework for GGH simulations *(**91**, **92**)*. Unlike specialized research code, CompuCell3D is a *simulation environment* that allows researchers to rapidly build and run shareable GGH-based simulations. It greatly reduces the need to develop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

CompuCell3D supports non-programmers by providing visualization tools, an *eXtensible Markup Language* (*XML*) interface for defining simulations, and the ability to extend the framework through specialized 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 initializes 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 generalized 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 generalized cells' volumes. When implicit plugin dependencies exist, CompuCell3D automatically loads and initializes 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.



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
Run steppables and visualize results
No
Done?
Yes
No
Run end-of-simulation steppables
End

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

CompuCell3D includes a *Graphical User Interface* (*GUI*) and visualization 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 visualizations of generalized cells, fields or various vector plots while the simulation is running and save them automatically for post-processing.

V. 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.

V.A A Short Introduction to XML

XML is a text-based data-description language, which allows standardized 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>
```

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* characterizing 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.

V.B 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 characterize 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

[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 $\boxed{\text{boxed}}$. Punctuation at the end of boxes is implicit.

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.

```
<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>
```

Lattice Section

Plugins Section

Steppables Section

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

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 generalized-cell-type names to numeric cell `TypeId`s for all generalized-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, $J(\mathtt{TypeName1}, \mathtt{TypeName2})$, 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 simple regions filled with generalized cells.

The `UniformInitializer` steppable in the grain-growth simulation defines one or more rectangular (parallelepiped in 3D) regions filled with generalized cells of user selectable types and sizes. We enclose each region definition within a `<Region>` tag

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

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) generalized cells and the `<Gap>` tag pair creates space between neighboring cells. The `<Types>` tag pair lists the types of generalized cells. The grain-growth simulation uses only one cell type, `Grain`, but we can also initialize cells using types randomly chosen from the list, as in Listing 2.

```
<Steppable Type="UniformInitializer">
   <Region>
      <BoxMin x="10" y="10" z="0"/>
      <BoxMax x="90" y="90" z="1"/>
      <Gap>0</Gap>
      <Width>5</Width>
      <Types>Condensing,NonCondensing</Types>
   </Region>
</Steppable>
```

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

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 generalized cells of that type. For example,

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

will allocate approximately 2/3 of the generalized 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, uninitialized pixels have type `Medium`.

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

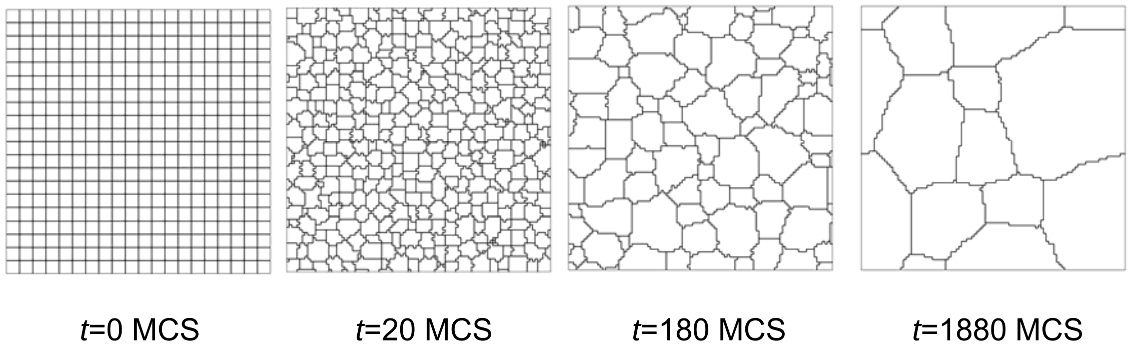


Figure 6. Snapshots of the cell-lattice configuration for the grain-growth simulation on a 100 x 100 pixel 3rd-neighbor square lattice, as specified in Listing 1. Boundary conditions are periodic in both directions.

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 x 100 x 100 lattice we only need to make the following replacements in Listing 1:

```
<Dimensions x="100" y="100" z="1"/> →
<Dimensions x="100" y="100" z="100"/>
```

and,

```
<BoxMax x="100" y="100" z="1"/> → <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> → <NeighborOrder>1</NeighborOrder>
```

Figure 7 shows snapshots for this simulation.

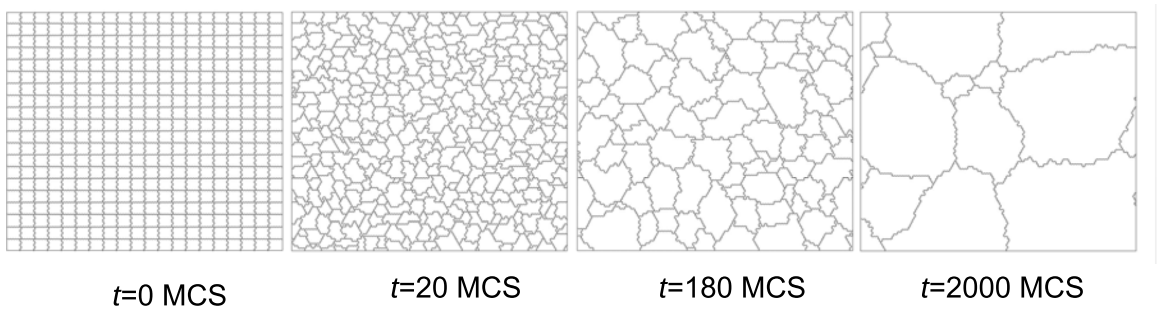


Figure 7. Snapshots of the cell-lattice configuration for the grain-growth simulation on a 100 x 100 pixel 1st -neighbor hexagonal lattice as specified in Listing 1 with substitutions described in the text. The *x* and *y* length units in an hexagonal lattice differ, resulting in differing *x* and *y* 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.

V.C Cell-Sorting Simulation

Cell sorting is an experimentally-observed phenomenon in which cells with different adhesivities are randomly mixed and reaggregated. 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 generalized 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).

```
<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>
```

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.

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 $\lambda_{vol} = 2.0$ 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 initialize 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 center 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)***.

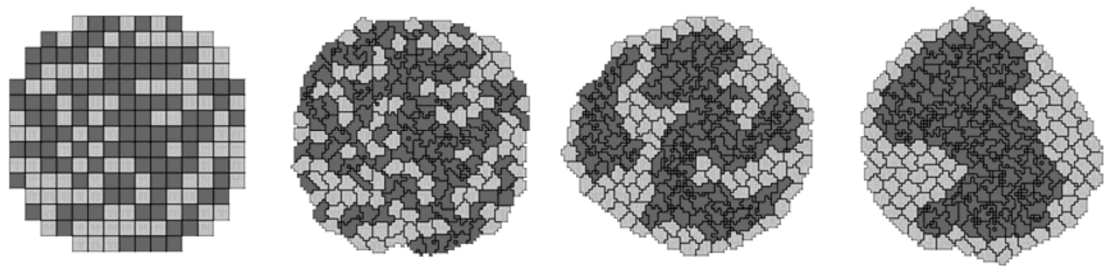


Figure 8. Snapshots of the cell-lattice configurations for the cell-sorting simulation in Listing 3. The boundary-energy hierarchy drives `NonCondensing` (light grey) cells to surround `Condensing` (dark grey) cells. The white background denotes surrounding `Medium`.

V.D 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' behaviors depend on the fields. To illustrate the use of fields, we model the *in vitro* behavior 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 behaviors 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 *chemorepellant*) which repels the bacterium *(**97**)*.

Listing 4 shows the 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>
```

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

The simulation has five generalized-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 generalized-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 generalized 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 generalized-cell type, specifying the type name and the target volume (or target surface area), and λ_{vol} (or λ_{surf}) for that generalized-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 generalized cell of a given type responds to a field. The `Chemotaxis` plugin biases a cell's motion up or down a field gradient by changing the calculated effective-energy change used in the acceptance function, equation (7). For a field $c(\vec{i})$:

$$\Delta H_{\text{chem}} = -\lambda_{\text{chem}}\left(c(\vec{i}) - c(\vec{i}')\right), \tag{9}$$

where $c(\vec{i})$ is the chemical field at the index-copy target pixel, $c(\vec{i}')$ the field at the index-copy source pixel, and λ_{chem} the strength and direction of chemotaxis. If $\lambda_{\text{chem}} > 0$ and $c(\vec{i}) > c(\vec{i}')$, 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 $\sim \lambda_{\text{chem}}\vec{\nabla}c$. 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*).

$$\Delta H_{\text{chem}} = -\lambda_{\text{chem}} \left(c(\vec{i}) - c(\vec{i}') \right)$$
$$\lambda_{\text{chem}} > 0$$

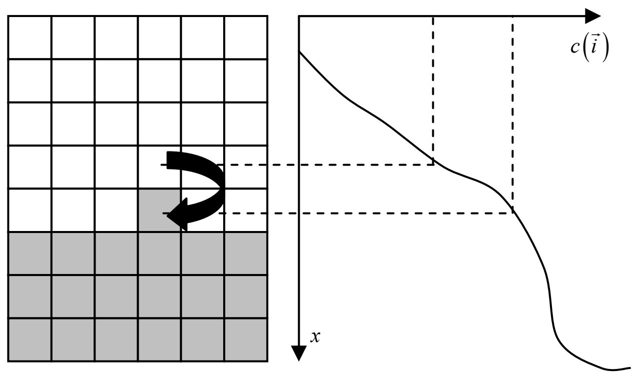


Figure 9. Connecting a field to GGH dynamics using a chemotaxis-energy term. The difference in the value of the field c at the source, $\vec{i}'$, and target, $\vec{i}$, pixels changes the ΔH of the index-copy attempt. Here $c(\vec{i}) > c(\vec{i}')$ and $\lambda > 0$, so $\Delta H_{\text{chem}} < 0$,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

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

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

As in the `Chemotaxis` plugin, we may define the behaviors 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, $\vec{i}$, is occupied by a cell, $\sigma(\vec{i})$, 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 generalized-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 generalized 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 x 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 x 6 square to which we attach additional pixels.

```
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
```

Listing 5. Simple PIF initializing two cells, one each of type `Bacterium` and `Amoeba`.

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:

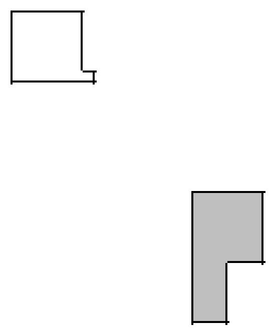


Figure 10. Initial configuration of the cell lattice based on the PIF in Listing 5.

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.

```
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
```

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'.

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.

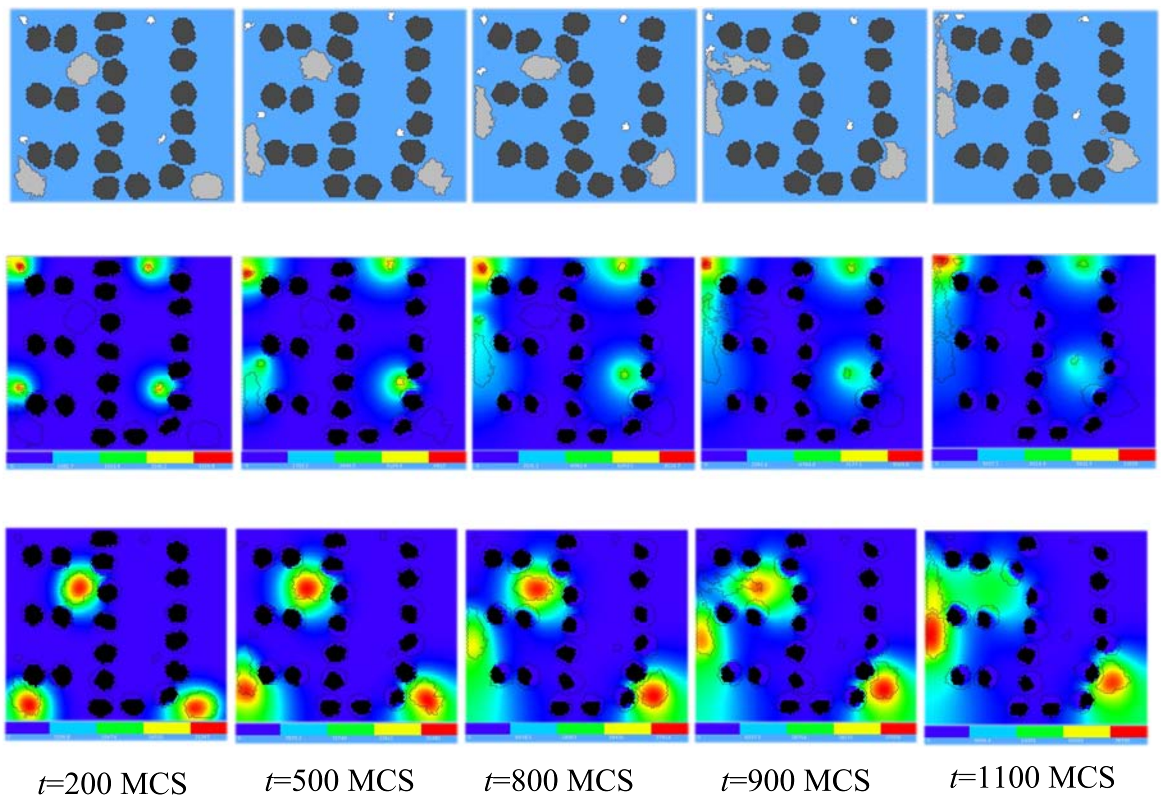


Figure 11. Snapshots of the bacterium-and-macrophage simulation from Listing 4 and the PIF in Listing 6 saved in the file 'bacterium_macrophage_2D_wall_v3.pif'. The upper row shows the cell-lattice configuration with the `Macrophage`s in grey, `Bacteria` in white, `red` blood cells in dark grey and `Medium`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

VI. 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.

VI.A 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*).

