Introduction to CompuCell3D

Version 3.6.2

Maciej H. Swat, Julio Belmonte, 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.

1 Introduction

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

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

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

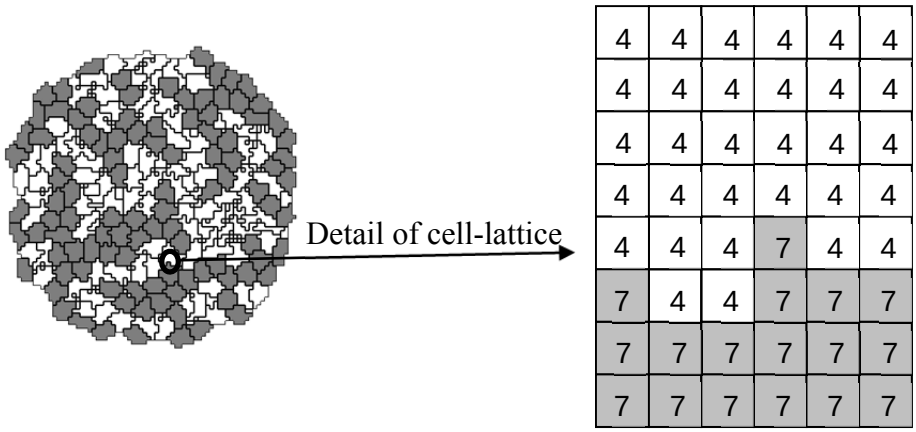


Figure 1. Detail of a typical two-dimensional GGH cell-lattice configuration. Each colored domain represents a single spatially-extended cell. The detail shows that each generalized cell is a set of cell-lattice sites (or *pixel*), $\vec{i}$, with a unique index, $\sigma(\vec{i})$, here 4 or 7. The color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

Multi-cell simulations are useful to interpolate between single-cell and continuum-tissue extremes because cells provide a natural level of abstraction for simulation of tissues,

organs and organisms ***(4)***. Treating cells phenomenologically reduces the millions of interactions of gene products to several behaviors: most cells can move, divide, die, differentiate, change shape, exert forces, secrete and absorb chemicals and electrical charges, and change their distribution of surface properties. The *Glazier-Graner-Hogeweg* (*GGH*) approach facilitates multiscale simulations by defining spatially-extended *generalized cells*, which can represent clusters of cells, single cells, 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.

2 GGH Applications

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

3 GGH Simulation Overview

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

Objects in a GGH simulation are either generalized cells or *fields* in two dimensions (*2D*) or three dimensions (*3D*). Generalized cells are spatially-extended objects (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 *effective 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.

3.1 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):

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

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,

$$J\left(\tau(\sigma),\tau(\sigma')\right) = J\left(\tau(\sigma'),\tau(\sigma)\right). \qquad (2)$$

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:

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

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:

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

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:

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

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*:

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

3.2 Dynamics

A GGH simulation consists of many attempts to copy cell indices between neighboring pixels. In CompuCell3D the default dynamical algorithm is *modified Metropolis dynamics*. During each index-copy attempt, we select a *target* pixel, $\vec{i}$, 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 .

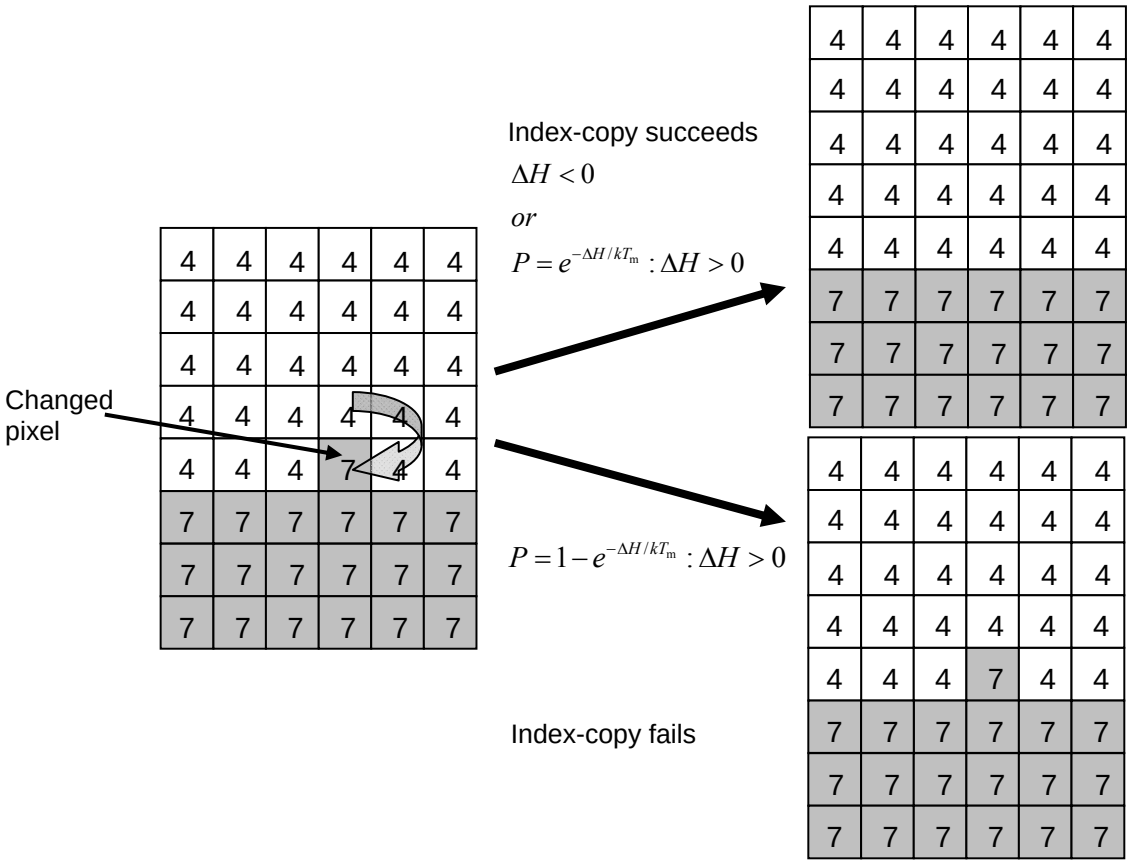


Figure 2. GGH representation of an index-copy attempt for two cells on a 2D square lattice – The "white" pixel (source) attempts to replace the "grey" pixel (target). The probability of accepting the index copy is given by equation (7).

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:

$$P\left(\sigma\left(\vec{i}\right) \to \sigma\left(\vec{i}'\right)\right) = \left\{\exp\left(-\Delta H / T_m : \Delta H > 0;\ 1 : \Delta H \le 0\right)\right\}, \qquad (7)$$

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

3.3 Algorithmic Implementation of Effective-Energy Calculations

Consider an effective energy consisting of boundary-energy and volume-constraint terms as in equation(6). After choosing the source ($\vec{i}'$) 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.

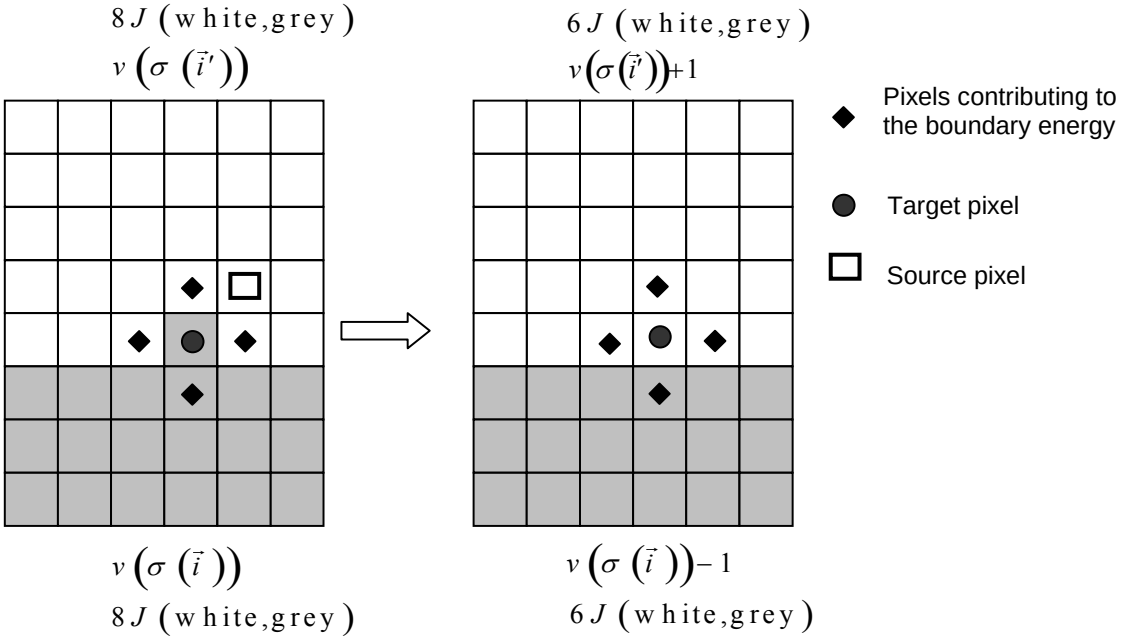


Figure 3. Calculating changes in the boundary energy and the volume-constraint energy on a nearest-neighbor square lattice.

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.

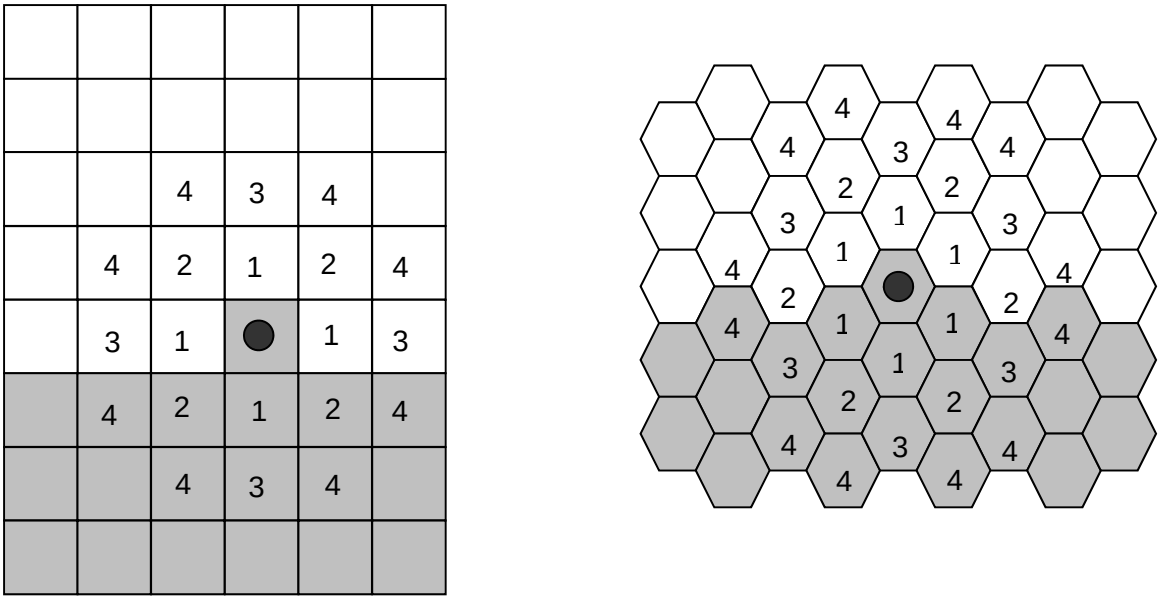


Figure 4. Locations of n^{th}-nearest neighbors on a 2D square lattice and a 2D hexagonal lattice.

	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.

4 CompuCell3D

CompuCell3D allows users to build sophisticated models more easily and quickly than does specialized custom code. It also facilitates model reuse and sharing.

A CC3D model consists of CC3DML scripts (an XML-based format), Python scripts, and files specifying the initial configurations of any fields and the cell lattice. The CC3DML

script specifies basic GGH parameters such as lattice dimensions, cell types, biological

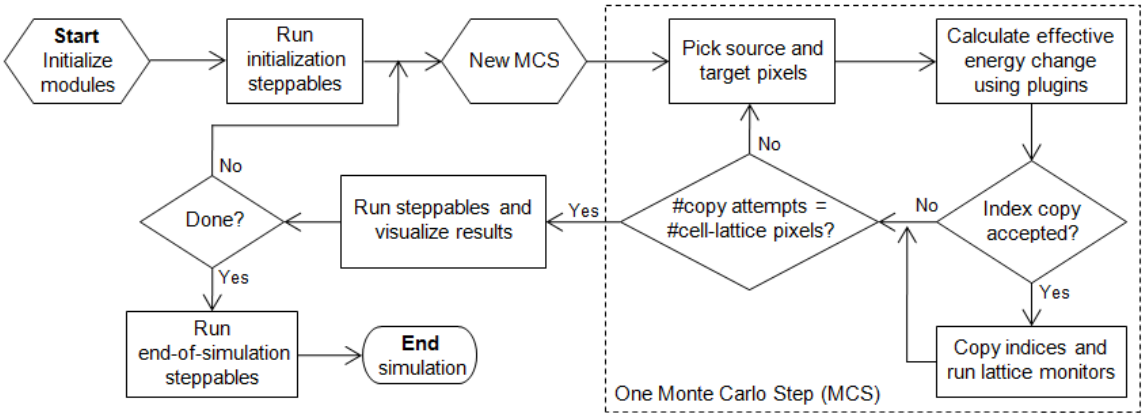


Figure 5 Flow chart of the GGH algorithm as implemented in CompuCell3D.

mechanisms and auxiliary information such as file paths. Python scripts primarily monitor the state of the simulation and implement changes in cell behaviors, *e.g.* changing the type of a cell depending on the oxygen partial pressure in a simulated tumor.

CompuCell3D is modular, loading only the modules needed for a particular model. Modules which calculate effective energy terms or monitor events on the cell lattice are called *plugins*. Effective-energy calculations are invoked every pixel copy attempt, while cell-lattice monitoring plugins run whenever an index copy occurs. Because plugins are the most frequently called modules in CC3D, most are coded in C++ for speed.

Modules called *steppables* usually performs operations on cells, not on pixels. Steppables are called at fixed intervals measured in Monte-Carlo steps. Steppables have three main uses: 1) to adjust cell parameters in response to simulation events[2], 2) to solve PDEs, 3) to load simulation initial conditions or save simulation results. Most steppables are implemented in Python. Much of the flexibility of CC3D comes from user-defined Python steppables.

The CC3D kernel supports parallel computation in shared-memory architectures (via OpenMP), providing substantial speedups on multi-core computers.

Besides the computational kernel of CC3D, the main components of the CC3D environment are: 1) Twedit++-CC3D – a model editor and code generator, 2) CellDraw – a graphical tool for configuring the initial cell lattice, 3) CC3D Player – a graphical tool for running, replaying and analyzing simulations.

Twedit++-CC3D provides a Simulation Wizard which generates draft CC3D model code based on high-level specification of simulation objects such as cell types and their behaviors, fields and interactions. Currently, the user must adjust default parameters in the auto-generated draft code, but later versions will provide interfaces for parameter specification. Twedit++-CC3D also provides a Python code-snippet generator, which simplifies coding Python CC3D modules.

[2] We will use the word *model* to describe the specification of a particular biological system and *simulation* to refer to a specific instance of the execution of such a model.

CellDraw allows users to draw regions which it fills with cells of user-specified types. It also imports microscope images for manual segmentation.
CC3D Player is a graphical interface which loads and executes CC3D models. It allows users to change model parameters during execution (*steering*), define multiple 2D and 3D visualizations of the cell lattice and fields and conduct real-time simulation analysis. CC3D Player also supports batch mode execution on clusters.

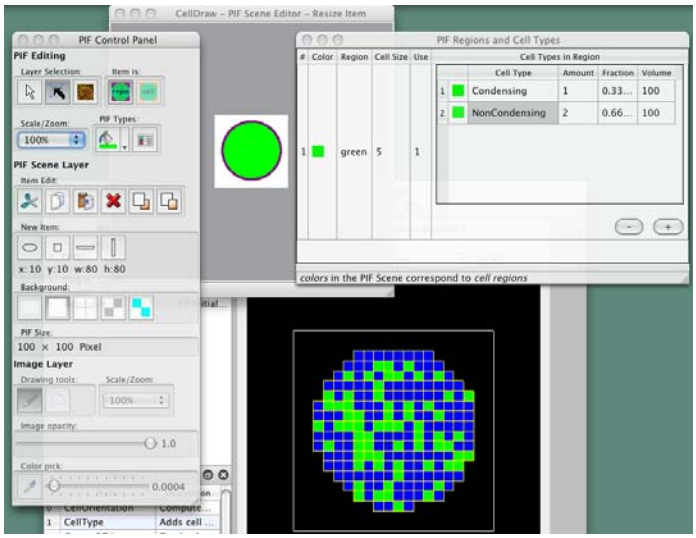


Figure 6 *CellDraw graphics tools and GUI.*

5 Building CC3DML-Based Simulations Using CompuCell3D

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

We use Twedit++-CC3D code generation and explain how to turn automatically-generated draft code into executable models. All of the parameters appearing in the autogenerated simulation scripts are set to their default values.

