

---

# CC3D Reference Manual

*Release 4.6.0*

**Maciej Swat, Julio Belmonte, T.J. Sego, Peter Fyffe, James A. Glazier**

Sep 18, 2025



# INTRODUCTION

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Python Tutorials Section</b>	<b>5</b>
<b>3</b>	<b>How to programmatically control every aspect of the simulation</b>	<b>65</b>
<b>4</b>	<b>Common CC3D Tasks</b>	<b>77</b>
<b>5</b>	<b>Example: Contact-Inhibited Cell Growth</b>	<b>83</b>
<b>6</b>	<b>Volume and Cell Growth</b>	<b>85</b>
<b>7</b>	<b>Surface and Cell Contact</b>	<b>89</b>
<b>8</b>	<b>Mitosis</b>	<b>91</b>
<b>9</b>	<b>Cell Death</b>	<b>95</b>
<b>10</b>	<b>Secretion Plugin Reference</b>	<b>99</b>
<b>11</b>	<b>Field Secretion   Interacting with PDE Solver Fields</b>	<b>101</b>
<b>12</b>	<b>Chemotaxis on a cell-by-cell basis</b>	<b>107</b>
<b>13</b>	<b>Steering – changing CC3DML parameters on-the-fly.</b>	<b>111</b>
<b>14</b>	<b>Steering – changing Python parameters using Graphical User Interface.</b>	<b>115</b>
<b>15</b>	<b>Replacing CC3DML with equivalent Python syntax</b>	<b>119</b>
<b>16</b>	<b>Cell Motility. Applying force to cells.</b>	<b>123</b>
<b>17</b>	<b>Setting cell membrane fluctuation ona cell-by-cell basis</b>	<b>125</b>
<b>18</b>	<b>Dividing Clusters (aka compartmental cells)</b>	<b>127</b>
<b>19</b>	<b>Changing cluster id of a cell.</b>	<b>133</b>
<b>20</b>	<b>SBML Solver</b>	<b>135</b>
<b>21</b>	<b>Building SBML models using Tellurium</b>	<b>143</b>
<b>22</b>	<b>Building SBML models efficiently with Antimony and CellML</b>	<b>147</b>

<b>23 Building and Using MaBoSS Models</b>	<b>151</b>
<b>24 Real-World Examples Section</b>	<b>159</b>
<b>25 Configuring Multiple Screenshots</b>	<b>183</b>
<b>26 Parameter Scans</b>	<b>187</b>
<b>27 Restarting Simulations</b>	<b>195</b>
<b>28 Steppable Section</b>	<b>199</b>
<b>29 Plugins Section</b>	<b>203</b>
<b>30 XML Expression Evaluator - muParser</b>	<b>239</b>
<b>31 Diffusion (PDE Solvers) in CompuCell3D</b>	<b>241</b>
<b>32 Potts and Lattice Section</b>	<b>275</b>
<b>33 Algorithms &amp; How They Work Section</b>	<b>283</b>
<b>34 Transitioning from CC3D 3.x to CC3D 4.x</b>	<b>295</b>
<b>35 Secretion Plugin (legacy version for pre-v3.5.0)</b>	<b>301</b>
<b>36 Appendix A: List of Base Steppable Functions</b>	<b>303</b>
<b>37 Appendix B: List of Cell Attributes</b>	<b>309</b>
<b>38 Direct Call to CompuCell3D from Python</b>	<b>311</b>
<b>39 Changing number of Worknodes</b>	<b>317</b>
<b>40 Authors</b>	<b>319</b>
<b>41 Funding</b>	<b>321</b>
<b>42 References</b>	<b>323</b>

This manual teaches how to leverage Python for complex CompuCell3D simulations. You do not need to be an expert in Python, but you should know how to write simple Python scripts that use functions, classes, dictionaries, and lists. You can find decent tutorials online (e.g. [Instant Python Hacking](#)) or watch the [CompuCell3D Workshop Python tutorial videos](#).



---

**CHAPTER  
ONE**

---

## **INTRODUCTION**

### **Simulations**

CompuCell3D simulations are a combination of CC3DML (CompuCell3D XML model specification format) and Python code. By itself, CC3DML is “static.” That means you specify initial cellular behaviors, and, throughout the simulation, those behavior descriptions remain unchanged. Python scripting, meanwhile, enables you to build complex simulations wherein the behaviors of individual cells change (according to user specification) as the simulation progresses. We assure you that unless you use Python “unwisely,” you will not hit any performance barrier for CompuCell3D simulations. It’s true that there will be things that should be done in C++ because Python will be way too slow to handle certain tasks. However, throughout our years experience with CompuCell3D, we found that 90% of times Python will make your life way easier and will not impose ANY noticeable degradation in the performance. Based on our experience with biological modeling, it is by far more important to be able to develop models quickly than to have a clumsy but over-optimized code. If you have any doubts about this philosophy ask any programmer or professor of Software Engineering about the effects of premature optimization. With Python scripting you will be able to dramatically increase your productivity and it really does not matter if you know C++ or not. With Python you do not compile anything, just write script and run. If a small change is necessary, you edit source code and run again. You will waste no time dealing with compilation/installation of C/C++ modules and Python script you will write will run on any operating system (Mac, Windows, Linux). However, if you still need to develop high performance C++ modules, CompuCell3D and Twedit++ have excellent tools which make even C++ programming quite pleasurable. For C++ programming, check out the Developer Zone menu in Twedit++ to easily create your own module.



## PYTHON TUTORIALS SECTION

### 2.1 Create Your First CompuCell3D Project

#### Learning Objectives:

- Set up your first simulation project
  - Take a tour of Twedit++
- 

Twedit++ makes it easy to write template Python scripts for CC3D with just a few clicks. Click on either “CC3DML” (CompuCell3D XML) or “CC3D Python”, then hover over the menu to find a code snippet. It will insert the code wherever your cursor is.

Additionally, each CC3D installation includes a Demos folder with simple simulations. Studying these will give you a lot of insight into how to build Python scripts in CC3D.

#### Which Files Do I edit?

- **Python Steppables:** The Python code you write will live here. Use this to define custom behaviors that occur as the simulation progresses. The next lesson explains the “steppable” concept more.
- **Main Python Script:** This sets up the steppables you write, and, typically, you do not need to edit it much.
- **XML Script:** The CC3DML (CompuCell3D XML) file mostly holds built-in behaviors where less customization is desired. It also contains “initializer” code used to create cells in the beginning.

#### 2.1.1 Main Python Script

Every CC3D simulation that uses Python consists of the so-called main Python script. The structure of this script is fairly “rigid” (templated) which implies that, unless you know exactly what you are doing, you should make changes in this script only in a few distinct places, leaving the rest of the template untouched. The goal of the main Python script is to set up a CC3D simulation and make sure that all modules are initialized in the correct order. Typically, the only place where you will modify this script is towards the end of the script where you register your extension modules (steppables and plugins).

Another task of the main Python script is to load the CC3DML file which contains an initial description of cellular behaviors.

#### 2.1.2 XML Script

CC3DML provides an *initial* description of cell behaviors, and we will modify those behaviors as simulation runs using Python. Many behaviors that can be defined in CC3DML are also available in Python if you prefer to have greater customization. All you have to do is to use Twedit++’s snippet menu.

### 2.1.3 Example: How to Set up a Project

To create a new project, click CC3DProject->New CC3D Project in the menu bar. In the first screen of the Simulation Wizard, we specify the name of the model (cellsorting), its storage directory (C:\CC3DProjects), and whether we will store the model as pure CC3DML (CC3D XML), Python and CC3DML, or pure Python. We recommend always using both Python and XML.

**Remark:** Simulation code for cellsorting will be generated in C:\CC3DProjects\cellsorting. On Linux/OSX/Unix systems, it will be generated in <your home directory>/CC3DProjects/cellsorting

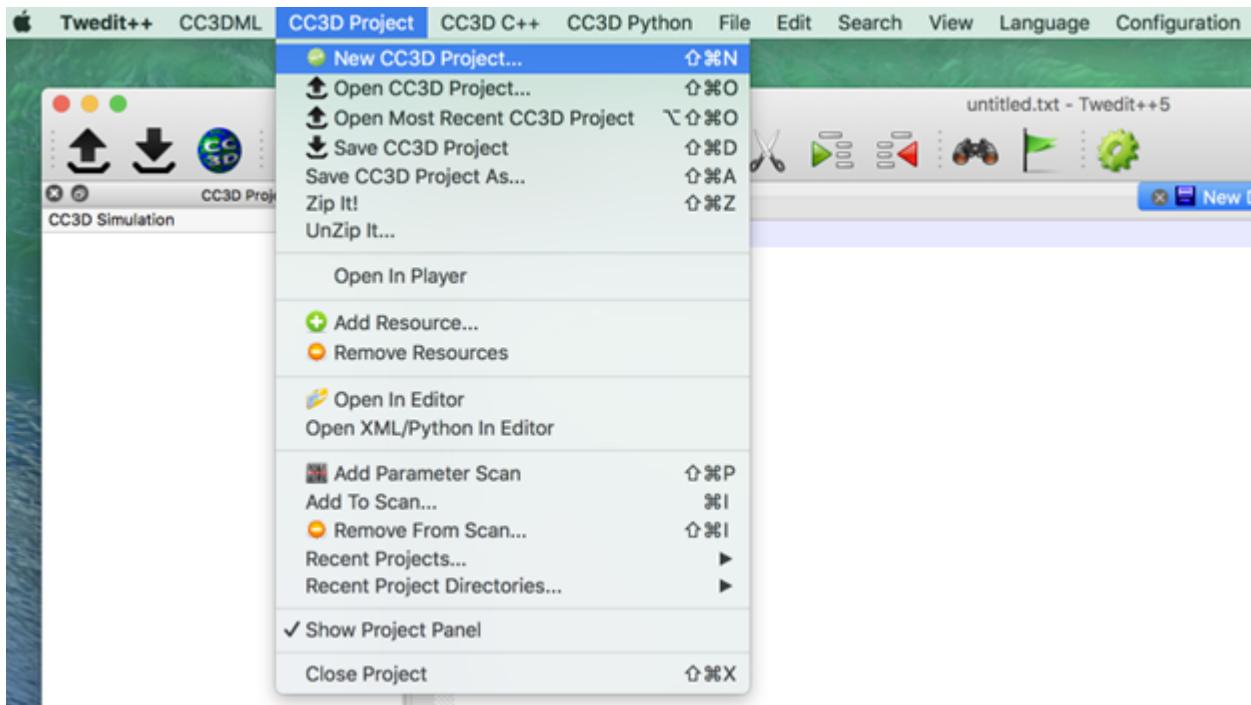


Fig. 1: Figure 1 Select CC3DProject->New CC3D Project

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 (temperature) of 10, a simulation duration of 10000 MCS, and a neighbor order of 2. BlobInitializer initializes the simulation with a disk of cells of a specified size.

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

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 them 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 **VolumeFlex Constraint** module.

Finally, Twedit++ generates the draft simulation code. Double-click on `cellsorting.cc3d` to open both the CC3DML (`cellsorting.xml`) and Python scripts for the model.

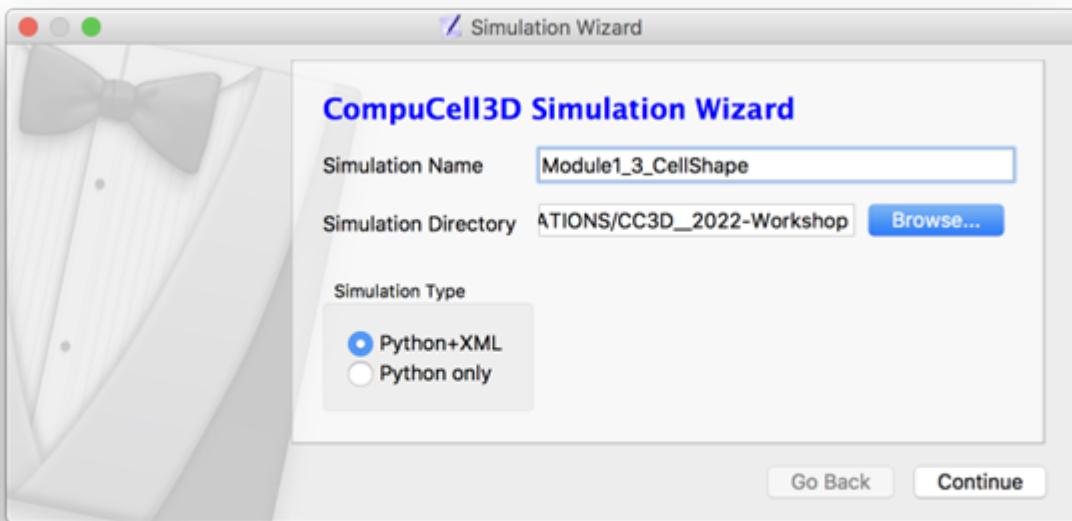
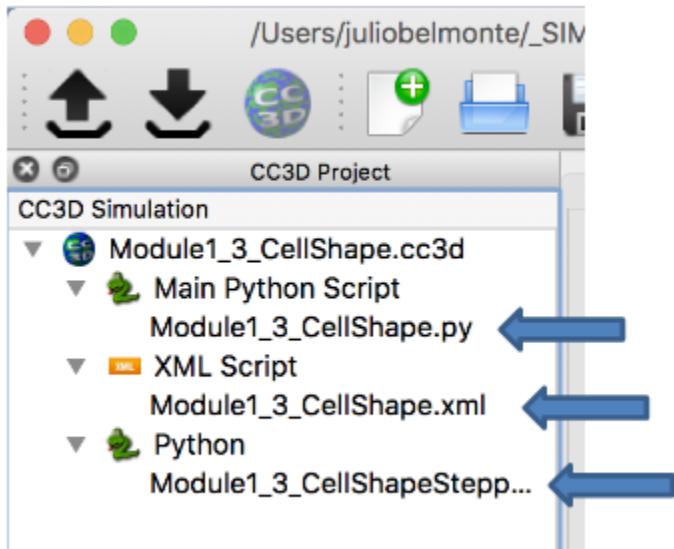


Fig. 2: Figure 1.1 Choose Python+XML in Simulation Wizard.



The names of the files for the newly-generated CC3D simulation code are stored in the .cc3d file (`C:\\\\CC3DProjects\\\\cellsorting`). Whenever you want to **re-open the project**, you should **select the .cc3d file**.

Besides that, you generally will not need to touch the .cc3d file since CC3D reads it to link the project files together automatically. When CC3D sees a `<PythonScript>` tag in that file, it knows that you are using Python scripting. Likewise, the tag `<XMLScript>` signifies that you are using XML.

Later, we'll discuss the full [Cell Sorting](#) exercise. For now, save your project.

---

Let's first look at a generated Python code:

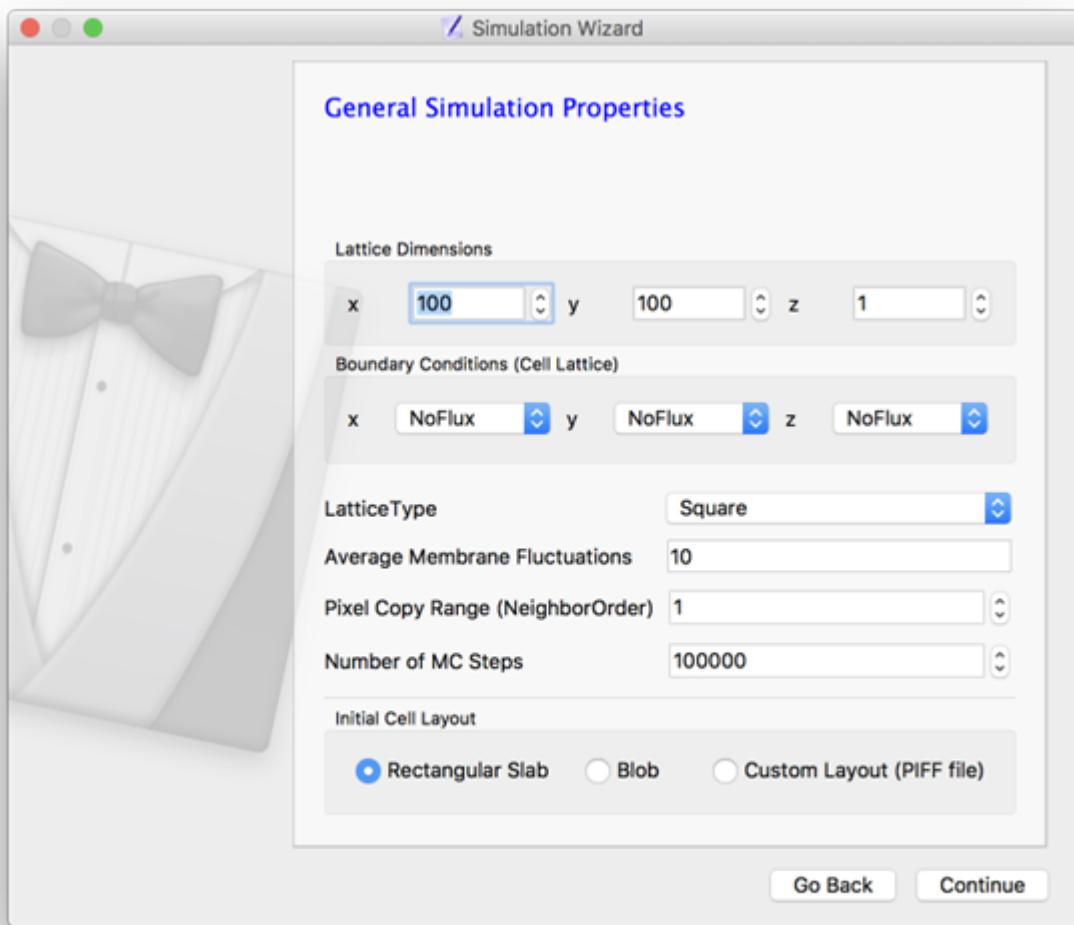


Fig. 3: Figure 2 Specification of basic cell-sorting properties in Simulation Wizard. Change NeighborOrder to 2.

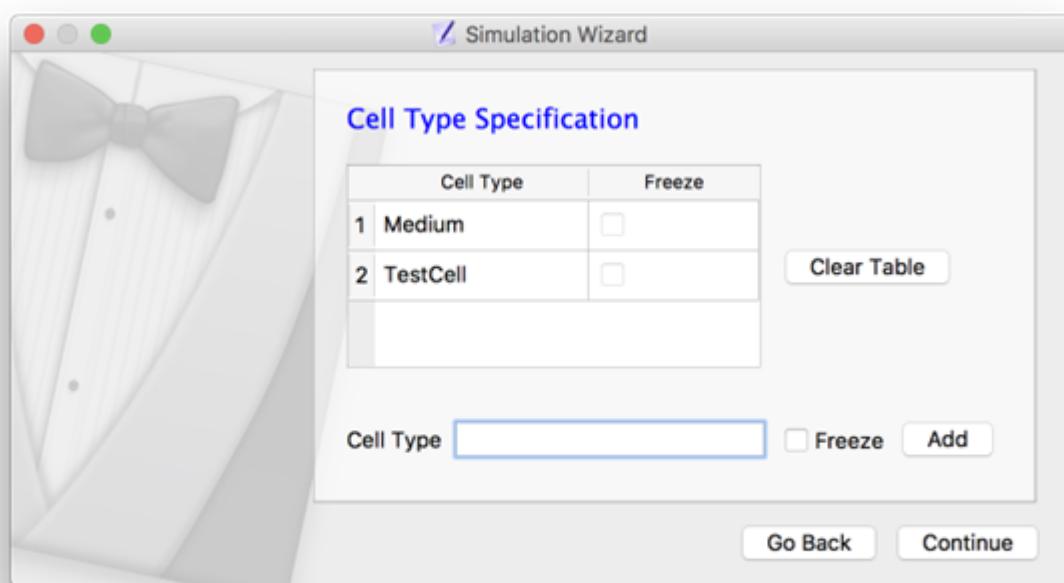
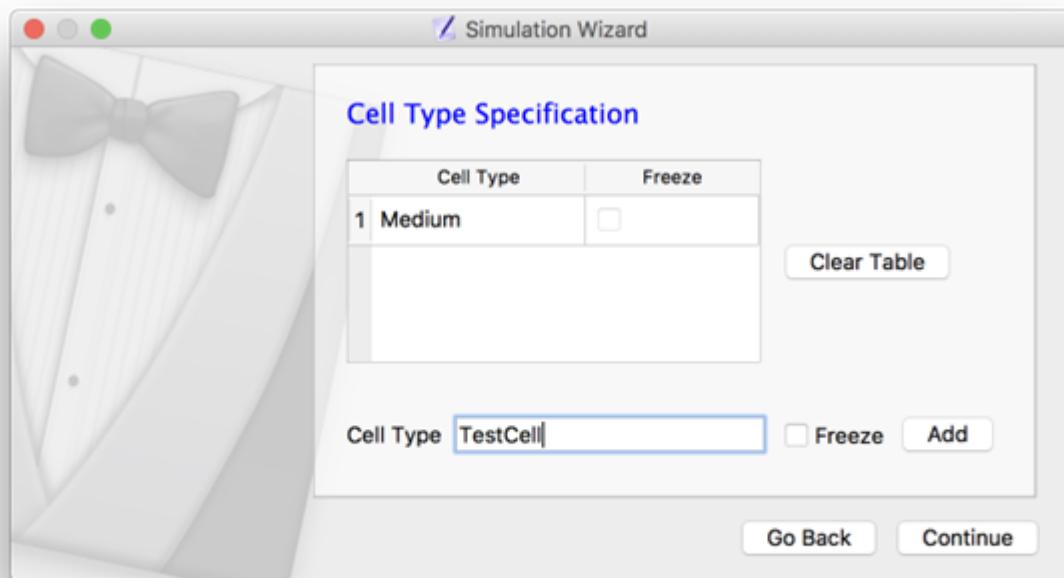


Fig. 4: Figure 3 Add a cell type by writing in the name and then clicking ‘Add’.