VI.B 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.

```
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)
```

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

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 initialization 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.

VI.C 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**).

```
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)
```

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**.

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 initializes 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 generalized 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.

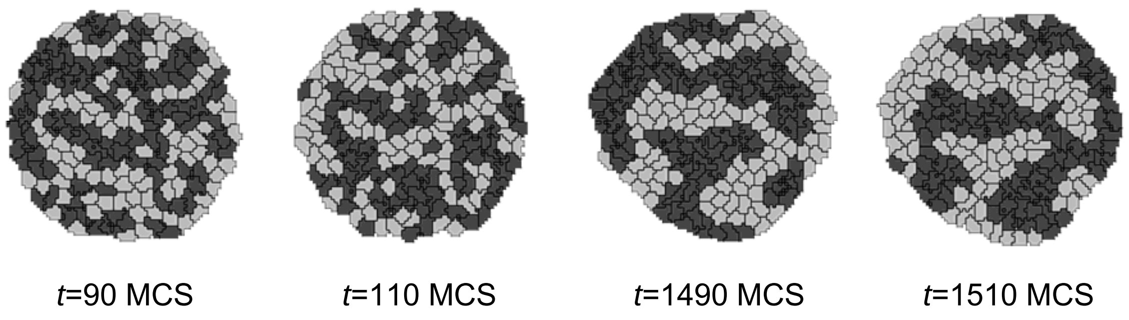


Figure 12. Results of the Python cell-type-oscillator simulation using the `TypeSwitcherSteppable` steppable implemented in Listing 8 in conjunction with the CC3DML cell-sorting simulation in Listing 3. Cells exchange types and corresponding adhesivities and colors every 100 MCS; *i.e.*, between *t*=90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

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**.

```
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)
```

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**.

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 *`SectionName`*`ParseData`. 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.

VI.D 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.

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**)*.

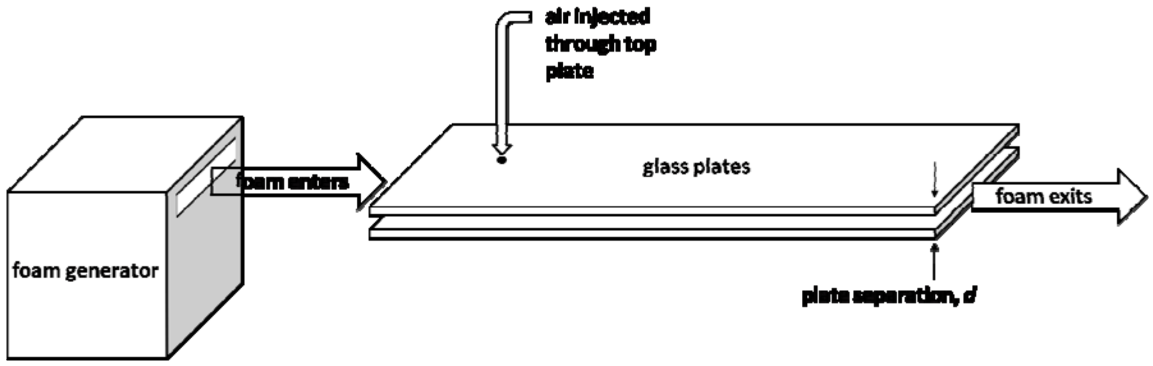


Figure 13. Schematic of experiment for studying quasi-2D foam flow.

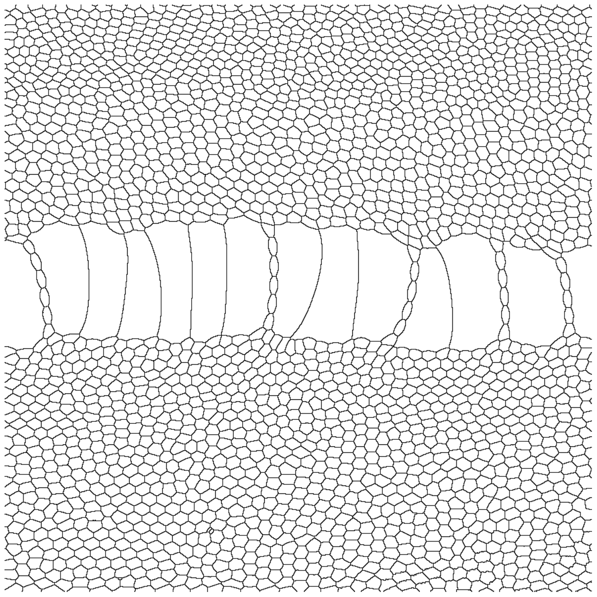


Figure 14. Detail of processed experimental image of flowing quasi-2D bubbles. Image size is 15 cm x 15 cm.

Generalized 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 generalized 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).

```
<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>
```

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

The CC3DML configuration file is simple: it initializes 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 `CenterOfMass` plugin in CompuCell3D actually calculates $\vec{x}_\sigma^C$, the centroid of the generalized cell multiplied by volume of the cell:

$$\vec{x}_\sigma^C = \sum_{\vec{i}} \vec{i}\,\delta\left(\sigma'(\vec{i}),\sigma\right), \tag{11}$$

so the actual centroid of the bubble is:

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

The ability to track a generalized-cell's 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'.

```
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)
```

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

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 bubble's `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 `BubbleNucleator`, `AirInjector`, and `BubbleCellRemover` and save them in the file 'foamairSteppables.py'.

Listing 12 shows the code for the `BubbleNucleator` steppable.

```
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()
```

Listing 12. Python code for the `BubbleNucleator` steppable, saved in the file 'foamairSteppables.py'. This module creates bubbles at points with random *y* coordinates and *x* coordinate of 3.

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 $\lambda_{vol} = lambdaVolume$:

```
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.

```
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
```

Listing 13. Python code for the *`AirInjector`* steppable which simulates air injection into the bubble currently occupying the cell-lattice pixel at location (x,y,z). Air injection begins after 5000 MCS to allow the channel to partially fill with bubbles. The steppable is saved in file 'foamairSteppables.py'.

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:

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

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

```
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`:

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

and then increment that bubble's 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).

```
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
```

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. Like the other steppables in the foam-flow simulation, we save it in the file 'foamairSteppables.py'.

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 generalized-cell inventory:

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

and in the fourth line we make a list containing all of the generalized 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 cell's centroid is greater than the `cutoffValue:`

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

and, if it is, set that cell's `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.

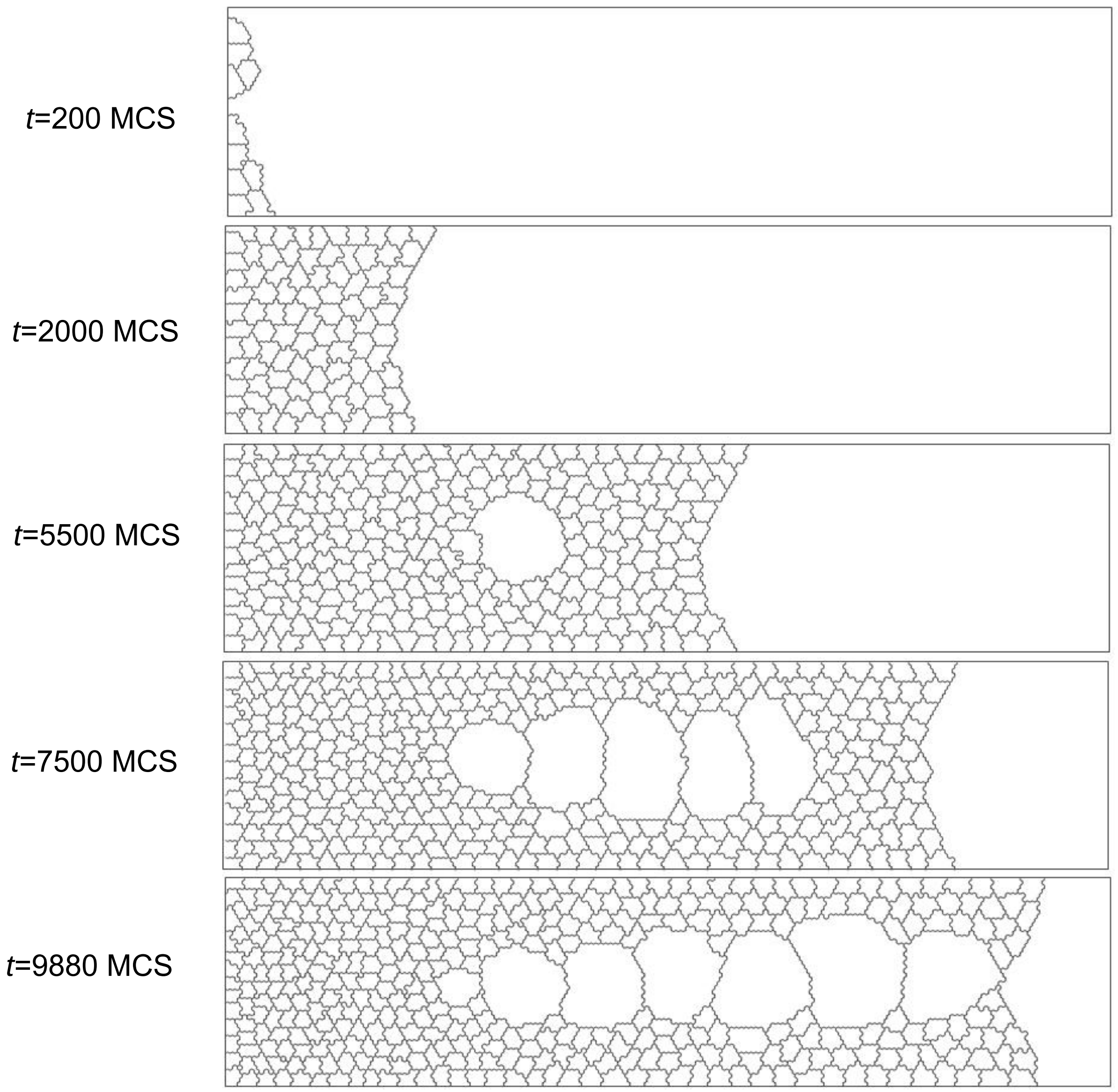


Figure 15. Results of the foam-flow simulation on a 2D 3rd-neighbor hexagonal lattice. Simulation code is given in Listing 10,Listing 11, Listing 12, Listing 13 and Listing 14.

VI.E. 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:

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

where $\left[FGF\right]$ 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:

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

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 generalized 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. *E.g.*, in the current simulation, we will record data on each cell division in a list attached to each cell. Generalized 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.

```
<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>
```

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

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 **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()

#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)
```

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

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.

```
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
```

Listing 17. Python code for the `VolumeConstraintSteppable`, saved in the file 'cellsort_2D_field_modules.py', for the diffusing-field-based cell-growth simulation. The `VolumeConstraintSteppable` *provides* dynamic volume constraint parameters for each cell, which depend on the cell type and the chemical field concentration at the cell's centroid.

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` ($\lambda_{\text{vol}}(\sigma) = 2.0$) 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:

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

and retrieve the FGF concentration at that point:

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

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

```
cell.targetVolume+=0.01*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.

```
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)
```

Listing 18. Python code for the `CellsortMitosis` plugin for the diffusing-field-based cell-growth simulation, saved in the file 'cellsort_2D_field_modules.py'. The plugin handles division of cells when they reach a threshold volume.

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. $\lambda_{vol.}$ 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.

```
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)
```

Listing 19. Python code for the `MitosisData` class for the diffusing-field-based cell-growth simulation, saved in the file 'cellsort_2D_field_modules.py'. `MitosisData` objects store information about cell divisions involving the parent and daughter cells.

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.

```
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
```

Listing 20. The Python code for the `MitosisDataPrinter` steppable for the diffusing-field-based cell-growth simulation, saved in the file 'cellsort_2D_field_modules.py'. The steppable prints the cell-division history for dividing cells (see Figure 18).

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.