5.1 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:[3]

[3] 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

```
<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.

5.2 Cell-Sorting Simulation

Cell sorting due to differential adhesion between cells of different types is one of the basic mechanisms creating tissue domains during development and wound healing and in maintaining domains in homeostasis. In a classic *in vitro* cell sorting experiment to determine relative cell adhesivities in embryonic tissues, mesenchymal cells of different types are dissociated, then randomly mixed and reaggregated. Their motility and differential adhesivities then lead them to rearrange to reestablish coherent homogenous domains with the most cohesive cell type surrounded by the less. The simulation of the sorting of two cell types was the original motivation for the development of GGH methods. Such simple simulations show that the final configuration depends only on the hierarchy of adhesivities, while the sorting dynamics depends on the ratio of the adhesive energies to the amplitude of cell fluctuations.

To invoke the simulation wizard to create a simulation, we click `CC3DProject->New CC3D Project` in the menu bar. In the initial screen we specify the name of the model (`cellsorting`), its storage directory (*C:\CC3DProjects*) and whether we will store the model as pure CC3DML, Python and CC3DML or pure Python. This tutorial will use Python and CC3DML.

---

listings are either boxed or presented with line numbers. Punctuation at the end of boxes is implicit.

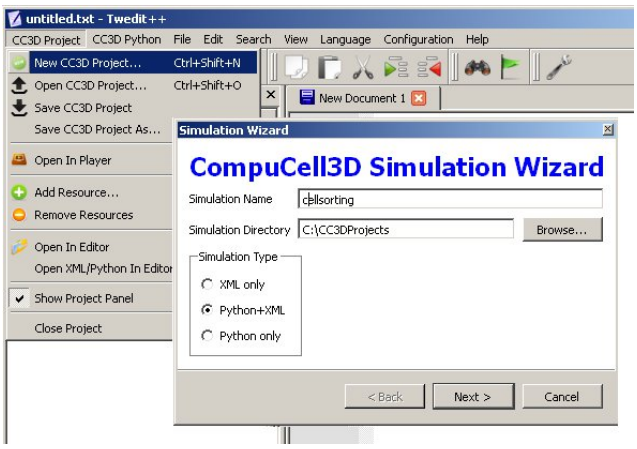


Figure 7 *Invoking the CompuCell3D Simulation Wizard from Twedit++.*

On the next page of the Wizard we specify GGH global parameters, including cell-lattice dimensions, the cell fluctuation amplitude, the duration of the simulation in Monte-Carlo steps and the initial cell-lattice configuration.
In this example, we specify a `100x100x1` cell-lattice, *i.e.*, a 2D model, a fluctuation amplitude of 10, a simulation duration of 10000 MCS and a pixel-copy range of 2. `BlobInitializer` initializes the simulation with a disk of cells of specified size.

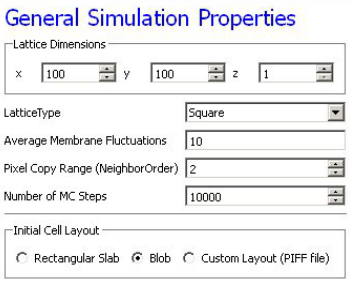


Figure 8 *Specification of basic cell-sorting properties in Simulation Wizard.*

On the next Wizard page we name the cell types in the model. We will use two cells types: `Condensing` (more cohesive) and `NonCondensing` (less cohesive). CC3D by default includes a special generalized-cell type `Medium` with unconstrained volume which fills otherwise unspecified space in the cell-lattice.

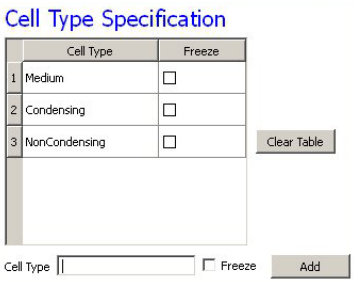


Figure 9 Specification of cell-sorting cell types in Simulation Wizard.

We skip the Chemical Field page of the Wizard and move to the Cell Behaviors and Properties page. Here we select the biological behaviors we will include in our model. **Objects in CC3D have no properties or behaviors unless we specify then explicitly**. Since cell sorting depends on differential adhesion between cells, we select the *Contact Adhesion* module from the Adhesion section and give the cells a defined volume using the *Volume Constraint* module.

Figure 10 Selection of cell-sorting cell behaviors in Simulation Wizard.[4]

We skip the next page related to Python scripting, after which Twedit++-CC3D generates the draft simulation code. Double clicking on `cellsorting.cc3d` opens both the CC3DML (*cellsorting.xml*) and Python scripts for the model. Because the CC3DML file contains the complete model in this example, we postpone discussion of the Python script. A CC3DML file has 3 distinct sections. The first, the *Lattice Section* (lines 2-7) specifies global parameters like the cell-lattice size. The *Plugin Section* (lines 8-30) lists all the plugins used, *e.g.* `CellType` and `Contact`. The *Steppable Section* (lines 32-39) lists all steppables, here we use only `BlobInitializer`.

```
01   <CompuCell3D version="3.6.0">
02    <Potts>
03     <Dimensions x="100" y="100" z="1"/>
04     <Steps>10000</Steps>
05     <Temperature>10.0</Temperature>
06     <NeighborOrder>2</NeighborOrder>
07    </Potts>
08
09    <Plugin Name="CellType">
10     <CellType TypeId="0" TypeName="Medium"/>
11     <CellType TypeId="1" TypeName="Condensing"/>
12     <CellType TypeId="2" TypeName="NonCondensing"/>
13    </Plugin>
14
15    <Plugin Name="Volume">
16     <VolumeEnergyParameters CellType="Condensing"
         LambdaVolume="2.0" TargetVolume="25"/>
17     <VolumeEnergyParameters CellType="NonCondensing"
         LambdaVolume="2.0" TargetVolume="25"/>
18    </Plugin>
19
20    <Plugin Name="CenterOfMass"/>
21
22    <Plugin Name="Contact">
23     <Energy Type1="Medium" Type2="Medium">10</Energy>
24     <Energy Type1="Medium" Type2="Condensing">10</Energy>
25     <Energy Type1="Medium" Type2="NonCondensing">10</Energy>
```

[4] We have graphically edited screenshots of Wizard pages to save space.

```
26     <Energy Type1="Condensing"Type2="Condensing">10</Energy>
27     <Energy Type1="Condensing" Type2="NonCondensing">10</Energy>
28     <Energy Type1="NonCondensing" Type2="NonCondensing">10</Energy>
29     <NeighborOrder>2</NeighborOrder>
30    </Plugin>
31
32    <Steppable Type="BlobInitializer">
33     <Region>
34      <Center x="50" y="50" z="0"/>
35      <Radius>20</Radius>
36      <Width>5</Width>
37      <Types>Condensing,NonCondensing</Types>
38     </Region>
39    </Steppable>
40   </CompuCell3D>
```

Listing 1 *Simulation-Wizard-generated draft CC3DML (XML) code for cell-sorting.*[5]

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*. They can be changed 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 is quite special as it does not participate directly in index copies, but is used by other plugins for cell-type-to-cell-index mapping.It 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. `Medium` (appearing in *Listing 1*)is a special cell type with unconstrained

[5] We use indent each nested block by two spaces in all listings in this paper to avoid distracting rollover of text at the end of the line. However, both Simulation Wizard and standard Python use an indentation of four spaces per block.

volume and surface area that fills all cell-lattice pixels unoccupied by cells of other types.[6]

Steppables section consists of module declaration which follow the following patern:

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

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

By autogenerating CC3DML code, Twedit++-CC3D releases user from remembering all the rules necessary to construct a valid CC3DML simulation script. All parameters appearing in the autogenerated CC3DML script have default values inserted by Simulation Wizard.

We must edit the parameters in the draft CC3DML script to build a functional cell-sorting model (Listing 1). The `CellType` plugin (lines 9-13) already provides three generalized-cell types: `Condensing` (C), `NonCondensing` (N) and `Medium` (M), so we need not change it.

However, the boundary-energy (Contact-energy) matrix in the `Contact` plugin (lines 22-30) is initially filled with identical values, *i.e.*, the cell types are identical. For cell-sorting, `Condensing` cells must adhere strongly to each other (so we set J_{CC}=2), `Condensing` and `NonCondensing` cells must adhere more weakly (here we set J_{CN}=11) and all other adhesion must be very weak (we set $J_{NN}=J_{CM}=J_{NM}$=16), as discussed in section. The value of J_{MM} =0 is irrelevant, since the `Medium` generalized cell does not contact itself.

To reduce artifacts due to the anisotropy of the square cell-lattice we increase the neighbor-order range in the contact energy to 2 so the contact-energy sum in equation (1) will include nearest and second-nearest neighbors (line 29).

In the `Volume` plugin, which calculates the Volume-constraint energy given in equation (4) the attributes `CellType`, `LambdaVolume` and `TargetVolume` inside the `<VolumeEnergyParameters>` tags specify $\lambda(\tau)$ and $V_t(\tau)$ for each cell type. In our simulations we set $V_t(\tau) = 25$ and $\lambda(\tau) = 2.0$ for both cell types.

We initialize the cell lattice using the `BlobInitializer`, which creates one or more disks (solid spheres in 3D) of cells. Each region is enclosed between `<Region>` tags. The `<Center>` tag with syntax `<Center x="x_position" y="y_position" z="z_position"/>` specifies the position of the center of the disk. The `<Width>` tag specifies the size of the initial square (cubical in 3D) generalized cells and the `<Gap>` tag creates space between neighboring cells. The `<Types>` tag lists the cell types to fill the disk. Here, we change the `Radius` in the draft `BlobInitializer` specification to 40. These few changes produce a working cell-sorting simulation.

To run the simulation we right click `cellsorting.cc3d` in the left panel and choose the `Open In Player` option. We can also run the simulation by opening CompuCellPlayer and selecting `cellsorting.cc3d` from the `File-> Open Simulation File…` dialog. Figure 11 shows snapshots of a simulation of the cell-sorting model. The less cohesive `NonCondensing` cells engulf the more cohesive `Condensing` cells, which cluster and form

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

a single central domain. By changing the boundary energies we can produce other cell-sorting patterns (***77***). In particular, if we reduce the contact energy between the `Condensing` cell type and the `Medium`, we can force inverted cell sorting, where the `Condensing` cells surround the `NonCondensing` cells. If we set the heterotypic contact energy to be less than either of the homotypic contact energies, the cells of the two types will mix rather than sort. If we set the cell-medium contact energy to be very small for one cell type, the cells of that type will disperse into the medium, as in cancer invasion. With minor modifications, we can also simulate the scenarios for three or more cell types, for situations in which the cells of a given type vary in volume, motility or adhesivity, or in which the initial condition contains coherent clusters of cells rather than randomly mixed cells (engulfment).

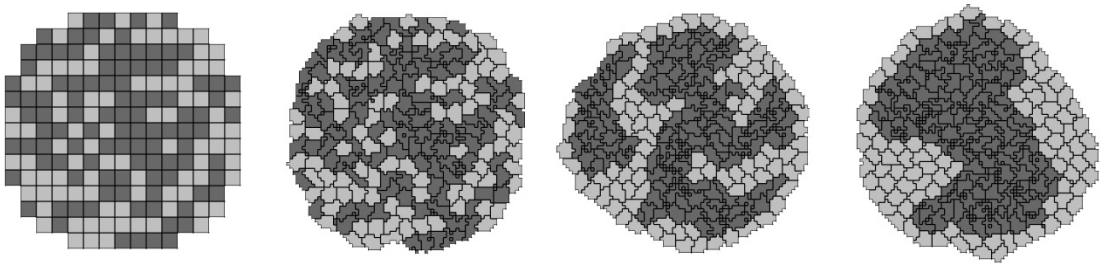


Figure 11 *Snapshots of the cell-lattice configurations for the cell-sorting simulation in* ***Listing 1****. The boundary-energy hierarchy drives* `NonCondensing` *(light grey) cells to surround* `Condensing` *(dark grey) cells. The white background denotes surrounding* `Medium`*.*

5.3 Angiogenesis Model

Vascular development is central to both development and cancer progression. We present a simplified model of the earliest phases of capillary network assembly by endothelial cells based on cell adhesion and contact-inhibited chemotaxis. This model does a good job of reproducing the patterning and dynamics which occur if we culture Human Umbilical Vein Endothelial Cells (*HUVEC*) on matrigel in a quasi-2D *in vitro* experiment (***101***,***102***). In addition to generalized cells modeling the HUVEC, we will need a diffusing chemical object, here, Vascular Endothelial Growth Factor (*VEGF*), cell secretion of VEGF and cell-contact-inhibited chemotaxis to VEGF.

We will use a 3D voxel (pixel) with a side of 4 μm, *i.e.* a volume of 64 μm^3. Since the experimental HUVEC speed is about 0.4 μm/min and cells in this simulation move at an average speed of 0.1 pixel/MCS, one MCS represents one minute.

In the Simulation Wizard, we name the model `ANGIOGENESIS`, set the cell- and field-lattice dimensions to 50×50×50, the membrane fluctuation amplitude to 20, the pixel-copy range to 3, the number of MCS to 10000 and select `BlobFieldInitializer` to produce the initial cell-lattice configuration. We have only one cell type – `Endothelial`. In the `Chemical Fields` page we create the `VEGF` field and select `FlexibleDiffusionSolverFE` from the `Solver` pull-down list.

Figure 12 Specification of the angiogenesis chemical field in Simulation Wizard.

Next, on the `Cell Properties and Behaviors` page, we select the `Contact` module from the `Adhesion-behavior` group and add `Secretion`, `Chemotaxis` and `Volume-constraint` behaviors by checking the appropriate boxes.

Figure 13 Specification of angiogenesis cell behaviors in Simulation Wizard.

Because we have invoked `Secretion` and `Chemotaxis`, the Simulation Wizard opens their configuration screens. On the `Secretion` page, from the pull-down list, we select the chemical to secrete by selecting `VEGF` in the `Field` pull-down menu and the cell type secreting the chemical (`Endothelial`), and enter the rate of 0.013 (50 pg (cell h)$^{-1}$ = 0.013 pg (voxel MCS)$^{-1}$, compare to (***103***)). We leave the `Secretion Type` entry set to `Uniform`, so each pixel of an endothelial cell secretes the same amount of `VEGF` at the same rate. Uniform volumetric secretion or secretion at the cell's center of mass may be most appropriate in 2D simulations of planar geometries (*e.g.* cells on a petrie dish or agar) where the biological cells are actually secreting up or down into a medium that carries the diffusant. CC3D also supplies a secrete-on-contact option to secrete outwards from the cell boundaries and allows specification of which boundaries can secrete, which is more realistic in 3D. However, users are free to employ any of these methods in either 2D or 3D depending on their interpretation of their specific biological situation. CompuCell3D does not have intrinsic units for fields, so the amount of a chemical can be interpreted in units of moles, number of molecules or grams. We click the `Add Entry` button to add the secretion information, then proceed to the next page to define the cells' chemotaxis properties.

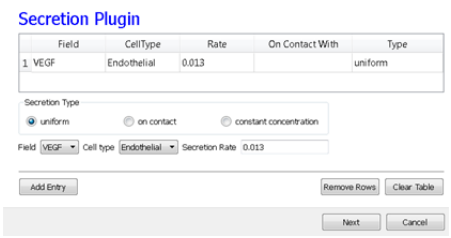


Figure 14 Specification of angiogenesis secretion parameters in Simulation Wizard.

On the `Chemotaxis` page, we select `VEGF` from the `Field` pull-down list and `Endothelial` for the cell type, entering a value for `Lambda` of 5000. When the `chemotaxis type` is `regular`, the cell's response to the field is linear, *i.e.* the effective strength of chemotaxis depends on the product of `Lambda` and the secretion rate of `VEGF`, *e.g.* a `Lambda` of 5000 and a `secretion rate` of 0.013 has the same effective chemotactic strength as a `Lambda` of 500 and a `secretion rate` of 0.13. Since endothelial cells do not chemotax at surfaces where they contact other endothelial cells (contact-inhibition), we select `Medium` from the pull-down menu next to the `Chemotax Towards` button and click this button to add `Medium` to the list of generalized cell types whose interfaces with `Endothelial` cells support chemotaxis. We click the `Add Entry` button to add the chemotaxis information, then proceed to the final Simulation Wizard page.

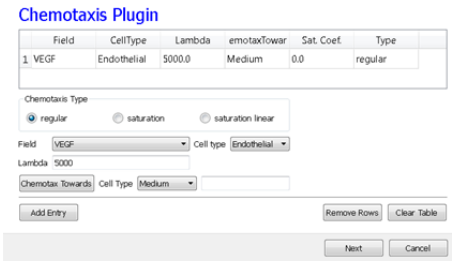


Figure 15 *Specification of angiogenesis chemotaxis properties in Simulation Wizard.*

Next, we adjust the parameters of the draft model. Pressure from chemotaxis to VEGF reduces the average endothelial-cell volume by about 10 voxels from the target volume. So, in the `Volume` plugin we set `TargetVolume` to 74 (64+10) and `LambdaVolume` to 20.0. In experiments, in the absence of chemotaxis no capillary network forms and cells adhere to each other to form clusters. We therefore set J_{MM}=0, J_{EM}=12 and J_{EE}=5 in the `Contact` plugin (`M`: `Medium`, `E`: `Endothelial`). We also set the `NeighborOrder` for the `Contact` energy calculations to 4.

The diffusion equation that governs VEGF ($V\left(\vec{x}\right)$) field evolution is:

$$\frac{\partial V(\vec{x})}{\partial t} = D_{\mathrm{VEGF}}^{\mathrm{EC}} \nabla^2 V(\vec{x}) - \gamma_{\mathrm{VEGF}} V(\vec{x}) \delta(\tau(\sigma(\vec{x})), \mathrm{M}) + S^{\mathrm{EC}} \delta(\tau(\sigma(\vec{x})), \mathrm{EC}), \qquad (9)$$

where $\delta(\tau(\sigma(\vec{x})), \mathrm{EC}) = 1$ inside `Endothelial` cells and 0 elsewhere and $\delta(\tau(\sigma(\vec{x})), \mathrm{M}) = 1$ inside `Medium` and 0 elsewhere. We set the diffusion constant D_{VEGF} =0.042 μm^2/sec (0.16 voxel^2/MCS, about two orders of magnitude smaller than experimental values),[7] the decay coefficient γ_{VEGF}=1 h^{-1} (0.016 MCS^{-1}) for `Medium` pixels and γ_{VEGF}=0 inside `Endothelial` cells, and the secretion rate S^{EC}=0.013 pg (voxel MCS)$^{-1}$.

[7] `FlexibleDiffusionSolverFE` becomes unstable for values of D_{VEGF}>0.16 voxel^2/MCS. For larger diffusion constants we must call the algorithm multiple times per MCS (See the *Three-Dimensional Vascular Solid Tumor Growth* section).

In the CC3DML script describing `FlexibleDiffusionSolverFE` (Listing 2, lines 38-47) we set the values of the `<DiffusionConstant>` and `<DecayConstant>` tags to 0.16 and 0.016 respectively. To prevent chemical decay inside Endothelial cells we add the line `<DoNotDecayIn>Endothelial</DoNotDecayIn>` inside the `<DiffusionData>` tag pair. Finally, we edit `BlobInitializer` (lines 49-56) to start with a solid sphere 10 pixels in radius centered at x=25, y=25, z=25 with initial cell width 4, as in **Listing 2**.