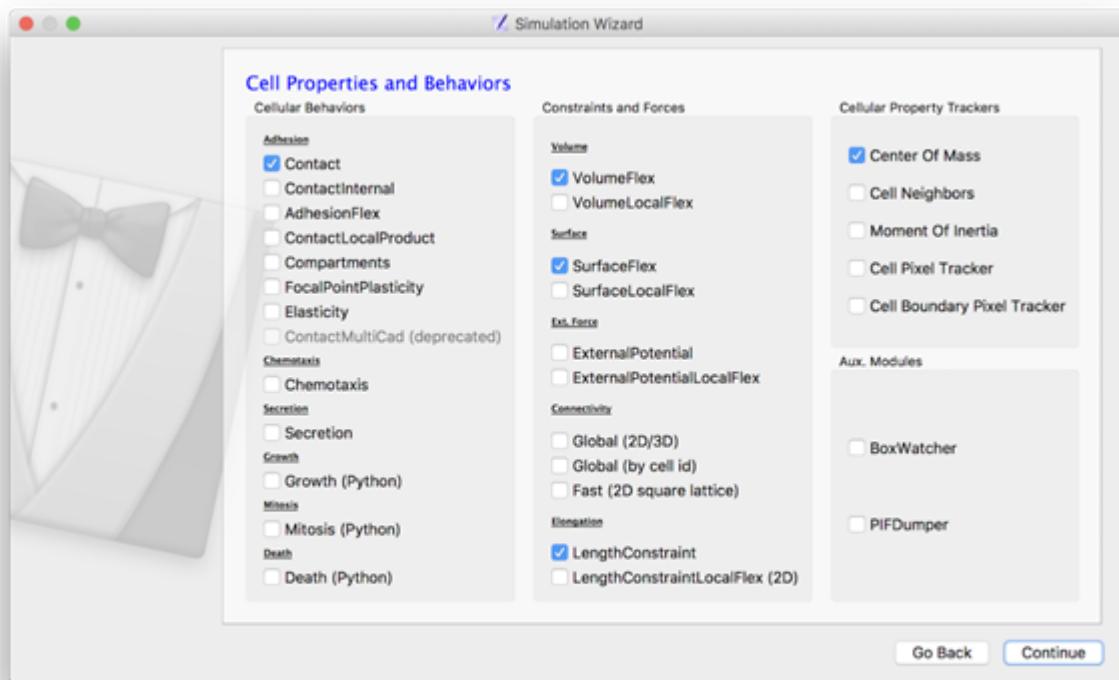


Fig. 5: Figure 4 Selection of cell-sorting cell behaviors in Simulation Wizard. [1]

File: C:\CC3DProjects\cellsorthing\Simulation\cellsorthing.py

```

1   from cc3d import CompuCellSetup
2   from cellsorthingSteppables import cellsorthingSteppable
3
4   CompuCellSetup.register_steppable(steppable=cellsorthingSteppable(frequency=1))
5
6   CompuCellSetup.run()

```

At the top of the simulation's Main Python Script, we import CompuCellSetup module from the cc3d package. The CompuCellSetup module has a few helpful functions that are used in setting up the simulation and starting execution of the CC3D model.

Next, we import newly generated steppable

```
from cellsorthingSteppables import cellsorthingSteppable
```

#### Note

If the module from which we import a steppable (here `cellsorthingSteppables`) or the steppable class (here `cellsorthingSteppable`) itself contains the word `steppable` (capitalization is not important), we can put `.` in front of the module: `from .cellsorthingSteppables import cellsorthingSteppable`. This is not necessary, but some development environments (e.g. PyCharm) will autocomplete syntax. This is quite helpful and speeds up the development process.

After this, we register the steppable by instantiating it using the constructor and specifying the frequency with which it will be called

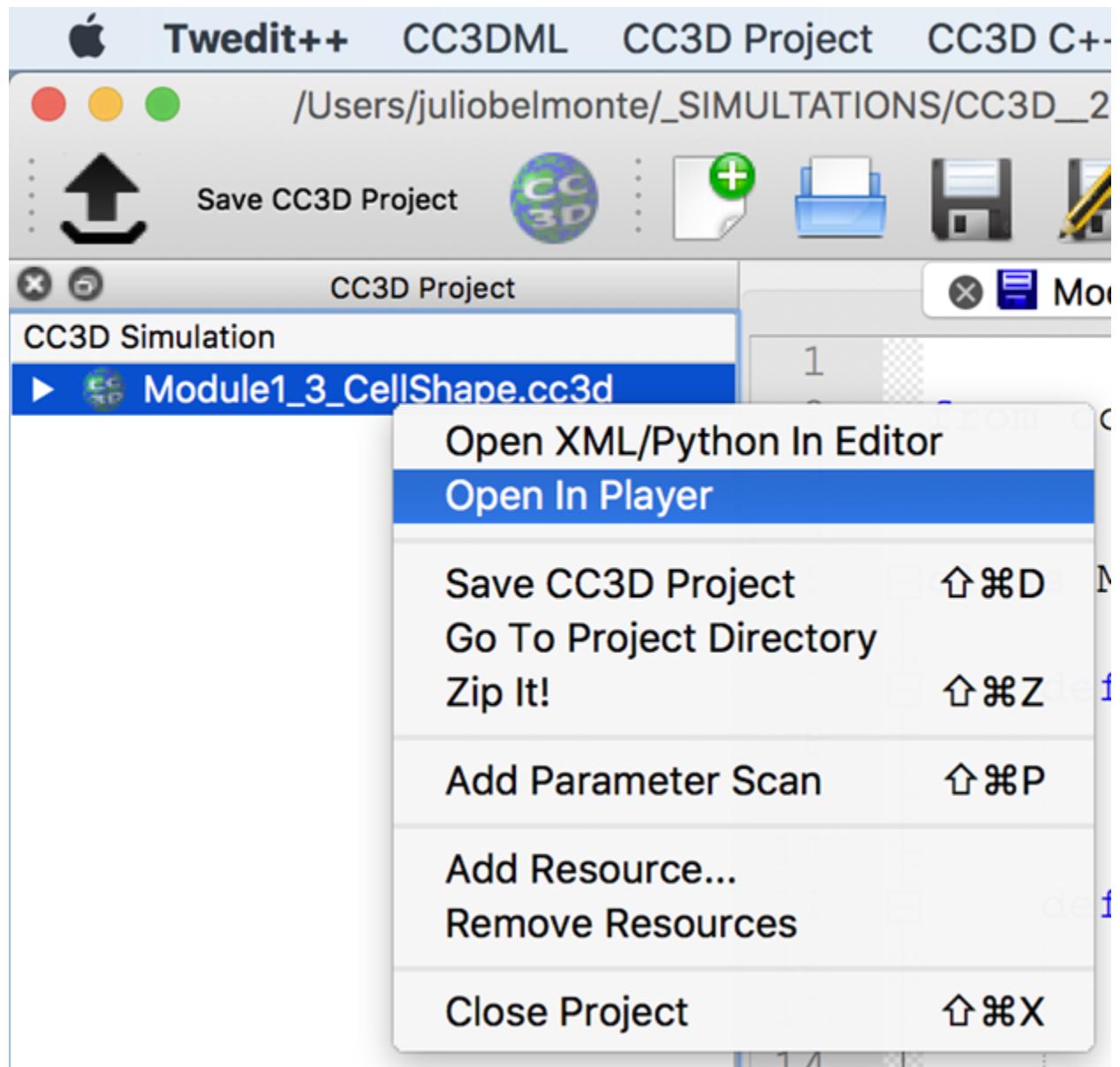
```
CompuCellSetup.register_steppable(steppable=cellsorthingSteppable(frequency=1))
```

Finally, we start simulation using

```
CompuCellSetup.run()
```

### 2.1.4 Exercise: Run the Simulation and Print Cell IDs

Right-click on the project name in Twedit, then click **Open In Player**. Or, if you already have Player ready, click **File->Open Simulation File (.cc3d)** then hit Play .



Once we open the .cc3d file in CompuCell3D Player, the simulation begins to run. When you look at the console output from this simulation, it will look something like this:

```
FAST numberOfAttempts=10000
Number of Attempted Energy Calculations=430
Step 155 Flips 171/10000 Energy -232 Cells 45 Inventory=45
cell.id= 1
cell.id= 2
cell.id= 3
cell.id= 4
cell.id= 5
cell.id= 6
cell.id= 7
```

Figure 5 Printing cell IDs using Python script

By default, the `step` function inside `cellsorthingSteppables.py` prints the ID of every cell on every time step.

```
from cc3d.core.PySteppables import *
```

(continues on next page)

(continued from previous page)

```

class cellsortingSteppable(SteppableBasePy):

    def __init__(self,frequency=1):
        SteppableBasePy.__init__(< b>self,frequency)

    def start(self):
        """
        any code in the start function runs before MCS=0
        """

    def step(self,mcs):
        """
        type here the code that will run every frequency MCS
        :param mcs: current Monte Carlo step
        """

        for cell in self.cell_list:
            print("cell.id=",cell.id)

    def finish(self):
        """
        Finish Function is called after the last MCS
        """

```

Inside the step function, we have the following code snippet:

```

for cell in self.cell_list:
    print("cell.id=",cell.id)

```

It prints the ID of every cell in the simulation. The step function is called after every Monte Carlo Step (MCS), so you will see the list print many times.

**Exercise:** Add an if statement so that this only prints the cell ID if mcs is less than 10.

## 2.1.5 Essential Python Functions

In addition to the step function, you can see start and finish functions, which are explained in the next module (SteppableBasePy class).

When writing Python extension modules, you have the flexibility to implement any combination of these 3 functions (start, step, and finish). You can, of course, leave them empty.

These 3 functions form the essence of Python scripting in CC3D:

1. start(**self**)
2. step(**self**,mcs)
3. finish(**self**)

---

Next Module: What is a Steppable? (SteppableBasePy class).

## 2.2 Running and Debugging CC3D Simulations Using PyCharm

Twedit++ provides many convenience tools when it comes to setting up simulation and also quickly modifying the content of the simulation using provided code helpers (see Twedit's CC3D Python and CC3D XML menus). Twedit also allows rapid creation of CC3D C++ plugins and steppables (something we cover in a separate developer's manual). However, as of current version, Twedit++ is just a code editor not an Integrated Development Environment (**IDE**). A real development environment offers many convenience features that make development faster. In this chapter we will teach you how to set up and use PyCharm to debug and quickly develop CC3D simulations. The ability of stepping up through simulation code is essential during simulation development. There you can inspect every single variable without using pesky **print** statements. You can literally see how your simulation is executed , statement-by-statement. In addition PyCharm provides nice context-based syntax completion so that by typing few characters from e.g. steppable method name (they do not need to be beginning characters) PyCharm will display available options, freeing you from memorizing every single method in CompuCell3D API.

First thing we need to do is to download and install PyCharm. Because PyCharm is written in Java it is available for every single platform. Visit <https://www.jetbrains.com/pycharm/download/> and get Community version of PyCharm for your operating system. You can also get professional version but you need to pay for this one so depending on your needs you have to make a choice here. We are using Community version because it is feature-rich and unless you do a lot of specialized Python development you will be fine with the free option.

After installing and doing basic configuration of PyCharm you are ready to open and configure CC3D to be executed from the IDE.

### 2.2.1 Step 1 - opening CC3D code in PyCharm and configuring Python environment

To open CC3D code in PyCharm, navigate to **File->Open...** and go to the folder where you installed CC3D and open the following subfolder <CC3D\_install\_folder>/lib/site-packages. In my case CC3D is installed in **c:\CompuCell3D-py3-64bit\** so I am opening **c:\CompuCell3D-py3-64bit\lib\site-packages\**

After you choose **site-packages** folder to open you may get another prompt that will ask you whether to open this folder in new window or attach to current window. Choose **New Window** option:

Next, you should see the following window

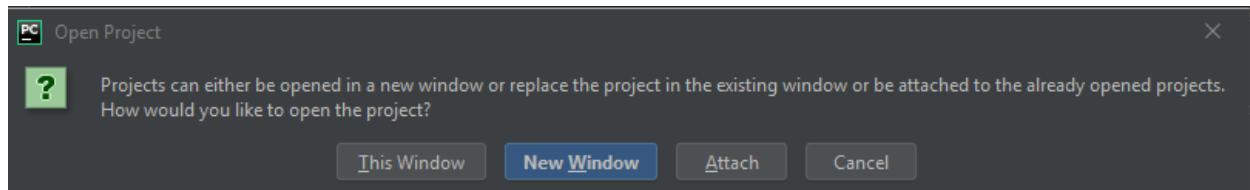
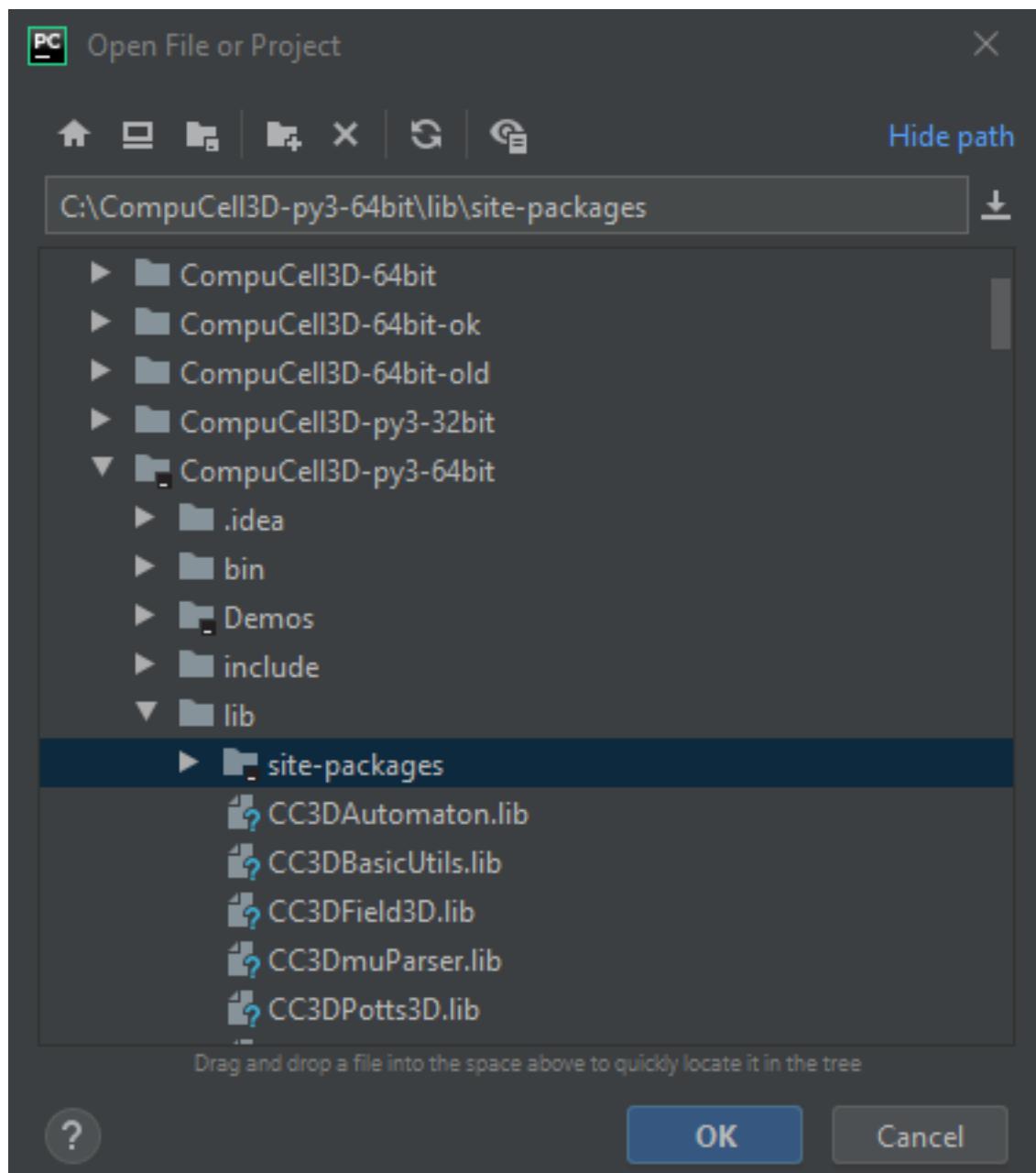
In order to be able to debug CC3D simulations it is best if the **Demos** folder (or any folder where you keep your simulations) also resides under **site-packages**. Simply copy **Demos** folder to **site-packages** folder so that your PyCharm Project Explorer looks as follows (left panel in PyCharm) - see **Demos** directory listed under **cc3d**:

### 2.2.2 Step 2 - running CC3D simulation from PyCharm. Configuring Python Environment and PREFIX\_CC3D

At this point we may attempt to run Player from PyCharm. To do so we expand **cc3d** folder in the left PyCharm panel and navigate to **cc3d->player5->compcell3d.pyw** and first we double-click to open **compcell3d.pyw** script in the editor and then **right-click** to open up a context menu and from there we choose **Run "compcell3d.pyw"** as shown below:

After we choose this option most likely we will get an error that will indicate that we need to set up Python environment to run CC3D in PyCharm.

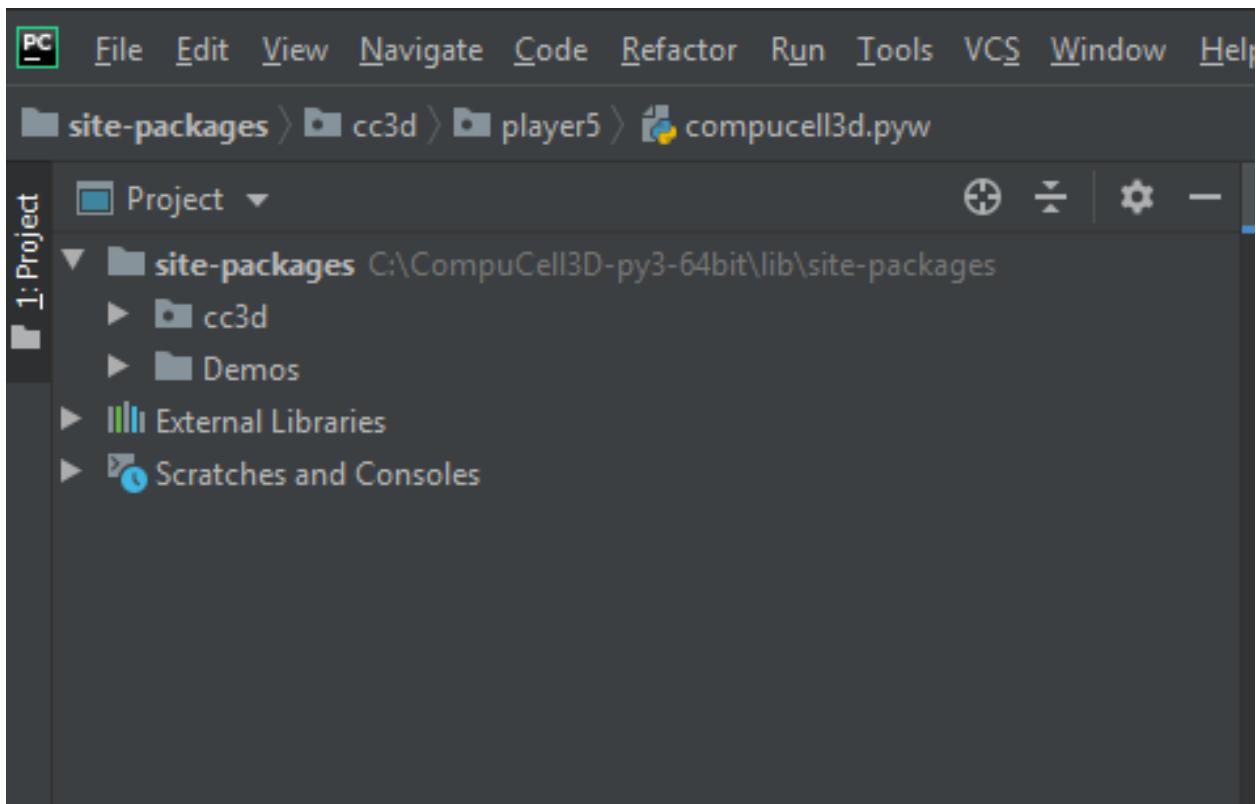
The actual error message might look different from the one shown below but regardless of it we need to setup proper Python environment inside PyCharm that we will use to run CC3D. Note, setting up environment is a task that you do only once because PyCharm remembers the environments you set up and it also remembers settings with which you ran particular projects and scripts. Setting up Python environment is actually quite easy because CompuCell3D ships with fully functional Python environment and in fact all we need to do is to point PyCharm where Python executable that CC3D uses is located. To do so we open up PyCharm Settings by going to **File->Settings** (or **PyCharm->Preferences...** if you are on Mac) and in the the search box of the Preferences dialog we type **interpreter** and select **Project Interpreter** option in the left panel:



```

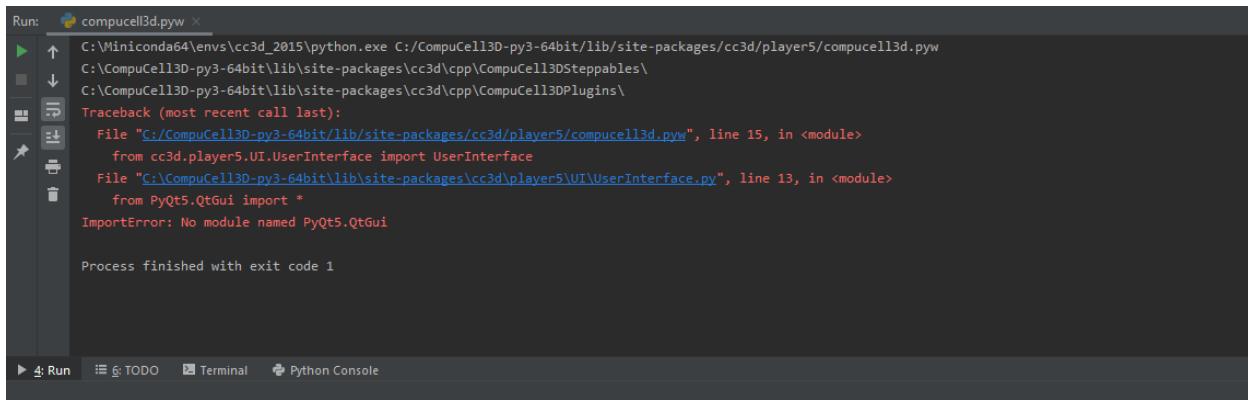
1 """
2 CC3D main script that lunches CC3D in the GUI mode
3 when running automated testing f Demo suite use the following cml options:
4
5 --exitWhenDone --testOutputDir=/Users/m/cc3d_tests --numSteps=100
6
7 or for automatic starting of a particular simulation you use :
8 -input=/home/m/376_dz/Demos/Models/cellsort/cellsort_2D/cellsort_2D.cc3d
9
10 """
11 import ...
12
13 setDebugging(0)
14
15 if sys.platform.lower().startswith('linux'):
16     # On linux have to import rr early on to avoid
17     # PyQt-related crash - appears to only affect VirtualBox Installs
18     # of linux
19     try:
20         import roadrunner
21     except ImportError:
22         print('Could not import roadrunner')
23         pass
24
25 if sys.platform.startswith('win'):
26     # this takes care of the need to distribute qwindows.dll with the qt5 application
27     # it needs to be located in the directory <library_path>/platforms
28     QCoreApplication.addLibraryPath("./bin/")
29
30
31
32
33
34
35
36
37
38
39
40
41