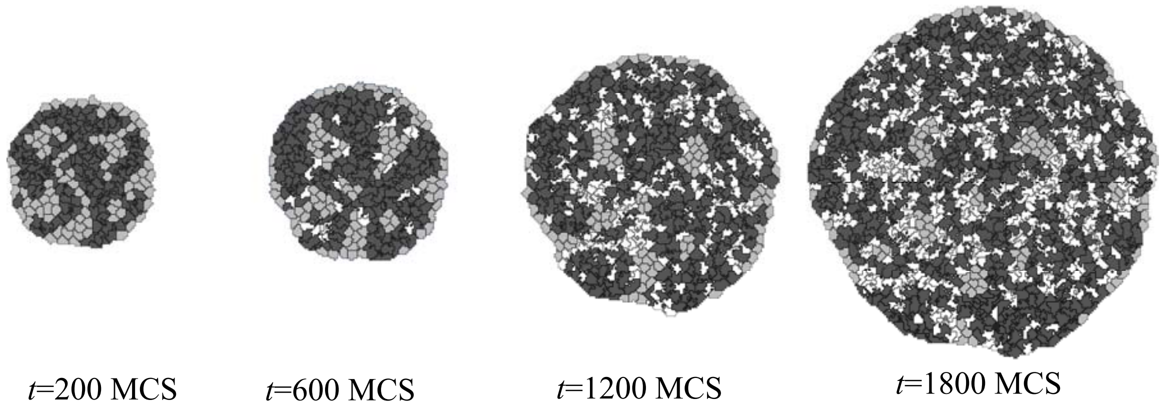


Figure 16. Snapshots of the diffusing-field-based cell-growth simulation obtained by running the CC3DML file in Listing 15 in conjunction with the Python file in Listing 16. As the simulation progresses, `NonCondensing` cells (light gray) secrete diffusing chemical, FGF, which causes `Condensing` (dark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

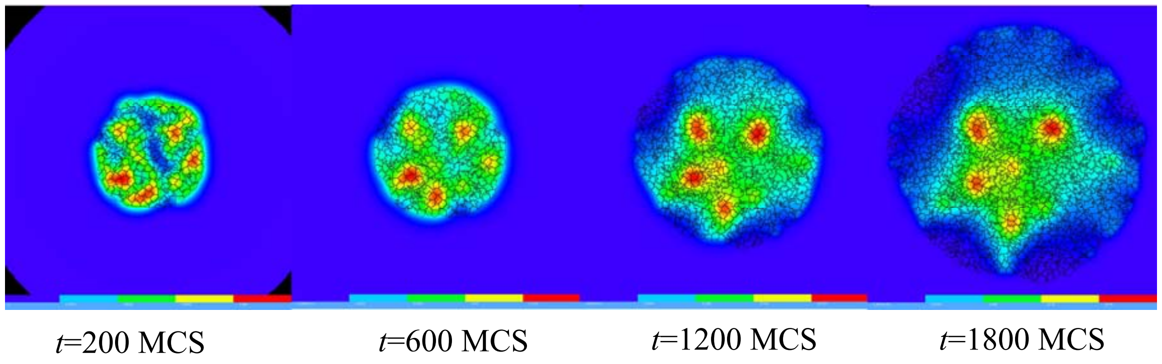
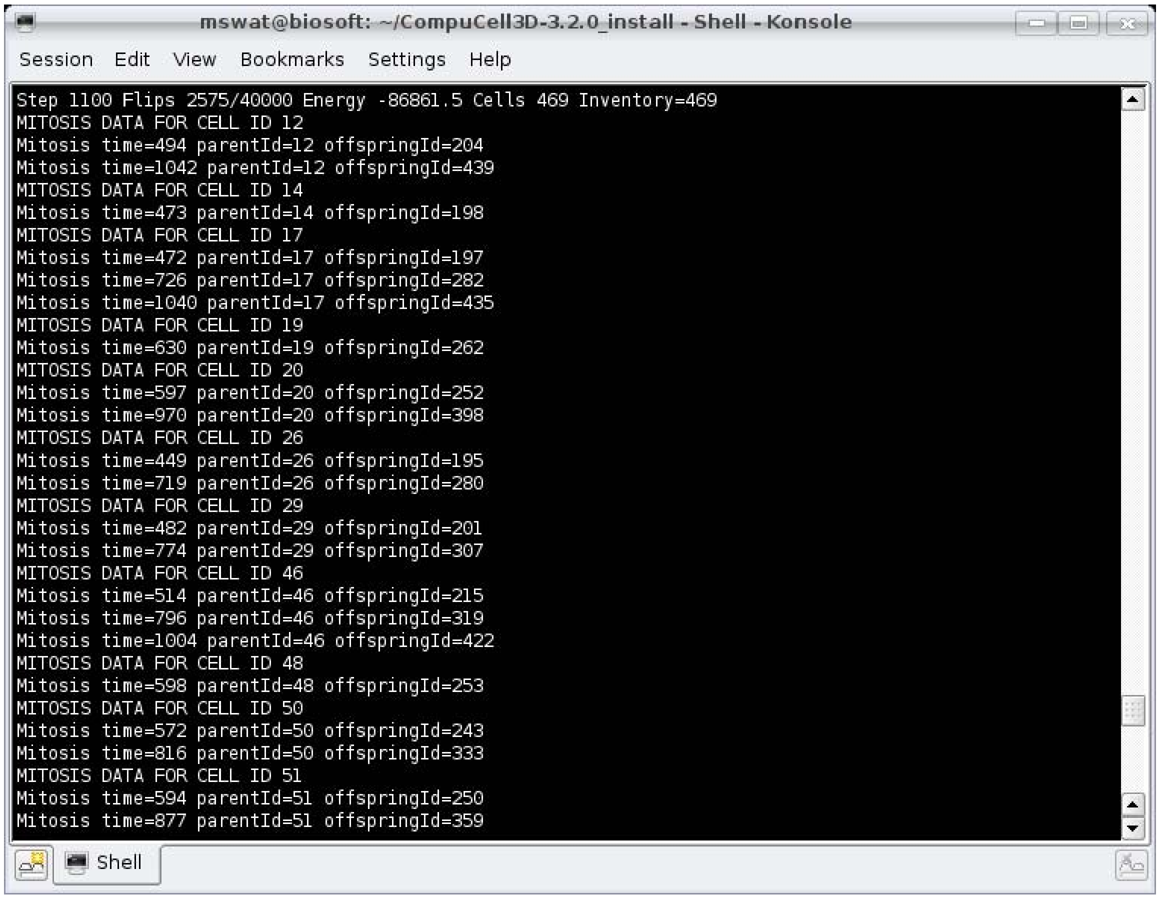


Figure 17. Snapshots of FGF concentration in the diffusing-field-based cell-growth simulation obtained by running the CC3DML file in Listing 15 in conjunction with the Python files in Listing 16, Listing 17**,** Listing 18, Listing 19**,** Listing 20. The bars at the bottom of the field images show the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


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***.

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.

VII. 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, CompuCell3D's 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.

VIII. 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 University's 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.

IX. XML Syntax of CompuCell3D modules

IX.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>
<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 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. 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 x number of lattice sites attempts per MCS – in our case one MCS is 101x101x1 spin-flip attempts. To set 2.5x101x101x1 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 artifacts 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 seveare 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 initialize 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 customize 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
$P = e^{-(\Delta E-\delta)/kT}$ where δ and *k* are specified by Offset and KBoltzman tags respectively. By default δ=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 Pots section:
```
<AcceptanceFunctionName>FirstOrderExpansion</AcceptanceFunctionName>
```

IX.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 (#D) 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 sqaure 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:

$$(x,y,z)_{hex} = \left(x, \frac{\sqrt{3}}{2} y + \frac{\sqrt{3}}{3}, \frac{\sqrt{6}}{3} z \right) \quad for \quad y \quad odd \quad \wedge \quad z \quad odd$$

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

$$(x,y,z)_{hex} = \left(x, \frac{\sqrt{3}}{2} y, \frac{\sqrt{6}}{3} z \right) \quad for \quad y \quad odd \quad \wedge \quad z \quad even$$

$$(x,y,z)_{hex} = \left(x + \frac{1}{2}, \frac{\sqrt{3}}{2} y, \frac{\sqrt{6}}{3} z \right) \quad for \quad y \quad even \quad \wedge \quad z \quad even$$

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:

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

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

where $S_{hex-unit}$ denotes length of the hexagon and $L_{hex-unit}$ denotes a distance between centers 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:

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

and unit length by:

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

IX.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.

IX.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 TypeId are consecutive positive integers (e.g. 0,1,2,3...). Medium is traditionally given TypeId=0 but this is not a requirement.

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.

IX.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:

$$E_{volume} = \lambda_{volume}(V_{cell} - V_{target})^2$$

$$E_{surface} = \lambda_{surface}(S_{cell} - S_{target})^2$$

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

IX.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 Volume Tracker) or Surface (calls Surface Tracker) 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 explicitely invoke VolumeTracker or Surface Tracker plugin with the following syntax:

```
<Plugin Name=”VolumeTracker”/>
<Plugin Name=”SurfaceTracker”/>
```

IX.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.

IX.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>
```

IX.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"/>
```

IX.2.7. SurfaceLocalFlex Plugin

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

Example syntax:

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

IX.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 the drawing below:

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 19. 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 , 7 pixels, cell 1 4 pixels and cell 2 6 pixels.

Example syntax:

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

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

IX.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:

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

where

$c\left(\vec{x}_{neighbor}\right)$, $c\left(\vec{x}\right)$ 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.

Ocassionally 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:

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

or

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

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.

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)

IX.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.

IX.2.11. CenterOfMass Plugin

This plugin monitors changes n the lattice and updates centroids of the cell:
$x_{CM}=\sum_i x_i \ , \ y_{CM}=\sum_i y_i \ , \ z_{CM}=\sum_i z_i$ 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:

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

This plugin is aware of boundary conditions and centroids are calculated properly regardless which boundary conditions are used.

IX.2.12. Contact Energy

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:

$$E_{adhesion} = \sum_{i,j,neighbors} J(\tau_{\sigma(i)},\tau_{\sigma(j)})(1-\delta_{\sigma(i),\sigma(j)}),$$

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 a contact energy per unit area of 3 and Foam and Medium and Medium and Medium have a contact energy of 0 per unit area.

IX.2.13. ContactLocalProduct Plugin

This plugin calculates contact energy based on local (i.e. per cell) cadhering 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:

$$E = \sum_{i,j-neighbors} \left(E_{offset} - k_{\sigma(i),\sigma(j)} f\left(N(i),N(j)\right)\right) \quad if \quad \sigma(i) \quad \wedge \quad \sigma(j) \quad \neq \quad medium$$

$$E = \sum_{i,j-neighbors} \left(E_{offset} - k_{\sigma(i),\sigma(j)} \right) \quad if \quad \sigma(i) \quad \vee \quad \sigma(j) \quad = \quad medium$$

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

In the case of the second formula $E_{offset} - k_{\sigma(i),\sigma(j)}$ 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 - $N(i)^2 N(j)^2$ and Min - $min(N(i),N(j))$). EnergyOffset can be set to user specified value using above syntax. Depth 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 cell2 denotes cadherin for another cell). Then in the Expression tag we give mathematical expression involving the two cadherin levels. The expression should follow C++ style of writing mathematical expressions.

IX.2.14. 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:

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

where indexes i,j label pixels, $J\left(\sigma(i),\sigma(j)\right)$ 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 f 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 lexographic 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.

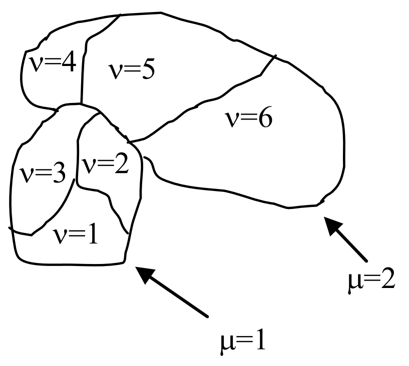


Figure 20. Two compartmental cells (cluster id μ=1 and cluster id μ=2) Compartmentalized cell μ=1 consists of subcells with cell id ν=1,2,3 and compartmentalized cell μ=2 consists of subcells with cell id ν=4,5,6

IX.2.15. ContactCompartment

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

$$E_{contactcompartment} = \sum_{i,j-neighbors} J\left(\sigma\left(\mu_i,\nu_i\right),\sigma\left(\mu_j,\nu_j\right)\right)$$

where *i* and *j* denote pixels , σ(μ,ν) 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. 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 σ(0,0) denotes medium

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

$$\begin{aligned} J\left(\sigma\left(\mu_i,\nu_i\right),\sigma\left(\mu_j,\nu_j\right)\right) = {} & J^{external}\left(\sigma\left(\mu_i,\nu_i\right),\sigma\left(\mu_j,\nu_j\right)\right) \quad if \quad \mu_i \neq \mu_i \\ & J^{internal}\left(\sigma\left(\mu_i,\nu_i\right),\sigma\left(\mu_j,\nu_j\right)\right) \quad if \quad \mu_i = \mu_i \end{aligned}$$

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.

IX.2.16. 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:

```
<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

For 3D simulations we can only define elongation parameters pn 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 ad 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 don't 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 (this 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.

IX.2.17. Connectivity Plugins

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.

Quite often in the simulation we don't need to impose connectivity constraint on all cells. 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"/>
```

IX.2.18. 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. For this reason we 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 CompuCell3D 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.

IX.2.19. 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).

IX.2.20. Elasticity Plugin and ElasticityTracker Plugin

This plugin is responsible for handling the following energy term:

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

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 IncudeTypes 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 functions :

```
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.

IX.2.21.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>
```

As can be seen from above syntax all the keywords correspond to an action in the Player. Project2D sets up the values of the projection on the Player steering bar. Here we set the player to start 2D display in the 'xz' projection with 'y' coordinate set to 50. Borders and contours properties are handled using Border and Contour elements. Specifying cell colors is done using Cell element. VisualControl element allows users to specify zoom factor and screen update and screenshot frequencies. Notice, screen update frequency migh not work properly when using Python script. In this case CompuCell will use whatever screen update frequency was stored in the config file (by default 1). We may also change things such as screen update frequency or screenshot frequency and choose whether or not to close the player after the simulation.

Although the set of allowed changes of player settings is fairly small at the moment we believe that the options that users have right now are quite sufficient for configuring the Player from the XML or python level. We will continue adding new options though.