```
01  <CompuCell3D version="3.6.0">
02
03  <Potts>
04   <Dimensions x="50" y="50" z="50"/>
05   <Steps>10000</Steps>
06   <Temperature>20.0</Temperature>
07   <NeighborOrder>3</NeighborOrder>
08  </Potts>
09
10  <Plugin Name="CellType">
11   <CellType TypeId="0" TypeName="Medium"/>
12   <CellType TypeId="1" TypeName="Endothelial"/>
13  </Plugin>
14
15  <Plugin Name="Volume">
16   <VolumeEnergyParameters CellType="Endothelial"
       LambdaVolume="20.0" TargetVolume="74"/>
17  </Plugin>
18
19  <Plugin Name="Contact">
20   <Energy Type1="Medium" Type2="Medium">0</Energy>
21   <Energy Type1="Medium" Type2="Endothelial">12</Energy>
22   <Energy Type1="Endothelial" Type2="Endothelial">5</Energy>
23   <NeighborOrder>4</NeighborOrder>
24  </Plugin>
25
26  <Plugin Name="Chemotaxis">
27   <ChemicalField Name="VEGF" Source="FlexibleDiffusionSolverFE">
28    <ChemotaxisByType ChemotactTowards="Medium" Lambda="5000.0"
        Type="Endothelial"/>
29   </ChemicalField>
30  </Plugin>
31
32  <Plugin Name="Secretion">
33   <Field Name="VEGF">
34    <Secretion Type="Endothelial">0.013</Secretion>
35   </Field>
36  </Plugin>
37
38  <Steppable Type="FlexibleDiffusionSolverFE">
39   <DiffusionField>
40    <DiffusionData>
41     <FieldName>VEGF</FieldName>
42     <DiffusionConstant>0.16</DiffusionConstant>
43     <DecayConstant>0.016</DecayConstant>
44     <DoNotDecayIn> Endothelial</DoNotDecayIn>
45    </DiffusionData>
46   </DiffusionField>
47  </Steppable>
48
49  <Steppable Type="BlobInitializer">
50   <Region>
51    <Center x="25" y="25" z="25"/>
52    <Radius>10</Radius>
```

```
53     <Width>4</Width>
54     <Types>Endothelial</Types>
55    </Region>
56   </Steppable>
57
58   </CompuCell3D>
```

Listing 2 *CC3DML code for the angiogenesis model.*

The main behavior that drives vascular patterning is contact-inhibited chemotaxis (Listing 2, lines 26-30). VEGF diffuses away from cells and decays in `Medium`, creating a steep concentration gradient at the interface between `Endothelial` cells and `Medium`. Because `Endothelial` cells chemotax up the concentration gradient only at the interface with `Medium` the `Endothelial` cells at the surface of the cluster compress the cluster of cells into vascular branches and maintain branch integrity.
We show screenshots of a simulation of the angiogenesis model in Figure 16 (***102***, ***104***). We can reproduce either 2D or 3D primary capillary network formation and the rearrangements of the network agree with experimentally-observed dynamics. If we eliminate the contact inhibition, the cells do not form a branched structure (as observed in chick allantois experiments, (***102***)). We can also study the effects of surface tension, external growth factors and changes in motility and diffusion constants on the pattern and its dynamics. However, this simple model does not include the strong junctions HUVEC cells make with each other at their ends after a period of prolonged contact. It also does not attempt to model the vacuolation and linking of vacuoles that leads to a connected network of tubes.

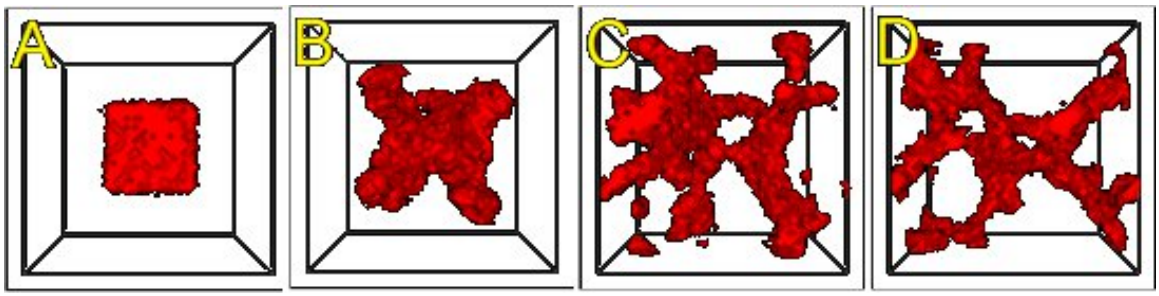


Figure 16 *An initial cluster of adhering endothelial cells forms a capillary-like network via sprouting angiogenesis. A: 0 hours (0 MCS), B: ~2 hours (100 MCS), C: ~5 hours (250 MCS), D: ~18 hours (1100 MCS).*

Since real endothelial cells are elongated, we can include the `Cell-elongation` plugin in the Simulation Wizard to better reproduce individual cell morphology. However, excessive cell elongation causes cell fragmentation. Adding either the `Global` or `Fast Connectivity Constraint` plugin prevents cell fragmentation.

5.4 Bacterium-and-Macrophage Simulation

Another example which illustrates the use of chemical fields is based on 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)***. The basic procedure to construct the simulation is very similar to the one we followed in constructing angiogenesis model.

In Twedit++-CC3D we open new project and name it bacterium_macrophage. We declare 3 cell types – Bacterium, Macrophage and Red (red blood cells). We assume that diffusing chemoattractant is secreted by bacteria, therefore on the Chemical Field page of the Simulation Wizard we declare ATTR chemical field which we will solve using DiffusionSolverFE. On the Cell Behaviors and Properties page we select Contact, Chemotaxis, VolumeFlex and Surface Flex. Clicking 'Next' button brings us to chemotaxis page where we set chemotaxis parameters as shown on Figure 17:

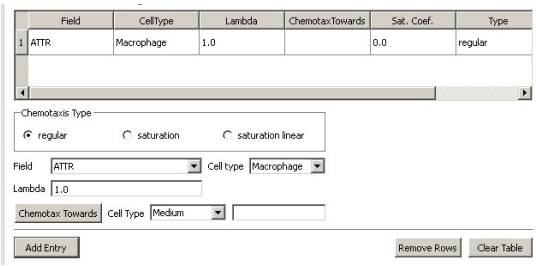


Figure 17 Setting up chemotaxis properties forMacrophages

After code-autogenerating is done we have to do several adjustments to the CC3DML script.

```
01   <CompuCell3D version="3.6.2">
02
03      <Potts>
04         <Dimensions x="100" y="100" z="1"/>
05         <Steps>10000</Steps>
06         <Temperature>40.0</Temperature>
07         <NeighborOrder>2</NeighborOrder>
08         <Boundary_x>Periodic</Boundary_x>
09         <Boundary_y>Periodic</Boundary_y>
10      </Potts>
11
12      <Plugin Name="CellType">
13         <CellType TypeId="0" TypeName="Medium"/>
14         <CellType TypeId="1" TypeName="Bacterium"/>
15         <CellType TypeId="2" TypeName="Macrophage"/>
16         <CellType TypeId="3" TypeName="Red"/>
17      </Plugin>
18
19      <Plugin Name="Volume">
20         <VolumeEnergyParameters CellType="Bacterium" LambdaVolume="60.0"
TargetVolume="10"/>
21         <VolumeEnergyParameters CellType="Macrophage" LambdaVolume="15.0"
TargetVolume="150"/>
22         <VolumeEnergyParameters CellType="Red" LambdaVolume="30.0"
TargetVolume="100"/>
23      </Plugin>
```

```
24
25      <Plugin Name="Surface">
26         <SurfaceEnergyParameters CellType="Bacterium" LambdaSurface="4.0"
TargetSurface="10"/>
27         <SurfaceEnergyParameters CellType="Macrophage" LambdaSurface="20.0"
TargetSurface="50"/>
28         <SurfaceEnergyParameters CellType="Red" LambdaSurface="0.0"
TargetSurface="40"/>
29      </Plugin>
30
31      <Plugin Name="Contact">
32         <Energy Type1="Medium" Type2="Medium">10.0</Energy>
33         <Energy Type1="Medium" Type2="Bacterium">8.0</Energy>
34         <Energy Type1="Medium" Type2="Macrophage">8.0</Energy>
35         <Energy Type1="Medium" Type2="Red">30.0</Energy>
36         <Energy Type1="Bacterium" Type2="Bacterium">150.0</Energy>
37         <Energy Type1="Bacterium" Type2="Macrophage">15.0</Energy>
38         <Energy Type1="Bacterium" Type2="Red">150.0</Energy>
39         <Energy Type1="Macrophage" Type2="Macrophage">150.0</Energy>
40         <Energy Type1="Macrophage" Type2="Red">150.0</Energy>
41         <Energy Type1="Red" Type2="Red">150.0</Energy>
42         <NeighborOrder>2</NeighborOrder>
43      </Plugin>
44
45      <Plugin Name="Chemotaxis">
46         <ChemicalField Name="ATTR" Source="DiffusionSolverFE">
47            <ChemotaxisByType Lambda="1.0" Type="Macrophage"/>
48         </ChemicalField>
49      </Plugin>
50
51      <Steppable Type="DiffusionSolverFE">
52         <DiffusionField>
53            <DiffusionData>
54               <FieldName>ATTR</FieldName>
55               <GlobalDiffusionConstant>0.1</GlobalDiffusionConstant>
56               <GlobalDecayConstant>5e-05</GlobalDecayConstant>
57               <DiffusionCoefficient CellType="Red">0.0</DiffusionCoefficient>
58            </DiffusionData>
59            <SecretionData>
60               <Secretion Type="Bacterium">100</Secretion>
61            </SecretionData>
62            <BoundaryConditions>
63               <Plane Axis="X">
64                  <Periodic/>
65               </Plane>
66               <Plane Axis="Y">
67                  <Periodic/>
68               </Plane>
69            </BoundaryConditions>
70         </DiffusionField>
71      </Steppable>
72
73      <Steppable Type="PIFInitializer">
74         <PIFName>Simulation/bacterium_macrophage.piff</PIFName>
75      </Steppable>
76
77   </CompuCell3D>
78
```

Listing 3. CC3DML code for Bacterium Macrophage simulation. Note that the code has been modified from its autogenetrated version

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. For a field $c\left(\vec{i}\right)$:

$$\Delta H_{\text{chem}} = -\lambda_{\text{chem}}\left(c\left(\vec{i}\right) - c\left(\vec{i}'\right)\right), \tag{10}$$

where $c\left(\vec{i}\right)$ is the chemical field at the index-copy target pixel, $c\left(\vec{i}'\right)$ the field at the index-copy source pixel, and λ_{chem} the strength and direction of chemotaxis. If $\lambda_{\text{chem}} > 0$ and $c\left(\vec{i}\right) > c\left(\vec{i}'\right)$, then ΔH_{chem} is negative, increasing the probability of accepting the index copy. 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 description of the chemotaxis plugin in the second part of this manual).

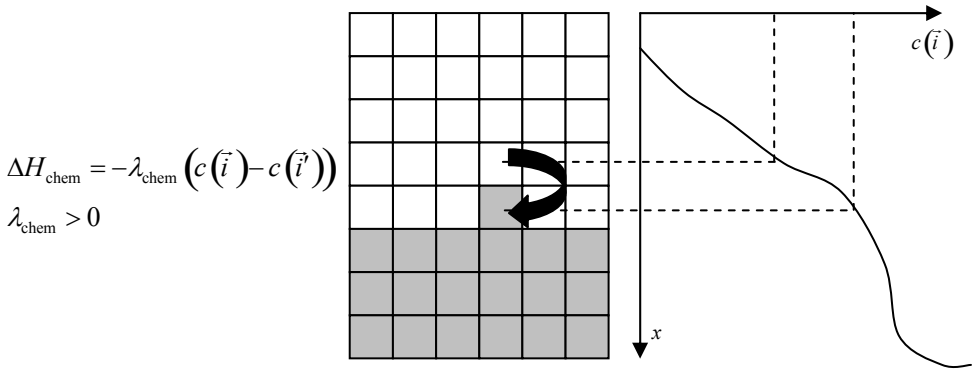


Figure 18. Connecting a field to GGH dynamics using a chemotaxis-energy term. The difference in the value of the field c at the source, $\vec{i}'$, and target, $\vec{i}$, pixels changes the ΔH of the index-copy attempt. Here $c\left(\vec{i}\right) > c\left(\vec{i}'\right)$ and $\lambda > 0$, so $\Delta H_{\text{chem}} < 0$, increasing the probability of accepting the index-copy attempt.

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 `DiffusionSolverFE`. `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 `DiffusionSolverFE` steppable as the source of our fields. The `DiffusionSolverFE` steppable is the main CompuCell3D tool for defining diffusing fields, which evolve according to the diffusion equation:

$$\frac{\partial c(\vec{i})}{\partial t} = D(\vec{i})\nabla^2 c(\vec{i}) - k(\vec{i})c(\vec{i}) + s(\vec{i}), \qquad (11)$$

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 global diffusion constant (using the `<GlobalDiffusionConstant>` tag pair) , and the global decay constant (using the `<GlobalDiffusionConstant>` tag pair). We can also specify diffusion constant for particular cell types using the following syntax:

```
<DiffusionCoefficient CellType="cell_type_1">coefficient</DiffusionCoefficient>
<DecayCoefficient CellType="cell_type_1">coefficient</DecayCoefficient>
```

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 and using the `DiffusionSolverFE` to solve diffusion equation releases users from specifying how many extra times the solver needs to be called.

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`.

To complete specification of the PDE diffusion equation we also set boundary conditions (by default they are set to no flux). Here however we set them to Periodic along x and y directions using the following syntax:

```
<BoundaryConditions>
 <Plane Axis="X">
```

```
  <Periodic/>
 </Plane>
 <Plane Axis="Y">
  <Periodic/>
 </Plane>
</BoundaryConditions>
```

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* (*PIFFs*). A PIFF 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.piff', we can load it into a simulation using the following syntax:

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

Listing 4 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 4. 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 19 shows the initial cell-lattice configuration specified in Listing 4:

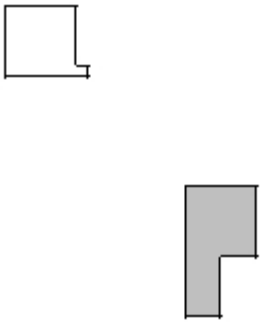


Figure 19. Initial configuration of the cell lattice based on the PIF in Listing 4.

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 5 shows the example PIF for the bacterium-and-macrophage simulation.

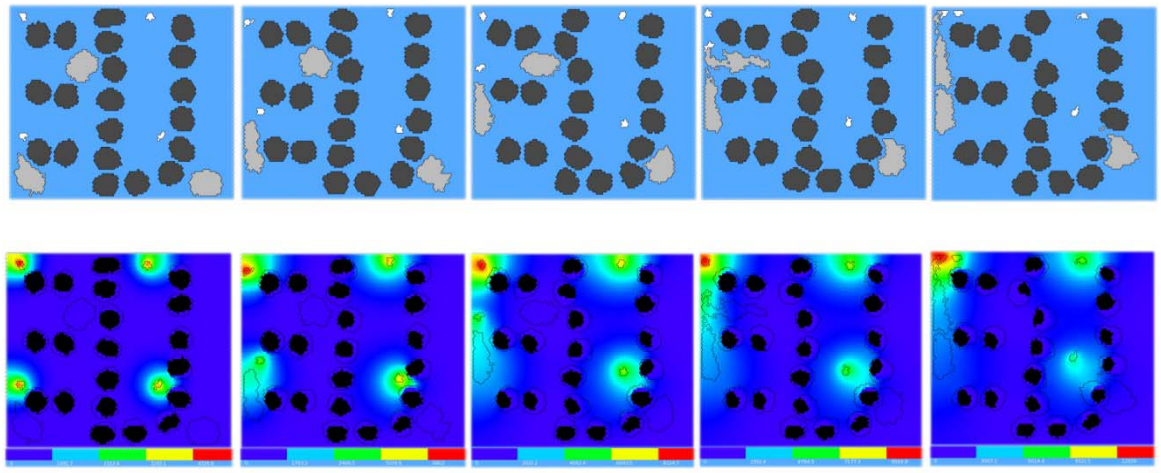
```
0 Red 10 20 10 20 0 0
1 Red 10 20 40 50 0 0
2 Red 10 20 70 80 0 0
3 Red 40 50 0 10 0 0
4 Red 40 50 30 40 0 0
5 Red 40 50 60 70 0 0
6 Red 40 50 90 95 0 0
7 Red 70 80 10 20 0 0
8 Red 70 80 40 50 0 0
9 Red 70 80 70 80 0 0
11 Bacterium 5 5 5 5 0 0
12 Macrophage 35 35 35 35 0 0
13 Bacterium 65 65 65 65 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 5. 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 3 we read the cell lattice configuration from the file 'bacterium_macrophage_2D_wall_v3.pif' using the lines:

```
<Steppable Type="PIFInitializer">
  <PIFName>Simulation/bacterium_macrophage.piff</PIFName>
 </Steppable>
```

Figure 20 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.



t=200 MCS *t*=500 MCS *t*=800 MCS *t*=900 MCS *t*=1100 MCS

Figure 20. Snapshots of the bacterium-and-macrophage simulation from Listing 3 and the PIF in Listing 5 saved in the file 'bacterium_macrophage_2D_wall_v3.pif'. The upper row shows the cell-lattice configuration with the `Macrophages` in grey, `Bacteria` in white, `red` blood cells in dark grey and `Medium` in blue. Second row shows the concentration of the chemoattractant ATTR secreted by the `Bacteria`. The bars at the bottom of the field images show the concentration scales (blue, low concentration, red, high concentration).

6 Python Scripting

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

Tasks performed by CompuCell3D modules either relate to index-copy attempts (plugins) or run either once, at the beginning or end of a simulation, or once every several MCS (steppables). Tasks run every index-copy attempt, like effective-energy-term calculations, are the most frequently-called tasks in a GGH simulation and writing them

in Python may slow simulations. However, steppables and lattice monitors are good candidates for Python implementation and cause negligible performance degradation. Python implementations are suitable for most cell-parameter adjustments that depend on the state of the simulation, *e.g.*, simulating cell growth in response to a chemical, cell-type differentiation and changes in cell-cell adhesion.

In the models we presented above, all cells had parameter values fixed in time. To allow cell behaviors to change, we need to be able to adjust cell properties during a simulation. CompuCell3D can execute Python scripts (CC3D supports Python versions 2.x) to modify the properties of cells in response to events occurring during a simulation, such as the concentration of a nutrient dropping below a threshold level, a cell reaching a doubling volume or a cell changing its neighbors. Most such Python scripts have a simple structure based on `print` statements, `if-elif-else` statements, `for` loops, `lists` and simple `classes` and do not require in-depth knowledge of Python to create.

6.1 A Short Introduction to Python

This section briefly introduces the main features of Python in the CompuCell3D context. For a more formal introduction to Python see (***98***) and http://python.org.
Python defines blocks of code, such as those appearing inside `if` statements or `for` loops (in general after ":"), by an increased level of indentation. This chapter uses 2 spaces per indentation level. For example, in Listing 3, we indent the body of the `if` statement by 2 spaces and the body of the inner `for` loop by an additional 2 spaces. The `for` loop is executed inside the `if` statement, which checks if we are in the second MCS of the simulation. The command `pixelOffset=10` assigns to the variable `pixelOffset` a value of `10`. The `for` loop assigns to the variable `x` values ranging from `0` through `self.dim.x-1`, where `self.dim.x` is a CC3D internal variable containing the size of the cell-lattice in the *x*-direction. When executed, Listing 3 prints consecutive integers from `10` to `10+self.dim.x-1`.

```
01  if (mcs==2):
02    pixelOffset = 10
03    for x in range(self.dim.x):
04      pixel = pixelOffset + x
05      print pixel
```

Listing 6 *Simple Python loop.*

The great advantage of Python compared to older languages like Fortran is that it can also iterate over members of a Python *list*, a *container* for grouping objects. Listing 4 executes a `for` loop over a list containing all cells in the simulation and prints the type of each cell.

```
01  for cell in self.cellList:
02    print "cell type=", cell.type
```

Listing 7 *Iterating over the inventory of CC3D cells in Python.*

Lists can combine objects of any type, including integers, strings, complex numbers, lists, and, in this case, CC3D cells. CompuCell3D uses lists extensively to keep track of cells, cell neighbors, cell pixels, *etc.*….

CompuCell3D allows users to construct custom Python code as independent modules called *steppables*, which are represented as classes. Listing 5 shows a typical CC3D Python steppable class. The first line declares the class name together with an argument (`SteppableBasePy`) inside the parenthesis which makes the main CC3D objects, including cells, lattice properties, *etc.*..., available inside the class. The `def __init__( self,_simulator,_frequency=1):` declares the initializing function `__init__` which is called automatically during class object instantiation. After initializing the class and inheriting CC3D objects, we declare 3 main functions called at different times during the simulation: `start` is called before the simulation starts; `step` is called at specified intervals in MCS throughout the simulation; and `finish` is called at the end of the simulation. The `start` function iterates over all cells, setting their target volume and inverse compressibility to 25 and 5, respectively. Generically, we use the `start` function to define model initial conditions. The `step` function increases the target volumes of all cells by 0.001 after the tenth MCS, a typical way to implement cell growth in CC3D. The `finish` function prints the cell volumes at the end of the simulation.

```
01  class Example(SteppableBasePy):
02    def __init__(self,_simulator,_frequency=1):
03      SteppableBasePy.__init__(self,_simulator,_frequency)
04
05    def start(self):
06      print “Called at the beginning of the simulation”
07      for cell in self.cellList:
08        cell.targetVolume=25
09        cell.lambdaVolume=5
10
11    def step(self,mcs):
12      print “Called every MCS”
13      if (mcs>10):
14        for cell in self.cellList:
15          cell.targetVolume+=0.001
16
17    def finish(self):
18      print “Called at the end of the simulation”
19      for cell in self.cellList:
20        print “cell volume = ”, cell.volume
```

Listing 8 *Sample CC3D steppable class.*

`start`, `step` and `finish` functions have default implementations in the base class `SteppableBasePy`. Therefore we only need to provide definition of those functions which we want to override. In addition, we can add our own functions to the class. The next section uses Python scripting to build a complex CompuCell3D model.

6.2 General structure of CC3D Python scripts

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 9 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()

#Steppable registration
from CustomSteppables import CustomSteppable
customSteppableInstance= CustomSteppable (sim,_frequency=100)
steppableRegistry.registerSteppable(customSteppableInstance)

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

Listing 9. 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()

#Steppable registration
from CustomSteppables import CustomSteppable
customSteppableInstance= CustomSteppable (sim,_frequency=100)
steppableRegistry.registerSteppable(customSteppableInstance)

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

Here we show example of how to instantiate and register a steppable (`CustomSteppable`). `CustomSteppable` is stored in file CustomSteppables.py and we import its content by typing:

```
from CustomSteppables import CustomSteppable
```

When Twedit++ generates simulation scripts the above script is generated automatically and it rarely needs any modifications.

In the next section, we will explain how to modify autogenerated steppable to implement dynamically changing cell properties.

6.3 Cell-Type-Oscillator Simulation

Suppose that we would like to add a caricature of oscillatory gene expression to our cell-sorting simulation so that cells exchange types every 100 MCS. All we have to do in is to then is to generate in Twedit++ cellsorting simulation but making sure that on the first page of the wizard screen we choose `Python+XML` option. As a result we will get simulation scripts (CC3DML and Python) which we will modify to create this simple new simulation. We will implement the changes of cell types using a Python steppable, since it occurs at intervals of 100 MCS. The skeleton of the steppable is autogenerated by Twedit++ and *Listing 10* shows the modification which are needed to turn boiler-plate code into functional simulation:

```
from PySteppables import *
import CompuCell
import sys
class CellTypeOscillatorSteppable(SteppableBasePy):

    def __init__(self,_simulator,_frequency=10):
        SteppableBasePy.__init__(self,_simulator,_frequency)
    def start(self):
        print "START FUNCTION"
    def step(self,mcs):
          for cell in self.cellList:
              if cell.type==self.CONDENSING:
                  cell.type=self.NONCONDENSING
              elif cell.type==self.NONCONDENSING:
                  cell.type=self.CONDENSING
    def finish(self):
        print "FINISH FUNCTION"
```

Listing 10. Autogenerated Python steppable with modification necessary to implement oscillatory cell type switching. The changes to to script are indicated in **bold** font.

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. The class definition :

```
class CellTypeOscillatorSteppable (SteppableBasePy):
```

causes the `TypeSwitcherSteppable` to inherit components of the `SteppableBasePy` class. `SteppableBasePy` contains default definitions of the `start(self)`, `step(self,mcs)` and `finish(self)` functions as well as many othr convenience objects

such as e.g. `cellList`. Here we implement all three functions but often we may skip either one of them. Inheritance reduces the length of the user-written Python code and ensures that the CellTypeOscillatorSteppable 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 `SteppableBasePy` *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)
```

Here we call the constructor for the inheritance class, `SteppableBasePy`, as required by Python. This instantiates many convenience objects which are available from within our `CellTypeOscillatorSteppable` class. For example we can write `self.simulator` to reference to `simulator` object, passed from the main script or by typing `self.cellList` we access a reference to *cell inventory* managed by CC3D kernel. 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. Cell inventory 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 `CellTypeOscillatorSteppable` 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==self.CONDENSING:
                cell.type=self.NONCONDENSING
            elif cell.type==self.NONCONDENSING:
                cell.type=self.CONDENSING
```

Here we use the cell inventory to iterate over all cells in the simulation and reassign their cell types between `self.CONDENSING` and `self.NONCONDENSING`. are constants which are equal to id's of cell types in our simulation (recall that cellsording had two cell types `Condensing` and `NonCondensing`). The names of the constants can be easily created from type name by capitalizing all the letters and prepending them with `self.` – for example

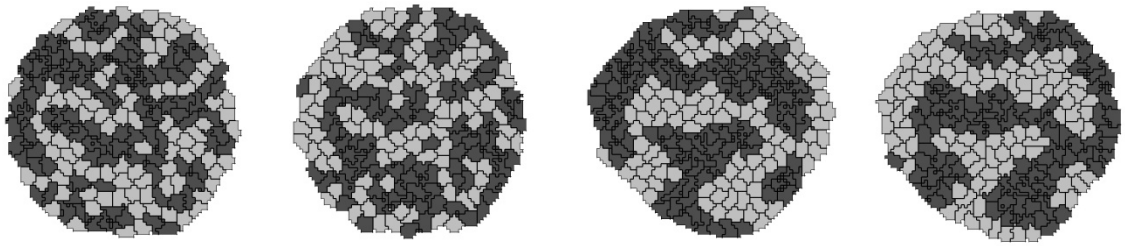
```
NonCondensing → self.NONCONDENSING
```

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

```
from CellTypeOscillatorSteppables import CellTypeOscillatorSteppable
steppableInstance=CellTypeOscillatorSteppable(sim,_frequency=100)
steppableRegistry.registerSteppable(steppableInstance)
```

CompuCell3D will not run unregistered steppables.

Figure 21 shows snapshots of the cell-type-oscillator simulation.



t=90 MCS *t*=110 MCS *t*=1490 MCS *t*=1510 MCS

Figure 21. Results of the Python cell-type-oscillator simulation using the `TypeSwitcherSteppable` steppable implemented in Listing 10 in conjunction with the CC3DML cell-sorting simulation. 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. The easiest and fastes way to convert CC3DML to equivalent Python syntax is to use Twedit++. By right-clicking on a CC3MDL file tag in the CC3D Project panel we get access to context menu "Convert XML to Python". By choosing it , with one click we convert entire CC3DML to Python syntax as shown

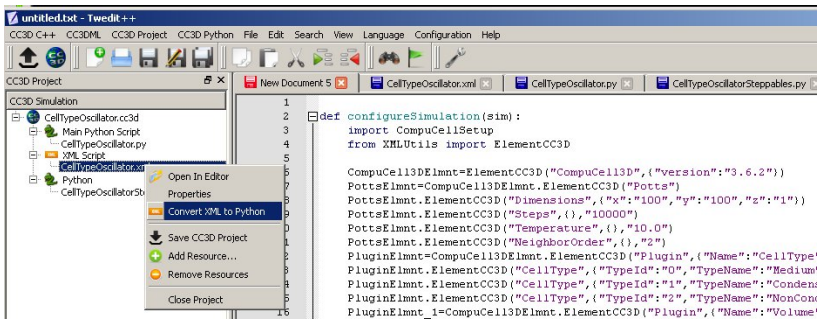


Figure 22 Converting CC3DML to Python syntax.