```

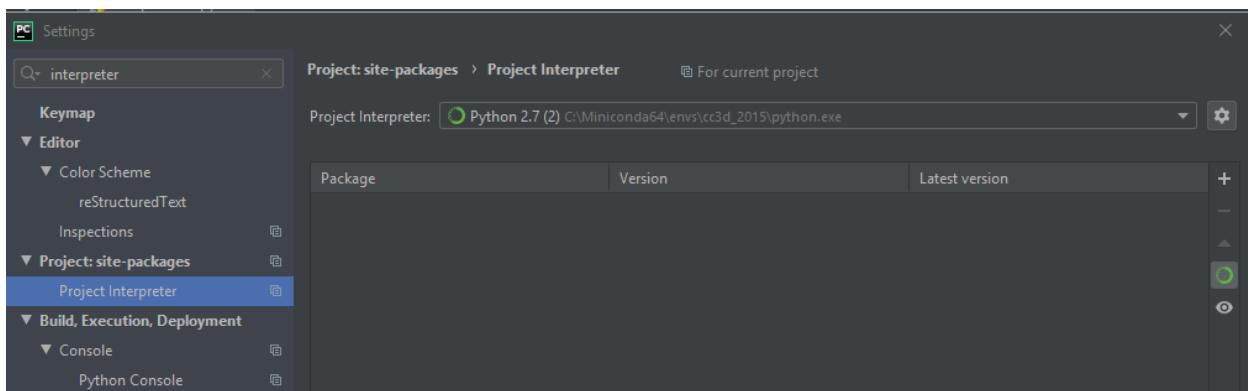


The screenshot shows the PyCharm IDE interface with the following details:

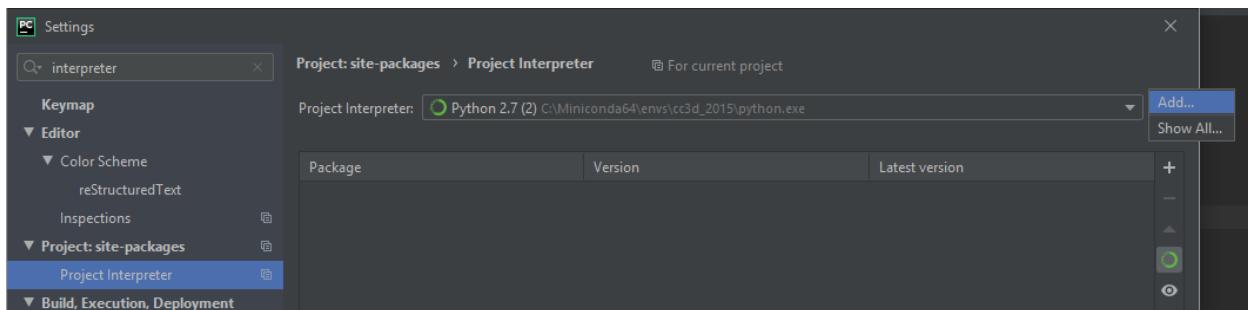
- File Path:** site-packages [C:\CompuCell3D-py3-64bit\lib\site-packages] - ...\\cc3d\\player5\\compucell3d.pyw
- Project Tree:** Shows the directory structure under site-packages: cc3d, core, cpp, player5, Configuration, Configuration\_settings, CQt, CustomGui, Graphics, icons, Launchers, Messaging, Plugins, Simulation, steering, tests, UI, Utilities, ViewManager.
- Editor Content:** The code for compucell3d.pyw. It includes comments about running CC3D in GUI mode or for automated testing, and imports for roadrunner and roadrunner. It handles imports differently based on the platform (linux vs win).
- Context Menu:** Opened at line 32, showing options:
  - New
  - Cut (Ctrl+X)
  - Copy (Ctrl+C)
  - Copy Path (Ctrl+Shift+C)
  - Copy Relative Path (Ctrl+Alt+Shift+C)
  - Paste (Ctrl+V)
  - Find Usages (Alt+F7)
  - Inspect Code...
  - Refactor
  - Clean Python Compiled Files
  - Add to Favorites
  - Reformat Code (Ctrl+Alt+L)
  - Optimize Imports (Ctrl+Alt+O)
  - Delete...
  - Run cmd script
  - Run cmd shell
  - Run 'compucell3d.pyw' (Ctrl+Shift+F10)
  - Debug 'compucell3d.pyw' (Ctrl+Shift+D)
  - Create 'compucell3d.pyw'...
  - Show in Explorer
  - File Path (Ctrl+Alt+F12)
  - Open in Terminal
  - Local History
  - Synchronize 'compucell3d.pyw'



The screenshot shows the PyCharm interface with the 'Run' tab selected. The 'Run' dropdown menu is open, showing the path: C:\Miniconda64\envs\cc3d\_2015\python.exe C:/CompuCell3D-py3-64bit/lib/site-packages/cc3d/player5/compuCell3d.pyw. Below this, a 'Traceback (most recent call last):' section shows an ImportError: No module named PyQt5.QtGui. At the bottom, it says 'Process finished with exit code 1'.



Next we click Gear box in the top right and the pop-up mini-dialog with Add... option opens up:



We select **Add...** and this brings us to the dialog where we configure point PyCharm to the Python executable we would like to use to run CC3D:

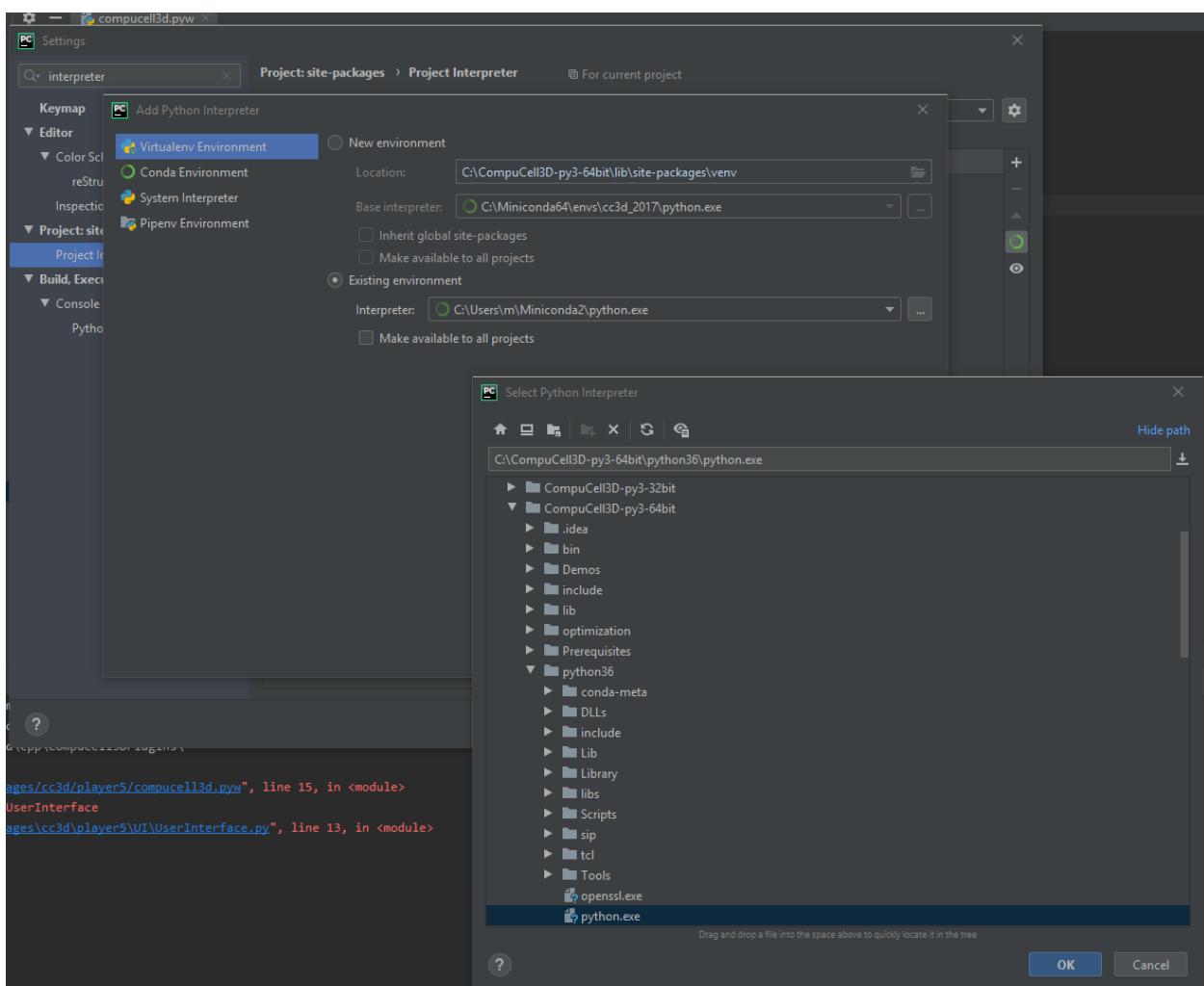
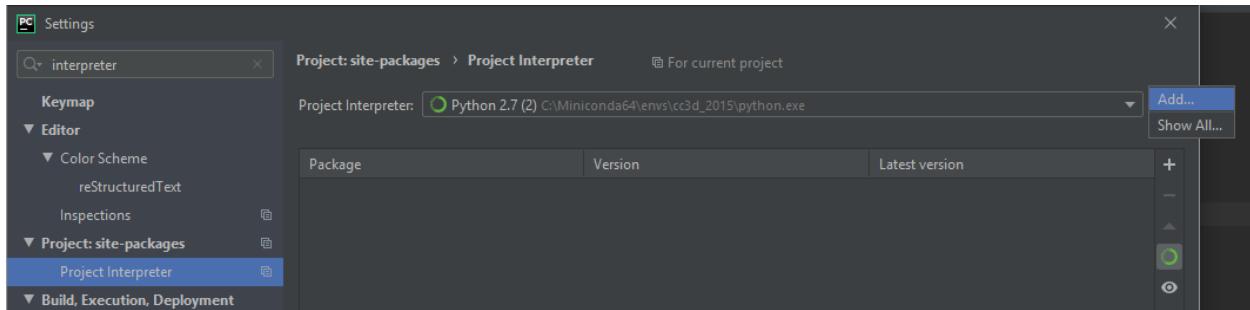
In this dialog we make sure to select options **Virtualenv Environment** and check **Existing Environment** radio-box and then select correct Python interpreter executable. In my case it is located in c:\CompuCell3D-py3-64bit\python36\python.exe and if you are on e.g. osx or linux you will need to navigate to <COMPUCELL3D\_INSTALL\_FOLDER>/Python3.x/bin/python :

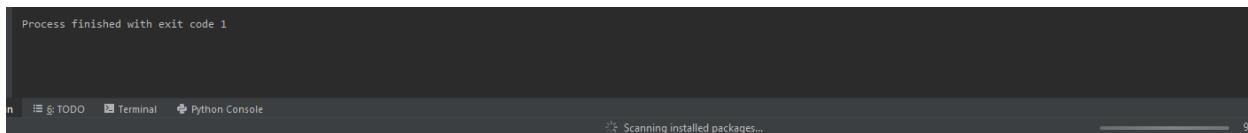
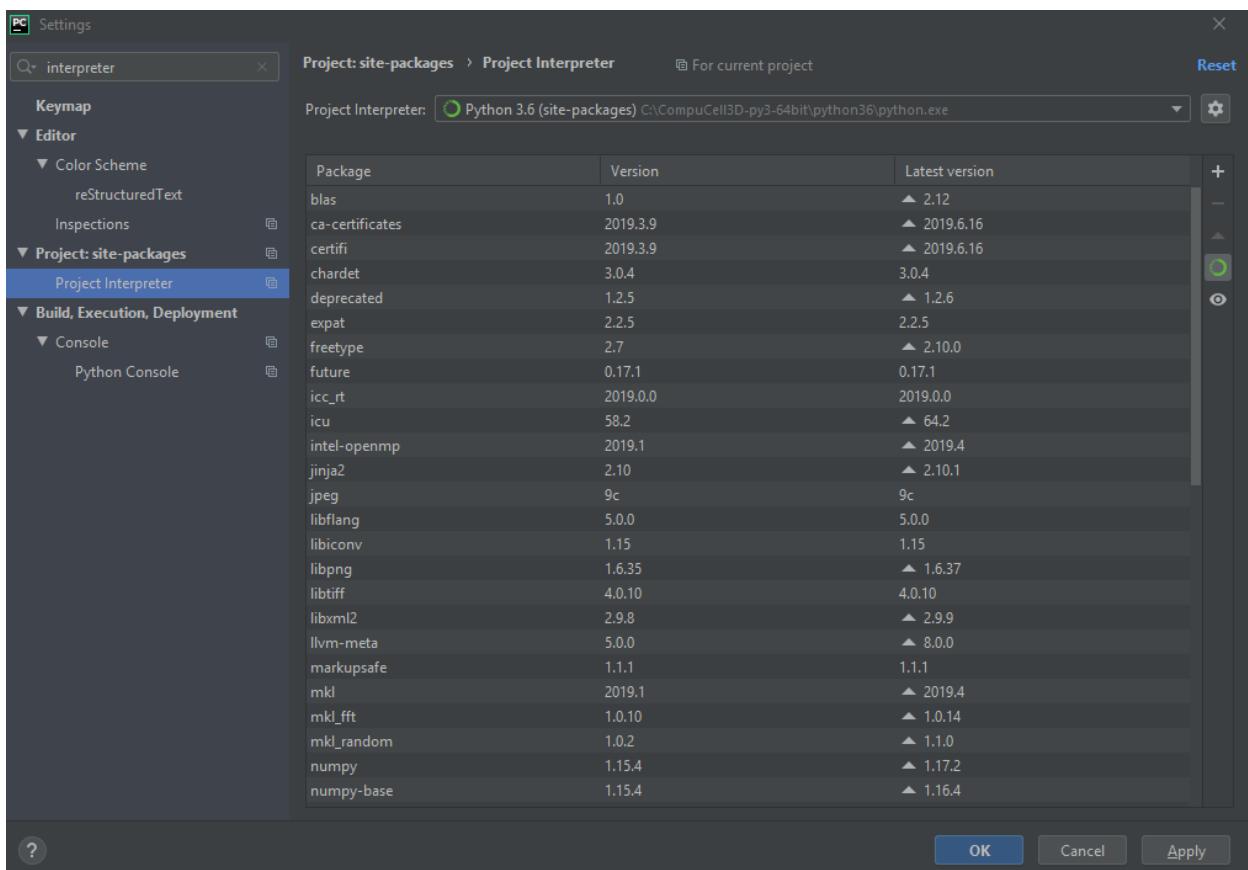
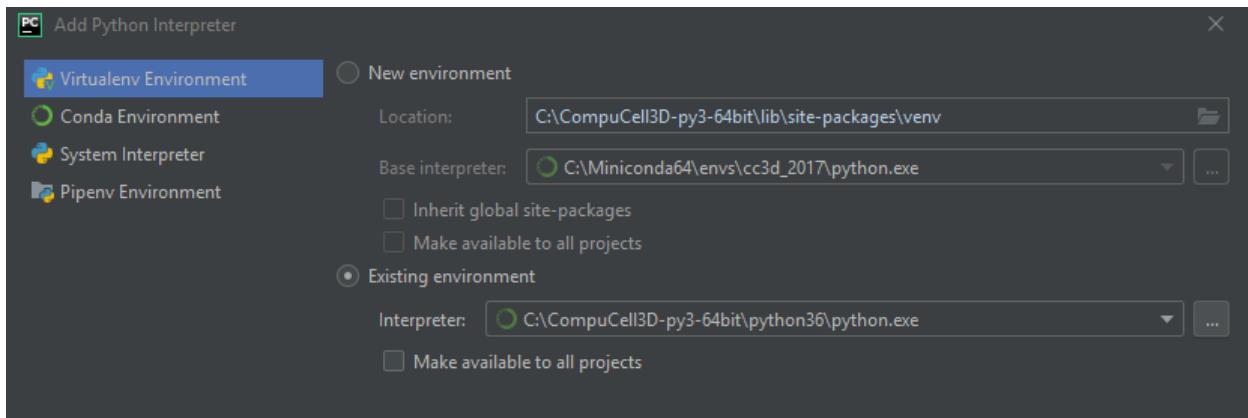
After we make a selection of the interpreter your **Add Python Interpreter** dialog should look as follows:

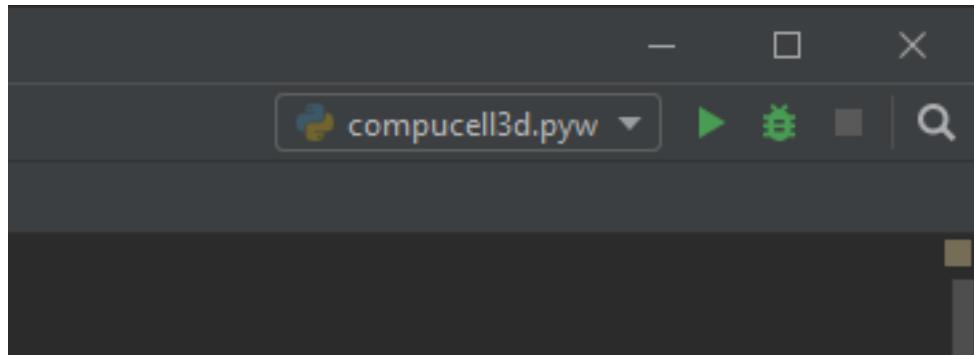
after we click OK PyCharm will scan the interpreter content for installed packages and display those packages in the dialog window:

Note, scanning may take a while so be patient. PyCharm will display progress bar below

and after it is done we may rerun compucell3d.pyw main script again. This time we will use PyCharm's convenience Run button located in the upper-right corner:







And, yes, we will get an error that tells us that we need to set environment variable PREFIX\_CC3D

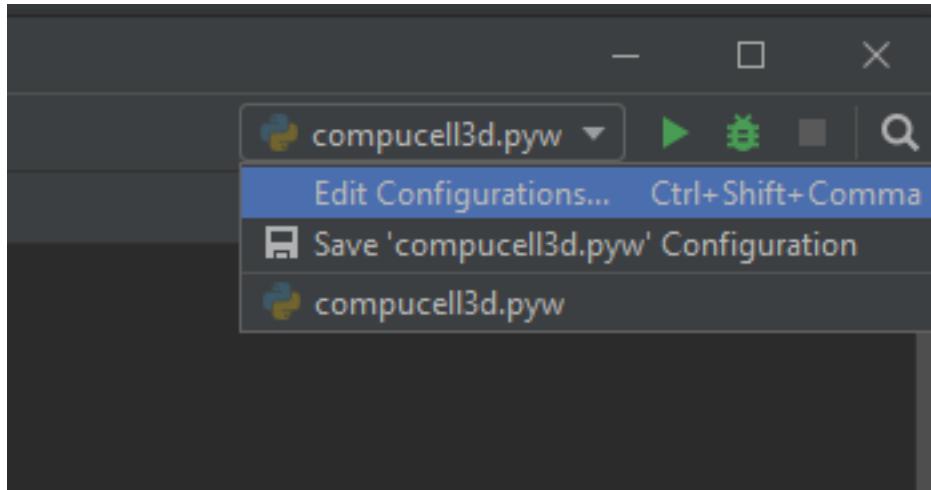
```

Run: compuCell3d.pyw ×
File "C:\CompuCell3D-py3-64bit\lib\site-packages\cc3d\player5\UI\userinterface.py", line 556, in __createLayout
    self.console = Console(self.consoleDock)
File "C:\CompuCell3D-py3-64bit\lib\site-packages\cc3d\player5\UI\Console.py", line 28, in __init__
    self.__errorConsole=ErrorConsole(self)
File "C:\CompuCell3D-py3-64bit\lib\site-packages\cc3d\player5\UI\ErrorConsole.py", line 103, in __init__
    self.cc3dSender = CC3DSender(self)
File "C:\CompuCell3D-py3-64bit\lib\site-packages\cc3d\player5\UI\CC3DSender.py", line 65, in __init__
    self.tweditCC3DPath = os.path.join(os.environ['PREFIX_CC3D'], 'twedit++.bat')
established empty port= 47406
File "C:\CompuCell3D-py3-64bit\python36\lib\os.py", line 669, in __getitem__
    raise KeyError(key) from None
KeyError: 'PREFIX_CC3D'

Process finished with exit code 1

```

The PREFIX\_CC3D is the path to the folder where you installed CC3D to set it up within PyCharm we open pull-down menu next to the Run button and choose Edit Configurations...:

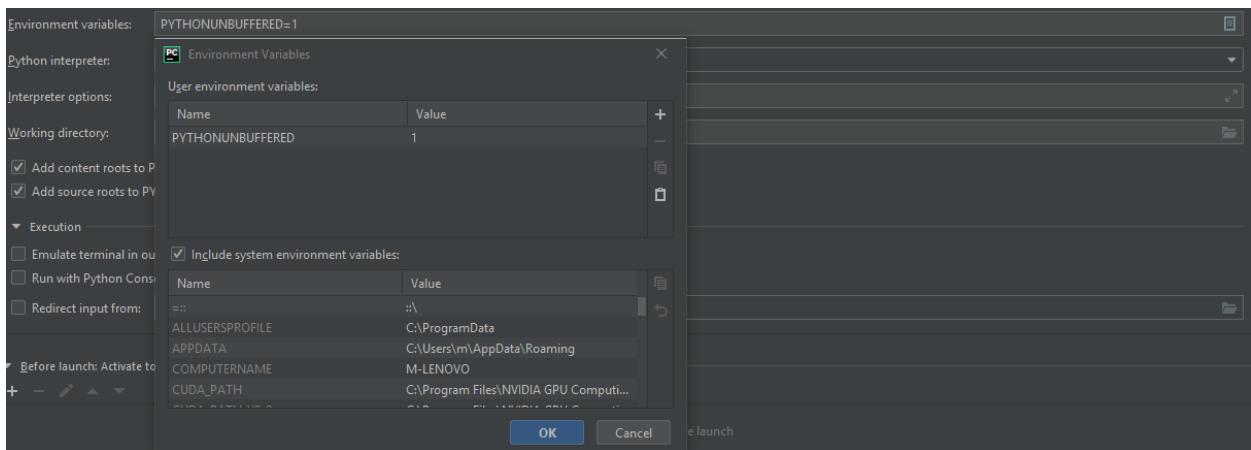
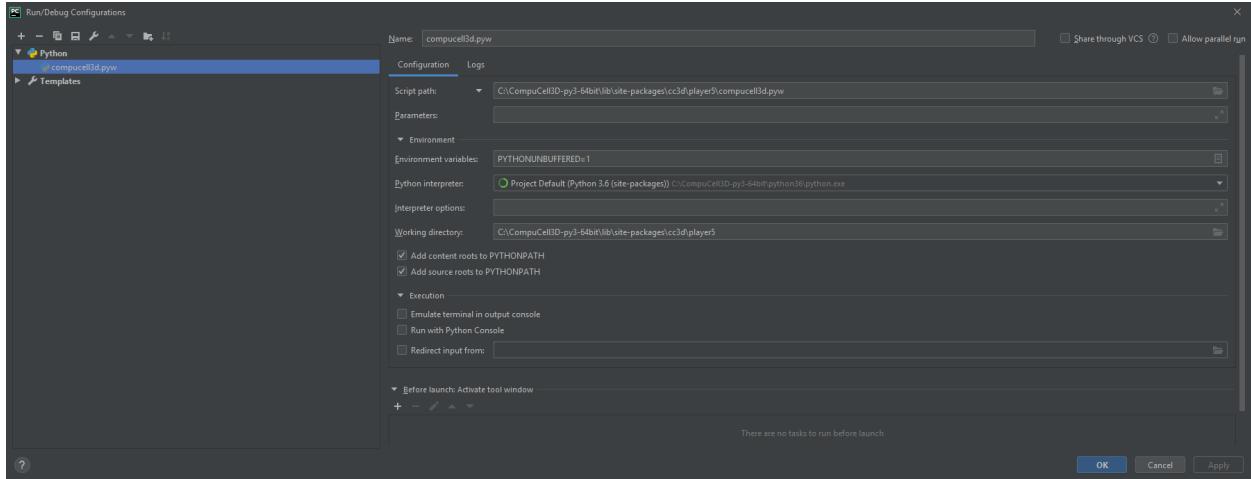


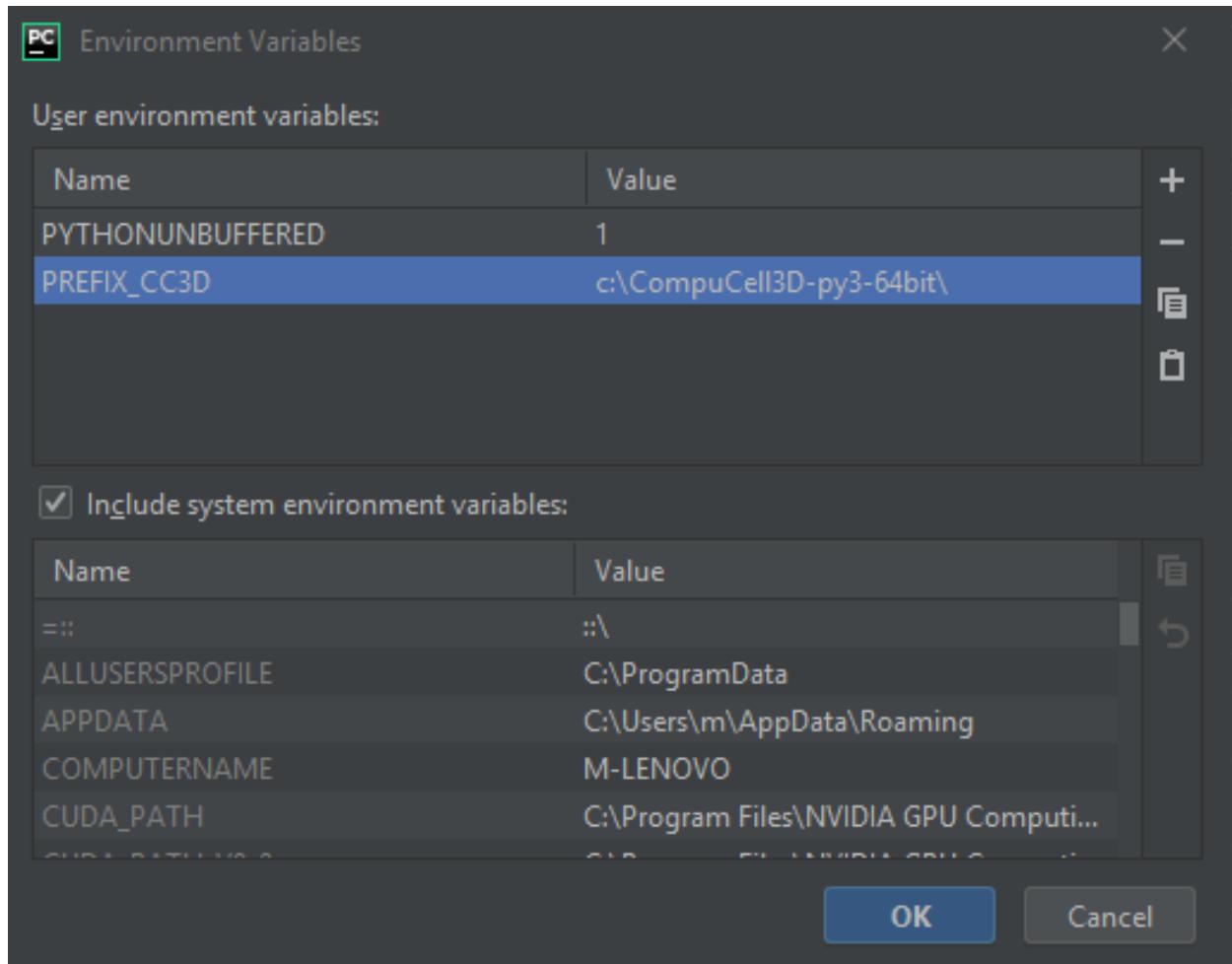
and the following dialog will open up:

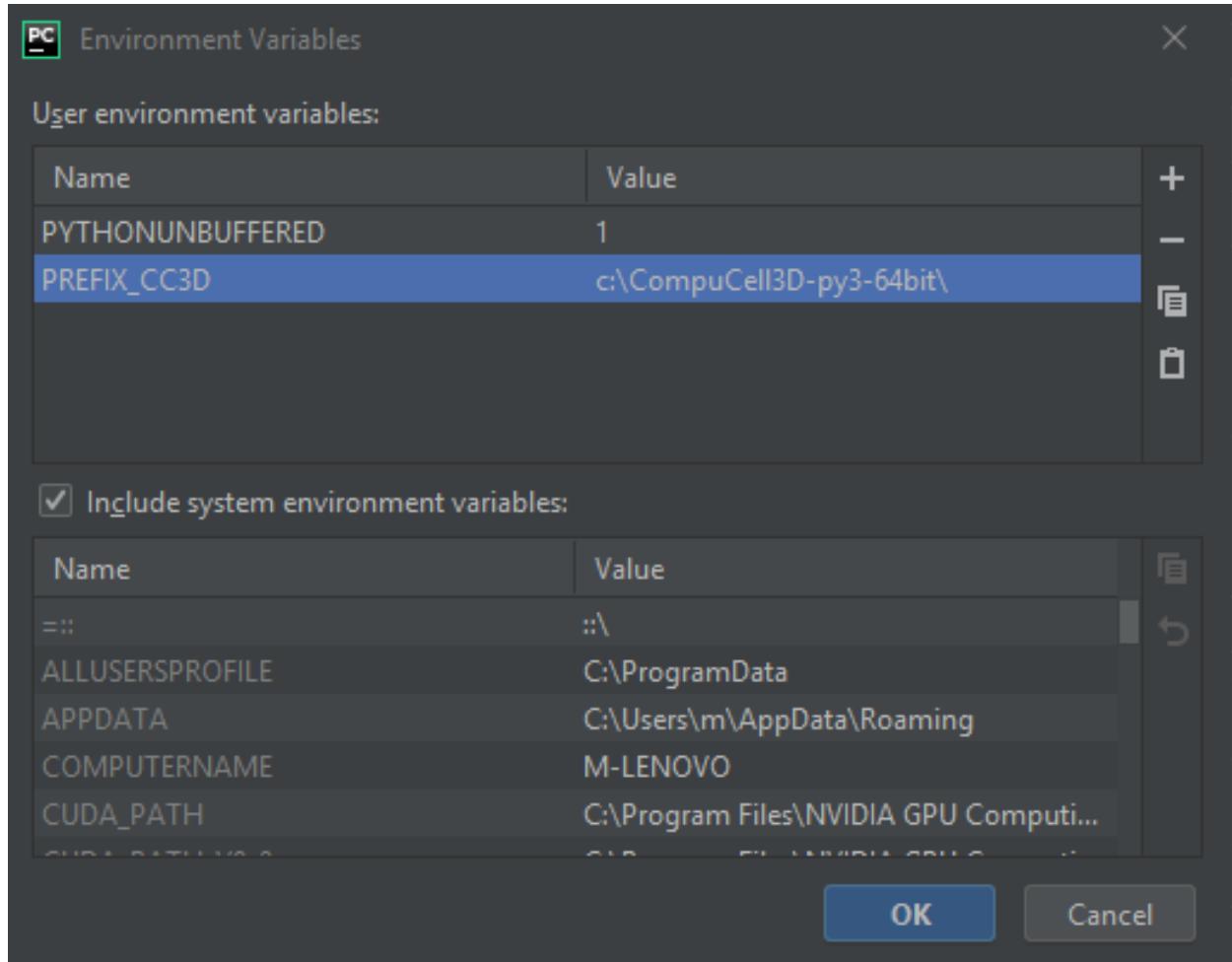
We select Environment Variables pull-down menu by clicking the icon in the right-end of the Environment Variables line and the following dialog will open up:

We click + icon on the right of the dialog and input there PREFIX\_CC3D as the name of the environment variable and c:\CompuCell3D-py3-64bit\ as its value.

We click OK buttons and retry running CC3D again. This time Player should open up:







We are done with configuring PyCharm. This section seem a bit long due to number of screenshots we present but once you perform those tasks 2-3 times they will become a second nature and you will be ready to explore what PyCharm has to offer and it does offer quite a lot. Time for next section

### Configuration of PyCharm on Macs (applies to linux as well)

Most of the steps outlined above apply to configuring PyCharm on OSX however if we do not set up all environment variables we might end up with cryptic looking error like the one below:

```
Traceback (most recent call last):
  File "/Users/j/Demo/CC3D_4.1.0/lib/site-packages/cc3d/player5/compuCell3d.pyw", line 15, in <module>
    from cc3d.player5.UI.UserInterface import UserInterface
  File "/Users/j/Demo/CC3D_4.1.0/lib/site-packages/cc3d/player5/UI/UserInterface.py", line 26, in <module>
    from cc3d.player5.Plugins.ViewManagerPlugins.SimpleTabView import SimpleTabView
  File "/Users/j/Demo/CC3D_4.1.0/lib/site-packages/cc3d/player5/Plugins/ViewManagerPlugins/SimpleTabView.py", line 23, in <module>
    from cc3d.player5.Graphics.GraphicsFrameWidget import GraphicsFrameWidget
  File "/Users/j/Demo/CC3D_4.1.0/lib/site-packages/cc3d/player5/Graphics/GraphicsFrameWidget.py", line 11, in <module>
    from cc3d.core.GraphicsOffScreen.GenericDrawer import GenericDrawer
  File "/Users/j/Demo/CC3D_4.1.0/lib/site-packages/cc3d/core/GraphicsOffScreen/GenericDrawer.py", line 32, in <module>
    from .MVCDrawModel2D import MVCDrawModel2D
  File "/Users/j/Demo/CC3D_4.1.0/lib/site-packages/cc3d/core/GraphicsOffScreen/MVCDrawModel2D.py", line 1, in <module>
    from .MVCDrawModelBase import MVCDrawModelBase
  File "/Users/j/Demo/CC3D_4.1.0/lib/site-packages/cc3d/core/GraphicsOffScreen/MVCDrawModelBase.py", line 3, in <module>
    from cc3d import CompuCellSetup
  File "/Users/j/Demo/CC3D_4.1.0/lib/site-packages/cc3d/CompuCellSetup/_init_.py", line 1, in <module>
    from .utils import *
  File "/Users/j/Demo/CC3D_4.1.0/lib/site-packages/cc3d/CompuCellSetup/utils.py", line 6, in <module>
    from cc3d.core import XMLUtils
  File "/Users/j/Demo/CC3D_4.1.0/lib/site-packages/cc3d/core/XMLUtils.py", line 2, in <module>
    from cc3d.cpp import CC3DXML
  File "/Users/j/Demo/CC3D_4.1.0/lib/site-packages/cc3d/cpp/CC3DXML.py", line 28, in <module>
    _CC3DXML = swig_import_helper()
  File "/Users/j/Demo/CC3D_4.1.0/lib/site-packages/cc3d/cpp/CC3DXML.py", line 24, in swig_import_helper
    _mod = imp.load_module('_CC3DXML', fp, pathname, description)
  File "/Users/j/Demo/CC3D_4.1.0/python37/lib/python3.7/_imp.py", line 242, in load_module
    return load_dynamic(name, filename, file)
  File "/Users/j/Demo/CC3D_4.1.0/python37/lib/python3.7/_imp.py", line 342, in load_dynamic
    return _load(spec)
ImportError: dlopen(/Users/j/Demo/CC3D_4.1.0/lib/site-packages/cc3d/cpp/_CC3DXML.so, 2): Symbol not found: __ZNKSt5ctypeIcE13_M_widen_initEv
  Referenced from: /Users/j/Demo/CC3D_4.1.0/lib/site-packages/cc3d/cpp/lib/libCC3DBasicUtils.dylib
  Expected in: /usr/lib/libstdc++.6.dylib
in /Users/j/Demo/CC3D_4.1.0/lib/site-packages/cc3d/cpp/lib/libCC3DBasicUtils.dylib
```

This happens because besides setting PREFIX\_CC3D we need to set other environment variables within PyCharm. What are those additional environment variables? To answer this question it is best to look inside run script that CC3D shipt with. On OSX we open up in editor compucell3d.command and we see the following code:

```
#!/bin/bash

# echo " "
# echo " dollar-zero AKA the first argument to this .command script is: "
# echo $0
# echo " "
export PYTHON_MINOR_VERSION=
cd "${0%/*}"

# language settings
export LANG=en_EN
export __CF_USER_TEXT_ENCODING=""

export COMPUCELL3D_MAJOR_VERSION=4
export COMPUCELL3D_MINOR_VERSION=0
```

(continues on next page)

(continued from previous page)

```

export COMPUCELL3D_BUILD_VERSION=0

echo " "
echo "-----"
echo " CompuCell3D version ${COMPUCELL3D_MAJOR_VERSION}.${COMPUCELL3D_MINOR_VERSION}."
echo " ${COMPUCELL3D_BUILD_VERSION}"
echo "      (OS X 10.8 x86_64 build)"
echo "-----"

export PREFIX_CC3D=$(pwd)

export PYTHON_EXEC_FILE=${PREFIX_CC3D}/python37/bin/python

export QT_QPA_PLATFORM_PLUGIN_PATH=${PREFIX_CC3D}/python37/plugins

export CC3D_PYTHON_APP=${PREFIX_CC3D}/python37/compuccell3d.app/Contents/MacOS/python

export DYLD_LIBRARY_PATH=${PREFIX_CC3D}/lib:${DYLD_LIBRARY_PATH}

...

```

All the lines that begin with `export` are used to set local environment variables that are necessary to run CC3D. In our case we need to set `PREFIX_CC3D`, `QT_QPA_PLATFORM_PLUGIN_PATH` and `DYLD_LIBRARY_PATH`. For runs within PyCharm we may skip `PYTHON_EXEC_FILE` and `CC3D_PYTHON_APP`.

In my case I had CC3D installed into `/Users/j/Demo/CC3D_4.1.0` and therefore the environment variable configuration screen looks as follows:

where I set :

```

PREFIX_CC3D=/Users/j/Demo/CC3D_4.1.0
QT_QPA_PLATFORM_PLUGIN_PATH=/Users/j/Demo/CC3D_4.1.0/python37/plugins
DYLD_LIBRARY_PATH=/Users/j/Demo/CC3D_4.1.0/lib

```

After making those changes you should be able to open CC3D within PyCharm and start debugging your simulations.

**Linux Users:** You may follow analogous process on your linux machines - simply check in the `compuccell3d.sh` script what are environment variables that are set there and make sure to set them in the PyCharm environment variable panel.

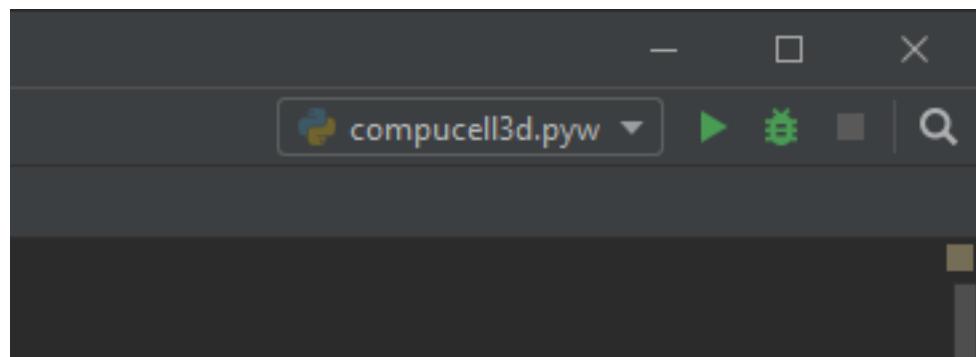
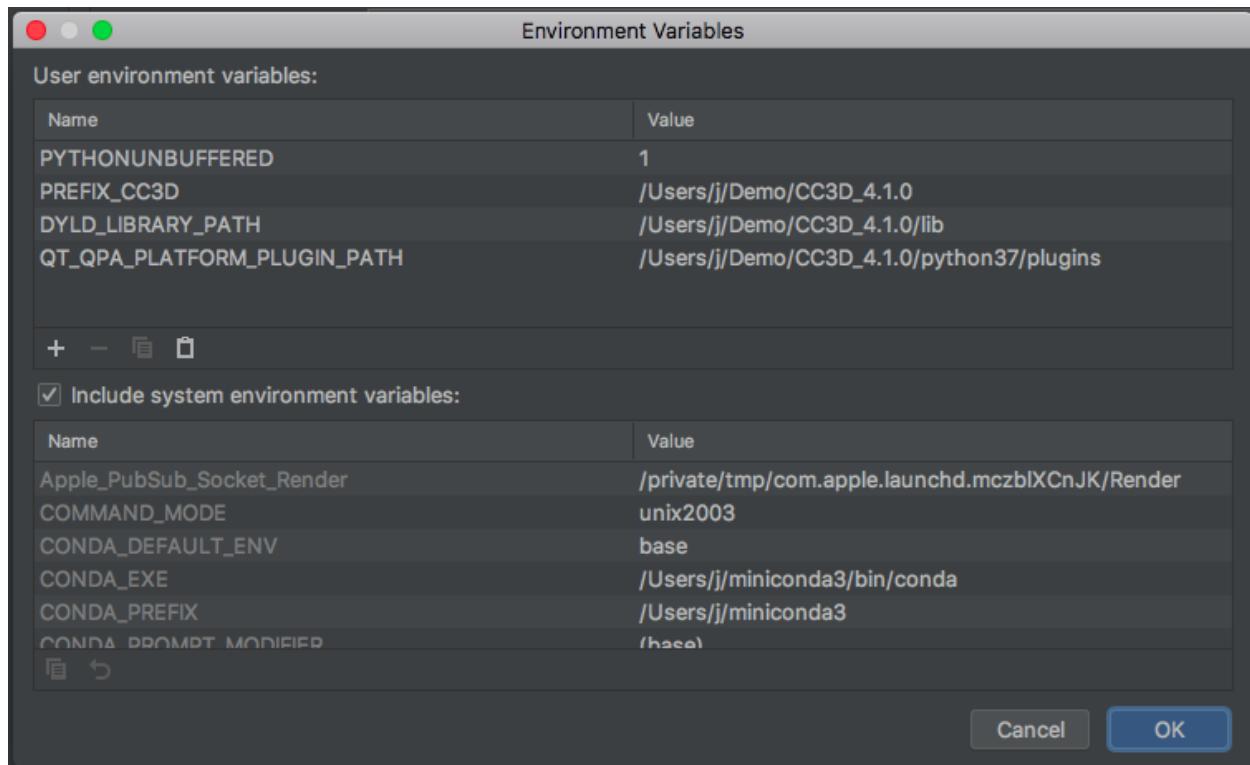
### 2.2.3 Step 3 - Debugging (stepping through) CC3D simulation and exploring other PyCharm features

All the hard work you have done so far will pay off in this section. We will show you how to step through simulation, how to inspect variables, how to fix errors, how to quickly type steppable code using PyCharm syntax completion and autoformat your code. Let us start with debugging first.