IX.2.22.BoundaryPixelTracker Plugin

This plugin allows storing list of boundary pixels for each cell. The syntax is as follows:

```
<Plugin Name="BoundaryPixelTracker">
    <NeighborOrder>1</NeighborOrder>
 </Plugin>
```

This plugin is also used by other plugins as a helper module.

IX.2.23. GlobalBoundaryPixelTracker

This plugin tracks boundary pixels of all the cells including medium It is used in a Boundary Walker algorithm where instead of blindly picking pixel copy candidate we pick it from the set of pixels comprising boundaries of non frozen cells. In situations when lattice is large and there are not that many cells it makes sense to use BoundaryWalker algorithm to limit number of "wrong" pixel picks when perfming pixel copy attempts. Take a look at the following example:

```
<Potts>
   <Dimensions x="100" y="100" z="1"/>
   <Anneal>10</Anneal>
   <Steps>10000</Steps>
   <Temperature>5</Temperature>
   <Flip2DimRatio>1</Flip2DimRatio>
   <NeighborOrder>2</NeighborOrder>
   <MetropolisAlgorithm>BoundaryWalker</MetropolisAlgorithm>
   <Boundary_x>Periodic</Boundary_x>
 </Potts>

 <Plugin Name="GlobalBoundaryPixelTracker">
    <NeighborOrder>2</NeighborOrder>
 </Plugin>
```

Here we are using BoundaryWalker algorithm (Potts section) and subsequently we list GlobalBoundaryTracker plugin where we set neighbor order to match that in the Potts section. The neighbor order determines how "thick" the overall boundary of cells will be. The higher this number the more pixels will belong to the boundary.

IX.2.24. PixelTracker Plugin

This plugin allows storing list of all pixels belonging to a given cell. The syntax is as follows:

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

This plugin is also used by other plugins as a helper module.

IX.2.25. MomentOfInertia plugin

This plugin updates tensor of inertia for every cell. Internally it uses parallel axis theorem to calculate most up-to-date tensor of inertia. It can be called directly:

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

However, most commonly it is called indirectly by other plugins like Elongation plugin.

IX.2.26. PolarizationVector plugin

This plugin adds a vector as a cell attribute:

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

IX.2.27. SimpleClock plugin

This plugin adds an integer as a cell attribute:

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

IX.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.

IX.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:

```
 <Steppable Type="UniformInitializer">
    <Region>
       <BoxMin x="35" y="0" z="30"/>
       <BoxMax x="135" y="1" z="430"/>
       <Gap>0</Gap>
       <Width>5</Width>
      <Types>psm</Types>
    </Region>
</Steppable>
```

Above we have defined a 2D rectangular box filled with 5x5 cells touching each other (Gap=0) and having type 'psm'. Notice that if you want to initialize 2D box in xz plane as above then y_min and y_max have to be 0 an 1 respectively.

Users can include as many regions as they want. The regions can overlap each other. Simply cells that are overwritten will either disappear or be truncated.

Additionally users can initialize region with random cell types chosen from provided list of cell types:

```
 <Steppable Type="UniformInitializer">
    <Region>
       <BoxMin x="35" y="0" z="30"/>
       <BoxMax x="135" y="1" z="430"/>
       <Gap>0</Gap>
       <Width>5</Width>
      <Types>psm,ncad,ncam</Types>
    </Region>
</Steppable>
```

When user specifies more than one cell type between <Types> tags (notice, the types have to be separated with ',' and there should be no spaces) then cells for this region will be initialized with types chosen randomly from the provided list (here the choices would be psm, ncad, ncam).
Remark: If one of the type names is repeated inside <Types> element this type will get greater weighting means probability of assigning this type to a cell will be greater. So for

example <Types>psm,ncad,ncam,ncam,ncam</Types> ncam will assigned to a cell with probability 3/5 and psm and ncad with probability 1/5.

IX.3.2. BlobInitializer Steppable

This steppable is used to lay out circular blob of cells on the lattice. This plugin does not have yet the flexibility of UniformInitializer but this will change in the future release. Original syntax of this plugin looks as follows:

```
 <Steppable Type="BlobInitializer">
   <Gap>0</Gap>
   <Width>5</Width>
   <CellSortInit>yes</CellSortInit>
   <Radius>40</Radius>
 </Steppable>
```

The blob is centered in the middle of th lattice and has radius given by <Radius> parameter all cells are initially squares (or cubes in 3D) - <Width> determines the length of the cube or square side and <Gap> determines space between squares or cubes. <CellSortInit> tag and value 'yes' is used to initialize cells randomly with type id being either 1 or 2. Otherwise all cells will have type id 1. This can be easily modified in Python .

The most recent syntax for this plugin gives users additional flexibility in initializing cell field using BlobFieldInitializer:

```
<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 Type="BlobInitializer">
```

Similarly as for the UniformFieldInitializer users can define many regions each of which is a blob of a particular center point , radius and list of cell types that will be assigned to cells forming the blob.

IX.3.3. PIF Initializer

To initialize the configuration of the simulation lattice you can can write your own **lattice initialization file**. Our experience suggests that you will probably have to write your own initialization files rather than relying on built-in initializers. The reason is simple: the built-in initializers implement very simple cell layouts, and if you want to study more complicated cell arrangements, the built-in initializers will not be very helpful. Therefore we encourage you to learn how to prepare lattice initialization files. Again, file definition is not complicated and we will explain every step. The lattice initialization file tells CompuCell3D how to lay out assign the simulation lattice pixels to cells.

The **Potts Initial File** (***PIF***) is a simple file format that we created for easy specification of initial cell positions. The PIF consists of multiple lines of the following format:

cell# celltype x1 x2 y1 y2 z1 z2

Where cell# is the unique integer index of a cell, celltype is a string representing the cell's initial type, and x1 and x2 specify a *range* of x-coordinates contained in the cell (similarly y1 and y2 specify a range of y-coordinates and z1 and z2 specify a range of z-coordinates). Thus each line assigns a rectangular volume to a cell. If a cell is not perfectly rectangular, multiple lines can be used to build up the cell out of rectangular sub-volumes (just by reusing the cell# and celltype).

A PIF can be provided to CompuCell3D by including the steppable object **PIFInitializer.**

Let's look at a PIF example for foams:

```
0 Medium 0 101 0 101 0 0
1 Foam 13 25 0 5 0 0
2 Foam 25 39 0 5 0 0
3 Foam 39 46 0 5 0 0
4 Foam 46 57 0 5 0 0
5 Foam 57 65 0 5 0 0
6 Foam 65 76 0 5 0 0
7 Foam 76 89 0 5 0 0
```

These lines define a background of Medium which fills the whole lattice and is then overwritten by seven rectangular cells of type Foam numbered 1 through 7. Notice that these cells lie in the *xy* plane (z1=0 z2=0 implies that cells have thickness =1) so this example is a two-dimensional initialization.

You can write the PIF file manually, but using a script or program that will write PIF file for you in the language of your choice (Perl, Python, Matlab, Mathematica, C, C++, Java or any other programming language) will save a great deal of typing. You may also use tools like PIFTracer which allow you to "paint" the lattice by tracing regions of the experimental pictures.

Notice, that for compartmental cell model the format of the PIF file is different:

```
Include Clusters
cluster # cell# celltype x1 x2 y1 y2 z1 z2
```

For example:

```
Include Clusters
1 1 Side1 23 25 47 56 10 14
1 2 Center 26 30 50 54 10 14
```

```
1 3 Side2 31 33 47 56 10 14
1 4 Top 26 30 55 59 10 14
1 5 Bottom 26 30 45 49 10 14
2 6 Side1 35 37 47 56 10 14
2 7 Center 38 42 50 54 10 14
2 8 Side2 43 45 47 56 10 14
2 9 Top 38 42 55 59 10 14
2 10 Bottom 38 42 45 49 10 14
```

IX.3.4. PIFDumper Steppable

This steppable does opposite to PIFIitializer – it writes PIF file of current lattice configuration. The syntax similar to the syntax of PIFInitializer:

```
 <Steppable Type="PIFDumper" Frequency=”100”>
    <PIFName>line</PIFName>
</Steppable>
```

Notice that we used Frequency attribute of steppable to ensure that PIF files are written every 100 MCS. Without it they would be written every MCS. The file names will have the following format:
PIFName.MCS.pif

In our case they would be line.0.pif, line.100.pif, line.200.pif etc...

This plugin is actually quite useful For example, let say you want to start your simulation from a more configuration of cells (not rectangular cells as this is the case when you use Uniform or Blob initializers) . In such a case you would run a simulation with a PIFDumper included and once the cell configuration reaches desired shape you would stop and use PIF file corresponding to this state. Once you have PIF initial configuration you may run many simulation starting from the same, realistic initial condition.

Now we will discuss how to use PDE solvers in ComuCell3D. Most of the PDE solvers solve PDE with diffusive terms. Let's take a look at them

IX.3.5. AdvectionDiffusionSolver.

This steppable solves advection diffusion equation on a cell field as opposed to grid. Of course, the inaccuracies are bigger than in the case of PDE being solved on the grid but on the other hand solving the PDE on a cell field means that we associate cocentration with a given cell (not just with a lattice point). This means that as cells move so does the concentration. In other words we get advection for free. The mathematical treatment of this kind of approximation was spelled out in Phys. Rev. E 72, 041909 (2005) paper by D.Dan et al.
The equation solved by this steppable is of the type:

$$\frac{\partial c}{\partial t} = D\nabla^2 c + kc + \vec{v}\cdot\vec{\nabla}c + secretion$$

where c denotes concentration , D is diffusion constant, k decay constant, $\vec{v}$ is velocity field.

In addition to just solving advection-diffusion equation this module allows users to specify secretion rates of the cells as well as different secretion modes. More about it in a moment. First let's see how one uses AdvectionDiffusionSolver:

This is an example syntax:

```
<Steppable Type="AdvectionDiffusionSolverFE">
     <DiffusionField>
        <DiffusionData>
            <FieldName>FGF</FieldName>
            <DiffusionConstant>0.05</DiffusionConstant>
            <DecayConstant>0.003</DecayConstant>
            <ConcentrationFileName>flowFieldConcentration2D.txt</ConcentrationFileName>
            <DoNotDiffuseTo>Wall</DoNotDiffuseTo>
        </DiffusionData>
        <SecretionData>
            <Secretion Type="Fluid">0.5</Secretion>
            <SecretionOnContact Type="Fluid"
            SecreteOnContactWith="Wall">0.3</SecretionOnContact>
        </SecretionData>

    </DiffusionField>
 </Steppable>
```

Inside AdvectionDiffusionSolver you need to define sections that describe a field on which the steppable is to operate. In our case we declare just one diffusion field. Inside the diffusion field we specify sections describing diffusion and secretion. Let's take a look at DiffusionData section first

```
        <DiffusionData>
            <FieldName>FGF</FieldName>
            <DiffusionConstant>0.05</DiffusionConstant>
            <DecayConstant>0.003</DecayConstant>
            <ConcentrationFileName>flowFieldConcentration2D.txt</ConcentrationFileName>
            <DoNotDiffuseTo>Wall</DoNotDiffuseTo>
        </DiffusionData>
```

We give a name (FGF) to the diffusion field – this is required as we will refer to this field in other modules. Next we specify diffusion constant and decay constant.

CAUTION: We use Forward Euler Method to solve these equations. This is not a stable method for solving diffusion equation and we do not perform stability checks. If you enter too high diffusion constant for example you may end up with unstable (wrong) solution. Always test your parameters to make sure you are not in the unstable region.

`ConcentrationFileName` is an optional tag and lets you specify a text file that contains a values of concentration for every pixel. The value of concentratio of the last pixel read for a given cell becomes an overall value of concentration for a cell. That is if cell has , say 8 pixels and you specify different concentration at every pixel, then cell concentration will be the last one read from the file.

Concentration file format is as follows:

x y z c

where x,y,z, denote coordinate of the pixel. c is the value of the concentration.

Example:

0 0 0 1.2
0 0 1 1.4
...

You may also specify cells which will not participate in the diffusion. You do it using `<DoNotDiffuseTo>` tag. In this example you do not let any FGF diffuse into Wall cells. You may of course use as many as necessary `<DoNotDiffuseTo>` tags .

In addition to diffusion parameters we may specify how secretion should proceed. SecretionData section contains all the necessary information to tell CompuCell how to handle secretion. Let's study the example:

```
<SecretionData>
  <Secretion Type="Fluid">0.5</Secretion>
  <SecretionOnContact Type="Fluid" SecreteOnContactWith="Wall">0.3</SecretionOnContact>
</SecretionData>
```

Here we have a definition two major secretion modes. Line `<Secretion Type="Fluid">0.5</Secretion>` ensures that every cell of type Fluid will get 0.5 increase in concentration every MCS. Line
`<SecretionOnContact Type="Fluid" SecreteOnContactWith="Wall">0.3</SecretionOnContact>`
means that cells of type Fluid will get additional 0.3 increase in concentration but only when they touch cell of type Wall. This mode of secretion is called SecretionOnContact.