Listing 11 shows the cell-type-oscillator simulation written entirely in Python, with changes to Listing 10 shown in **bold**.

```
def configureSimulation(sim):
    import CompuCellSetup
    from XMLUtils import ElementCC3D

    CompuCell3DElmnt=ElementCC3D("CompuCell3D",{"version":"3.6.2"})

    PottsElmnt=CompuCell3DElmnt.ElementCC3D("Potts")

    # Basic properties of CPM (GGH) algorithm
    PottsElmnt.ElementCC3D("Dimensions",{"x":"100","y":"100","z":"1"})
    PottsElmnt.ElementCC3D("Steps",{},"10000")
    PottsElmnt.ElementCC3D("Temperature",{},"10.0")
    PottsElmnt.ElementCC3D("NeighborOrder",{},"2")
```

```
    PluginElmnt=CompuCell3DElmnt.ElementCC3D("Plugin",{"Name":"CellType"})

    # Listing all cell types in the simulation
    PluginElmnt.ElementCC3D("CellType",{"TypeId":"0","TypeName":"Medium"})
    PluginElmnt.ElementCC3D("CellType",{"TypeId":"1","TypeName":"Condensing"})
    PluginElmnt.ElementCC3D("CellType",{"TypeId":"2","TypeName":"NonCondensing"})

    PluginElmnt_1=CompuCell3DElmnt.ElementCC3D("Plugin",{"Name":"Volume"})
    PluginElmnt_1.ElementCC3D("VolumeEnergyParameters",\
    {"CellType":"Condensing","LambdaVolume":"2.0","TargetVolume":"25"})
    PluginElmnt_1.ElementCC3D("VolumeEnergyParameters",\
    {"CellType":"NonCondensing","LambdaVolume":"2.0","TargetVolume":"25"})

    PluginElmnt_2=CompuCell3DElmnt.ElementCC3D("Plugin",{"Name":"NeighborTracker"})

    # Module tracking center of mass of each cell

    PluginElmnt_3=CompuCell3DElmnt.ElementCC3D("Plugin",{"Name":"Contact"})
    # Specification of adhesion energies
    PluginElmnt_3.ElementCC3D("Energy",{"Type1":"Medium","Type2":"Medium"},"10.0")
    PluginElmnt_3.ElementCC3D("Energy",{"Type1":"Medium","Type2":"Condensing"},"16.0")
    PluginElmnt_3.ElementCC3D("Energy",{"Type1":"Medium","Type2":"NonCondensing"},"16.0")
    PluginElmnt_3.ElementCC3D("Energy",{"Type1":"Condensing","Type2":"Condensing"},"2.0")
    PluginElmnt_3.ElementCC3D("Energy",\
    {"Type1":"Condensing","Type2":"NonCondensing"},"11.0")
    PluginElmnt_3.ElementCC3D("Energy",\
    {"Type1":"NonCondensing","Type2":"NonCondensing"},"15.0")
    PluginElmnt_3.ElementCC3D("NeighborOrder",{},"2")

    SteppableElmnt=CompuCell3DElmnt.ElementCC3D("Steppable",{"Type":"BlobInitializer"})

    # Initial layout of cells in the form of spherical (circular in 2D) blob
    RegionElmnt=SteppableElmnt.ElementCC3D("Region")
    RegionElmnt.ElementCC3D("Center",{"x":"50","y":"50","z":"0"})
    RegionElmnt.ElementCC3D("Radius",{},"40")
    RegionElmnt.ElementCC3D("Gap",{},"0")
    RegionElmnt.ElementCC3D("Width",{},"5")
    RegionElmnt.ElementCC3D("Types",{},"Condensing,NonCondensing")

    CompuCellSetup.setSimulationXMLDescription(CompuCell3DElmnt)

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)

# add extra attributes here

CompuCellSetup.initializeSimulationObjects(sim,simthread)
# Definitions of additional Python-managed fields go here

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

from CellTypeOscillatorSteppables import CellTypeOscillatorSteppable
steppableInstance=CellTypeOscillatorSteppable(sim,_frequency=100)
steppableRegistry.registerSteppable(steppableInstance)

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

Listing 11. Stand-alone Python cell-type-oscillator script containing an initial section that replaces the CC3DML from Listing 1.

The `configureSimulation` function replaces the CC3DML file from Listing 1. 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 conversion from XML to Python follows simple algorithm for nested data structures and is explained in detail in Python Scripting Manual. In essentially all cases users never need to bother with the details of this algorithm as Twedit++ does all the work behind the scenes. Great advantage of using Python-only simulations is that parameters appearing in the configureSim function (this function replaces the CC3DML script) can be Python variables or Python expressions and this may help users in building simulation codes which are easit to maintain.

6.4 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**)*. However the implmenetation of those behaviors in CC3D is relatively straightforward as we will show.

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

$$\frac{\partial [FGF](\vec{i})}{\partial t} = 0.10\nabla^2 [FGF](\vec{i}) + 0.05\,\delta\left(\tau\left(\sigma(\vec{i})\right), \texttt{NonCondensing}\right), \qquad (12)$$

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

$$\frac{dV_t(\sigma)}{dt} = 0.01[FGF](\vec{x}_\sigma). \qquad (13)$$

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.

We start constructing our simulation by invoking CC3D Simulation Wizard from Twedit++ and naming simulation `cellsort_2D_field`. We set lattice dimension to 200x200x1, number of steps to 300 and NeighborOrder to 3. We then declare 3 cell types `Condensing`, `NonCondensing` and `DifferentiatedCondensing`. On the Chemical Fields

page we declare one diffusant - FGF and choose DiffusionSolverFE to evolve the field. On the Cell Properties and Behaviors page we check Contact, VolumeLocalFlex and Mitosis and complete wizard workflow by clicking next until we get to final page which generates simulation template.

First thing we do with autogenerated code we edit CC3DML file as follows (changes are shown in **bold**):

```
<CompuCell3D version="3.6.2">

   <Potts>
      <Dimensions x="200" y="200" z="1"/>
      <Steps>3000</Steps>
      <Temperature>10.0</Temperature>
      <NeighborOrder>2</NeighborOrder>
   </Potts>

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

   <Plugin Name="Volume"/>

   <Plugin Name="CenterOfMass"/>

  <Plugin Name="Contact">
   <Energy Type1="Medium" Type2="Medium">0.0</Energy>
   <Energy Type1="Medium" Type2="Condensing">16.0</Energy>
   <Energy Type1="Medium" Type2="NonCondensing">16.0</Energy>
   <Energy Type1="Medium" Type2="DifferentiatedCondensing">16.0</Energy>
   <Energy Type1="Condensing" Type2="Condensing">2.0</Energy>
   <Energy Type1="Condensing" Type2="NonCondensing">11.0</Energy>
   <Energy Type1="Condensing" Type2="DifferentiatedCondensing">2.0</Energy>
   <Energy Type1="NonCondensing" Type2="NonCondensing">15.0</Energy>
   <Energy Type1="NonCondensing" Type2="DifferentiatedCondensing">11.0</Energy>
   <Energy Type1="DifferentiatedCondensing" Type2="DifferentiatedCondensing">2.0</Energy>
   <NeighborOrder>2</NeighborOrder>
  </Plugin>

   <Steppable Type="DiffusionSolverFE">
      <DiffusionField>
         <DiffusionData>
            <FieldName>FGF</FieldName>
            <GlobalDiffusionConstant>0.1</GlobalDiffusionConstant>
            <GlobalDecayConstant>5e-05</GlobalDecayConstant>
         </DiffusionData>
         <SecretionData>
            <Secretion Type="NonCondensing">0.05</Secretion>
         </SecretionData>
         <BoundaryConditions>
            <Plane Axis="X">
               <ConstantDerivative PlanePosition="Min" Value="0.0"/>
               <ConstantDerivative PlanePosition="Max" Value="0.0"/>
            </Plane>
            <Plane Axis="Y">
               <ConstantDerivative PlanePosition="Min" Value="0.0"/>
               <ConstantDerivative PlanePosition="Max" Value="0.0"/>
            </Plane>
         </BoundaryConditions>
      </DiffusionField>
   </Steppable>

   <Steppable Type="BlobInitializer">
      <Region>
```

```
        <Center x="100" y="100" z="0"/>
        <Radius>40</Radius>
        <Gap>0</Gap>
        <Width>5</Width>
        <Types>Condensing,NonCondensing</Types>
      </Region>
   </Steppable>
</CompuCell3D>
```

Listing 12. CC3DML code for the diffusing-field-based cell-growth simulation.Changes from the autogenerated template are shown in **bold** font.

The CC3DML code is a slightly extended version of the cell-sorting code in Listing 1 plus the `DiffusionSolverFE` discussed in the bacterium-and-macrophage simulation. Note we have specified no-flux boundary conditions even though we could have completely removed this section as no-flux boundary conditions are default choice in CC3D.

The initial cell-lattice does not contain any `CondensingDifferentiated` cells. These cells appear only as the result of mitosis. We use the `VolumeLocalFlex` plugin (specifying `<Plugin Name="Volume"/>` is sufficient) to allow the target volume to vary individually for each cell, allowing cell growth. 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, `ConstraintInitializerSteppable` 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 – we will clal this steppable every 10 MCS as opposed to 100 MCS as generated by Twedit++ . 2) A steppable, `MitosisSteppable`, that divides cell once it reaches a threshold volume and then adjusts the parameters of the resulting parent and daughter cells. This steppable also appends information about the time and type of cell division to a list attached to each cell. We call this steppable every MCS 3) A steppable, `MitosisDataPrinterSteppable`, that prints the cell-division information from the lists attached to each cell. We call this steppable every 100 MCS. 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_fieldSteppables.py'.

Listing 13 shows the main Python script for the diffusing-field-based cell-growth simulation.

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

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

import CompuCellSetup
```

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

# add extra attributes here
pyAttributeAdder,dictAdder=CompuCellSetup.attachListToCells(sim)

CompuCellSetup.initializeSimulationObjects(sim,simthread)
# Definitions of additional Python-managed fields go here

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

from cellsort_2D_fieldSteppables import ConstraintInitializerSteppable
ConstraintInitializerSteppableInstance=ConstraintInitializerSteppable(sim,_frequency=10)
steppableRegistry.registerSteppable(ConstraintInitializerSteppableInstance)

from cellsort_2D_fieldSteppables import MitosisSteppable
MitosisSteppableInstance=MitosisSteppable(sim,_frequency=1)
steppableRegistry.registerSteppable(MitosisSteppableInstance)

from cellsort_2D_fieldSteppables import MitosisDataPrinterSteppable
instanceOfMitosisDataPrinterSteppable=MitosisDataPrinterSteppable\
(_simulator=sim,_frequency=100)
steppableRegistry.registerSteppable(instanceOfMitosisDataPrinterSteppable)

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

Listing 13. Main Python script for the diffusing-field-based cell-growth simulation. Changes to the template code shown in **bold**. Attaching Python dictionaries or lists to cells can be accomplished from Twedit++ by calling CC3DPython->CellAttributes->Add Dictionary to Cells on the main Python script.

As compared to 'vanilla' CC3D main Python script Listing 13 contains new line

```
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. We can also attach Python dictionary by using:

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

Since location of this statement is important (you cannot put it just anywhere in the script) it is best to use Twedit++ by calling CC3DPython->CellAttributes->Add Dictionary to Cells on the main Python script.

Moving on to the Python modules, we can see that most of the code looks a lot like other steppable file found in e.g. Listing 10. In addition to steppables, we created one non-steppable MitosisData used to register mitotic events.

```
from PySteppables import *
import CompuCell
import sys
from random import random
from PySteppablesExamples import MitosisSteppableBase

class ConstraintInitializerSteppable(SteppableBasePy):
    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator,_frequency)
    def start(self):
        for cell in self.cellList:
```

```
            cell.targetVolume=25
            cell.lambdaVolume=2.0

    def step(self,mcs):
        field=CompuCell.getConcentrationField(self.simulator,"FGF")
        comPt=CompuCell.Point3D()
        for cell in self.cellList:
            if cell.type==self.CONDENSING: #Condensing cell
                comPt.x=int(round(cell.xCOM))
                comPt.y=int(round(cell.yCOM))
                comPt.z=int(round(cell.zCOM))
                concentration=field.get(comPt) # get concentration at comPt
                cell.targetVolume+=0.1*concentration # increase cell's target volume

#MItosis data has to have base class "object" otherwise if cell will be deleted CC3D may
crash due to improper garbage collection
class MitosisData(object):
    def __init__(self, _MCS=-1, _parentId=-1, _parentType=-1,\
    _offspringId=-1, _offspringType=-1):
        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)


class MitosisSteppable(MitosisSteppableBase):
    def __init__(self,_simulator,_frequency=1):
        MitosisSteppableBase.__init__(self,_simulator, _frequency)

    def step(self,mcs):
        cells_to_divide=[]
        for cell in self.cellList:
            if cell.volume>50:

                cells_to_divide.append(cell)

        for cell in cells_to_divide:
            self.divideCellRandomOrientation(cell)

    def updateAttributes(self):
        parentCell=self.mitosisSteppable.parentCell
        childCell=self.mitosisSteppable.childCell

        parentCell.targetVolume/=2.0
        childCell.targetVolume=parentCell.targetVolume
        childCell.lambdaVolume=parentCell.lambdaVolume

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

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

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


class MitosisDataPrinterSteppable(SteppableBasePy):
    def __init__(self,_simulator,_frequency=10):
        SteppableBasePy.__init__(self,_simulator,_frequency)
```

```
    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 14. Python steppable code stored in 'cellsort_2D_fieldSteppables.py' for the diffusing-field-based cell-growth simulation.

Let us first consider `ConstraintInitializerSteppable`. The `start(self)` function executes only once, at the beginning of the simulation. It iterates over each cell 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 `DiffusionSolverFE` 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` equal to `self.CONDENSING`, we calculate its centroid, round it to nearest integers:

```
comPt.x=int(round(cell.xCOM))
comPt.y=int(round(cell.yCOM))
comPt.z=int(round(cell.zCOM))
```

and retrieve the FGF concentration at that point:

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

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

```
cell.targetVolume+=0.01*concentration
```

We must include the `CenterOfMass` plugin in the CC3DML code. Otherwise the centroid (`cell.xCOM, cell.yCOM, cell.zCOM`) will have the default value (0,0,0). By default Twedit++ includes CenterOfMass plugin in the autogenerated code.
The `MitosisSteppable` 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.

`MitosisSteppable` inherits the content of the `MitosisSteppableBase` class. `MitosisSteppableBase` contains several convenience functions associated with Mitosis even (many of those functions are implemented in C++) and also inherits `SteppableBase` providing many convenient features of this class. At each MCS (or with user specified frequency) we scan cells and decide which cells are to undergo mitosis:

```
    def step(self,mcs):
        cells_to_divide=[]
        for cell in self.cellList:
            if cell.volume>50:

                cells_to_divide.append(cell)

        for cell in cells_to_divide:
            self.divideCellRandomOrientation(cell)
```

if a cell has volume greater than 50 we attach this cell to the list of cells to be divided (`cells_to_divide`). Subsequentyl we walk over all the cells stored in `cells_to_divide` and split them into two cells. Each function from Mitosis Steppable which divides cells calls (immediately after the division takes place) `updateAttributes(self)function.` Therefore we also need to reimplement the function `updateAttributes(self)`, 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 `DifferentiatedCondensing`:

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

Note that we also have to import `random()` function from `random` module to ensure that Python finds it:

```
from random import random
```

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)
```

The `MitosisData` class, which stores the data on the cell division that we append to the cells' attribute lists after each cell division. 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.

The `MitosisDataPrinterSteppable` steppable prints the mitosis data to the user's screen and is shown here mainly to demonstrate that certain plugins inCC3D do not always have to adjust cell parameters – they can access then, print them to the screen or even do something which has little to do with simulation itself (e.g. output file for post processing). To autogenerate steppable for the existing simulation we use Twedit++ CC3D project manager as shown on **Figure *23***.

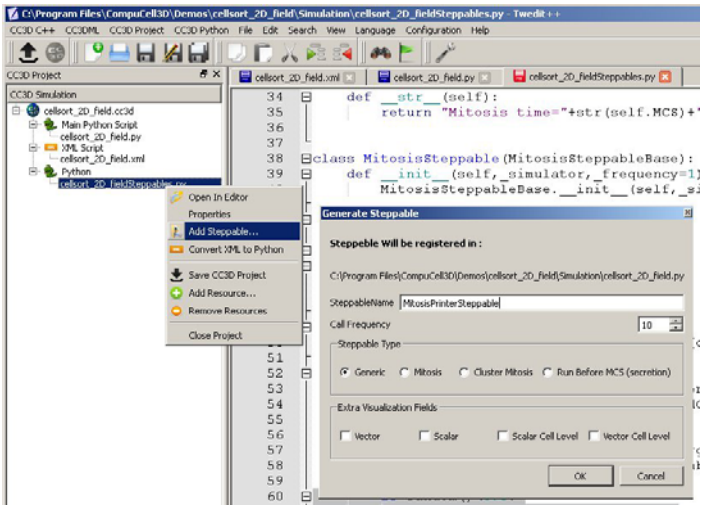
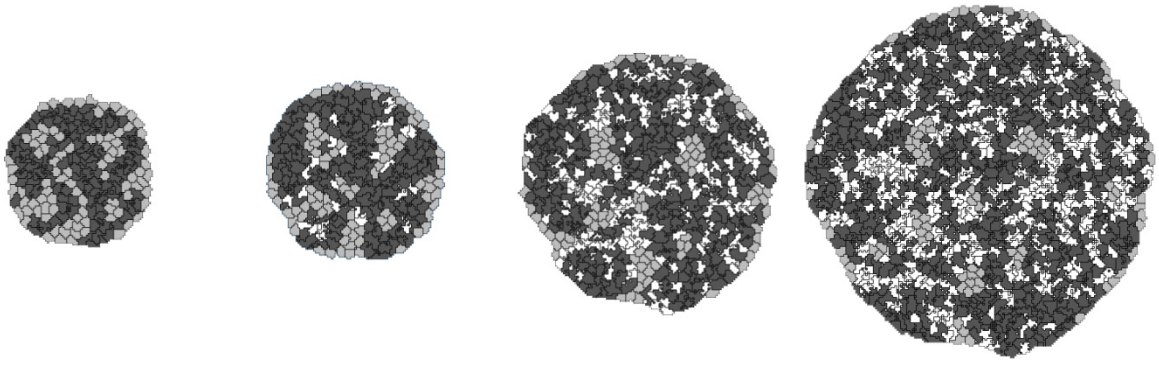


Figure 23 To autogenerate steppable using Twedit++, right-click on steppable file in the CC3D Project Manager panel and choose `Add Steppable…` menu optioin which pops-up a simple dialog window.

Within the `step(self,mcs)` function of the `MitosisDataPrinterSteppable`, we iterate over each cell 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
```

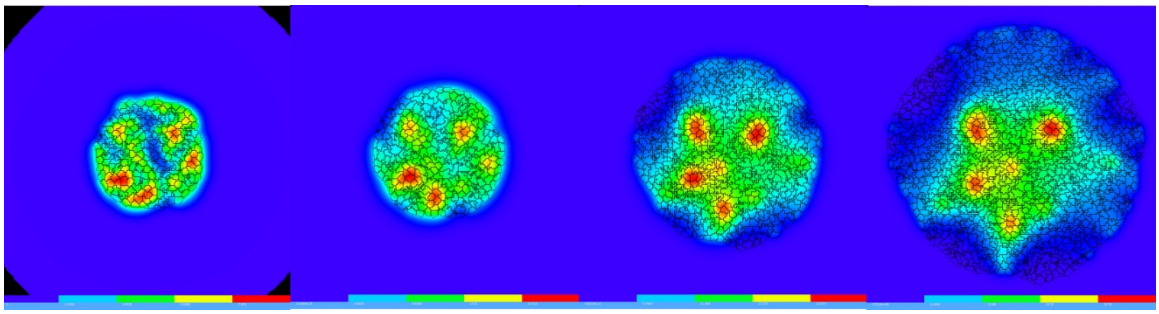
Figure 24 and Figure 25 show snapshots of the diffusing-field-based cell-growth simulation. Figure 26 shows a sample screen output of the cell-division history.



t=200 MCS *t*=600 MCS *t*=1200 MCS *t*=1800 MCS

Figure 24. Snapshots of the diffusing-field-based cell-growth simulation obtained by running the CC3DML file in Listing 12 in conjunction with the Python file in Listing 13. 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.



t=200 MCS *t*=600 MCS *t*=1200 MCS *t*=1800 MCS

Figure 25. Snapshots of FGF concentration in the diffusing-field-based cell-growth simulation obtained by running the CC3DML file in Listing 12 in conjunction with the Python files in Listing 13, Listing 14**.** The bars at the bottom of the field images show the concentration scales (blue, low concentration; red, high concentration).