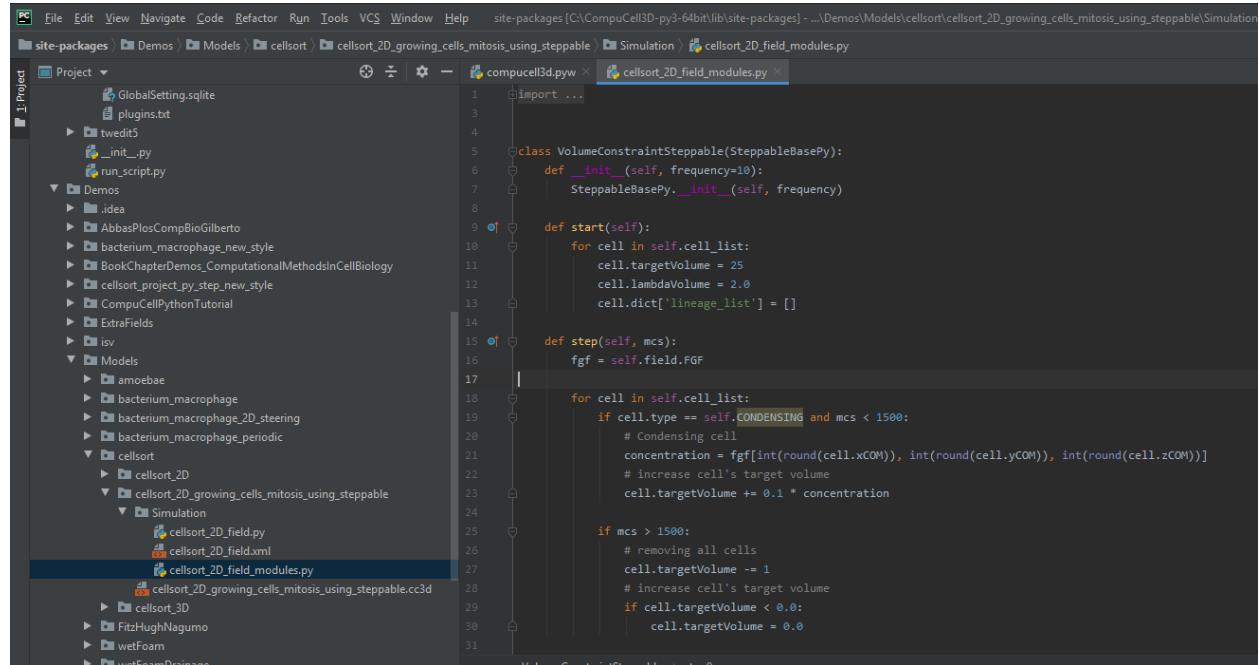
#### Debugging Simulation

To Debug a simulation we open CompuCell3D in the debug mode by clicking `Debug` located to the right of the `Run` button:

The player will open up. You may start the simulation by pressing `Step` button on the player. While the simulation is running we would like to inspect actual variable inside Python steppable. To do so we open up a simulation script we



want to debug. In my case I will open simulation in `c:\CompuCell3D-py3-64bit\lib\site-packages\site-packages [C:\CompuCell3D-py3-64bit\lib\site-packages] - ...\\Models\\cellsort\\cellsort_2D_growing_cells_mitosis_using_stoppable\\` and in particular I would like to step through every single line of the steppable. So I open the steppable `c:\CompuCell3D-py3-64bit\lib\site-packages\site-packages [C:\CompuCell3D-py3-64bit\lib\site-packages] - ...\\Models\\cellsort\\cellsort_2D_growing_cells_mitosis_using_stoppable\\Simulation\\cellsort_2D_field_modules.py` in PyCharm editor.



```

File Edit View Navigate Code Refactor Run Tools VCS Window Help site-packages [C:\CompuCell3D-py3-64bit\lib\site-packages] - ...\\Models\\cellsort\\cellsort_2D_growing_cells_mitosis_using_stoppable\\Simulation\\cellsort_2D_field_modules.py
Project GlobalSetting.sqlite
plugins.txt
twedit5
__init__.py
run_script.py
Demos .idea
AbbasPlosCompBioGilberto
bacterium_macrophage_new_style
BookChapterDemos_ComputationalMethodsInCellBiology
cellsort_project_py_step_new_style
CompuCellPythonTutorial
ExtraFields
isv
Models amoeba
bacterium_macrophage
bacterium_macrophage_2D_steering
bacterium_macrophage_periodic
cellsort
cellsort_2D
cellsort_2D_growing_cells_mitosis_using_stoppable
Simulation
cellsort_2D_field.py
cellsort_2D_field.xml
cellsort_2D_field_modules.py
cellsort_2D_growing_cells_mitosis_using_stoppable.cc3d
cellsort_3D
FitzHughNagumo
wetFoam
wetFoamDrainage
compcell3d.pyw
cellsort_2D_field_modules.py
1 import ...
3
4
5 class VolumeConstraintSteppable(SteppableBasePy):
6     def __init__(self, frequency=10):
7         SteppableBasePy.__init__(self, frequency)
8
9     def start(self):
10        for cell in self.cell_list:
11            cell.targetVolume = 25
12            cell.lambdaVolume = 2.0
13            cell.dict['lineage_list'] = []
14
15    def step(self, mcs):
16        fgf = self.field.FGF
17
18        for cell in self.cell_list:
19            if cell.type == self.CONDENSING and mcs < 1500:
20                # Condensing cell
21                concentration = fgf[int(round(cell.xCOM)), int(round(cell.yCOM)), int(round(cell.zCOM))]
22                # increase cell's target volume
23                cell.targetVolume += 0.1 * concentration
24
25            if mcs > 1500:
26                # removing all cells
27                cell.targetVolume -= 1
28                # increase cell's target volume
29                if cell.targetVolume < 0.0:
30                    cell.targetVolume = 0.0
31
VolumeConstraintSteppable.step()

```

Next, we put a breakpoint (red circle) by clicking on the left margin of the editor. Breakpoint is the place in the code where the debugger will stop execution of the code and give you options to examine variables of the simulation:

After we places our breakpoint(s) let's hit Step button on the player. The execution of the code will resume and will be stopped exactly at teh place where we placed our breakpoint. The debug console will open up in the PyCharm (see bottom panel) and the blue line across editor line next to red circle indicates current position of code execution:

Once the code is stopped we typically want to inspect values of variaables. To do so we open “Evaluate Expression” by either clicking the icon or using keyboard shortcut (Alt+F8). note that keyboard shortcuts can be different on different operating systems:

Once Evaluate Expression window opens up you can evaluate variables in the current code frame. Let us evaluate the content of `cell` variable by typing `cell` in the line of the Evaluate Expression window:

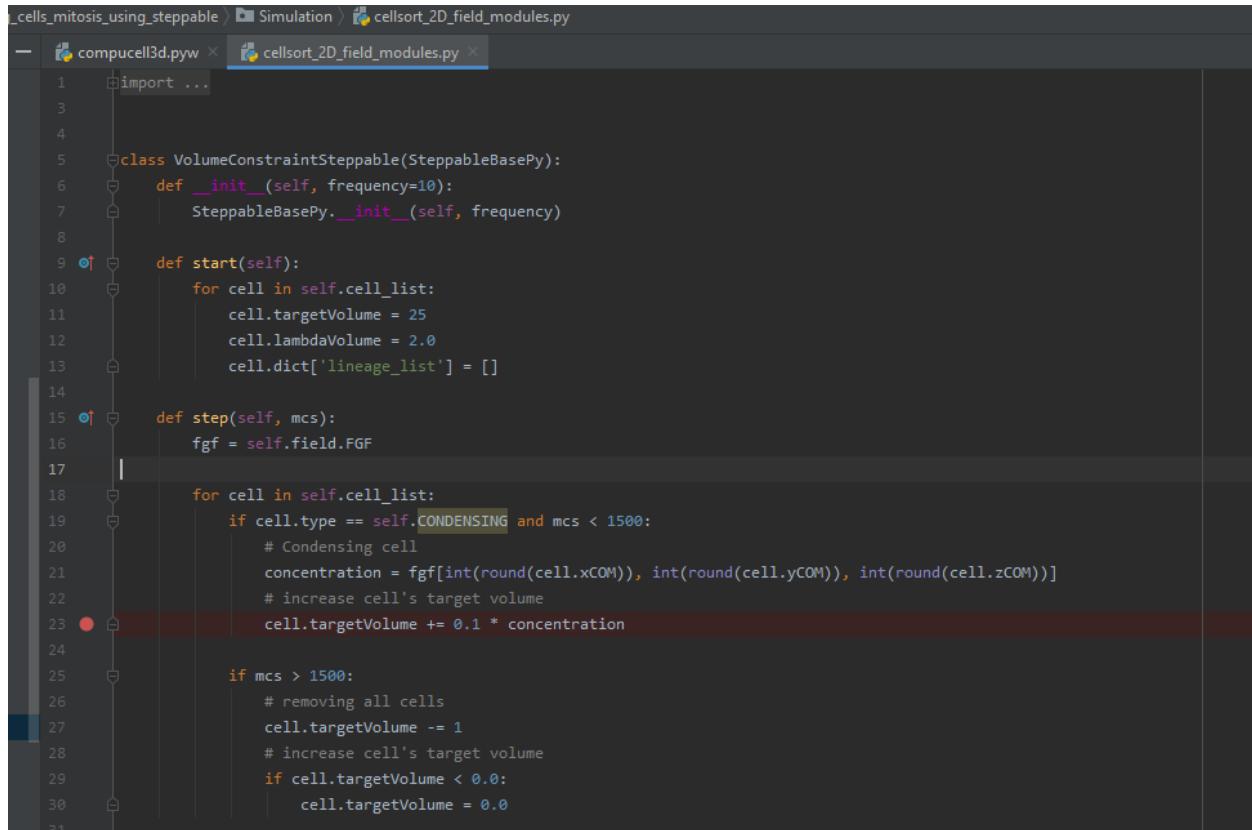
As you can see this displays attributes of `cell` object and we can inspect every single attribute of this particular `cell` object:

We can advance code execution by one line by hitting F8 or clicking Step Over from the debug menu. This will advance us to the next line of steppable. At this point we may open second Evaluate Expression window and this time type `concentration` to check the value of `concentration` variable and in another window we type `cell.targetVolume + 0.1 * concentration` to show that not only we can check values of single variables but also evaluate full expressions:

A very important feature of a breakpoint is the ability to enable them if certain condition is met. For example we want to break when `concentration` is greater than 0.5. To do so we right-click on the breakpoint red-circle

and in the line below we enter `concentration > 0.5` and click Done :

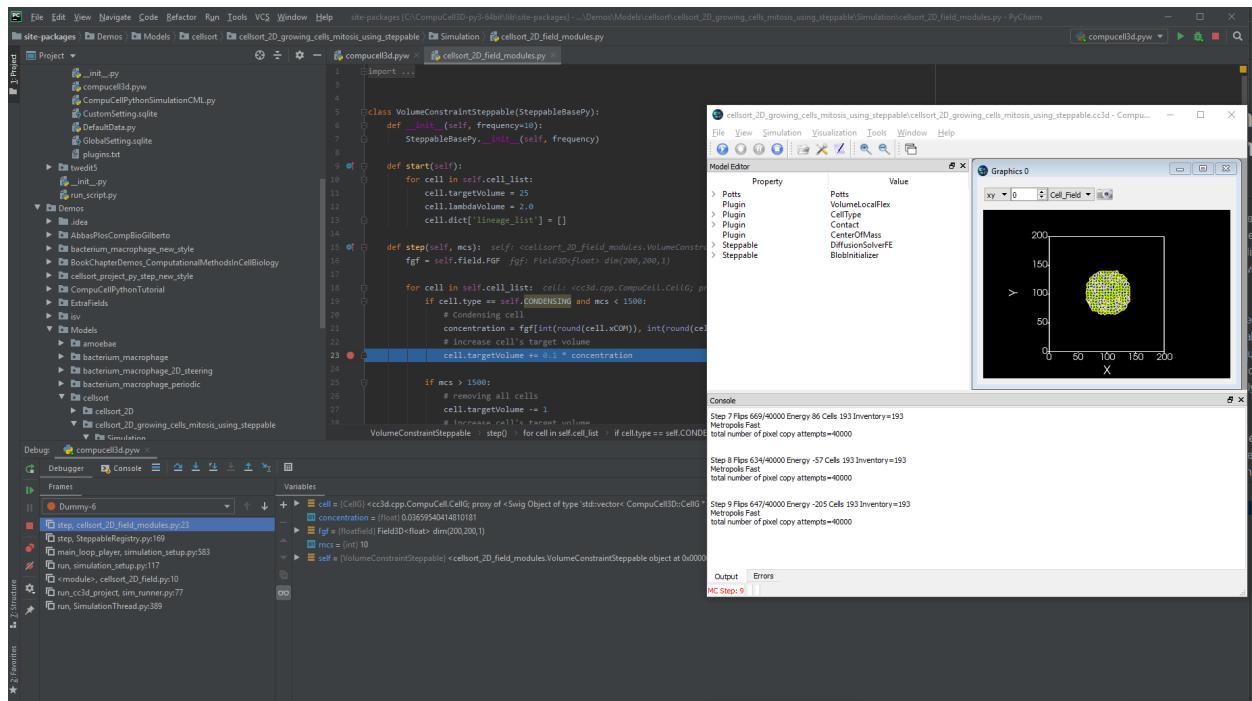
Next we resume stopped program by clicking Resume program in the lower-left corner:



```

cells_mitosis_using_steppable > Simulation > cellsort_2D_field_modules.py
compucell3d.pyw < cellsort_2D_field_modules.py >
1 import ...
2
3
4
5 class VolumeConstraintSteppable(SteppableBasePy):
6     def __init__(self, frequency=10):
7         SteppableBasePy.__init__(self, frequency)
8
9     def start(self):
10        for cell in self.cell_list:
11            cell.targetVolume = 25
12            cell.lambdaVolume = 2.0
13            cell.dict['lineage_list'] = []
14
15     def step(self, mcs):
16         fgf = self.field.FGF
17
18         for cell in self.cell_list:
19             if cell.type == self.CONDENSING and mcs < 1500:
20                 # Condensing cell
21                 concentration = fgf[int(round(cell.xCOM)), int(round(cell.yCOM)), int(round(cell.zCOM))]
22                 # increase cell's target volume
23                 cell.targetVolume += 0.1 * concentration
24
25             if mcs > 1500:
26                 # removing all cells
27                 cell.targetVolume -= 1
28                 # increase cell's target volume
29                 if cell.targetVolume < 0.0:
30                     cell.targetVolume = 0.0
31

```

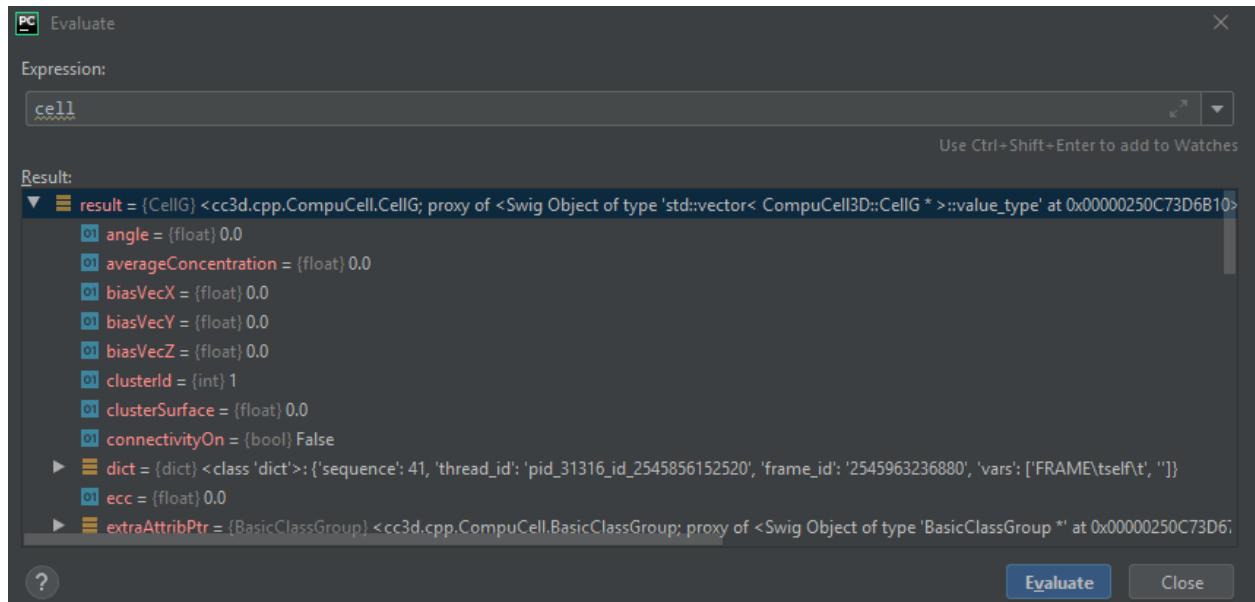
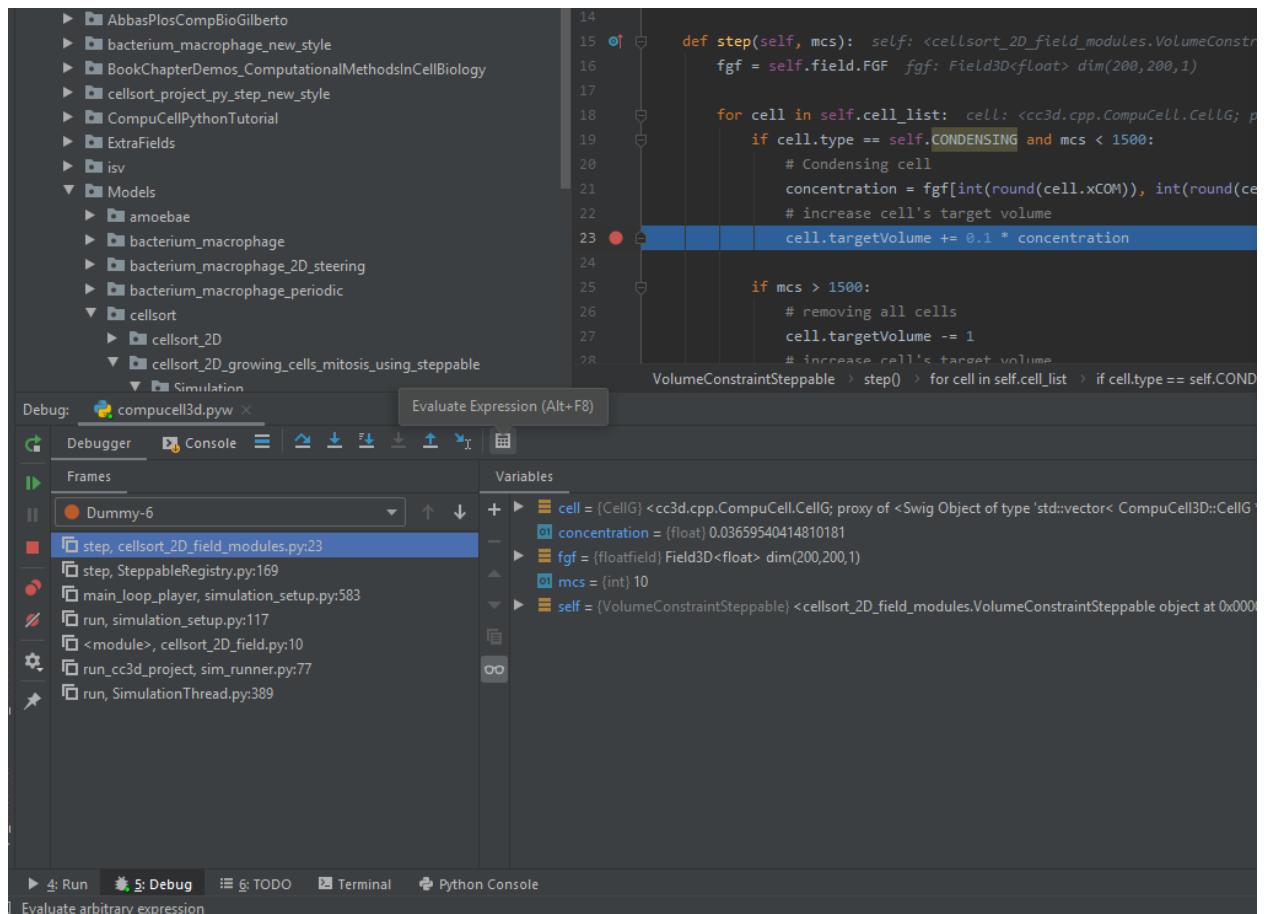