IX.3.6. FlexibleDiffusionSolver

This steppable is one of the basic and most important modules in CompuCell3D simulations. As the name suggests it is responsible for solving diffusion equation but in addition to this it also handles chemical secretion which by itself maybe thought of as being part of general diffusion equation.

$$\frac{\partial c}{\partial t} = D\nabla^2 c + kc + secretion$$

where *k* is a decay constant of concentration *c* and D is the diffusion constant. The term called *secretion* has the meaning as described below.
The principles of operations are analogous as in the case of AdvectionDiffusionSolver so most of has been said there applies to FlexibleDiffusionSolve. Also the syntax is very similar. Let's see an example

```
 <Steppable Type="FlexibleDiffusionSolverFE">
     <DiffusionField>
        <DiffusionData>
            <FieldName>FGF8</FieldName>
            <DiffusionConstant>0.1</DiffusionConstant>
            <DecayConstant>0.002</DecayConstant>
            <DeltaT>0.1</DeltaT>
            <DeltaX>1.0</DeltaX>
            <DoNotDiffuseTo>Bacteria</DoNotDiffuseTo>
        </DiffusionData>
        <SecretionData>
            <Secretion Type="Amoeba">0.1</Secretion>
        </SecretionData>
    </DiffusionField>

    <DiffusionField>
        <DiffusionData>
            <FieldName>FGF</FieldName>
            <DiffusionConstant>0.02</DiffusionConstant>
            <DecayConstant>0.001</DecayConstant>
            <DeltaT>0.01</DeltaT>
            <DeltaX>0.1</DeltaX>
        <DoNotDiffuseTo>Bacteria</DoNotDiffuseTo>
        </DiffusionData>
        <SecretionData>
            <SecretionOnContact Type="Medium"
             SecreteOnContactWith="Amoeba">0.1</SecretionOnContact>
            <Secretion Type="Amoeba">0.1</Secretion>
        </SecretionData>
    </DiffusionField>
 </Steppable>
```

We can see new xml tags `<DeltaT>` and `<DeltaX>`. Their values determine the correspondence between MCS and actual time and between lattice spacing and actual spacing size. In this example for the first diffusion field one MCS corresponds to 0.1 units of actual time and lattice spacing is equal 1 unit of actual length. What is happening here is that the diffusion constant gets multiplied by `DeltaT/ (DeltaX* DeltaX)`provided the decay constant is set to 0. If the decay constant is not zero `DeltaT` appears additionally in the term (in the explicit numerical approximation of the diffusion equation solution) containing decay constant so in this case it is more than simple diffusion constant rescaling.

SecretionData sections are analogous to those defined in AdvectionDiffusionSolver. here however, the secretion is done done on per-pixel basis (as opposed to per cell basis for AdvectionDiffusionSolver). For example when we use the following xml statement
`<Secretion Type="Amoeba">0.1</Secretion>`
this means that every pixel that belongs to cells of type Amoebae will get boost in concentration by 0.1. That is the secretion proceeds uniformly in the whole body of a cell.

Alternative secretion mode would be the one described by the following line:
`<SecretionOnContact Type="Medium" SecreteOnContactWith="Amoeba">0.1</SecretionOnContact>`
Here the secretion will take place in medium and only in those pixels belonging to Medium that touch directly Amoeba.
More secretion schemes will be added in the future.

The FlexibleDiffusionSolver is also capable of solving simple coupled diffusion type PDE of the form:

$$\frac{\partial c}{\partial t} = D\nabla^2 c + kc + secretion + m_d cd + m_f cf$$
$$\frac{\partial d}{\partial t} = D\nabla^2 d + kd + secretion + n_c dc + n_f df$$
$$\frac{\partial f}{\partial t} = D\nabla^2 f + kf + secretion + p_c fc + p_d fd$$

where m_c , m_g , n_c , n_f , p_c , p_d are coupling coefficients. To code the above equations in xml CompuCell3D syntax you need to use the following syntax:

```
<Steppable Type="FlexibleDiffusionSolverFE">
     <DiffusionField>
        <DiffusionData>
            <FieldName>c</FieldName>
            <DiffusionConstant>0.1</DiffusionConstant>
            <DecayConstant>0.002</DecayConstant>
             <CouplingTerm InteractingFieldName="d" CouplingCoefficent="0.1"/>
             <CouplingTerm InteractingFieldName="f" CouplingCoefficent="0.2"/>
            <DeltaT>0.1</DeltaT>
            <DeltaX>1.0</DeltaX>
            <DoNotDiffuseTo>Bacteria</DoNotDiffuseTo>
        </DiffusionData>
        <SecretionData>
            <Secretion Type="Amoeba">0.1</Secretion>
        </SecretionData>
    </DiffusionField>

    <DiffusionField>
        <DiffusionData>
            <FieldName>d</FieldName>
            <DiffusionConstant>0.02</DiffusionConstant>
            <DecayConstant>0.001</DecayConstant>
             <CouplingTerm InteractingFieldName="c" CouplingCoefficent="-0.1"/>
             <CouplingTerm InteractingFieldName="f" CouplingCoefficent="-0.2"/>
            <DeltaT>0.01</DeltaT>
            <DeltaX>0.1</DeltaX>
        <DoNotDiffuseTo>Bacteria</DoNotDiffuseTo>
        </DiffusionData>
        <SecretionData>
            <Secretion Type="Amoeba">0.1</Secretion>
        </SecretionData>
    </DiffusionField>
    <DiffusionField>
        <DiffusionData>
            <FieldName>f</FieldName>
            <DiffusionConstant>0.02</DiffusionConstant>
            <DecayConstant>0.001</DecayConstant>
             <CouplingTerm InteractingFieldName="c" CouplingCoefficent="-0.2"/>
             <CouplingTerm InteractingFieldName="d" CouplingCoefficent="0.2"/>
            <DeltaT>0.01</DeltaT>
            <DeltaX>0.1</DeltaX>
        <DoNotDiffuseTo>Bacteria</DoNotDiffuseTo>
        </DiffusionData>
        <SecretionData>
            <Secretion Type="Amoeba">0.1</Secretion>
        </SecretionData>
    </DiffusionField>
 </Steppable>
```

As one can see the only addition that is required to couple diffusion equations has simple syntax:

```
             <CouplingTerm InteractingFieldName="c" CouplingCoefficent="-0.1"/>
             <CouplingTerm InteractingFieldName="f" CouplingCoefficent="-0.2"/>
```

IX.3.7. FastDiffusionSolver2D

FastDiffusionSolver2DFE steppable is a simplified version of the FlexibleDiffusionSolverFE steppable. It runs several times faster that flexible solver but lacks some of its features. Typical syntax is shown below:

```
<Steppable Type="FastDiffusionSolver2DFE">
      <DiffusionField>
        <DiffusionData>
          <UseBoxWatcher/>
            <FieldName>FGF</FieldName>
            <DiffusionConstant>0.010</DiffusionConstant>
            <DecayConstant>0.003</DecayConstant>
            <DoNotDecayIn>Wall</DoNotDecay>

<ConcentrationFileName>Demos/diffusion/diffusion_2D_fast_box.pulse.txt<
/ConcentrationFileName>
        </DiffusionData>

    </DiffusionField>
 </Steppable>
```

In particular for fast solver you cannot specify cells into which diffusion is prohibited. However, you may specify cell types where diffusant decay is prohibited

IX.3.8. KernelDiffusionSolver

This diffusion solver has the advantage over previous solvers that it can handle large diffusion constants. It is also stable. However, it does not accept options like <DoNotDiffuseTo> or <DoNotDecayIn>. It also requires periodic boundary conditions. Simply put KernelDiffusionSolver solves diffusion equation

$$\frac{\partial c}{\partial t} = D\nabla^2 c + kc + secretion$$

With fixed, periodic boundary conditions on the edges of the lattice. This is different from FlexibleDiffusionSolver where the boundary conditions evolve. You also need to choose a proper Kernel range (K) according to the value of diffusion constant. Usually when $K^2\, e^{-(K\hat{}2\,/\,(4D))}$ is small (this is the main part of the integrand), the approximation convergers to the exact value.

The syntax for this solver is as follows:

```
<Steppable Type="KernelDiffusionSolver">
     <DiffusionField>
        <Kernel>4</Kernel>
        <DiffusionData>
            <FieldName>FGF</FieldName>
            <DiffusionConstant>1.0</DiffusionConstant>
            <DecayConstant>0.000</DecayConstant>

<ConcentrationFileName>Demos/diffusion/diffusion_2D.pulse.txt</Concentr
ationFileName>
        </DiffusionData>
```

```
    </DiffusionField>
 </Steppable>
```

Inside <DiffusionField> tag one may also use option <CoarseGrainFactor> to
For example:

```
<Steppable Type="KernelDiffusionSolver">
     <DiffusionField>
        <Kernel>4</Kernel>
        <CoarseGrainFactor>2</CoarseGrainFactor>
        <DiffusionData>
            <FieldName>FGF</FieldName>
            <DiffusionConstant>1.0</DiffusionConstant>
            <DecayConstant>0.000</DecayConstant>

<ConcentrationFileName>Demos/diffusion/diffusion_2D.pulse.txt</Concentr
ationFileName>
        </DiffusionData>
    </DiffusionField>
 </Steppable>
```

IX.3.9. ReactionDiffusionSolver

The reaction diffusion solver solves the following system of N reaction diffusion equations:

$$\frac{\partial c_1}{\partial t} = D_1\nabla^2 c_1 + f_1\left(c_1, c_2, \dots, c_N\right)$$

$$\frac{\partial c_2}{\partial t} = D_2\nabla^2 c_2 + f_2\left(c_1, c_2, \dots, c_N\right)$$

$$\vdots$$

$$\frac{\partial c_N}{\partial t} = D_N\nabla^2 c_N + f_N\left(c_1, c_2, \dots, c_N\right)$$

Let's consider a simple example of such system:

$$\frac{\partial F}{\partial t} = 0.1\nabla^2 F + (-0.1H)$$

$$\frac{\partial H}{\partial t} = 0.0\nabla^2 H + 0.1F$$

It can be coded as follows:

```
<Steppable Type="ReactionDiffusionSolverFE">
     <DiffusionField>
        <DiffusionData>
            <FieldName>F</FieldName>
            <DiffusionConstant>0.010</DiffusionConstant>

<ConcentrationFileName>Demos/diffusion/diffusion_2D.pulse.txt</Concentr
ationFileName>
            <AdditionalTerm>-0.01*H</AdditionalTerm>
        </DiffusionData>
    </DiffusionField>
```

```
     <DiffusionField>
        <DiffusionData>
            <FieldName>H</FieldName>
            <DiffusionConstant>0.0</DiffusionConstant>
            <AdditionalTerm>0.01*F</AdditionalTerm>
        </DiffusionData>
    </DiffusionField>
 </Steppable>
```

Notice how we implement functions *f* from the general system of reaction diffusion equations. We simply use <AdditionalTerm> tag and there we type arithmetic expression involving field names (tags <FieldName>). In addition to this we may include in those expression word CellType. For example:

```
<AdditionalTerm>0.01*F*CellType</AdditionalTerm>
```

This means that function *f* will depend also on CellType . CellType hodls the value of the type of the cell at particular location x,y,z of the lattice. The inclusion of the cell type might be useful if you want to use additional terms which may change depending of the cell type. Then all you have to do is to either use if statements inside <AdditionalTerm> or form equivalent mathematical expression using functions allowed by muParser (http://muparser.sourceforge.net/mup_features.html#idDef2)

For example, let's assume that additional term for second equation is the following:

$$f_H = \begin{cases} 0.1*F & \text{if} \quad \text{CellType}=1 \\ 0.15*F & \text{otherwise} \end{cases}$$

In such a case additional term would be coded as follows:

```
<AdditionalTerm>if (CellType==1,0.01*F,0.15*F) </AdditionalTerm>
```

Notice that we have used here muParser function called if. The syntax of it is as follows:

```
if(condition,expression if condition true, expression if condition
false)
```

One thing to remember is that computing time of the additional term depends on the level of complexity of this term. Thus it is not the best idea to code very complex expressions using muParser.

IX.3.10. BoxWatcher Steppable

This steppable can potentially speed-up your simulation. Every MCS (or every Frequency MCS) it determines maximum and minimum coordinates of cells and then imposes slightly bigger box around cells and ensures that in the subsequent MCS spin flip attempts take place only inside this box containing cells (plus some amount of medium on the sides). Thus instead of sweeping entire lattice and attempting random spin flips CompuCell3D will only spend time trying flips inside the box. Depending on the simulation the performance gains are up to approx. 30%. The steppable will work best if you have simulation with cells localized in one region of the lattice with lots of empty space. The steppable will adjust box every MCS (or every Frequency MCS) according to evolving cellular pattern.

The syntax is as follows:

```
<Steppable Type="BoxWatcher">
  <XMargin>5</XMargin>
  <YMargin>5</YMargin>
  <ZMargin>5</ZMargin>
</Steppable>
```

All that is required is to specify amount of extra space (expressed in units of pixels) that needs to be added to the a tight box i.e. the box whose sides just touch most peripheral cells' pixels .

IX.4. Additional Plugins and Modules

Besides the modules that were introduced above CompuCell3D contains other modules which were developed to solve particular problem. For example module called DictyFieldInitializer is used to prepare initial cell configuration for the simulation of *Dictyostelium discoideum* morphogenesis based on the paper by P.Hogeweg and N.Savill **Modelling morphogenesis: from single cells to crawling slugs. J. theor. Biol. 184, 229-235.**
Such modules have limited area of applicability and are mostly used in a single simulation. For this reason we will not describe them in more detail here. Interested user may consult CompuCell3D manual 3.2.0 where all such modules were described. It is our goal however to eliminate a need to write customized modules as much as possible. For example DictyFieldInitializer can be easily replaced by using UniformInitializer and defining several regions there. Similarly Reaction diffusion solver for this simulation can be replaced by a general Reaction Diffusion solver described above.
While we might run into performance issues when using general as opposed to customized, the flexibility and portability associated with using general use modules are worth extra run time.