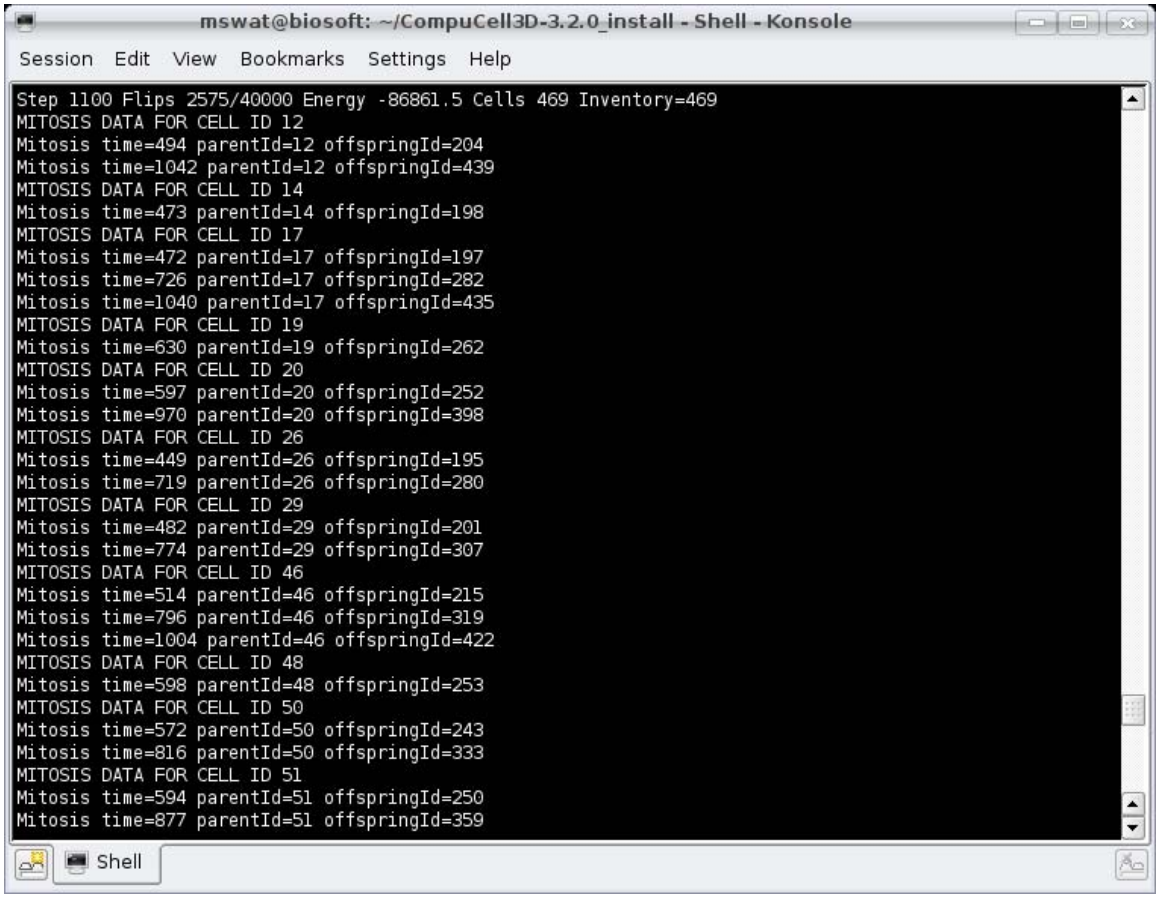


Figure 26. Sample output from the MitosisDataPrinterSteppable steppable in Listing 14

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.

6.5 Three-Dimensional Vascular Tumor Growth Model

The development of a primary solid tumor starts from a single cell that proliferates in an inappropriate manner, dividing repeatedly to form a cluster of tumor cells. Nutrient and waste diffusion limits the diameter of such *avascular tumor spheroids* to about 1 mm. The central region of the growing spheroid becomes necrotic, with a surrounding layer of cells whose hypoxia triggers VEGF-mediated signaling events that initiate tumor neovascularization by promoting growth and extension (*neoangiogenesis*) of nearby blood vessels. Vascularized tumors are able to grow much larger than avascular spheroids and are more likely to metastasize.

Here, we present a simplified 3D model of a generic vascular tumor which can be easily extended to describe specific vascular tumor types and host tissues. We begin with a cluster of proliferating tumor cells, `P`, and normal vasculature. Initially, tumor cells proliferate as they take up diffusing glucose from the field, *`GLU`*, which the pre-existing vasculature supplies (in this model we neglect possible changes in concentration along the blood vessels in the direction of flow and set the secretion parameters uniform over all blood-vessel surfaces). We assume that the tumor cells (both in the initial cluster and later) are always hypoxic and secrete a long-diffusing isoform of VEGF-A, *`L_VEGF`*. When *`GLU`* drops below a threshold, tumor cells become necrotic, gradually shrink and finally disappear. The initial tumor cluster grows and reaches a maximum diameter characteristic of an avascular tumor spheroid. To reduce execution time in our demonstration, we choose our model parameters so that the maximum spheroid diameter will be about 10 times smaller than in experiment. A few pre-selected neovascular endothelial cells, `NV`, in the pre-existing vasculature respond both by chemotaxing towards higher concentrations of pro-angiogenic factors and by forming new blood vessels via neoangiogenesis. The tumor-induced vasculature increases the growth rate of the resulting vascularized solid tumor compared to an avascular tumor, allowing the tumor to grow beyond the spheroid's maximum diameter. Despite our rescaling of the tumor size, the model produces a range of biologically reasonable morphologies that allow study of how tumor-induced angiogenesis affects the growth rate, size and morphology of tumors.

We use the basic angiogenesis simulation from the previous section to simulate both pre-existing vasculature and tumor-induced angiogenesis, adding a set of finite-element links between the endothelial cells to model the strong junctions that form between endothelial cells *in vivo*. We denote the short-diffusing isoform of VEGF-A, *S_VEGF*. Both endothelial cells and neovascular endothelial cells chemotax up gradients of `S_VEGF`, but only neovascular endothelial cells chemotax up gradients of `L_VEGF`.

In the Simulation Wizard we name the model `TumorVascularization`, set the cell- and field-lattice dimensions to 50×50×80, the membrane fluctuation amplitude to 20, the pixel-copy range to 3, number of MCS to 10000 and choose `UniformInitializer` to produce the initial tumor and vascular cells, since it automatically creates a mixture of the cell types. We specify four cell types: `P`: proliferating tumor cells, `N`: necrotic cells, `EC`: endothelial cells and `NV`: neovascular endothelial cells.

On the `Chemical Fields` page we create the `S_VEGF` and `L_VEGF` fields and select `FlexibleDiffusionSolverFE` for both from the `Solver` pull-down list. We also check `Enable multiple calls of PDE solvers` to work around the numerical instabilities of the PDE solvers for large diffusion constants.

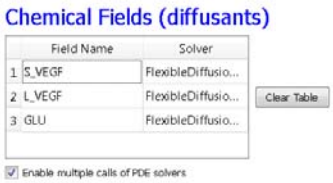


Figure 27 Specification of vascular tumor chemical fields in Simulation Wizard.

On the `Cell Behavior and Properties` page we select both the `Contact` and `FocalPointPlasticity` modules from the `Adhesion` group, and add `Chemotaxis`, `Growth` and `Mitosis`, `Volume Constraint` and `Global Connectivity` by checking the appropriate boxes. We also track the `Center-of-Mass` (to access field concentrations) and `Cell Neighbors` (to implement contact-inhibited growth). Unlike in our angiogenesis simulation, we will implement secretion as a part of the `FlexibleDiffusionSolverFE` syntax.

Figure 28 Specification of vascular tumor cell behaviors in Simulation Wizard.

In the `Chemotaxis` page, for each cell-type/chemical-field pair we click the `Add Entry` button to add the relevant chemotaxis information, *e.g.* we select `S_VEGF` from the `Field` pull-down list and `EC` and `NV` from the `cell-type` list and set `Lambda` to 5000. To enable contact inhibition of `EC` and `NV` chemotaxis we select `Medium` from the pull-down menu next to the `Chemotax Towards` button and click the button to add `Medium` to the list. We repeat this process for the `T` and `N` cell types, so that `NV` cells chemotax up gradients of `L_VEGF`. We then proceed to the final `Simulation Wizard` page.

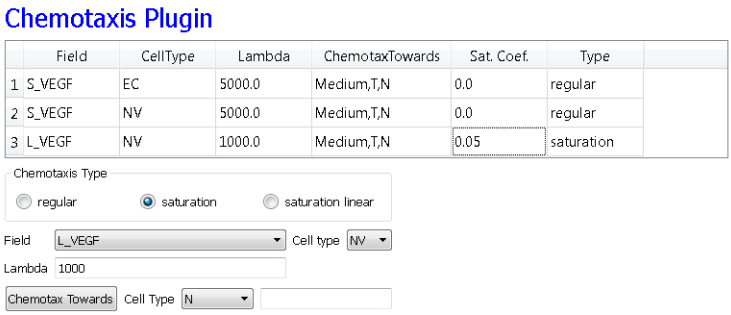


Figure 29 Specification of vascular tumor chemotaxis properties in Simulation Wizard.

Twedit++ generates 3 simulation files – a CC3DML file specifying the energy terms, diffusion solvers and initial cell layout, a main Python file which loads the CC3DMLfile, sets up the CompuCell environment and executes the Python steppables and a Python steppables file. The main Python file is typically constructed by modifying the standard template in Listing 15. Lines 1-12 set up the CC3D simulation environment and load the simulation. Lines 14-20 create instances of two steppables – `MitosisSteppable` and `VolumeParamSteppable` – and register them with the CC3D kernel. Line 22 starts the main CC3D loop, which executes Monte Carlo Steps and periodically calls the steppables.

```
01   import sys
02   from os import environ
03   import string
04   sys.path.append(environ["PYTHON_MODULE_PATH"])
05
06   import CompuCellSetup
07   sim,simthread = CompuCellSetup.getCoreSimulationObjects()
08   CompuCellSetup.initializeSimulationObjects(sim,simthread)
09   import CompuCell
10
11   from PySteppables import SteppableRegistry
12   steppableRegistry=SteppableRegistry()
13
14   from VascularTumorSteppables import MitosisSteppable
15   mitosisSteppable=MitosisSteppable(sim,1)
16   steppableRegistry.registerSteppable(mitosisSteppable)
17
18   from VascularTumorSteppables import VolumeParamSteppable
19   volumeParamSteppable=VolumeParamSteppable(sim,1)
20   steppableRegistry.registerSteppable(volumeParamSteppable)
21
22   CompuCellSetup.mainLoop(sim,simthread,steppableRegistry)
```

Listing 15 The Main Python script initializes the vascular tumor simulation and runs the main simulation loop.

Next, we edit the draft autogenerated simulation CC3DML file in Listing 16.

```
01   <CompuCell3D>
02   <Potts>
03    <Dimensions x="50" y="50" z="80"/>
04    <Steps>100000</Steps>
05    <Temperature>20</Temperature>
06    <Boundary_x>Periodic</Boundary_x>
```

```
 <Boundary_y>Periodic</Boundary_y>
 <Boundary_z>Periodic</Boundary_z>
 <RandomSeed>313</RandomSeed>
 <NeighborOrder>3</NeighborOrder>
</Potts>

<Plugin Name="CellType">
 <CellType TypeName="Medium" TypeId="0"/>
 <CellType TypeName="P" TypeId="1"/>
 <CellType TypeName="N" TypeId="2"/>
 <CellType TypeName="EC" TypeId="3"/>
 <CellType TypeName="NV" TypeId="4"/>
</Plugin>

<Plugin Name="Chemotaxis">
 <ChemicalField Source="FlexibleDiffusionSolverFE" Name="S_VEGF">
  <ChemotaxisByType Type="NV" Lambda="5000" ChemotactTowards="Medium,P,N"/>
 </ChemicalField>
 <ChemicalField Source="FlexibleDiffusionSolverFE" Name="L_VEGF">
  <ChemotaxisByType Type="NV" Lambda="1000"
    ChemotactTowards="Medium,P,N" SaturationCoef="0.05"/>
 </ChemicalField>
 <ChemicalField Source="FlexibleDiffusionSolverFE" Name="S_VEGF">
  <ChemotaxisByType Type="EC" Lambda="5000" ChemotactTowards="Medium,P,N"/>
 </ChemicalField>
</Plugin>

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

<Plugin Name="Contact">
 <Energy Type1="Medium" Type2="Medium">0</Energy>
 <Energy Type1="P" Type2="Medium">10</Energy>
 <Energy Type1="P" Type2="P">8</Energy>
 <Energy Type1="N" Type2="Medium">15</Energy>
 <Energy Type1="N" Type2="P">8</Energy>
 <Energy Type1="N" Type2="N">3</Energy>
 <Energy Type1="EC" Type2="Medium">12</Energy>
 <Energy Type1="EC" Type2="P">30</Energy>
 <Energy Type1="EC" Type2="N">30</Energy>
 <Energy Type1="EC" Type2="EC">5</Energy>
 <Energy Type1="NV" Type2="Medium">12</Energy>
 <Energy Type1="NV" Type2="P">30</Energy>
 <Energy Type1="NV" Type2="N">30</Energy>
 <Energy Type1="NV" Type2="EC">5</Energy>
 <Energy Type1="NV" Type2="NV">5</Energy>
 <NeighborOrder>4</NeighborOrder>
</Plugin>

<Plugin Name="VolumeLocalFlex"/>

<Plugin Name="FocalPointPlasticity">
 <Parameters Type1="EC" Type2="NV">
  <Lambda>50.0</Lambda>
  <ActivationEnergy>-100.0</ActivationEnergy>
  <TargetDistance>5.0</TargetDistance>
  <MaxDistance>15.0</MaxDistance>
  <MaxNumberOfJunctions>2</MaxNumberOfJunctions>
 </Parameters>
 <Parameters Type1="EC" Type2="EC">
  <Lambda>400.0</Lambda>
  <ActivationEnergy>-100.0</ActivationEnergy>
  <TargetDistance>5.0</TargetDistance>
```

```
    <MaxDistance>15.0</MaxDistance>
    <MaxNumberOfJunctions>3</MaxNumberOfJunctions>
   </Parameters>
   <Parameters Type1="NV" Type2="NV">
    <Lambda>20.0</Lambda>
    <ActivationEnergy>-100.0</ActivationEnergy>
    <TargetDistance>5.0</TargetDistance>
    <MaxDistance>10.0</MaxDistance>
    <MaxNumberOfJunctions>2</MaxNumberOfJunctions>
   </Parameters>
   <NeighborOrder>1</NeighborOrder>
  </Plugin>

  <Plugin Name="ConnectivityGlobal">
   <Penalty Type="NV">10000</Penalty>
   <Penalty Type="EC">10000</Penalty>
  </Plugin>

  <Plugin Name="PDESolverCaller">
   <CallPDE PDESolverName="FlexibleDiffusionSolverFE" ExtraTimesPerMC="9"/>
  </Plugin>

  <Steppable Type="FlexibleDiffusionSolverFE">
   <!--endothelial-derived short diffusing VEGF isoform-->
   <DiffusionField>
    <DiffusionData>
     <FieldName>S_VEGF</FieldName>
     <ConcentrationFileName></ConcentrationFileName>
     <DiffusionConstant>0.016</DiffusionConstant>
     <DecayConstant>0.0016</DecayConstant>
     <DoNotDecayIn>EC</DoNotDecayIn>
     <DoNotDecayIn>NV</DoNotDecayIn>
    </DiffusionData>
    <SecretionData>
     <Secretion Type="NV">0.0013</Secretion>
     <Secretion Type="EC">0.0013</Secretion>
    </SecretionData>
   </DiffusionField>

   <!--tumor-derived long diffusing VEGF isoform-->
   <DiffusionField>
    <DiffusionData>
     <FieldName>L_VEGF</FieldName>
     <DiffusionConstant>0.16</DiffusionConstant>
     <DecayConstant>0.0016</DecayConstant>
    </DiffusionData>
    <SecretionData>
     <Secretion Type="P">0.001</Secretion>
     <Uptake Type="NV" MaxUptake="0.05" RelativeUptakeRate="0.5"/>
     <Uptake Type="EC" MaxUptake="0.05" RelativeUptakeRate="0.5"/>
    </SecretionData>
   </DiffusionField>

   <DiffusionField>
    <DiffusionData>
     <FieldName>GLU</FieldName>
     <ConcentrationFileName>GLU_300.dat</ConcentrationFileName>
     <DiffusionConstant>0.16</DiffusionConstant>
    </DiffusionData>
    <SecretionData>
     <Secretion Type="NV">0.4</Secretion>
     <Secretion Type="EC">0.8</Secretion>
     <Uptake Type="Medium" MaxUptake="0.0064" RelativeUptakeRate="0.1"/>
```

```
132      <Uptake Type="P" MaxUptake="0.1" RelativeUptakeRate="0.1"/>
133     </SecretionData>
134    </DiffusionField>
135   </Steppable>
136
137   <Steppable Type="UniformInitializer">
138    <Region>
139     <BoxMin x="0" y="24" z="16"/>
140     <BoxMax x="50" y="28" z="20"/>
141     <Width>4</Width>
142     <Types>EC</Types>
143    </Region>
144    <Region>
145     <BoxMin y="0" x="24" z="16"/>
146     <BoxMax y="50" x="28" z="20"/>
147     <Width>4</Width>
148     <Types>EC</Types>
149    </Region>
150    <Region>
151     <BoxMin x="10" y="24" z="16"/>
152     <BoxMax x="50" y="28" z="20"/>
153     <Width>4</Width>
154     <Gap>25</Gap>
155     <Types>NV</Types>
156    </Region>
157    <Region>
158     <BoxMin y="8" x="24" z="16"/>
159     <BoxMax y="50" x="28" z="20"/>
160     <Width>4</Width>
161     <Gap>25</Gap>
162     <Types>NV</Types>
163    </Region>
164    <Region>
165     <BoxMin x="26" y="26" z="40"/>
166     <BoxMax x="34" y="34" z="48"/>
167     <Width>2</Width>
168     <Types>P</Types>
169    </Region>
170   </Steppable>
171
172   </CompuCell3D>
```