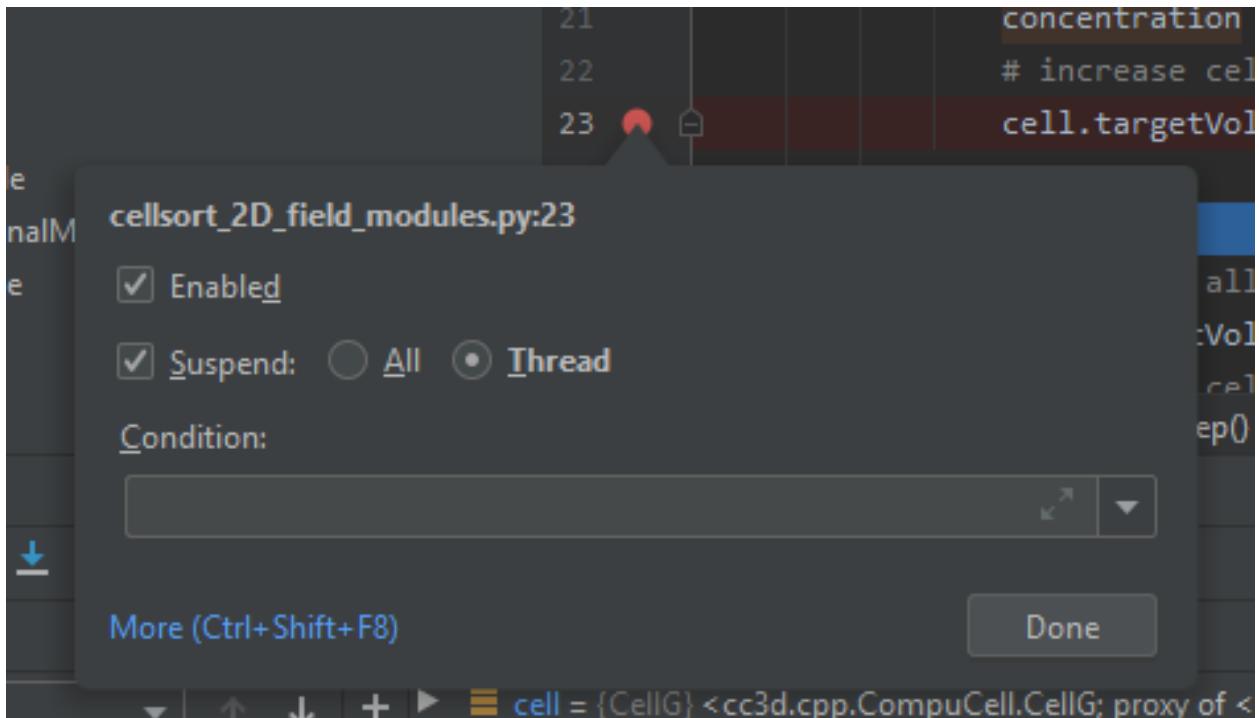
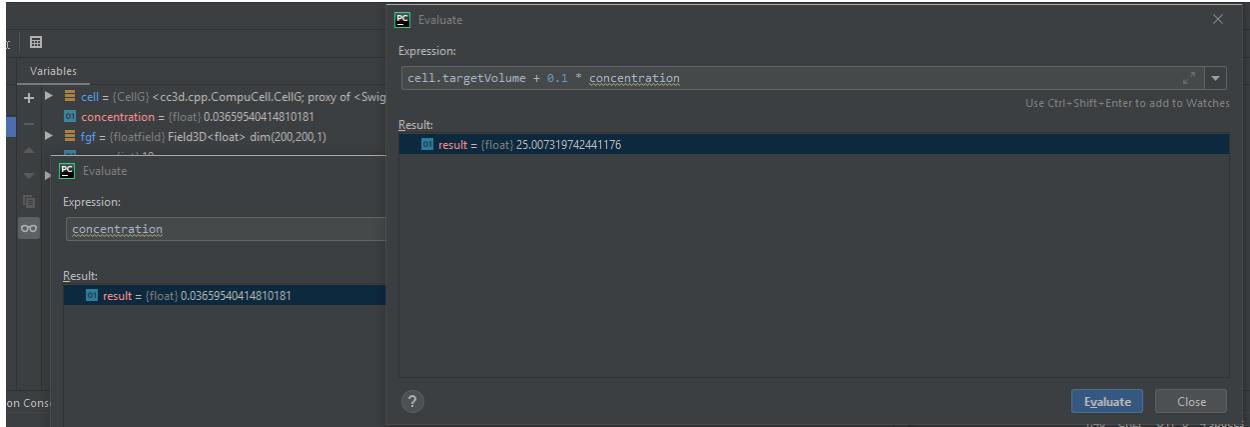
The screenshot shows the PyCharm IDE interface with several windows open:

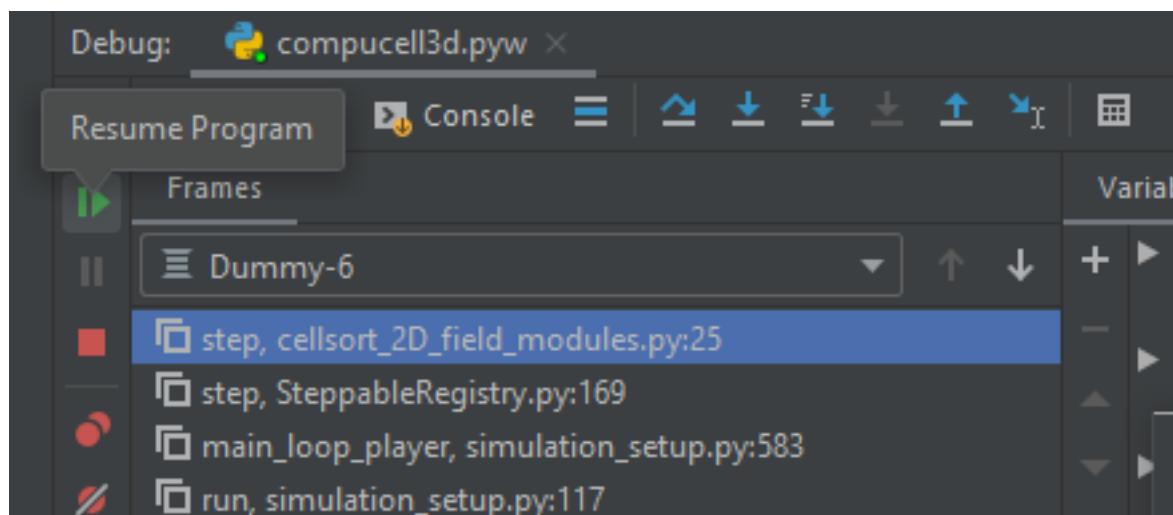
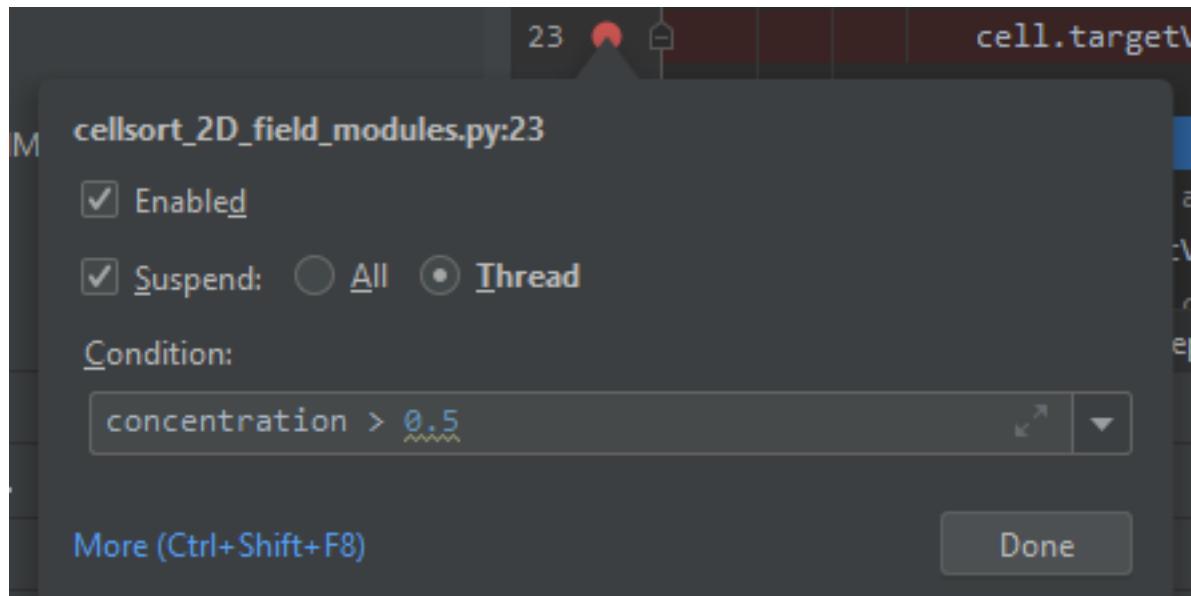
- Project View:** Shows the project structure with files like `compucell3d.pyw`, `cellsort_2D_field_modules.py`, and various configuration files.
- Code Editor:** Displays the same `VolumeConstraintSteppable` class code as above, with a red dot indicating the current line of execution (line 23).
- Model Editor:** A floating window showing properties for a `VolumeConstraintSteppable` object, including `Potts`, `Plugin`, `CellType`, `CellRadius`, `CenterOfMass`, `DiffusionSolverFE`, and `BlobInitializer`.
- Graphics 0:** A visualization panel showing a 2D grid with a yellow circular cluster of points centered around (100, 100), representing cell positions.
- Console:** Displays simulation logs, including step counts, energy levels, and pixel copy attempts.
- Frames:** A stack trace showing the call hierarchy, with the top frame being `Dummy-6`.
- Variables:** A list of variables and their values, such as `cell`, `concentration`, `fgf`, `mcs`, and `self`.
- Output and Errors:** Panels showing the standard output and error streams.

The screenshot shows a debugger interface with the following details:

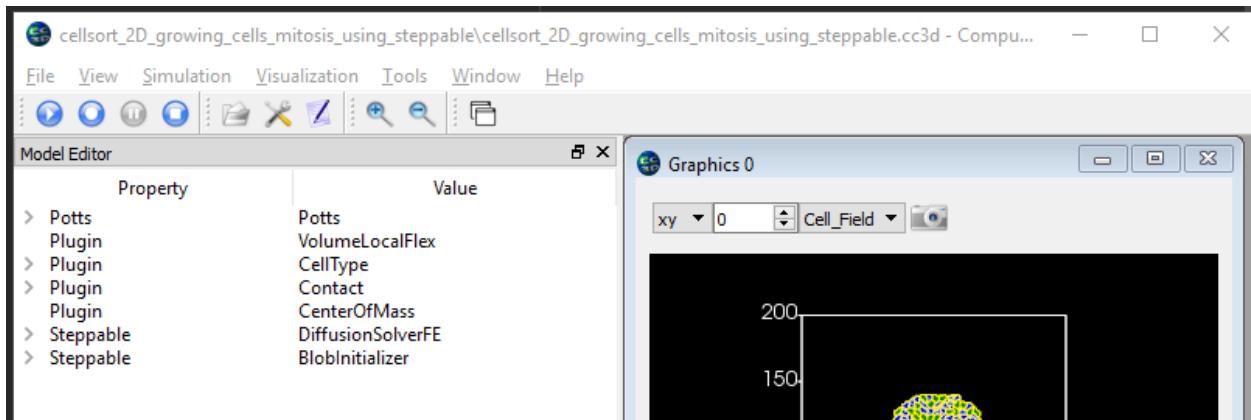
- Project Tree:** Shows various project files and modules, including `AbbasPlosCompBioGilberto`, `bacterium_macrophage_new_style`, `BookChapterDemos_ComputationalMethodsInCellBiology`, `cellsort_project_py_step_new_style`, `CompuCellPythonTutorial`, `ExtraFields`, `isv`, `Models` (with sub-modules `amoebae`, `bacterium_macrophage`, `bacterium_macrophage_2D_steer`, `bacterium_macrophage_periodic`), `cellsort`, `cellsort_2D`, and `cellsort_2D_growing_cells_mitosis_using_steppable`.
- Code Editor:** Displays Python code for a `VolumeConstraintSteppable` class. The current line is `cell.targetVolume += 0.1 * concentration`. A red dot marks the current position in the code.
- Variables View:** Shows local variables:
  - `cell`: A proxy object of type `<Swig Object of type 'std::vector< CompuCell3D::CellG>'>`.
  - `concentration`: A float value of `0.03659540414810181`.
  - `fgf`: A `Field3D<float>` object of size `(200,200,1)`.
  - `mcs`: An integer value of `10`.
  - `self`: A `VolumeConstraintSteppable` object.
- Stack Trace:** Shows the call stack from bottom to top:
  - `VolumeConstraintSteppable > step0 > for cell in self.cell_list > if cell.type == self.CONDENSING`
  - `if mcs > 1500:`
  - `# removing all cells`
  - `cell.targetVolume -= 1`
  - `# increase cell's target volume`
  - `if cell.type == self.CONDENSING`
  - `for cell in self.cell_list`
  - `if cell.type == self.CONDENSING and mcs < 1500:`
  - `# Condensing cell`
  - `concentration = fgf[int(round(cell.xCOM)), int(round(cell.yCOM))]`
  - `# increase cell's target volume`
  - `cell.targetVolume += 0.1 * concentration`
  - `if mcs > 1500:`
  - `# removing all cells`
  - `cell.targetVolume -= 1`
  - `# increase cell's target volume`
- Bottom Bar:** Includes buttons for Run, Debug, TODO, Terminal, and Python Console, along with an "Evaluate arbitrary expression" input field.



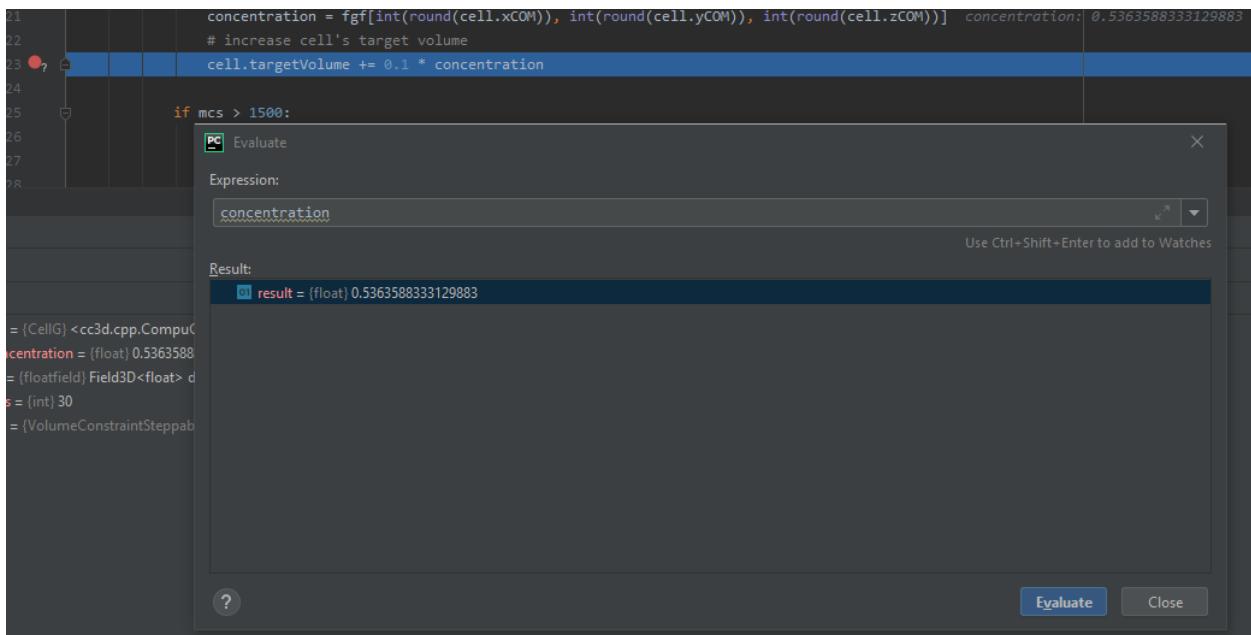




and we also need to press `Play`` on the Player because the Player code is resumed but the simulation may still be paused in the Player so by pressing ```Play` on the player we will resume it.



After a brief moment the PyCharm debugger will pause the execution of the program and if we inspect the value of the `concentration` variable we will see that indeed its value is greater than 0.5:



This technique of adding conditional breakpoints is quite useful when debugging simulations. If you have a lot of cells you do not want to step every single line of the loop by hitting F8. you want to press `Play` on the player and then have debugger inspect stop condition and stop once the condition has been satisfied.

This is a brief introduction and tutorial for using PyCharm debugger with CC3D simulation. There is more to debugging but we will not cover it here. You can find more complete PyCharm Debugging tutorial here: <https://www.jetbrains.com/help/pycharm/debugging-your-first-python-application.html>

## 2.2.4 Step 4 - writing steppable code with PyCharm code auto completion

While debugging features provide a strong argument for using this IDE in CC3D development, “regular” users can also benefit a lot by using code auto-completion capabilities. So far we have been showing fairly advanced features but what if you just want to write CC3D steppable and run your simulation. PyCharm provides excellent auto-completion capabilities. To motivate why this feature is useful, imagine a simple example where you are inside a steppable that you are writing and would like to add function that creates new cell. In Twedit++ we know that in such situation we go to CC3D Python menu and search for appropriate function. PyCharm offers actually on-line auto-completion based on available modules that are installed in the configured Python environment. This is precisely why we spent a little bit of time at the beginning of this chapter setting up PyCharm, in particular, setting Python environment. Let us come back to our example of adding new cell. We suspect that a function that adds new cell has a word “new” and “cell” in it. We will use this knowledge and start typing `self.cell` in the Steppable editor we will get a pop-up selectable options for the most closely matched function candidates, a function we are looking for is `self.new_cell` and is listed somewhere in the middle:

The screenshot shows a PyCharm interface with the file `cellsrt_2D_field_modules.py` open. The cursor is positioned at the start of a new function definition. A code completion dropdown is open, listing several methods starting with `new_`. At the top of this list is `new_cell`, which is highlighted. Other visible methods include `cell_list_by_type`, `cell_list`, `cell_field`, `cellField`, `cellList`, and `cellListByType`. The PyCharm interface includes toolbars, a status bar, and a bottom navigation bar.

```

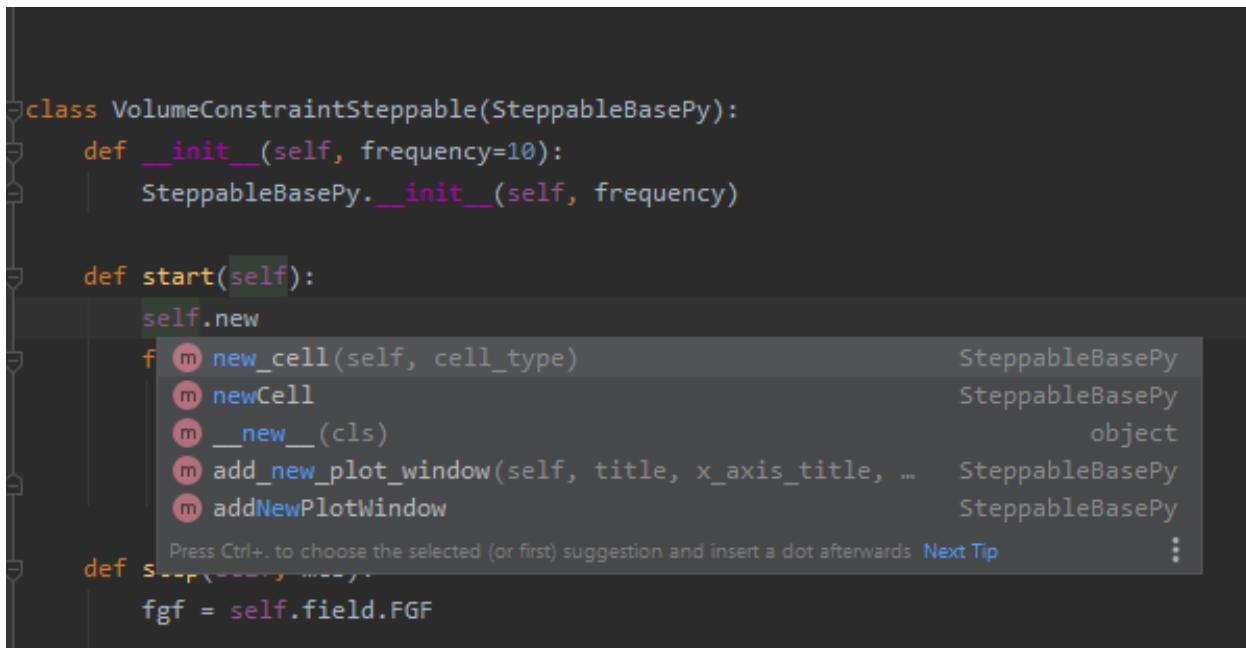
1 import ...
3
4
5 class VolumeConstraintSteppable(SteppableBasePy):
6     def __init__(self, frequency=10):
7         SteppableBasePy.__init__(self, frequency)
8
9     def start(self):
10        self.cell
11        f cell_list_by_type
12        f cell_list
13        f cell_field
14        f cellField
15        f cellList
16        f cellListByType
17        m new_cell(self, cell_type)
18        m add_sbml_to_cell(self, model_file, model_name,...)
19        m delete_cell(self, cell)
20        m delete_sbml_from_cell(self, model_name, cell)
21        m move_cell(self, cell, shift_vector)
22        m set_stop_size_for_cell(self, model_name, cell)
23        # concentration = int(round(cell.xConc)), int(round(cell.yConc)), int(round(cell.zConc))
24        # increase cell's target volume
25        cell.targetVolume += 0.1 * concentration
26

```

When we start typing `self.new` we will get different ordering of candidate functions with `self.new_cell` listed at the top of the list:

Finally when we select this `self.new_cell` option from the pop-up list PyCharm will also display a signature of the function:

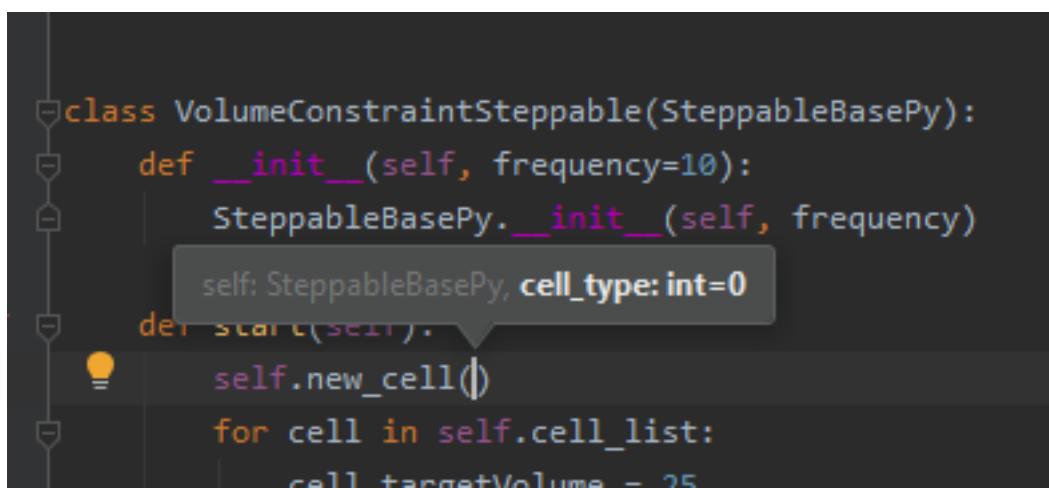
The auto-completion pop-up lists have also another benefit. They allow you to check out what other functions are available and if you see something interesting you can always lookup documentation to see if indeed this function matches your needs. Most importantly you can always suggest additional functions to be added to the steppables. The best way to do it is to open up a ticket at <https://github.com/CompuCell3D/CompuCell3D/issues>. All you need is a GitHub account (those are free) and you are ready to be part of CC3D development team.



```
class VolumeConstraintSteppable(SteppableBasePy):
    def __init__(self, frequency=10):
        SteppableBasePy.__init__(self, frequency)

    def start(self):
        self.new
        f m new_cell(self, cell_type) SteppableBasePy
        m newCell SteppableBasePy
        m __new__(cls) object
        m add_new_plot_window(self, title, x_axis_title, ...) SteppableBasePy
        m addNewPlotWindow SteppableBasePy

    def s Press Ctrl+ to choose the selected (or first) suggestion and insert a dot afterwards Next Tip
        ---r-----y-----y-----
        fgf = self.field.FGF
```



```
class VolumeConstraintSteppable(SteppableBasePy):
    def __init__(self, frequency=10):
        SteppableBasePy.__init__(self, frequency)
        self: SteppableBasePy, cell_type: int=0
    def start(self):
        self.new_cell()
        for cell in self.cell_list:
            cell.targetVolume = 25
```

## 2.2.5 Perspective

In this chapter we presented PyCharm features that make it an ideal IDE for CC3D code and simulation development. The question that you may have at this point is what is the role of Twedit++. Clearly, if we could port all Twedit++ wizards and helpers to PyCharm would probably be recommending using PyCharm. However, for the time being Twedit++ still offers a lot of time-saving tools. It can generate a template of functional simulation, it can generate C++ plugins and steppables (if you are working at the C++ level), it provides XML and Python helpers and overall it is a functional , rudimentary programmer's editor. We think that it is best to combine Twedit++ and PyCharm when you are developing your simulation. Ideally you would create simulation in Twedit++, you could manage .cc3d project in Twedit++ but when you want nice syntax auto-completion, and debugging capabilities you would switch to PyCharm. Obviously, you can have the two tools open at the same time and choose features from any of them that best fit your programing style.