X. References

1. Bassingthwaighte, J. B. (2000) Strategies for the Physiome project. *Annals of Biomedical Engineering* **28**, 1043-1058.
2. Merks, R. M. H., Newman, S. A., and Glazier, J. A. (2004) Cell-oriented modeling of *in vitro* capillary development. *Lecture Notes in Computer Science* **3305**, 425-434.
3. Turing, A. M. (1953) The Chemical Basis of Morphogenesis. *Philosophical Transactions of the Royal Society B* **237**, 37-72.
4. Merks, R. M. H. and Glazier, J. A. (2005) A cell-centered approach to developmental biology. *Physica A* **352**, 113-130.
5. Dormann, S. and Deutsch, A. (2002) Modeling of self-organized avascular tumor growth with a hybrid cellular automaton. *In Silico Biology* **2**, 1-14.
6. dos Reis, A. N., Mombach, J. C. M., Walter, M., and de Avila, L. F. (2003) The interplay between cell adhesion and environment rigidity in the morphology of tumors. *Physica A* **322**, 546-554.
7. Drasdo, D. and Hohme, S. (2003) Individual-based approaches to birth and death in avascular tumors. *Mathematical and Computer Modelling* **37**, 1163-1175.

8. Holm, E. A., Glazier, J. A., Srolovitz, D. J., and Grest, G. S. (1991) Effects of Lattice Anisotropy and Temperature on Domain Growth in the Two-Dimensional Potts Model. *Physical Review A* **43**, 2662-2669.
9. Turner, S. and Sherratt, J. A. (2002) Intercellular adhesion and cancer invasion: A discrete simulation using the extended Potts model. *Journal of Theoretical Biology* **216**, 85-100.
10. Drasdo, D. and Forgacs, G. (2000) Modeling the interplay of generic and genetic mechanisms in cleavage, blastulation, and gastrulation. *Developmental Dynamics* **219**, 182-191.
11. Drasdo, D., Kree, R., and McCaskill, J. S. (1995) Monte-Carlo approach to tissue-cell populations. *Physical Review E* **52**, 6635-6657.
12. Longo, D., Peirce, S. M., Skalak, T. C., Davidson, L., Marsden, M., and Dzamba, B. (2004) Multicellular computer simulation of morphogenesis: blastocoel roof thinning and matrix assembly in *Xenopus laevis*. *Developmental Biology* **271**, 210-222.
13. Collier, J. R., Monk, N. A. M., Maini, P. K., and Lewis, J. H. (1996) Pattern formation by lateral inhibition with feedback: A mathematical model of Delta-Notch intercellular signaling. *Journal of Theoretical Biology* **183**, 429-446.
14. Honda, H. and Mochizuki, A. (2002) Formation and maintenance of distinctive cell patterns by coexpression of membrane-bound ligands and their receptors. *Developmental Dynamics* **223**, 180-192.
15. Moreira, J. and Deutsch, A. (2005) Pigment pattern formation in zebrafish during late larval stages: A model based on local interactions. *Developmental Dynamics* **232**, 33-42.
16. Wearing, H. J., Owen, M. R., and Sherratt, J. A. (2000) Mathematical modelling of juxtacrine patterning. *Bulletin of Mathematical Biology* **62**, 293-320.
17. Zhdanov, V. P. and Kasemo, B. (2004) Simulation of the growth of neurospheres. *Europhysics Letters* **68**, 134-140.
18. Ambrosi, D., Gamba, A., and Serini, G. (2005) Cell directional persistence and chemotaxis in vascular morphogenesis. *Bulletin of Mathematical Biology* **67**, 195-195.
19. Gamba, A., Ambrosi, D., Coniglio, A., de Candia, A., di Talia, S., Giraudo, E., Serini, G., Preziosi, L., and Bussolino, F. (2003) Percolation, morphogenesis, and Burgers dynamics in blood vessels formation. *Physical Review Letters* **90**, 118101.
20. Novak, B., Toth, A., Csikasz-Nagy, A., Gyorffy, B., Tyson, J. A., and Nasmyth, K. (1999) Finishing the cell cycle. *Journal of Theoretical Biology* **199**, 223-233.
21. Peirce, S. M., van Gieson, E. J., and Skalak, T. C. (2004) Multicellular simulation predicts microvascular patterning and *in silico* tissue assembly. *FASEB Journal* **18**, 731-733.
22. Merks, R. M. H., Brodsky, S. V., Goligorksy, M. S., Newman, S. A., and Glazier, J. A. (2006) Cell elongation is key to *in silico* replication of *in vitro* vasculogenesis and subsequent remodeling. *Developmental Biology* **289**, 44-54.
23. Merks, R. M. H. and Glazier, J. A. (2005) Contact-inhibited chemotactic motility can drive both vasculogenesis and sprouting angiogenesis. *q-bio/0505033.*
24. Kesmir, C. and de Boer., R. J. (2003) A spatial model of germinal center

reactions: cellular adhesion based sorting of B cells results in efficient affinity maturation. *Journal of Theoretical Biology* **222**, 9-22.

25. Meyer-Hermann, M., Deutsch, A., and Or-Guil, M. (2001) Recycling probability and dynamical properties of germinal center reactions. *Journal of Theoretical Biology* **210**, 265-285.
26. Nguyen, B., Upadhyaya, A. van Oudenaarden, A., and Brenner, M. P. (2004) Elastic instability in growing yeast colonies. *Biophysical Journal* **86**, 2740-2747.
27. Walther, T., Reinsch, H., Grosse, A., Ostermann, K., Deutsch, A., and Bley, T. (2004) Mathematical modeling of regulatory mechanisms in yeast colony development. *Journal of Theoretical Biology* **229**, 327-338.
28. Borner, U., Deutsch, A., Reichenbach, H., and Bar, M. (2002) Rippling patterns in aggregates of *myxobacteria* arise from cell-cell collisions. *Physical Review Letters* **89**, 078101.
29. Bussemaker, H. J., Deutsch, A., and Geigant, E. (1997) Mean-field analysis of a dynamical phase transition in a cellular automaton model for collective motion. *Physical Review Letters* **78**, 5018-5021.
30. Dormann, S., Deutsch, A., and Lawniczak, A. T. (2001) Fourier analysis of Turing-like pattern formation in cellular automaton models. *Future Generation Computer Systems* **17**, 901-909.
31. Börner, U., Deutsch, A., Reichenbach, H., and Bär, M. (2002) Rippling patterns in aggregates of myxobacteria arise from cell-cell collisions. *Physical Review Letters* **89,** 078101.
32. Zhdanov, V. P. and Kasemo, B. (2004) Simulation of the growth and differentiation of stem cells on a heterogeneous scaffold. *Physical Chemistry Chemical Physics* **6**, 4347-4350.
33. Knewitz, M. A. and Mombach, J. C. (2006) Computer simulation of the influence of cellular adhesion on the morphology of the interface between tissues of proliferating and quiescent cells. *Computers in Biology and Medicine* **36**, 59-69.
34. Marée, A. F. M. and Hogeweg, P. (2001) How amoeboids self-organize into a fruiting body: Multicellular coordination in Dictyostelium discoideum. *Proceedings of the National Academy of Sciences of the USA* **98**, 3879-3883.
35. Marée, A. F. M. and Hogeweg, P. (2002) Modelling Dictyostelium discoideum morphogenesis: the culmination. *Bulletin of Mathematical Biology* **64**, 327-353.
36. Marée, A. F. M., Panfilov, A. V., and Hogeweg, P. (1999) Migration and thermotaxis of *Dictyostelium discoideum* slugs, a model study. *Journal of Theoretical Biology* **199**, 297-309.
37. Savill, N. J. and Hogeweg, P. (1997) Modelling morphogenesis: From single cells to crawling slugs. *Journal of Theoretical Biology* **184**, 229-235.
38. Hogeweg, P. (2000) Evolving mechanisms of morphogenesis: on the interplay between differential adhesion and cell differentiation. *Journal of Theoretical Biology* **203**, 317-333.
39. Johnston, D. A. (1998) Thin animals. *Journal of Physics A* **31**, 9405-9417.
40. Groenenboom, M. A. and Hogeweg, P. (2002) Space and the persistence of male-killing endosymbionts in insect populations. *Proceedings in Biological Sciences* **269**, 2509-2518.
41. Groenenboom, M. A., Maree, A. F., and Hogeweg, P. (2005) The RNA silencing

pathway: the bits and pieces that matter. *PLoS Computational Biology* **1**, 155-165.
42. Kesmir, C., van Noort, V., de Boer, R. J., and Hogeweg, P. (2003) Bioinformatic analysis of functional differences between the immunoproteasome and the constitutive proteasome. *Immunogenetics* **55**, 437-449.
43. Pagie, L. and Hogeweg, P. (2000) Individual- and population-based diversity in restriction-modification systems. *Bulletin of Mathematical Biology* **62**, 759-774.
44. Silva, H. S. and Martins, M. L. (2003) A cellular automata model for cell differentiation. *Physica A* **322**, 555-566.
45. Zajac, M., Jones, G. L., and Glazier, J. A. (2000) Model of convergent extension in animal morphogenesis. *Physical Review Letters* **85**, 2022-2025.
46. Zajac, M., Jones, G. L., and Glazier, J. A. (2003) Simulating convergent extension by way of anisotropic differential adhesion. *Journal of Theoretical Biology* **222**, 247-259.
47. Savill, N. J. and Sherratt, J. A. (2003) Control of epidermal stem cell clusters by Notch-mediated lateral induction. *Developmental Biology* **258**, 141-153.
48. Mombach, J. C. M, de Almeida, R. M. C., Thomas, G. L., Upadhyaya, A., and Glazier, J. A. (2001) Bursts and cavity formation in Hydra cells aggregates: experiments and simulations. *Physica A* **297**, 495-508.
49. Rieu, J. P., Upadhyaya, A., Glazier, J. A., Ouchi, N. B. and Sawada, Y. (2000) Diffusion and deformations of single hydra cells in cellular aggregates. *Biophysical Journal* **79**, 1903-1914.
50. Mochizuki, A. (2002) Pattern formation of the cone mosaic in the zebrafish retina: A cell rearrangement model. *Journal of Theoretical Biology* **215**, 345-361.
51. Takesue, A., Mochizuki, A., and Iwasa, Y. (1998) Cell-differentiation rules that generate regular mosaic patterns: Modelling motivated by cone mosaic formation in fish retina. *Journal of Theoretical Biology* **194**, 575-586.
52. Dallon, J., Sherratt, J., Maini, P. K., and Ferguson, M. (2000) Biological implications of a discrete mathematical model for collagen deposition and alignment in dermal wound repair. *IMA Journal of Mathematics Applied in Medicine and Biology* **17**, 379-393.
53. Maini, P. K., Olsen, L., and Sherratt, J. A. (2002) Mathematical models for cell-matrix interactions during dermal wound healing. *International Journal of Bifurcations and Chaos* **12**, 2021-2029.
54. Kreft, J. U., Picioreanu, C., Wimpenny, J. W. T., and van Loosdrecht, M. C. M. (2001) Individual-based modelling of biofilms. *Microbiology* **147**, 2897-2912.
55. Picioreanu, C., van Loosdrecht, M. C. M., and Heijnen, J. J. (2001) Two-dimensional model of biofilm detachment caused by internal stress from liquid flow. *Biotechnology and Bioengineering* **72**, 205-218.
56. van Loosdrecht, M. C. M., Heijnen, J. J., Eberl, H., Kreft, J., and Picioreanu, C. (2002) Mathematical modelling of biofilm structures. *Antonie Van Leeuwenhoek International Journal of General and Molecular Microbiology* **81**, 245-256.
57. Popławski, N. J., Shirinifard, A., Swat, M., and Glazier, J. A. (2008) Simulations of single-species bacterial-biofilm growth using the Glazier-Graner-Hogeweg model and the CompuCell3D modeling environment. *Mathematical Biosciences and Engineering* **5**, 355-388.
58. Chaturvedi, R., Huang, C., Izaguirre, J. A., Newman, S. A., Glazier, J. A., Alber,

M. S. (2004) A hybrid discrete-continuum model for 3-D skeletogenesis of the vertebrate limb. *Lecture Notes in Computer Science* **3305**, 543-552.

59. Popławski, N. J., Swat, M., Gens, J. S., and Glazier, J. A. (2007) Adhesion between cells, diffusion of growth factors, and elasticity of the AER produce the paddle shape of the chick limb. *Physica A* **373**, 521-532.

60. Glazier, J. A. and Weaire, D. (1992) The Kinetics of Cellular Patterns. *Journal of Physics: Condensed Matter* **4**, 1867-1896.

61. Glazier, J. A. (1993) Grain Growth in Three Dimensions Depends on Grain Topology. *Physical Review Letters* **70**, 2170-2173.

62. Glazier, J. A., Grest, G. S., and Anderson, M. P. (1990) Ideal Two-Dimensional Grain Growth. In *Simulation and Theory of Evolving Microstructures*, M. P. Anderson and A. D. Rollett, editors. The Minerals, Metals and Materials Society, Warrendale, PA, pp. 41-54.