Listing 16 CC3DML specification of the vascular tumor model's initial cell layout, PDE solvers and key cellular behaviors.

In Listing 16, in the `Contact` plugin (lines 36-53), we set J_{MM}=0, J_{EM}=12 and J_{EE}=5 (`M: Medium`, `E: EC`) and the `NeighborOrder` to 4. The `FocalPointPlasticity` plugin (lines 57-80) represents adhesion junctions by mechanically connecting the centers-of-mass of cells using a breakable linear spring (see (***104***)). `EC-EC` links are stronger than `EC-NV` links, which are, in turn, stronger than `NV-NV` links (see the CC3D manual for details). Since, the Simulation Wizard creates code to implement links between all cell-type pairs in the model, we must delete most of them, keeping only the links between `EC-EC`, `EC-NV` and `NV-NV` cell types.

We assume that `L_VEGF` diffuses 10 times faster than `S_VEGF`, so D_{L_VEGF}=0.42 μm^2/sec (1.6 voxel^2/MCS). This large diffusion constant would make the diffusion solver unstable. Therefore in the CC3DML file (Listing 16, lines 108-114), we set the values of the `<DiffusionConstant>` and `<DecayConstant>` tags of the `L_VEGF` field to 0.16 and 0.0016 respectively and use 9 extra calls per MCS to achieve a diffusion constant

equivalent to 1.6 (lines 87-89). We instruct P cells to secrete (line 116) into the L_VEGF field at a rate of 0.001 (3.85 pg (cell h)$^{-1}$=0.001 pg (voxel MCS)$^{-1}$). Both EC and NV absorb L_VEGF. To simulate this uptake, we use the <SecretionData> tag pair (lines 117-118).

Since the same diffusion solver will be called 10 times per MCS to solve S_VEGF, we must reduce the diffusion constant of S_VEGF by a factor of 10, setting the <DiffusionConstant> and <DecayConstant> tags of S_VEGF field to 0.016 and 0.0016 respectively. To prevent S_VEGF decay inside EC and NV cells we add <DoNotDecayIn>EC</DoNotDecayIn> and <DoNotDecayIn>NV</DoNotDecayIn> inside the <DiffusionData> tag pair (lines 99-100). We define S_VEGF to be secreted (lines 102-105) by both the EC and NV cells types at a rate of 0.013 per voxel per MCS (50 pg (cell h)$^{-1}$ = 0.013 pg (voxel MCS)$^{-1}$, compare to Leith and Michelson 1995).

The experimental glucose diffusion constant is about 600 μm^2/sec. We convert the glucose diffusion constant by multiplying by appropriate spatial and temporal conversion factors: 600 μm^2/sec×(voxel/4 μm)^2×(60 sec/MCS)=2250 voxel^2/MCS. To keep our simulation times short for the example we use a simulated glucose diffusion constant 1500 smaller, resulting in much steeper glucose gradients and smaller maximum tumor diameters. We could use the steady-state diffusion solver for the glucose field to be more realistic.

Experimental GLU uptake by P cells is ~ 0.3 μmol/g/min. We assume that stromal cells (represented here without individual cell boundaries by Medium) take up GLU at a slower rate of 0.1 μmol/g/min. A gram of tumor tissue has about 10^8 tumor cells, so the glucose uptake per tumor cell is 0.003 pmol/MCS/cell or about 0.1 fmol/MCS/voxel. We assume that (at homeostasis) the pre-existing vasculature supplies all the required GLU to Medium, which has a total mass of 1.28×10^{-5} grams and consumes GLU at a rate of 0.1 fmol/MCS/voxel, so the total GLU uptake (in the absence of a tumor) is 1.28 pmol/MCS. For this glucose to be supplied by 24 EC cells, their GLU secretion rate must be 0.8 fmol/MCS/voxel. We distribute the total GLU uptake (in the absence of a tumor) over all the Medium voxels, so the uptake rate is ~ 1.28 pmol/MCS/(~20000 Medium voxels)=6.4×10^{-3} fmol/MCS/voxel.

We specify the uptake of GLU by Medium and P cells in lines 131-132 and instruct NV and EC cells to secrete GLU at the rate 0.4 and 0.8 pg (voxel MCS)$^{-1}$ respectively (lines 129-130)

We use UniformInitializer (lines 137-170) to initialize the tumor-cell cluster and two crossing vascular cords. We also add two NV cells to each vascular cord, 25 pixels apart.

```
01  from PySteppables import *
02  from PySteppablesExamples import MitosisSteppableBase
03  import CompuCell
04  import sys
05  from random import uniform
06  import math
07
08  class VolumeParamSteppable(SteppableBasePy):
09    def __init__(self,_simulator,_frequency=1):
10      SteppableBasePy.__init__(self, _simulator,_frequency)
11      self.fieldL_VEGF = CompuCell.getConcentrationField('L_VEGF')
12      self.fieldGLU = CompuCell.getConcentrationField('GLU')
13
```

```
    def start(self):
      for cell in self.cellList:
        if (cell.type>=3):
          cell.targetVolume=64.0+10.0
          cell.lambdaVolume=20.0
        else:
          cell.targetVolume=32.0
          cell.lambdaVolume=20.0

    def step(self,mcs):
      pt=CompuCell.Point3D()
      for cell in self.cellList:
        if (cell.type==4): #Neovascular cells (NV)
          totalArea=0
          pt.x=int(round(cell.xCOM))
          pt.y=int(round(cell.yCOM))
          pt.z=int(round(cell.zCOM))
          VEGFconc=self.fieldL_VEGF.get(pt)
          cellNeighborList=self.getNeighborList(cell)
          for nsd in cellNeighborList:
            if (nsd.neighborAddress and nsd.neighborAddress.type>=3):
              totalArea+=nsd.commonSurfaceArea
          if (totalArea<45):
            cell.targetVolume+=2.0*VEGFconc/(0.01+VEGFconc)
        if (cell.type==1): #Proliferating Cells
          pt.x=int(round(cell.xCOM))
          pt.y=int(round(cell.yCOM))
          pt.z=int(round(cell.zCOM))
          gluConc=self.fieldGLU.get(pt)
          #Proliferating Cells become Necrotic when gluConc is low
          if (gluConc<0.001 and mcs>1000):
            cell.type=2
          else:
            cell.targetVolume+=0.022*gluConc/(0.05+gluConc)
        if cell.type==2: #Necrotic Cells
          cell.targetVolume-=0.1
          if cell.targetVolume<0.0:
            cell.targetVolume=0.0


class MitosisSteppable(MitosisSteppableBase):
  def __init__(self,_simulator,_frequency=1):
    MitosisSteppableBase.__init__(self,_simulator,_frequency)

  def step(self,mcs):
    cells_to_divide=[]
    for cell in self.cellList:
      if (cell.type==1 and cell.volume>64):
        cells_to_divide.append(cell)
      if (cell.type==4 and cell.volume>128):
        cells_to_divide.append(cell)
    for cell in cells_to_divide:
      self.divideCellRandomOrientation(cell)

  def updateAttributes(self):
    parentCell=self.mitosisSteppable.parentCell
    childCell=self.mitosisSteppable.childCell
    parentCell.targetVolume=parentCell.targetVolume/2
    parentCell.lambdaVolume=parentCell.lambdaVolume
    childCell.type=parentCell.type
    childCell.targetVolume=parentCell.targetVolume
    childCell.lambdaVolume=parentCell.lambdaVolume
```

Listing 17 Vascular tumor model Python steppables. The `VolumeParametersSteppable` adjusts the properties of the cells in response to simulation events and the `MitosisSteppable` implements cell division.

In the Python Steppable script in Listing 17, we set the initial target volume of both `EC` and `NV` cells to 74 (64+10) voxels and the initial target volume of tumor cells to 32 voxels (lines 14-21). All λ_{vol} are 20.0.

To model tumor cell growth, we increase the tumor cells' target volumes (lines 38-47) according to:

$$\frac{dV_t(\text{tumor})}{dt} = \frac{G_{\max} GLU(\vec{x})}{GLU(\vec{x}) + GLU_0}, \qquad (14)$$

where $GLU(\vec{x})$ is the `GLU` concentration at the cell's center-of-mass of and GLU_0 is the concentration at which the growth rate is half its maximum. We assume that the fastest cell cycle time is 24 hours, so $G_{\max}$ is 32 voxels/24 hours = 0.022 voxel/MCS.

To account for contact-inhibited growth of `NV` cells, when their common surface area with other `EC` and `NV` cells is less than a threshold, we increase their target volume according to:

$$\frac{dV_t(\text{NV})}{dt} = \frac{G_{\max}\ L\_VEGF(\vec{x})}{L\_VEGF(\vec{x}) + L\_VEGF_0}, \qquad (15)$$

where $L\_VEGF(\vec{x})$ is the concentration of `L_VEGF` at the cell's center-of-mass, L_VEGF_0 is the concentration at which the growth rate is half its maximum and $G_{\max}$ is the maximum growth rate for `NV` cells. We calculate the common surface area between each `NV` cell and its neighboring `NV` or `EC` cells in lines 32-35. If the common surface area is smaller than 45, then we increase its target volume (lines 36-37). When the volume of `NV` and `P` cells reaches a *doubling volume* (here, twice their initial target volumes), we divide them along a random axis, as shown in the `MitosisSteppable` (Listing 17, lines 54-75).

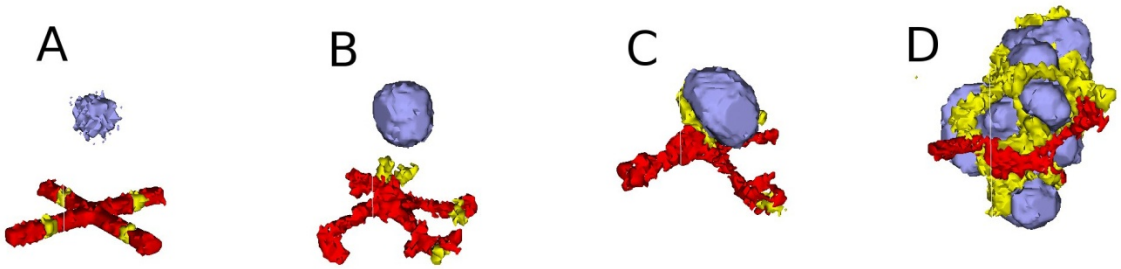


Figure 30 3D snapshots of the vascular tumor simulation taken at: A) 0 MCS , B) 500 MCS, C) 2000 MCS and D) 5000 MCS. Red and Yellow cells represent endothelial cells and neovascular endothelial cells, respectively.

With this simple model we can easily explore the effects of changes in cell adhesion, nutrient availability, cell motility, sensitivity to starvation or dosing with chemotherapeutics or antiangiogenics on the growth and morphology of the simulated tumor.

6.6 Subcellular Simulations Using BionetSolver

While our vascular tumor model showed how to change cell-level parameters like target volume, we have not yet linked macroscopic cell behaviors to intracellular molecular concentrations. Signaling, regulatory and metabolic pathways all steer the behaviors of biological cells by modulating their biochemical machinery. CC3D allows us to add and solve subcellular reaction-kinetic pathway models inside each generalized cell, specified using the SBML format (***105***), and to use such models (*e.g.* of their levels of gene expression) to control cell-level behaviors like adhesion or growth.
We can use the same SBML framework to implement classic physics-based pharmacokinetic (*PBPK*) models of supercellular chemical flows between organs or tissues. The ability to explicitly model such subcellular and supercellular pathways adds greatly to the range of hypotheses CC3D models can represent and test. In addition, the original formulation of SBML primarily focused on the behaviors of biochemical networks within a single cell, while real signaling networks often involve the coupling of networks between cells. BionetSolver supports such coupling, allowing exploration of the very complex feedback resulting from intercell interactions linking intracellular networks in an environment where the couplings change continuously due to cell growth, cell movement and changes in cell properties.
As an example of such interaction between signaling networks and cell behaviors, we will develop a multicellular implementation of Delta-Notch mutual inhibitory coupling. In this juxtacrine signaling process, a cell's level of membrane-bound Delta depends on its intracellular level of activated Notch, which in turn depends on the average level of membrane-bound Delta of its neighbors. In such a situation, the Delta-Notch dynamics of the cells in a tissue sheet will depend on the rate of cell rearrangement and the fluctuations it induces. While the example does not explore the richness due to the coupling of subcellular networks with intercellular networks and cell behaviors, it already shows how different such behaviors can be from those of their non-spatial simplifications. We begin with the Ordinary Differential Equation (*ODE*) Delta-Notch patterning model of Collier (***106***) in which juxtacrine signaling controls the internal levels of the cells' Delta and Notch proteins. The base model neglects the complexity of the interaction due to changing spatial relationships in a real tissue:

$$\frac{dD}{dt} = v \cdot \left(\frac{1}{1 + b \cdot N^h} - D \right), \qquad (16)$$

$$\frac{dN}{dt} = \frac{\overline{D}^k}{a + \overline{D}^k} - N, \qquad (17)$$

where D and N are the concentrations of activated Delta and Notch proteins inside a cell, $\overline{D}$ is the average concentration of activated Delta protein at the surface of the cell's neighbors, a and b are saturation constants, h and k are Hill coefficients, and v is a constant that gives the relative lifetimes of Delta and Notch proteins.

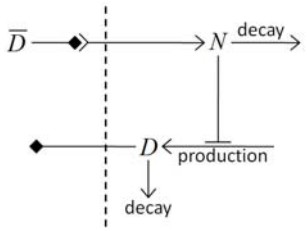


Figure 31 Diagram of Delta-Notch feedback regulation between and within cells.