63. Glazier, J. A., Anderson, M. P., and Grest, G. S. (1990) Coarsening in the Two-Dimensional Soap Froth and the Large-Q Potts Model: A Detailed Comparison. *Philosophical Magazine B* **62**, 615-637.

64. Grest, G. S., Glazier, J. A., Anderson, M. P., Holm, E. A., and Srolovitz, D. J. (1992) Coarsening in Two-Dimensional Soap Froths and the Large-Q Potts Model. *Materials Research Society Symposium* **237**, 101-112.

65. Jiang, Y. and Glazier, J. A. (1996) Extended Large-Q Potts Model Simulation of Foam Drainage. *Philosophical Magazine Letters* **74**, 119-128.

66. Jiang, Y., Levine, H., and Glazier, J. A. (1998) Possible Cooperation of Differential Adhesion and Chemotaxis in Mound Formation of *Dictyostelium*. *Biophysical Journal* **75**, 2615-2625.

67. Jiang, Y., Mombach, J. C. M., and Glazier, J. A. (1995) Grain Growth from Homogeneous Initial Conditions: Anomalous Grain Growth and Special Scaling States. *Physical Review E* **52**, 3333-3336.

68. Jiang, Y., Swart, P. J., Saxena, A., Asipauskas, M., and Glazier, J. A. (1999) Hysteresis and Avalanches in Two-Dimensional Foam Rheology Simulations. *Physical Review E* **59**, 5819-5832.

69. Ling, S., Anderson, M. P., Grest, G. S., and Glazier, J. A. (1992) Comparison of Soap Froth and Simulation of Large-Q Potts Model. *Materials Science Forum* **94-96**, 39-47.

70. Mombach, J. C. M. (2000) Universality of the threshold in the dynamics of biological cell sorting. *Physica A* **276**, 391-400.

71. Weaire, D. and Glazier, J. A. (1992) Modelling Grain Growth and Soap Froth Coarsening: Past, Present and Future. *Materials Science Forum* **94-96**, 27-39.

72. Weaire, D., Bolton, F., Molho, P., and Glazier, J. A. (1991) Investigation of an Elementary Model for Magnetic Froth. *Journal of Physics: Condensed Matter* **3**, 2101-2113.

73. Glazer, J. A., Balter, A., Popławski, N. (2007) Magnetization to Morphogenesis: A Brief History of the Glazier-Graner-Hogeweg Model. In *Single-Cell-Based Models in Biology and Medicine*. Anderson, A. R. A., Chaplain, M. A. J., and Rejniak, K. A., editors. Birkhauser Verlag Basel, Switzerland. pp. 79-106.

74. Walther, T., Reinsch, H., Ostermann, K., Deutsch, A. and Bley, T. (2005) Coordinated growth of yeast colonies: experimental and mathematical analysis of

possible regulatory mechanisms. *Engineering Life Sciences* **5**, 115-133.

75. Keller, E. F. and Segel., L. A. (1971) Model for chemotaxis. *Journal of Theoretical Biology* **30**, 225-234.
76. Glazier, J. A. and Upadhyaya, A. (1998) First Steps Towards a Comprehensive Model of Tissues, or: A Physicist Looks at Development. In *Dynamical Networks in Physics and Biology: At the Frontier of Physics and Biology*, D. Beysens and G. Forgacs editors. EDP Sciences/Springer Verlag, Berlin, pp. 149-160.
77. Glazier, J. A. and Graner, F. (1993) Simulation of the differential adhesion driven rearrangement of biological cells. *Physical Review E* **47**, 2128-2154.
78. Glazier, J. A. (1993) Cellular Patterns. *Bussei Kenkyu* **58**, 608-612.
79. Glazier, J. A. (1996) Thermodynamics of Cell Sorting. *Bussei Kenkyu* **65**, 691-700.
80. Glazier, J. A., Raphael, R. C., Graner, F., and Sawada, Y. (1995) The Energetics of Cell Sorting in Three Dimensions. In *Interplay of Genetic and Physical Processes in the Development of Biological Form*, D. Beysens, G. Forgacs, F. Gaill, editors. World Scientific Publishing Company, Singapore, pp. 54-66.
81. Graner, F. and Glazier, J. A. (1992) Simulation of biological cell sorting using a 2-dimensional extended Potts model. *Physical Review Letters* **69**, 2013-2016.
82. Mombach, J. C. M and Glazier, J. A. (1996) Single Cell Motion in Aggregates of Embryonic Cells. *Physical Review Letters* **76**, 3032-3035.
83. Mombach, J. C. M., Glazier, J. A., Raphael, R. C., and Zajac, M. (1995) Quantitative comparison between differential adhesion models and cell sorting in the presence and absence of fluctuations. *Physical Review Letters* **75**, 2244-2247.
84. Cipra, B. A. (1987) An Introduction to the Ising-Model. *American Mathematical Monthly* **94**, 937-959.
85. Metropolis, N., Rosenbluth, A., Rosenbluth, M. N., Teller, A. H., and Teller, E. (1953) Equation of state calculations by fast computing machines. *Journal of Chemical Physics* **21**, 1087-1092.
86. Forgacs, G. and Newman, S. A. (2005). *Biological Physics of the Developing Embryo*. Cambridge Univ. Press, Cambridge.
87. Alber, M. S., Kiskowski, M. A., Glazier, J. A., and Jiang, Y. On cellular automation approaches to modeling biological cells. In *Mathematical Systems Theory in Biology, Communication and Finance*. J. Rosenthal, and D. S. Gilliam, editors. Springer-Verlag, New York, pp. 1-40.
88. Alber, M. S., Jiang, Y., and Kiskowski, M. A. (2004) Lattice gas cellular automation model for rippling and aggregation in *myxobacteria*. *Physica D* **191**, 343-358.
89. Novak, B., Toth, A., Csikasz-Nagy, A., Gyorffy, B., Tyson, J. A., and Nasmyth, K. (1999) Finishing the cell cycle. *Journal of Theoretical Biology* **199**, 223-233.
90. Upadhyaya, A., Rieu, J. P., Glazier, J. A., and Sawada, Y. (2001) Anomalous Diffusion in Two-Dimensional Hydra Cell Aggregates. *Physica A* **293**, 549-558.
91. Cickovski, T., Aras, K., Alber, M. S., Izaguirre, J. A., Swat, M., Glazier, J. A., Merks, R. M. H., Glimm, T., Hentschel, H. G. E., Newman, S. A. (2007) From genes to organisms via the cell: a problem-solving environment for multicellular development. *Computers in Science and Engineering* **9**, 50-60.
92. Izaguirre, J.A., Chaturvedi, R., Huang, C., Cickovski, T., Coffland, J., Thomas,

G., Forgacs, G., Alber, M., Hentschel, G., Newman, S. A., and Glazier, J. A. (2004) CompuCell, a multi-model framework for simulation of morphogenesis. *Bioinformatics* **20**, 1129-1137.

93. Armstrong, P. B. and Armstrong, M. T. (1984) A role for fibronectin in cell sorting out. *Journal of Cell Science* **69**, 179-197.

94. Armstrong, P. B. and Parenti, D. (1972) Cell sorting in the presence of cytochalasin B. *Journal of Cell Science* **55**, 542-553.

95. Glazier, J. A. and Graner, F. (1993) Simulation of the differential adhesion driven rearrangement of biological cells. *Physical Review E* **47**, 2128-2154.

96. Glazier, J. A. and Graner, F. (1992) Simulation of biological cell sorting using a two-dimensional extended Potts model. *Physical Review Letters* **69**, 2013-2016.

97. Ward, P. A., Lepow, I. H., and Newman, L. J. (1968) Bacterial factors chemotactic for polymorphonuclear leukocytes. *American Journal of Pathology* **52**, 725-736.

98. Lutz, M. (1999) *Learning Python*. Sebastopol, CA: O'Reilly & Associates, Inc.

99. Balter, A. I., Glazier, J. A., and Perry, R. (2008) Probing soap-film friction with two-phase foam flow. *Philosophical Magazine, submitted.*

100. Dvorak, P., Dvorakova, D., and Hampl, A. (2006) Fibroblast growth factor signaling in embryonic and cancer stem cells. *FEBS Letters* **580**, 2869-2287.

Appendix

1Calculating Inertia Tensor in CompuCell3D.

For each cell the inertia tensor is defined as follows:

$$I = \begin{bmatrix} \sum_i y_i^2 + z_i^2 & -\sum_i x_i y_i & -\sum_i x_i z_i \\ -\sum_i x_i y_i & \sum_i x_i^2 + z_i^2 & -\sum_i y_i z_i \\ -\sum_i x_i z_i & -\sum_i y_i z_i & \sum_i x_i^2 + y_i^2 \end{bmatrix}$$

where index '*i*' denotes *i*-th pixel of a given cell and x_i, y_i,z_i are coordinates of that pixel in a given coordinate frame.

where index '*i*' denotes *i*-th pixel of a given cell and x_i, y_i,z_i are coordinates of that pixel in a given coordinate frame.

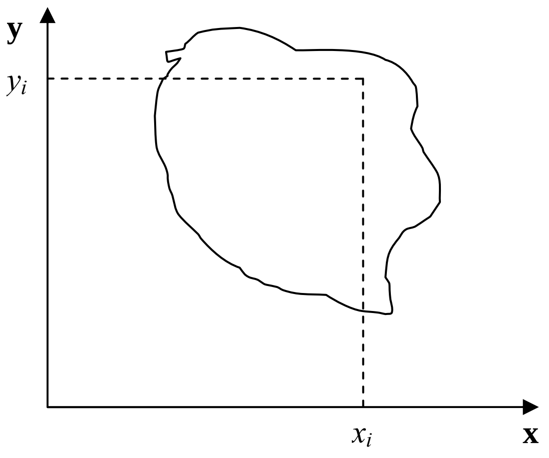


Figure 20. Cell and its coordinate frame in which we calculate inertia tensor

In Figure 20 we show one possible coordinate frame in which one can calculate inertia tensor. If the coordinate frame is fixed calculating components of inertia tensor for cell gaining or losing one pixel is quite easy. We will be adding and subtracting terms like $y_i^2 + z_i^2$ or $x_i y_i$. However, in CompuCell3D we are mostly interested in knowing tensor of inertia of a cell with respect to *xyz* coordinate frame with origin at the center of mass (*COM*) of a given cell as shown in Fig 21. Now, to calculate such tensor we cannot simply add or subtract terms like $y_i^2 + z_i^2$ or $x_i y_i$ to account for lost or gained pixel. If a cell gains or loses a pixel its COM coordinates change. If so then all the x_i, y_i,z_i coordinates that appear in the inertia tensor expression will have different value. Thus for each change in cell shape (gain or loss of pixel) we would have to recalculate inertia tensor from scratch. This would be quite time consuming and would require us to keep track of all the pixels belonging to a given cell. It turns out however that there is a better way of keeping track of inertia tensor for cells. We will be using parallel axis theorem to do the calculations. Paralel axis theorem states that if I_{COM} is a

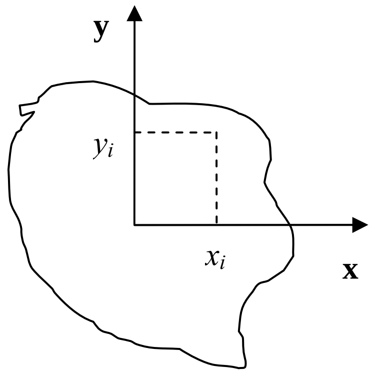


Figure 21. Cell and coordinate system passing through center of mass of a cell. Notice that as cell changes shape the position of center of mass moves.

moment of inertia with respect to axis passing through center of mass then we can calculate moment of inertia with respect to any parallel axis to the one passin through the COM by using the following formula:

$I_{x'x'} = I_{xx} + Md^2$

where I_{xx} denotes moment of inertia with respect to *x* axis passing through center of mass, $I_{x'x'}$ is a moment of inertia with respect to axis parallel to the *x* axis passing through center of mass, *d* is the distance between the axes and *M* is mass of the cell.

Let us now draw a picture of a cell gaining one pixel:

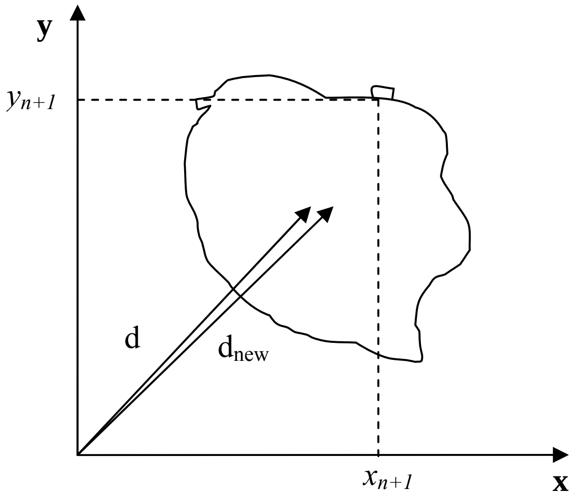


Figure 22. Cell gaining one pixel.d denotes a distance from origin of a fixed fram of reference to a center of mass of a cell before cell gains new pixel. d_{new} denotes same distance but after cell gains new pixel

Now using parallel axis theorem we can write expression for the moment of inertia after cell gains one pixel the following that:

$I^{new}{}_{xx} = I^{new}{}_{x'x'} - (V+1)d_{new}{}^2$

where as before $I^{new}{}_{xx}$ denotes moment of inertia of a cell with new pixel with respect to *x* axis passing through center of mass, $I^{new}{}_{x'x'}$ is a moment of inertia with respect to axis parallel to the *x* axis passing through center of mass, d_{new} is the distance between the axes and *V+1* is volume of the cell after it gained one pixel. Now let us rewrite above equation by adding ad subtracting Vd^2 term:

$$\begin{aligned} I^{new}{}_{xx} &= I^{old}{}_{x'x'} + y_{n+1}{}^2 + z_{n+1}{}^2 - Vd^2 + Vd^2 - (V+1)d_{new}{}^2 \\ &= I^{old}{}_{x'x'} - Vd^2 + Vd^2 - (V+1)d_{new}{}^2 + y_{n+1}{}^2 + z_{n+1}{}^2 \\ &= I^{old}{}_{xx} + Vd^2 - (V+1)d_{new}{}^2 + y_{n+1}{}^2 + z_{n+1}{}^2 \end{aligned}$$

Therefore we have found an expression for moment of inertia passing through the center of mass of the cell with additional pixel. Note that this expression involves moment of inertia but for the old cell (*i.e.* the original cell, not the one with extra pixel). When we add new pixel we know its coordinates and we can also easily calculate d_{new} .Thus when we need to calculate the moment of intertia for new cell instead of performing summation as given in the definition of the inertia tensor we can use much simpler expression.
This was diagonal term of the inertia tensor. What about off-diagonal terms? Let us write explicitely expression for I_{xy} :

$$I_{xy} = -\sum_i^N (x_i - x_{COM})(y_i - y_{COM}) = -\sum_i^N x_i y_i + x_{COM}\sum_i^N y_i + y_{COM}\sum_i^N x_i - x_{COM}y_{COM}\sum_i^N 1$$

$$= -\sum_i^N x_i y_i + x_{COM}Vy_{COM} + y_{COM}Vx_{COM} - x_{COM}y_{COM}V$$

$$= -\sum_i^N x_i y_i + Vx_{COM}y_{COM}$$

where x_{COM} denotes *x* COM position of the cell, similarly y_{COM} denotes *y* COM position of cell and *V* denotes cell volume. In the above formula we have used the fact that

$x_{COM} = \frac{\sum_i x_i}{V} \Rightarrow \sum_i x_i = x_{COM}V$ and similarly for the *y* coordinate.

Now, for the new cell with additional pixel we have the following relation:

$$I^{new}{}_{xy} = -\sum_i^{N+1} x_i y_i + (V+1)x^{new}{}_{COM}y^{new}{}_{COM}$$

$$= -\sum_i^N x_i y_i + Vx_{COM}y_{COM} - x_{COM}Vy_{COM} + (V+1)x^{new}{}_{COM}y^{new}{}_{COM} - x_{N+1}y_{N+1}$$

$$= I^{old}{}_{xy} - Vx_{COM}y_{COM} + (V+1)x^{new}{}_{COM}y^{new}{}_{COM} - x_{N+1}y_{N+1}$$

where we have added and subtracted $Vx_{COM}y_{COM}$ to be able to form

$I^{old}{}_{xy} - \sum_i^N x_i y_i + Vx_{COM}y_{COM}$ on the right hand side of the expression for $I^{new}{}_{xy}$. As it was the case for diagonal element, calculating off-diagonal of the inertia tensor involves $I^{old}{}_{xy}$ and positions of center of mass of the cell before and after gaining new pixel. All those quantities are either known a priori ($I^{old}{}_{xy}$) or can be easily calculated (center of mass position after gaining one pixel).
Therefore we have shown how we can calculate tensor of inertia for a given cell with respect to a coordinate frame with origin at cell's center of mass, without evaluating full sums. Such "local" calculations greatly speed up simulations

2.Calculating shape constraint of a cell – elongation term

The shape of single cell immersed in medium and not subject to too drastic surface or surface constraints will be spherical (circular in 2D). However in certain situation we may want to use cells which are elongated along one of their body axes. To facilitate this we can place constraint on principal lengths of cell. In 2D it is sufficient to constrain one of the principal lenghths of cell how ever in 3D we need to constrain 2 out of 3 principal lengths. Our first task is to diagonalize inertia tensor (i.e. find a coordinate frame transformation which brings inertia tensor to a giagonal form)

2.1. Diagonalizing inertia tensor

We will consider here more difficult 3D case. The 2D case is described in detail in M.Zajac, G.L.jones, J,A,Glazier "*Simulating convergent extension by way of anisotropic differential adhesion*" Journal of Theoretical Biology 222 (2003) 247–259.

In order to diagonalize inertia tensor we need to solve eigenvalue equation:
$\det(I-\lambda)=0$ or in full form

$$\begin{vmatrix} \sum_i y_i^2+z_i^2-\lambda & -\sum_i x_i y_i & -\sum_i x_i z_i \\ -\sum_i x_i y_i & \sum_i x_i^2+z_i^2-\lambda & -\sum_i y_i z_i \\ -\sum_i x_i z_i & -\sum_i y_i z_i & \sum_i x_i^2+y_i^2-\lambda \end{vmatrix} = \begin{vmatrix} I_{xx}-\lambda & I_{xy} & I_{xz} \\ I_{xy} & I_{yy}-\lambda & I_{yz} \\ I_{xz} & I_{yz} & I_{zz}-\lambda \end{vmatrix} = 0$$

The eigenvalue equation will be in the form of 3rd order polynomial. The roots of it are guaranteed to be real. The polynomial itself can be found either by explicit derivation, using symbolic calculation or simply in Wikipedia (http://en.wikipedia.org/wiki/Eigenvalue_algorithm)

$$\det\begin{bmatrix} a-\lambda & b & c \\ d & e-\lambda & f \\ g & h & i-\lambda \end{bmatrix} = -\lambda^3+\lambda^2(a+e+i)+\lambda(db+gc+fh-ae-ai-ei)+(aei-afh-dbi+dch+gbf-gce).$$

so in our case the eigenvalue equation takes the form:

$$-L^3+L^2(\mathrm{I}_{xx}+\mathrm{I}_{yy}+\mathrm{I}_{zz})+L(\mathrm{I}_{xy}{}^2+\mathrm{I}_{yz}{}^2+\mathrm{I}_{xz}{}^2-\mathrm{I}_{xx}\mathrm{I}_{yy}-\mathrm{I}_{yy}\mathrm{I}_{zz}-\mathrm{I}_{xx}\mathrm{I}_{zz})$$
$$+\mathrm{I}_{xx}\mathrm{I}_{yy}\mathrm{I}_{zz}-\mathrm{I}_{xx}\mathrm{I}_{yz}{}^2-\mathrm{I}_{yy}\mathrm{I}_{xz}{}^2-\mathrm{I}_{zz}\mathrm{I}_{xy}{}^2+2\mathrm{I}_{xy}\mathrm{I}_{yz}\mathrm{I}_{xz}=0$$

This equation can be solved analytically, again we may use Wikipedia (http://en.wikipedia.org/wiki/Cubic_function)
Now, the eigenvalues found that way are principal moments of inertia of a cell. That is they are components of inertia tensor in a coordinate frame rotated in such a way that off-diagonal elements of inertia tensor are 0:

$$I=\begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$$

In our cell shape constraint we will want to obtain ellipsoidal cells. Therefore the target tensor of inertia for the cell should be tensor if inertia for ellipsoid:

$$I = \begin{bmatrix} \frac{1}{5}\left(b^2+c^2\right) & 0 & 0 \\ 0 & \frac{1}{5}\left(a^2+c^2\right) & 0 \\ 0 & 0 & \frac{1}{5}\left(a^2+b^2\right) \end{bmatrix}$$

where *a,b,c* are parameters describing the surface of an ellipsoid:

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}+\frac{z^2}{c^2}=1$$

In other words *a,b,c* are half lengths of principal axes (they are analogues of circle's radius)
Now we can determine semi axes lengths in terms of principal moments of inertia by inverting the following set of equations:

$$I_{xx}=\frac{1}{5}\left(b^2+c^2\right)$$
$$I_{yy}=\frac{1}{5}\left(a^2+c^2\right)$$
$$I_{zz}=\frac{1}{5}\left(a^2+b^2\right)$$

Once we have calculated semiaxes lengths in terms of moments of inertia we can plug – in actual numbers for moment of inertia (the ones for actual cell) and obtain lengths of semiexes. Next we apply quadratic constraint on largest (semimajor) and smallest (seminimor axes). This is what elongation plugin does.

3 Forward Euler method for solving PDE's in CompuCell3D.

In CompuCell3D most of the solvers uses explicit schemes (Forward Euler method) to obtain PDE solutions. Thus for the diffusion equation we have:

$$\frac{\partial c}{\partial t}=\frac{\partial^2 c}{\partial x^2}+\frac{\partial^2 c}{\partial y^2}+\frac{\partial^2 c}{\partial z^2}$$

In a discretetized form we may write:

$$\frac{c(x,t+\Delta t)-c(x,t)}{\Delta t}=\frac{c(x+\Delta x,t)-2c(x,t)+c(x-\Delta x,t)}{\Delta x^2}+$$
$$\frac{c(y+\Delta y,t)-2c(x,t)+c(y-\Delta y,t)}{\Delta y^2}+$$
$$\frac{c(z+\Delta z,t)-2c(z,t)+c(z-\Delta z,t)}{\Delta z^2}$$

where to save space we used shorthand notation:

$$c(x+\Delta x,y,z,t)\equiv c(x+\Delta x,t)$$
$$c(x,y,z,t)\equiv c(x,t)$$

and similarly for other coordinates.
After rearranging terms we get the following expression:

$$c(x,t+\Delta t)=\left[\frac{\Delta t}{\Delta x^2}\sum_{i=neighbors}\big(c(i,t)-c(x,t)\big)\right]-c(x,t)$$

where the sum over index '*i*' goes over neighbors of point (*x,y,z*) and the neighbors will have the following concentrations: $c(x+\Delta x,t)$, $c(y+\Delta y,t)$, $\ldots$, $c(z+\Delta z,t)$.

4. Calculating center of mass when using periodic boundary conditions.

When you are running calculation with periodic boundary condition you may end up with situation like in the figure below:

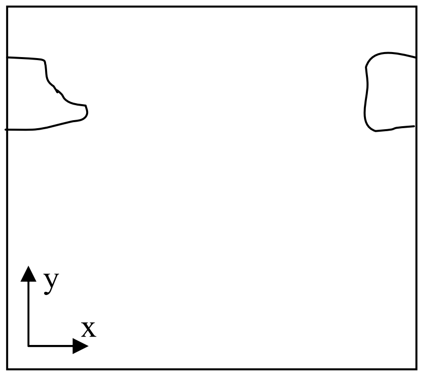


Figure 23. A connected cell in the lattice edge area – periodic boundary conditions are applied

Clearly what happens is that simply connected cell is wrapped around the lattice edge so part of it is in the region of high values of x coordinate and the other is in the region where x coordinates have low values. Consequently a naïve calculation of center of mass position according to :

$$x_{COM}=\frac{\sum_i x_i}{V}$$

would result in x_{COM} being somewhere in the middle of the lattice and abviously outside the cell.A better procedure could be as follows: Before calculating center of mass when new pixel is added or lost we "shift" a cell and new pixel (gained or lost)to the middle of the lattice do calculations "in the middle of the lattice" and shift back. Now if after shifting back it turns out that center of mass of a cell lies outside lattice position it in the center of mass by applygin a shift equal to the length of the lattice and whose direction should be such that the center of mass of the cell ends up inside the lattice (there is only one such shift and it might be be equal to zero vector).

This is how we do it using mathematical formulas:

$\vec{s} = \vec{x}_{COM} - \vec{c}$

First we define shift vector $\vec{s}$ as a vector difference between vector pointing to center of mass of the lattice and vector pointing to (approximately) the middle of the lattice.

Next we shift cell to the middle of the lattice using :

$\vec{x}'_{COM} = \vec{x}_{COM} - \vec{s}$

where $\vec{x}'_{COM}$ denotes center of mass position of a cell after shifting but before adding or subtracting a pixel.

Next we take into account the new pixel (either gained or lost) and calculate center of mass position (for the shifted cell):

$$\vec{x}'_{COM}{}^{new} = \frac{\vec{x}'_{COM} V + \vec{x}_i}{V+1}$$

Above we have assumed that we are adding one pixel.

Now all that we need to do is to shift back $\vec{x}'_{COM}{}^{new}$ by same vector $\vec{s}$ that brought cell to (approximately) center of the lattice.

$\vec{x}_{COM}{}^{new} = \vec{x}'_{COM}{}^{new} + \vec{s}$

We are almost done. We still have to check if $\vec{x}_{COM}{}^{new}$ is inside the lattice. If this is not the case we need to shift it back to the lattice but now we are allowed to use only a vector whose components are multiples of lattice dimensions (and we can safely restrict to +1 and -1 multiples of the lattice dimmensions) . For example we may have:

$\vec{P} = (x_{\max}, -y_{\max}, 0)$ where $x_{\max}$, $y_{\max}$, $z_{\max}$ are dimensions of the lattice.

There is no cheating here. In the lattice with periodic boundary conditions you are allowed to shift point coordinates a vector whose components are multiples of lattice dimensions.

All we need to do is to examine new center of mass position and form suitable vector $\vec{P}$.