Notch activity increases with the levels of Delta in neighboring cells, while Delta activity decreases with increasing Notch activity inside a cell (Figure 31). When the parameters in the ODE model are chosen correctly, each cell assumes one of two exclusive states: a *primary fate*, in which the cell has a high level of Delta and a low level of Notch activity, and a *secondary fate*, in which the cell has a low level of Delta and a high level of Notch. To build this model in CC3D, we assign a separate copy of the ODE model (eqs. *(16)* and *(17)*) to each cell and allow each cell to see the Delta concentrations of its neighbors. We use CC3D's BionetSolver library to manage and solve the ODEs, which are stored using the SBML standard.

The three files that specify the Delta-Notch model are included in the CompuCell3D installation and can be found at *<CC3D-installation-dir>/DemosBionetSolver/DeltaNotch*: the main Python file (*DeltaNotch.py*) sets the parameters and initial conditions; the Python steppable file (*DeltaNotch_Step.py*) calls the subcellular models; and the SBML file (*DN_Collier.sbml*) contains the description of the ODE model. The first two files can be generated and edited using Twedit++, the last can be generated and edited using an SBML editor like Jarnac or JDesigner (both are open source). Listing 9 shows the SBML file viewed using Jarnac (www.sys-bio.org).

```
01   p = defn cell
02     vol compartment;
03     var D, N;
04     ext Davg, X;
05     $X -> N; pow(Davg,k)/(a+pow(Davg,k))-N;
06     $X -> D; v*(1/(1+b*pow(N,h))-D);
07   end;
08
09   p.compartment = 1;
10   p.Davg = 0.4;
11   p.X = 0;
12   p.D = 0.5;
13   p.N = 0.5;
14   p.k = 2;
15   p.a = 0.01;
16   p.v = 1;
17   p.b = 100;
18   p.h = 2;
```

Listing 18 Jarnac specification of the Delta-Notch coupling model in Figure 31.

The main Python file (*DeltaNotch.py*) includes lines to define a steppable class (`DeltaNotchClass`) to include the ODE model and its interactions with the CC3D generalized cells (Listing 19).

```
01  from DeltaNotch_Step import DeltaNotchClass
02  deltaNotchClass=DeltaNotchClass(_simulator=sim,_frequency=1)
03  steppableRegistry.registerSteppable(deltaNotchClass)
```

Listing 19 Registering `DeltaNotchClass` in the main Python script, DeltaNotch.py in the Delta-Notch model.

The Python steppable file (Listing 20, *DeltaNotch_Step.py*) imports the BionetSolver library (line 1), then defines the class and initializes the solver inside it (lines 2-5).

```
01  import bionetAPI
02  class DeltaNotchClass(SteppableBasePy):
03    def __init__(self,_simulator,_frequency):
04      SteppableBasePy.__init__(self,_simulator,_frequency)
05      bionetAPI.initializeBionetworkManager(self.simulator)
06
07    def start(self):
08      #Loading model
09      Name = "DeltaNotch"
10      Key  = "DN"
11      Path = os.getcwd()+"\DemosBionetSolver\DeltaNotch\DN_Collier.sbml"
12      IntegrationStep = 0.2
13      bionetAPI.loadSBMLModel(Name, Path, Key, IntegrationStep)
14
15      bionetAPI.addSBMLModelToTemplateLibrary(sbmlModelName,"TypeA")
16      bionetAPI.initializeBionetworks()
17
18      import random
19      for cell in self.cellList:
20        D = random.uniform(0.9,1.0)
21        N = random.uniform(0.9,1.0)
22        bionetAPI.setBionetworkValue("DN_D",D,cell.id)
23        bionetAPI.setBionetworkValue("DN_N",N,cell.id)
24        cellDict=CompuCell.getPyAttrib(cell)
25        cellDict["D"]=D
26        cellDict["N"]=N
```

Listing 20 Implementation of the `__init__` and `start` functions of the `DeltaNotchClass` in the Delta-Notch model.

The first lines in the `start` function (Listing 20, lines 9-12) specify the name of the model, its nickname (for easier reference), the path to the location where the SBML model is stored, and the time-step of the ODE integrator, which fixes the relation between MCS and the time units of the ODE model (here, 1 MCS corresponds to 0.2 ODE model time units). In line 13 we use the defined names, path and time-step parameter to load the SBML model.
Listing 11, line 15 associates the subcellular model with the CC3D cells, creating an instance of the ODE solver (described by the SBML model) for each cell of type `TypeA`. Line 16 initializes the loaded subcellular models.

To set the initial levels of Delta (`D`) and Notch (`N`) in each cell, we visit all cells and assign random initial concentrations between 0.9 and 1.0 (Listing 20, lines 18-26). Line 18 imports the intrinsic Python random number generator. Lines 23-24 pass these values to the subcellular models in each cell. The first argument specifies the ODE model parameter to change with a string containing the nickname of the model, here `DN`, followed by an underscore and the name of the parameter as defined in the SBML file. The second argument specifies the value to assign to the parameter, and the last argument specifies the cell id. For visualization, we store the values of `D` and `N` in a dictionary attached to each cell (lines 25-26).

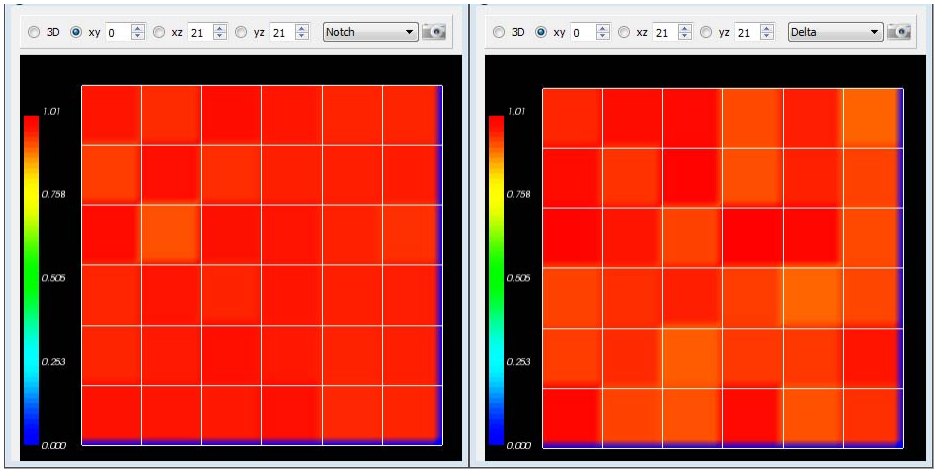


Figure 32 Initial Notch (left) and Delta (right) concentrations in the Delta-Notch model.

Listing 21 defines a `step` function of the class, which is called every MCS, to read the Delta concentrations of each cell's neighbors to determine the value of $\overline{D}$ (the average Delta concentration around the cell). The first three lines in listing 12 iterate over all cells. Inside the loop, we first set the variables `D` and `nn` to zero. They will store the total Delta concentration of the cell's neighbors and the number of neighbors, respectively. Next we get a list of the cell's neighbors and iterate over them. Line 9 reads the Delta concentration of each neighbor (the first argument is the name of the parameter and the second is the id of the neighboring cell) summing the total Delta and counting the number of neighbors. Note the `+=` syntax (*e.g.*, `nn+=1 is equivalent to nn=nn+1`). Lines 3 and 7 skip `Medium` (`Medium` has a value 0, so `if (Medium)` is false).

```
01  def step(self,mcs):
02    for cell in self.cellList:
03      if cell:
04        D=0.0; nn=0
05        cellNeighborList=self.getCellNeighbors(cell)
06        for nsd in cellNeighborList:
07          if nsd:
08            nn+=1
09            D+=bionetAPI.getBionetworkValue("DN_D",nsd.neighborAddress.id)
10        if (nn>0):
11          D=D/nn
12        bionetAPI.setBionetworkValue("DN_Davg",D,cell.id)
13        cellDict=CompuCell.getPyAttrib(cell)
14        cellDict["D"]=D
15        cellDict["N"]=bionetAPI.getBionetworkValue("DN_N",cell.id)
```

```
16    bionetAPI.timestepBionetworks()
```

Listing 21 Implementation of a `step` function to calculate $\overline{D}$ in the `DeltaNotchClass` in the Delta-Notch model.

After looping over the cell's neighbors, we set its new value of the variable $\overline{D}$, which in the SBML code has the name `Davg`, to the average neighboring Delta (`D`) concentration, ensuing that the denominator, `nn`, is not zero (Listing 21, lines 10-12).
The remaining lines (Listing 21, lines 13-15) access the cell dictionary and store the cell's current Delta and Notch concentrations. Line 16 then calls BionetSolver and tell it to integrate the ODE model with the new parameters for one integration step (0.2 time units in this case).
Figure 33 shows a typical cell configurations and states for the simulation. The random initial values gradually converge to a pattern with cells with low levels of Notch (primary fate) surrounded by cells with high levels of Notch (secondary fate).

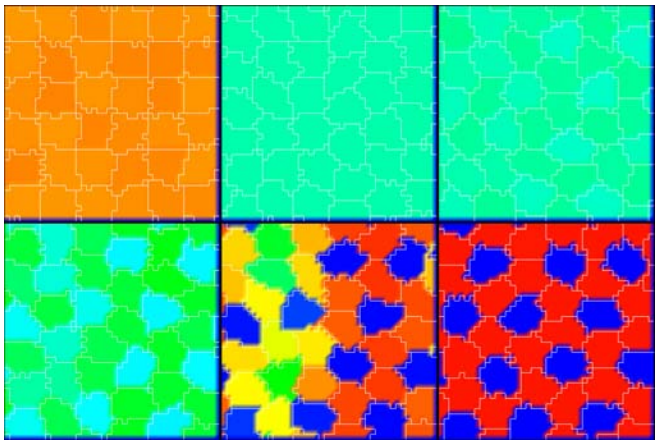


Figure 33 Dynamics of the Notch concentrations of cells in the Delta-Notch model. Snapshots taken at 10, 100, 300, 400, 450 and 600 MCS.

Listing 22 lines 2-4 define two new visualization fields in the main Python file (*DeltaNotch.py*) to visualize the Delta and Notch concentrations in CompuCell Player. To fill the fields with the Delta and Notch concentrations we call the steppable class, `ExtraFields` (Listing 22, lines 6-9). This code is very similar to our previous steppable calls, with the exception of line 8, which uses the function `setScalarFields()`to reference the visualization `Fields`.

```
01   #Create extra player fields here or add attributes
02   dim=sim.getPotts().getCellFieldG().getDim()
03   DeltaField=simthread.createScalarFieldCellLevelPy("Delta")
04   NotchField=simthread.createScalarFieldCellLevelPy("Notch")
05
06   from DeltaNotch_Step import ExtraFields
07   extraFields=ExtraFields(_simulator=sim,_frequency=5)
08   extraFields.setScalarFields(DeltaField,NotchField)
09   steppableRegistry.registerSteppable(extraFields)
```

Listing 22 Adding extra visualization fields in the main Python script DeltaNotch.py in the Delta-Notch model.

In the steppable file (Listing 23, *DeltaNotch_Step.py*) we use `setScalarFields()` to set the variables `self.scalarField1` and `self.scalarField2` to point to the fields `DeltaField` and `NotchField`, respectively. Lines 10 and 11 of the `step` function clear the two fields using `clearScalarValueCellLevel(). Line 12` loops over all cells, line 13 accesses a cell's dictionary and lines 14 and 15 use the `D` and `N` entries to fill in the respective visualization fields, where the first argument specifies the visualization field, the second the cell to be filled, and the third the value to use.

```
01  class ExtraFields(SteppableBasePy):
02    def __init__(self,_simulator,_frequency=1):
03      SteppableBasePy.__init__(self,_simulator,_frequency)
04
05    def setScalarFields(self,_field1,_field2):
06      self.scalarField1=_field1
07      self.scalarField2=_field2
08
09    def step(self,mcs):
10      clearScalarValueCellLevel(self.scalarField1)
11      clearScalarValueCellLevel(self.scalarField2)
12      for cell in self.cellList:
13        cellDict=CompuCell.getPyAttrib(cell)
14        fillScalarValueCellLevel(self.scalarField1,cell,cellDict["D"])
15        fillScalarValueCellLevel(self.scalarField2,cell,cellDict["N"])
```

Listing 23 Steppable to visualize the concentrations of Delta and Notch in each cell in the Delta-Notch model.

The two fields can be visualized in CompuCell Player using the `Field-selector` button of the `Main Graphics Window` menu (second-to-last button, Figure 32).
As we illustrate in figure 20, the result is a roughly hexagonal pattern of activity with one cell of low Notch activity for every two cells with high Notch activity. In the presence of a high level of cell motility, the identity of high and low Notch cells can change when the pattern rearranges. We could easily explore the effects of Delta-Notch signaling on tissue structure by linking the Delta-Notch pathway to one of its known downstream targets. *E.g.* if we wished to simulate embryonic feather-bud primordial in chicken skin or the formation of colonic crypts, we could start with an epithelial sheet of cells in 3D on a rigid support, and couple the growth of the cells to their level of Notch activity by having Notch inhibit cell growth. The result would be clusters of cell growth around the initial low-Notch cells, leading to a patterned 3D buckling of the epithelial tissue. Such mechanisms are capable of extremely complex and subtle patterning, as observed *in vivo*.

7 Conclusion

Multi-cell modeling, especially when combined with subcell (or supercell modeling) of biochemical networks, allows the creation and testing of hypotheses concerning many key aspects of embryonic development, homeostasis and developmental disease. Until now, such modeling has been out of reach to all but experienced software developers. CC3D makes the development of such models much easier, though it still does involve a minimal level of hand editing. We hope the examples we have shown will convince readers to evaluate the suitability of CompuCell3D for their research.

Furthermore, CC3D directly addresses the current difficulty researchers face in reusing, testing or adapting both their own and published models. Most published multi-cell, multi-scale models exist in the form of Fortran/C/C++ code which is often of little practical value to other potential users. Reusing such code involves digging into large code bases, inferring their function, extracting the relevant code and trying to paste it into a new context. CompuCell3D improves this status quo in three ways: 1) It is fully open-source. 2) CC3D model execution is cross-platform and does not require compilation. 3) CC3D models are modular, compact and shareable. Because Python-based CC3D models require much less development effort to develop than custom code, simulations are fast and easy to develop and refine. Despite this convenience, CC3D 3.6 often runs as fast or faster than custom code solving the same model. Current CC3D development focuses on adding GPU-based PDE solvers, MPI parallelization and additional cell behaviors. We are also developing a high-level cell-behavior model description language which will compile into executable Python, removing the last need for model builders to learn programming techniques.

We hope the examples we have shown will convince readers to evaluate the suitability of GGH simulations using CompuCell3D for their research.

Most of the the code examples presented in this part of the manual are available from *www.compucell3d.org* and are often included in the binary CC3D packages. They will be curated to ensure their correctness and compatibility with future versions of CompuCell3D.

8 Acknowledgements

We gratefully acknowledge support from the National Institutes of Health, National Institute of General Medical Sciences, grants 1R01 GM077138-01A1 and 1R01 GM076692-01, Environment Protection Agency research grants 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.

9 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.
101. Merks, R. M., and Glazier, J. A. (2006). Dynamic mechanisms of blood vessel growth. *Nonlinearity* **19**, C1-C10.
102. Merks, R. M., Perryn, E. D., Shirinifard, A., and Glazier, J. A. (2008). Contact-inhibited chemotactic motility can drive both vasculogenesis and sprouting angiogenesis. *PLoS Computational Biology* **4**, e1000163.
103. Leith, J.T., and Michelson, S. (1995). Secretion rates and levels of vascular endothelial growth factor in clone A or HCT-8 human colon tumour cells as a function of oxygen concentration. *Cell Prolif* **28**, 415-430.

104. Shirinifard, A., Gens, J. S., Zaitlen, B. L., Popławski, N. J., Swat, M. H., and Glazier, J. A. (2009). 3D Multi-Cell Simulation of Tumor Growth and Angiogenesis. *PLoS ONE* **4**, e7190.
105. Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., Kitano, H., Arkin, A. P., Bornstein, B. J., Bray, D., Cornish-Bowden, A. , Cuellar, A. A., Dronov, S., Gilles, E. D., Ginkel, M., Gor, V., Goryanin, I. I., Hedley, W. J., Hodgman, T. C., Hofmeyr, J.-H., Hunter, P. J., Juty, N. S., Kasberger, J. L., Kremling, A., Kummer, U., Le Novère, N., Loew, L. M., Lucio, D., Mendes, P., Minch, E., Mjolsness, E. D., Nakayama, Y., Nelson, M. R., Nielsen, P. F., Sakurada, T., Schaff, J. C., Shapiro, B. E., Shimizu, T. S., Spence, H. D., Stelling, J., Takahashi, K., Tomita, M., Wagner, J., Wang, J. (2003). The Systems Biology Markup Language (SBML): A medium for representation and exchange of biochemical network models. *Bioinformatics* **19:** 524-531.
106. 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.