## 2.3 What is a Steppable? (SteppableBasePy class)

SteppableBasePy has built-in **functions that are called automatically** during the simulation. The most important functions are `start` and `step`.

### 2.3.1 Functions

**def \_\_init\_\_(self, frequency=1):** This code runs as the simulation is set up, and, in most cases, you will not need to edit it.

**def start(self):** This is called after cells are created but before the simulation starts, so use it to assign custom cell properties or create `plots`.

**def step(self, mcs):** Almost everything will happen here. For example, you might grow, divide, or kill your cells here.

**def on\_stop(self):** This runs when you click the stop button.

**def finish(self):** This function is called at the end of the simulation, but it is used very infrequently. Be careful: it will only run if the simulation reaches the maximum number of steps as specified by the XML attribute `<Steps>`.

A very common line in a Python steppable will read:

```
for cell in self.cell_list:
```

`cell_list` is a variable of the `SteppableBasePy` class, which means that every steppable you create will automatically store every cell it creates in that list.

All steppables in CompuCell3D are extensions of `SteppableBasePy`, so they, too, can use `cell_list`.

**If you're new to programming:** The word `self` refers to the class we're inside, which would the steppable. Please see chapters on classes and inheritance from any Python manual if this looks unfamiliar. Also, if you want a full list of the cell attributes, see [Appendix B](#).

**If you're a programmer:** Under the hood, the `self.cell_list` is a handle, or a “pointer”, to the C++ object that stores all cells in the simulation. The content of cell inventory and cell ordering of cells there is fully managed by C++ code. You can easily see what member variables the C++ cell object has by calling `dir(cell)` with one of the cells from the `self.cell_list` or by checking out [Appendix B](#).

#### Note

Old syntax like `self.cellList` is still supported.

## 2.4 Adding Steppable to Simulation using Twedit++

In the CC3D Project Panel of Twedit++, right-click on a Steppable Python file and choose the ‘Add Steppable’ option:

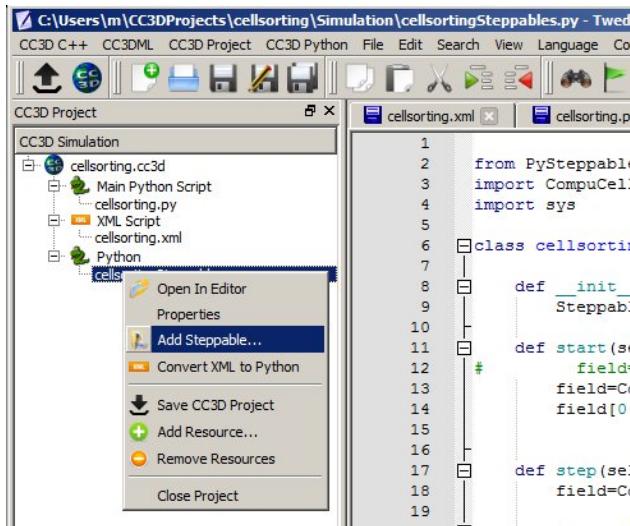


Figure 8 Adding steppable using Twedit++

A dialog will pop up where you can specify the name and type of the new Steppable as well as its call frequency. Click OK, and a new steppable gets added to your code.

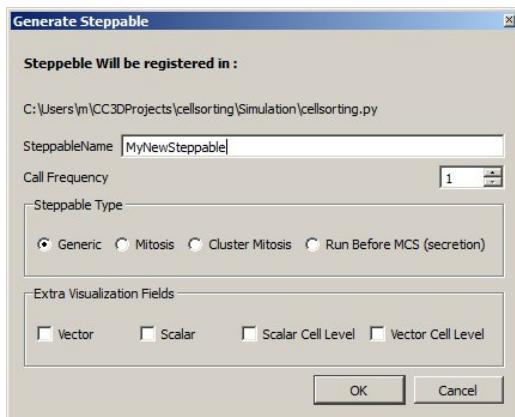


Figure 9 Configuring basic steppable properties in Twedit++.

Notice that Twedit++ takes care of adding steppable registration code in the main Python script:

```
from cellsoringSteppables import MyNewSteppable
CompuCellSetup.register_steppable(steppable=MyNewSteppable(freqency=1))
```

## 2.5 Iterating over cell neighbors

We have already learned how to iterate over cells in the simulation. Quite often in the multi-cell simulations there is a need to visit neighbors of a single cell. We define a neighbor as an adjacent cell which has common surface area with the cell in mind. To enable neighbor tracking you have to include NeighborTracker plugin in the XML or in Python code which replaces XML. For details see [CompuCellPythonTutorial/NeighborTracker](#) example. Take a look at the implementation of the step function where we visit cell neighbors:

```

from cc3d.core.PySteppables import *

class NeighborTrackerPrinterSteppable(SteppableBasePy):
    def __init__(self, frequency=100):
        SteppableBasePy.__init__(self, frequency)

    def step(self, mcs):

        for cell in self.cell_list:

            for neighbor, common_surface_area in self.get_cell_neighbor_data_list(cell):
                if neighbor:
                    print("neighbor.id", neighbor.id, " common_surface_area=", common_
-surface_area)
                else:
                    print("Medium common_surface_area=", common_surface_area)

```

In the outer for loop we iterate over all cells. During each iteration this loop picks a single cell. For each such cell we construct the inner loop where we access a list of cell neighbors:

```
for neighbor, common_surface_area in self.get_cell_neighbor_data_list(cell):
```

Notice that during each iteration loop Python returns two objects: neighbor and common surface area. neighbor points to a cell object that has nonzero common surface area with the cell from the outer loop. It can happen that the neighbor object returned by the inner loop is None. This means that this particular cell from the outer loop touches Medium. Take a look at the if-else statement in the example code above. If you want to paste neighbor iteration code template into your simulation go to CC3D Python->Visit->Cell Neighbors in Twedit++.

If you are puzzled why loop above has two variables after for it is because self.get\_cell\_neighbor\_data\_list(cell) object when iterated over will return tuples of two objects. Let's do an experiment:

```

for neighbor_tuple in self.get_cell_neighbor_data_list(cell):
    print neighbor_tuple
    if neighbor_tuple[0]:
        print('Cell id = ', neighbor_tuple[0].id)
    else:
        print('Got Medium Cell ')
    print('Common Surface Area = ', neighbor_tuple[1])

```

The output will be:

```

neighbor_tuple= (<CompuCell.CellG; proxy of <Swig Object of type 'std::vector<
-CompuCell3D::CellG * >::value_type' at 0x00000000007388EA0> >, 5)
Cell id = 11
Common Surface Area = 5
neighbor_tuple= (None, 4)
Got Medium Cell
Common Surface Area = 4

```

Now you can see neighbor\_tuple is indeed an object that has two components. First one neighbor\_tuple[0] points to cell object, second one neighbor\_tuple[1] is a common surface area.

In general, when Python iterates over a list-like object that returns tuples you have two choices how to write the for

loop. You can either use

```
for neighbor_tuple in self.get_cell_neighbor_data_list(cell):
    print(neighbor_tuple[0], neighbor_tuple[1])
```

and refer to the elements of the returned tuple using indices or you can be more explicit and unpack the tuple directly into two variables and access them by different “names”:

```
for neighbor, common_surface_area in self.get_cell_neighbor_data_list(cell):
    print neighbor, common_surface_area
```

## 2.5.1 Neighbor Iteration Helpers

In addition to a plain-vanilla iteration over neighbors the CellNeighborDataList object that you get using `self.get_cell_neighbor_data_list(cell)` has few useful tools that summarize properties of cell neighbors.

## 2.5.2 Common Surface Area With Cells of Given Types

Sometimes we are interested in a common surface area of a given cell with ALL neighbors that are of specific type. CellNeighborDataList has a convenience function `common_surface_area_with_cell_types` that computes it. Here is an example

```
for cell in self.cell_list:
    neighbor_list = self.get_cell_neighbor_data_list(cell)
    common_area_with_types = neighbor_list.common_surface_area_with_cell_types(cell_type_
    ↪list=[1, 2])
    print 'Common surface of cell.id={} with cells of types [1,2] = {}'.format(cell.id,_
    ↪common_area_with_types)
```

The example output is:

```
Common surface of cell.id=10 with cells of types [1,2] = 24
Common surface of cell.id=11 with cells of types [1,2] = 22
```

As you can see `common_surface_area_with_cell_types` returns a number that is a total common surface area of a given cell with other cells of the type that you specify as argument to `common_surface_area_with_cell_types` function as shown above

## 2.5.3 Common Surface Area With Cells of a Given Type - Detailed View

If you want to break the above common surface area by cell types. i.e. you want to know what was the common surface area with cells of type 1, what was the common surface area with cells of type 2, etc..., you want to use `neighbor_list.common_surface_area_by_type()` call :

```
for cell in self.cellList:
    neighbor_list = self.get_cell_neighbor_data_list(cell)
    common_area_with_types = neighbor_list.common_surface_area_with_cell_types(cell_type_
    ↪list=[1, 2])
    print 'Common surface of cell.id={} with cells of types [1,2] = {}'.format(cell.id,_
    ↪common_area_with_types)

    common_area_by_type_dict = neighbor_list.common_surface_area_by_type()
    print 'Common surface of cell.id={} with neighbors \ndetails {}'.format(cell.id,_
    ↪common_area_by_type_dict)
```

The output may look as follows:

```
Common surface of cell.id=10 with cells of types [1,2] = 20
Common surface of cell.id=10 with neighbors
details defaultdict(<type 'int'>, {1L: 15, 2L: 5})

Common surface of cell.id=11 with cells of types [1,2] = 24
Common surface of cell.id=11 with neighbors
details defaultdict(<type 'int'>, {1L: 15, 2L: 9})
```

For cell with `id=10` we have that the total common surface area with cell types 1 and 2 is 20 and if we “zoom-in” we can see that cell with `id=10` had common surface area of 15 with cell of types 1 and 5 with cells of type 2. The two contact areas by type add up to 20 as expected because this particular cell is in contact only with cells of type 1 and 2.

Similar thinking explains common surface areas for cell 11.

A more interesting thing is to look at cell with `id==`. In this particular simulation this cell was in contact with `Medium` and the output looks as follows:

```
Common surface of cell.id=1 with cells of types [1,2] = 12
Common surface of cell.id=1 with neighbors
details defaultdict(<type 'int'>, {0: 10, 1L: 1, 2L: 11})
```

Now you see that the overlap with cells of type 0, 1, 2 was 10, 1, 11 and this does not add up to 12 - the total contact area between cell with `id=1` and cells of type 1 and 2. However if we replaced

```
common_area_with_types = neighbor_list.common_surface_area_with_cell_types(cell_type_
↪list=[1, 2])
```

with

```
common_area_with_types = neighbor_list.common_surface_area_with_cell_types(cell_type_
↪list=[0, 1, 2])
```

all the surfaced areas for cell with `id=1` would add up as they did for cells with `id=10`

## 2.5.4 Counting Neighbors of Particular Type

If you want to know how many neighbors of a given type a given cell has you can do “manual” iteration of all neighbors and keep track of how many of them were of a particular type or you can use a convenience function `neighbor_count_by_type`. `neighbor_count_by_type` will return a dicitorary where the key is a type to the neighbor and the value is how many neighbors of this type are in contact with a given cell

Here is an example:

```
for cell in self.cell_list:
    neighbor_list = self.get_cell_neighbor_data_list(cell)
    neighbor_count_by_type_dict = neighbor_list.neighbor_count_by_type()
    print 'Neighbor count for cell.id={} is {}'.format(cell.id, neighbor_count_by_type_
↪dict)
```

and the output is:

```
Neighbor count for cell.id=1 is defaultdict(<type 'int'>, {0: 1, 1L: 1, 2L: 2})
Neighbor count for cell.id=2 is defaultdict(<type 'int'>, {0: 1, 1L: 2, 2L: 1})
...
```

(continues on next page)

(continued from previous page)

```
Neighbor count for cell.id=11 is defaultdict(<type 'int'>, {1L: 4, 2L: 2})
Neighbor count for cell.id=12 is defaultdict(<type 'int'>, {1L: 2, 2L: 3})
```

Here is an explanation: cell with `id==2` had one neighbor of type Medium (key 0), two neighbor of type 1 (key 1), and one neighbor of type 2 (key 2)

Cell with `id=11` was in contact with six cells - 4 of them were of type 1 and two were of type 2

## 2.6 Passing information between steppables

When you work with more than one steppable (and it is a good idea to work with several steppables each of which has a well defined purpose) you may sometimes need to access or change member variable of one steppable inside the code of another steppable. The most straightforward method to implement exchange of information between steppables is to utilize the python dictionary `shared_steppable_vars`, which is a shared dictionary that every steppable can access as an attribute.

Say we have two steppables, where one steppable updates a shared variable `x_shared`, and the other steppable prints the current value of `x_shared`. Let's call our steppables `UpdaterSteppable` and `PrinterSteppable` for our tasks of updating and printing `x_shared`, respectively:

```
class UpdaterSteppable(SteppableBasePy):
    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)

    def start(self):
        self.shared_steppable_vars['x_shared'] = 0

    def step(self, mcs):
        self.shared_steppable_vars['x_shared'] += 1

class PrinterSteppable(SteppableBasePy):
    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)

    def step(self, mcs):
        print('x_shared=', self.shared_steppable_vars['x_shared'])
```

We see that `UpdaterSteppable` initializes a key `x_shared` with a value of 0 in the shared dictionary and updates it every step. Meanwhile `PrinterSteppable` accesses and prints the value of the same key in the shared dictionary.

The same can be done with simulation objects. Say we have two steppables, where one steppable tests whether or not each cell undergoes mitosis, and the other steppable implements mitosis. For this case, we'll call our steppable that checks for mitosis `CheckMitosisSteppable`, and we'll use the mitosis steppable base class to implement mitosis with a steppable called `MitosisSteppable`. When checking for the occurrence of mitosis, we'll say that all cells with a volume greater than 75 undergo mitosis:

```
class CheckMitosisSteppable(SteppableBasePy):
    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)
```

(continues on next page)

(continued from previous page)

```

def step(self, mcs):

    for cell in self.cell_list:
        if cell.volume > 75:
            self.shared_steppable_vars['cells_to_divide'].append(cell)

class MitosisSteppable(MitosisSteppableBase):
    def __init__(self, frequency=1):
        MitosisSteppableBase.__init__(self, frequency)
        self.set_parent_child_position_flag(0)

    def start(self):
        self.shared_steppable_vars['cells_to_divide'] = []

    def step(self, mcs):

        for cell in self.shared_steppable_vars['cells_to_divide']:
            self.divide_cell_random_orientation(cell)

        self.shared_steppable_vars['cells_to_divide'] = []

    def update_attributes(self):

        self.clone_parent_2_child()

```

We see that `MitosisSteppable` initializes the key `cells_to_divide` with an empty list in the shared dictionary. `CheckMitosisSteppable` populates that same list every step with cell objects according to results of our test for mitosis, each of which `MitosisSteppable` then passes to the built-in method `divide_cell_random_orientation` for randomly-oriented mitosis. After processing all mitosis results, `MitosisSteppable` then returns the shared list to an empty list, so that each positive result of our check for mitosis corresponds to one division of a mitotic cell (otherwise a cell in `shared_steppable_vars['cells_to_divide']` would divide every step!).

## 2.7 How to Check if Two Cells are Unique

In the above examples we were printing cell attributes such as cell type, cell id etc. Sometimes in the simulations you will have two cells and you may want to test if they are different. The most straightforward Python construct would look as follows:

```

cell1 = self.cellField.get(pt)
cell2 = self.cellField.get(pt)
if cell1 != cell2:
    # do something
    ...

```

Because `cell1` and `cell2` point to cell at `pt` i.e. the same cell then `cell1 != cell2` should return false. Alas, written as above the condition is evaluated to true. The reason for this is that what is returned by `cellField` is a Python object that wraps a C++ pointer to a cell. Nevertheless two Python objects `cell1` and `cell2` are different objects because they are created by different calls to `self.cellField.get()` function. Thus, although logically they point to the same cell, you cannot use `!=` operator to check if they are different or not.

The solution is to use the following function

```
self.are_cells_different(cell1,cell2)
```

or write your own Python function that would do the same:

```
def are_cells_different(self, cell1, cell2):
    if (cell1 and cell2 and cell1.this != cell2.this) or\
        (not cell1 and cell2) or (cell1 and not cell2):
        return 1
    else:
        return 0
```

## 2.8 Creating and Deleting Cells. Cell-Type Names

The simulation that Twedit++ Simulation Wizard generates contains some kind of initial cell layout. Sometimes however we want to be able to either create cells as simulation runs or delete some cells. CC3D makes such operations very easy and Twedit++ is of great help. Let us first start with a simulation that has no cells. All we have to do is to comment out BlobInitializer section in the CC3DML code in our cellsorting simulation:

File: C:\\\\CC3DProjects\\\\cellsoring\\\\Simulation\\\\cellsoring.xml

```
<CompuCell3D version="3.6.2">
  <Potts>
    <Dimensions x="100" y="100" z="1"/>
    <Steps>10000</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"/>
  </Plugin>

  <Plugin Name="Volume">
    <VolumeEnergyParameters CellType="Condensing" LambdaVolume="2.0"
      TargetVolume="25"/>
    <VolumeEnergyParameters CellType="NonCondensing" LambdaVolume="2.0"
      TargetVolume="25"/>
  </Plugin>

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

  <Plugin Name="Contact">
    <Energy Type1="Medium" Type2="Medium">10.0</Energy>
    <Energy Type1="Medium" Type2="Condensing">10.0</Energy>
    <Energy Type1="Medium" Type2="NonCondensing">10.0</Energy>
    <Energy Type1="Condensing" Type2="Condensing">10.0</Energy>
    <Energy Type1="Condensing" Type2="NonCondensing">10.0</Energy>
    <Energy Type1="NonCondensing" Type2="NonCondensing">10.0</Energy>
  </Plugin>
```

(continues on next page)

(continued from previous page)

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

</CompuCell3D>
```

When we run this simulation and try to iterate over list of all cells (see earlier example) we won't see any cells:

```
Step 127 Flips 0/10000 Energy 0 Cells 0 Inventory=0
FAST numberoftAttempts=10000
Number of Attempted Energy Calculations=0
Step 128 Flips 0/10000 Energy 0 Cells 0 Inventory=0
FAST numberoftAttempts=10000
Number of Attempted Energy Calculations=0
Step 129 Flips 0/10000 Energy 0 Cells 0 Inventory=0
FAST numberoftAttempts=10000
Number of Attempted Energy Calculations=0
Step 130 Flips 0/10000 Energy 0 Cells 0 Inventory=0
FAST numberoftAttempts=10000
```

Figure 10 Output from simulation that has no cells

To create a single cell in CC3D we type the following code snippet:

```
def start(self):
    self.cell_field[10:14, 10:14, 0] = self.new_cell(self.CONDENSING)
```

### Note

`self.cellField` still works. We can also access this field via `self.field.cell_field` although `self.cell_field` looks much simpler

In Twedit++ go to CC3D Python->Cell Manipulation->Create Cell:

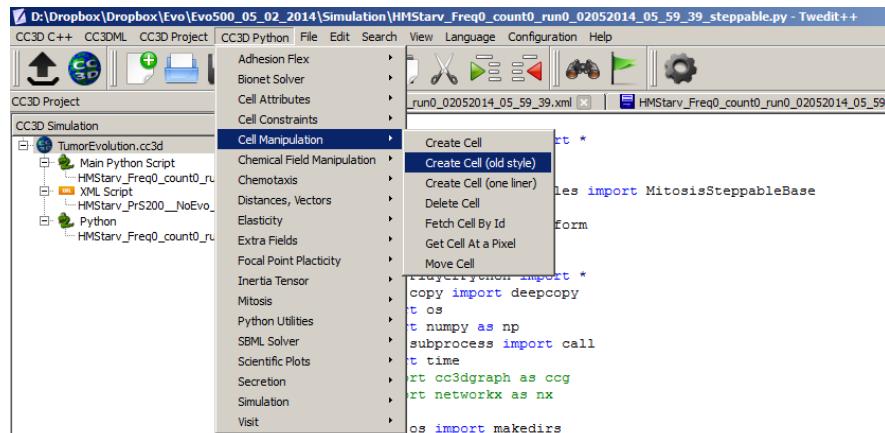


Figure 11 Inserting code snippet in Twedit++ to create cells. Notice that this is a generic code that usually needs minor customizations.

Notice that we create cell in the start function. We can create cells in step functions as well. We create a C++ cell object using the following statement:

```
self.new_cell(self.CONDENSING)
```

We initialize its type using `self.CONDENSING` class variable that corresponds to an integer assigned to type Condensing. Cell type is an integer value from 1 to 255 and CompuCell3D automatically creates class variables corresponding to each type. By looking at the definition of the CellType plugin in CC3DML for cellsorting simulation you can easily infer that number 1 denotes cells of type Condensing and 2 denotes cells of type NonCondensing. Because

it is much easier to remember names of cell types than keeping track which cell type corresponds to which number. SteppableBasePy provides very convenient member variables denoting cell type numbers. The name of such variable is obtained by capitalizing all letters in the name of the cell type and prepending if with self. In our example we will have 3 such variables `self.MEDIUM`, `self.CONDENSING`, `self.NONCONDENSING` with values 0, 1, 2 respectively.

**Note**

To ensure that cell type names are correctly translated into Python class variables avoid using spaces in cell type name.

Consequently,

```
cell.type = self.CONDENSING
```

is equivalent to

```
cell.type = 1
```

but the former makes the code more readable. After assigning cell type all that remains is to initialize lattice sites using newly created cell object so that atleast one lattice site points to this cell object.

The syntax which assigns cell object to 25 lattice sites

```
self.cell_field[10:14, 10:14, 0] = cell
```

is based on Numpy syntax. `self.cell_field` is a pointer to a C++ lattice which stores pointers to cell objects. In this example our cell is a 5x5 square collection of pixels. Notice that the `10:14` has 5 elements because the both the lower and the upper limits are included in the range. As you can probably tell, `self.cellField` is member of `SteppableBasePy`. To access cell object occupying lattice site, x, y, z, we type:

```
cell=self.cell_field[x,y,z]
```

The way we access cell field is very convenient and should look familiar to anybody who has used Matlab, Octave or Numpy.

Deleting CC3D cell is easier than creating one. The only thing we have to remember is that we have to add PixelTracker Plugin to CC3DML (in case you forget this CC3D will throw error message informing you that you need to add this plugin).

The following snippet will erase all cells of type Condensing:

```
def step(self, mcs):
    for cell in self.cell_list:
        if cell.type == self.CONDENSING:
            self.delete_cell(cell)
```

We use member function of `SteppableBasePy` – `deleteCell` where the first argument is a pointer to cell object.

## 2.9 Calculating distances in CC3D simulations.

This may seem like a trivial task. After all, Pitagorean theorem is one of the very first theorems that people learn in basic mathematics course. The purpose of this section is to present convenience functions which will make your code more readable. You can easily code such functions yourself but you probably will save some time if you use ready solutions. One of the complications in the CC3D is that sometimes you may run simulation using periodic boundary conditions. If that's the case, imagine two cells close to the right hand side border of the lattice and moving to the

right. When we have periodic boundary conditions along X axis one of such cells will cross lattice boundary and will appear on the left hand side of the lattice. What should be a distance between cells before and after once of them crosses lattice boundary? Clearly, if we use a naïve formula the distance between cells will be small when all cells are close to right hand side border but if one of them crosses the border the distance calculated using the simple formula will jump dramatically. Intuitively we feel that this is incorrect. The way solve this problem is by shifting one cell to approximately center of the lattice and than applying the same shift to the other cell. If the other cell ends up outside of the lattice we add a vector whose components are equal to dimensions of the lattice but only along this axes along which we have periodic boundary conditions. The point here is to bring a cell which ends up outside the lattice to beinside using vectors with components equal to the lattice dimensions. The net result of these shifts is that we have two cells in the middle of the lattice and the distance between them is true distance regardless the type of boundary conditions we use. You should realize that when we talk about cell shifting we are talking only about calculations and not physical shifts that occur on the lattice.

Example CellDistance from CompuCellPythonTutorial directory demonstrates the use of the functions calculating distance between cells or between any 3D points:

```
class CellDistanceSteppable(SteppableBasePy):

    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)
        self.cellA = None
        self.cellB = None

    def start(self):
        self.cellA = self.potts.createCell()
        self.cellA.type = self.A
        self.cell_field[10:12, 10:12, 0] = self.cellA

        self.cellB = self.potts.createCell()
        self.cellB.type = self.B
        self.cell_field[92:94, 10:12, 0] = self.cellB

    def step(self, mcs):
        dist_vec = self.invariant_distance_vector_integer(p1=[10, 10, 0], p2=[92, 12, 0])

        print('dist_vec=', dist_vec, ' norm=', self.vector_norm(dist_vec))

        dist_vec = self.invariant_distance_vector(p1=[10, 10, 0], p2=[92.3, 12.1, 0])
        print('dist_vec=', dist_vec, ' norm=', self.vector_norm(dist_vec))

        print('distance invariant=', self.invariant_distance(p1=[10, 10, 0], p2=[92.3, 12.1, 0]))

        print('distance =', self.distance(p1=[10, 10, 0], p2=[92.3, 12.1, 0]))

        print('distance vector between cells =', self.distance_vector_between_cells(self.cellA, self.cellB))
        print('invariant distance vector between cells =',
              self.invariant_distance_vector_between_cells(self.cellA, self.cellB))
        print('distanceBetweenCells = ', self.distance_between_cells(self.cellA, self.cellB))
        print('invariantDistanceBetweenCells = ', self.invariant_distance_between_
              cells(self.cellA, self.cellB))
```

In the start function we create two cells – `self.cellA` and `self.cellB`. In the step function we calculate invariant

distance vector between two points using `self.invariant_distance_vector_integer` function. Notice that the word Integer in the function name suggests that the result of this call will be a vector with integer components. Invariant distance vector is a vector that is obtained using our shifting operations described earlier.

The next function used inside step is `self.vector_norm`. It returns length of the vector. Notice that we specify vectors or 3D points in space using `[]` operator. For example to specify vector, or a point with coordinates  $x, y, z = (10, 12, -5)$  you use the following syntax:

```
[10, 12, -5]
```

If we want to calculate invariant vector but with components being floating point numbers we use `self.invariant_distance_vector` function. You may ask why not using floating point always? The reason is that sometimes CC3D expects vectors/points with integer coordinates to e.g. access specific lattice points. By using appropriate distance functions you may write cleaner code and avoid casting and rounding operators. However this is a matter of taste and if you prefer using floating point coordinates it is perfectly fine. Just be aware that when converting floating point coordinate to integer you need to use round and int functions.

Function `self.distance` calculates distance between two points in a naïve way. Sometimes this is all you need. Finally the set of last four calls `self.distance_vector_between_cells`, `self.invariant_distance_vector_between_cells`, `self.distance_between_cells`, `self.invariant_distance_between_cells` calculates distances and vectors between center of masses of cells. You could replace

```
self.invariant_distance_vector_between_cells(self.cellA, self.cellB)
```

with

```
self.invariant_distance_vector_between(  
    p1=[ self.cellA.xCOM, self.cellA.yCOM, self.cellA.zCOM],  
    p2=[ self.cellB.xCOM, self.cellB.yCOM, self.cellB.zCOM]  
)
```

but it is not hard to notice that the former is much easier to read.

## 2.10 Looping over select cell types. Finding cell in the inventory.

We have already see how to iterate over list of all cells. However, quite often we need to iterate over a subset of all cells e.g. cells of a given type. The code snippet below demonstrates howto accomplish such task (in Twedit++ go to CC3D Python->Visit->All Cells of Given Type):

```
for cell in self.cell_list_by_type(self.CONDENSING):  
    print("id=", cell.id, " type=", cell.type)
```

As you can see `self.cell_list` is replaced with `self.cell_list_by_type(self.CONDENSING)` which limits the integration to only those cells which are of type Condensing. We can also choose several cell types to be included in the iteration. For example the following snippet

```
for cell in self.cell_list_by_type(self.CONDENSING, self.NONCONDENSING):  
    print("id=", cell.id, " type=", cell.type)
```

will make CC3D visit cells of type Condensing and NonCondensing. The general syntax is:

```
self.cell_list_by_type(cellType1, cellType2, ...)
```

Occasionally we may want to fetch from a cell inventory a cell object with specific a cell id. This is how we do it (CC3D Python -> Cell Manipulation->Fetch Cell By Id):

```
cell = self.fetch_cell_by_id(10)
print(cell)
```

The output of this code will look as shown below:

```
Step 0 Flips 76/10000 Energy 634 Cells 46 Inventory=46
<CompuCell.CellG; proxy of <Swig Object of type 'std::map< CompuCell3D::CellG *,Coordinates3D< float > >::key_type' at 0x7391d58>
```

Figure 12 Fetching cell with specified id

Function `self.fetch_cell_by_id` will return cell object with specified cell id if such object exists in the cell inventory. Otherwise it will return Null pointer or None object. In fact, to fully identify a cell in CC3D we need to use cell id and cluster. However, when we are not using compartmentalized cells single id , as shown above, insufficient. We will come back to cell ids and cluster ids later in this manual.

## 2.11 Writing data files in the simulation output directory.

Quite often when you run CC3D simulations you need to output data files where you store some information about the simulation. When CC3D saves simulation snapshots it does so in the special directory which is created automatically and whose name consists of simulation core name and timestamp. By default, CC3D creates such directories as sub-folders of `<your_home_directory>/CC3DWorkspace`. You can redefine the location of CC3D output in the Player or from the command line. If standard simulation output is placed in a special directory it makes a lot of sense to store your custom data files in the same directory. The following code snippet shows you how to accomplish this (the code to open file in the simulation output directory can be inserted from Twedit++ - simply go to CC3D Python->Python Utilities):

```
def step(self, mcs):
    output_dir = self.output_dir

    if output_dir is not None:
        output_path = Path(output_dir).joinpath('step_' + str(mcs).zfill(3) + '.dat')
        with open(output_path, 'w') as fout:

            attr_field = self.field.ATTR
            for x, y, z in self.every_pixel():
                fout.write('{:} {:} {:} {}\n'.format(x, y, z, attr_field[x, y, z]))
```

In the step function we create `output_path` by concatenating `output_dir` with a string that contains word `step_`, current mcs (zero-filled up to 3 positions - note how we use standard string function `zfill`) and extension `.dat`

### Note

`self.output_dir` is a special variable in each steppable that stores directory where the output

of the current simulation will be written.

### Note

Path concatenation in `Path(output_dir).joinpath(...)` is done using standard Python package `pathlib`. we import this functionality using `from pathlib import Path`

Next, we open file using `with` statement - if you are unfamiliar with this way of interacting with files in Python please check new Python tutorials online:

```
with open(output_path, 'w') as fout:
    # file is open at this point and there is no need to close it
    # because "with" statement will take care of it automatically
    ...
```

Inside `with` statement (where the file is open) we access chemical field ATTR and use `self.every_pixel` operator to access and write field values to the file:

```
with open(output_path, 'w') as fout:

    attr_field = self.field.ATTR
    for x, y, z in self.every_pixel():
        fout.write('{x} {y} {z} {attr}\n'.format(x, y, z, attr=attr_field[x, y, z]))
```

If we want to create directory inside simulation output folder we can use the following functionality of `pathlib`

```
new_dir = Path(self.output_dir).joinpath('new_dir/new_subdir')
new_dir.mkdir(exist_ok=True, parents=True)
```

We first create path to the new directory using `pathlib`'s `Path` object and its `joinpath` method. and then use `Path`'s method `mkdir` to create directory. The `exist_ok=True`, `parents=True` arguments ensure that the function will not crash if the directory already exists and all the paths along directory path will be created as needed (`parents=True`). In our case it means that `new_dir` and `new_subdir` will be created. `self.output_dir` will be created as well but in a different place by the CC3D simulation setup code.

## 2.12 Adding Plots to the Simulation

Some modelers like to monitor simulation progress by displaying “live” plots that characterize the current state of the simulation. In CC3D, it is very easy to add to the Player windows. The best way to add plots is via Twedit++ CC3D Python->Scientific Plots menu. Take a look at this example code to get a flavor of what is involved when you want to work with plots in CC3D:

```
class CellSortingSteppable(SteppableBasePy):
    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)

    def start(self):
        self.plot_win = self.add_new_plot_window(
            title='Average Volume And Volume of Cell 1',
            x_axis_title='MonteCarlo Step (MCS)',
            y_axis_title='Variables',
            x_scale_type='linear',
            y_scale_type='log',
            grid=True # only in 3.7.6 or higher
            config_options={'legend':True} # Turn on legends
        )

        self.plot_win.add_plot("AverageVol", style='Dots', color='red', size=5)
        self.plot_win.add_plot('Cell1Vol', style='Steps', color='black', size=5)
```

(continues on next page)

(continued from previous page)

```
def step(self, mcs):

    avg_vol = 0.0
    number_of_cells = 0

    for cell in self.cell_list:
        avg_vol += cell.volume
        number_of_cells += 1

    avg_vol /= float(number_of_cells)

    cell1 = self.fetch_cell_by_id(1)
    print(cell1)

    # name of the data series, x, y
    self.plot_win.add_data_point('AverageVol', mcs, avg_vol)
    # name of the data series, x, y
    self.plot_win.add_data_point('Cell1Vol', mcs, cell1.volume)
```

In the `start` function, we create a plot window (`self.plot_win`) – the arguments of this function are self-explanatory. After we have a plot window object (`self.plot_win`), we add data to it at every time step. Here, we will plot two time-series data, one showing the average volume of all cells and one showing the instantaneous volume of a cell with id 1:

```
self.plot_win.add_plot('AverageVol', style='Dots', color='red', size=5)
self.plot_win.add_plot('Cell1Vol', style='Steps', color='black', size=5)
```

The first argument is the name of the data series. This name has two purposes – **1.** It is used in the legend to identify data points and **2.** It is used as an identifier when appending new data. We can also specify a logarithmic axis by using `y_scale_type='log'` as in the example above.

Calling `add_plot` multiple times will produce independent plots as long as you name them distinctly.

In the `step` function, we calculate the average volume of all cells and extract the instantaneous volume of the cell with id 1. Then, we add that result to the time series:

```
# name of the data series, x, y
self.plot_win.add_data_point('AverageVol', mcs, avg_vol)
# name of the data series, x, y
self.plot_win.add_data_point('Cell1Vol', mcs, cell1.volume)
```

Notice that we are using data series identifiers (`AverageVol` and `Cell1Vol`) to add new data. The second argument in the above function calls is the current Monte Carlo Step (`mcs`) whereas the third is an actual quantity that we want to plot on the Y axis. We are done at this point.

The results of the above code may look something like this:

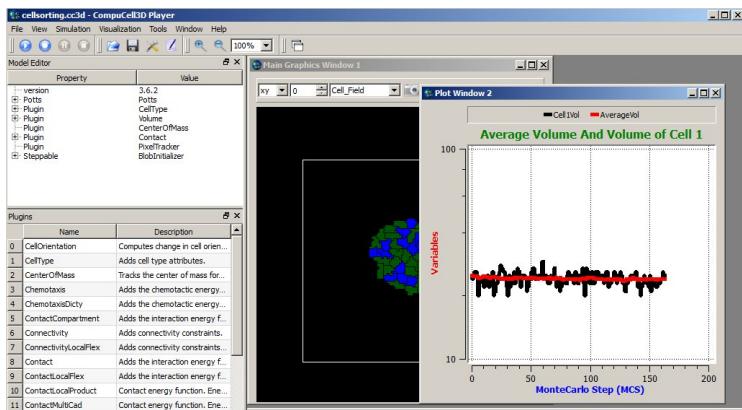


Figure 13 Displaying plot window in the CC3D Player with 2 time-series data.

**Styling:** If you need prettier plots, we recommend saving the data you need to plot to a separate CSV file, then use a framework like Seaborn or Matplotlib to refine your plots. Plots provided in CC3D are used mainly as a convenience feature and to monitor the current state of the simulation.

## 2.12.1 Histograms

When using a histogram, you plot a list of data at each time step rather than a single value. Numpy has the tools to make this task relatively simple. An example `scientificHistBarPlots` in `CompuCellPythonTutorial` demonstrates the use of histograms. Let us look at the example steppable (you can also find relevant code snippets in CC3D Python-> Scientific Plots menu):

```
from cc3d.core.PySteppables import *
import random
import numpy as np
from pathlib import Path

class HistPlotSteppable(SteppableBasePy):
    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)
        self.plot_win = None

    def start(self):

        # initialize setting for Histogram
        self.plot_win = self.add_new_plot_window(title='Histogram of Cell Volumes', x_
→axis_title='Number of Cells',
                                                y_axis_title='Volume Size in Pixels')
        # alpha is transparency 0 is transparent, 255 is opaque
        self.plot_win.add_histogram_plot(plot_name='Hist 1', color='green', alpha=100)
        self.plot_win.add_histogram_plot(plot_name='Hist 2', color='red', alpha=100)
        self.plot_win.add_histogram_plot(plot_name='Hist 3', color='blue')

    def step(self, mcs):

        vol_list = []
        for cell in self.cell_list:
            vol_list.append(cell.volume)
```

(continues on next page)

(continued from previous page)

```

gauss = np.random.normal(0.0, 1.0, size=(100,))

    self.plot_win.add_histogram(plot_name='Hist 1', value_array=gauss, number_of_
bins=10)
    self.plot_win.add_histogram(plot_name='Hist 2', value_array=vol_list, number_of_
bins=10)
    self.plot_win.add_histogram(plot_name='Hist 3', value_array=vol_list, number_of_
bins=50)

    if self.output_dir is not None:
        output_path = Path(self.output_dir).joinpath("HistPlots_" + str(mcs) + ".txt"
)
        self.plot_win.save_plot_as_data(output_path, CSV_FORMAT)

        png_output_path = Path(self.output_dir).joinpath("HistPlots_" + str(mcs) + "."
)
        self.plot_win.save_plot_as_png(png_output_path, 1000, 1000)

    # here we specify size of the image saved - default is 400 x 400
    self.plot_win.save_plot_as_png(png_output_path, 1000, 1000)

```

In the start function, we call `self.add_new_plot_window` to add a new plot window -`self.plot_win`- to the Player. Subsequently, we specify the display properties of different data series (histograms). Notice that we can specify opacity using the `alpha` parameter.

In the step function, we first iterate over each cell and append their volumes to the Python list. Later, we plot a histogram of the array using a very simple call:

```
self.plot_win.add_histogram(plot_name='Hist 2', value_array=vol_list, number_of_bins=10)
```

#### Parameters:

- `value_array`: holds an unordered collection of data at one time step, such as the volume of 100 cells.
- `number_of_bins`: controls how many “bars” will appear, which can make the plot look more coarse- or fine-grained.

### 2.12.2 Example: Create a Histogram from a Random Distribution

The following snippet:

```

gauss = []
for i in range(100):
    gauss.append(random.gauss(0,1))

(n2, bins2) = numpy.histogram(gauss, bins=10)

```

declares `gauss` as Python list and appends to it 100 random numbers which are taken from Gaussian distribution centered at 0.0 and having standard deviation equal to 1.0. We histogram those values using the following code:

```
self.plot_win.add_histogram(plot_name='Hist 1' , value_array = gauss ,number_of_bins=10)
```

When we look at the code in the `start` function we will see that this data series will be displayed using green bars.

### 2.12.3 Save Plot as an Image

At the end of the steppable, we can output the histogram plot as a PNG image file using:

```
self.plot_win.save_plot_as_png(png_output_path, 1000, 1000)
```

The last two arguments of this function represent the x and y sizes of the image.

The image file will be written in the simulation output directory.

#### Note

As of writing this manual, we do not support scaling of the plot image output. This might change in a future release. However, we strongly recommend that you save all the data you plot in a separate file and post-process it in a full-featured plotting program.

### 2.12.4 Save Plot as CSV Data File

Finally, for any plot, we can output plotted data in the form of a text file. All we need to do is to call `save_plot_as_data` from the plot windows object:

```
output_path = "HistPlots_"+str(mcs)+".txt"
self.plot_win.save_plot_as_data(output_path, CSV_FORMAT)
```

This file will be written in the simulation output directory. You can use it later to post-process plot data using external plotting software.

### 2.12.5 How to Improve Plot Performance

Create a separate steppable specifically for plotting. In your Main Python Script, increase the `frequency` property of the plot steppable so that it updates less often.

Of course, this plot will not look as smooth for demonstrations; it's just an efficient monitoring tool.

```
from cc3d import CompuCellSetup

from MyProjectSteppables import MyMainSteppable
CompuCellSetup.register_steppable(steppable=MyMainSteppable(frequency=1))

from MyProjectSteppables import UpdatePlotsSteppable
CompuCellSetup.register_steppable(steppable=UpdatePlotsSteppable(frequency=200))

CompuCellSetup.run()
```

## 2.13 How to Detect Contact

### 2.13.1 Single Contact Event

At every time step, you should check each cell's neighbors, then mark a dictionary property to prevent the event from happening again for the same cell. CC3D is not event-driven, so code to handle contact events should go in the `step` function. Here is how to trigger an action when two cells collide.

```

def start(self):
    #Add the `touching_macrophage_cell` attribute to every cell.
    for cell in self.cell_list:
        cell.dict['touching_macrophage_cell'] = False

def step(self, mcs):
    """
    Check every bacteria to see if it is touching a macrophage.
    If it is, and it has never touched a macrophage, then set the property to True.
    """
    for cell in self.cell_list_by_type(self.BACTERIA):
        for neighbor, common_surface_area in self.get_cell_neighbor_data_list(cell):
            if neighbor: #Ensure we are not looking at the Medium
                if neighbor.type == self.MACROPHAGE:
                    if cell.dict['touching_macrophage_cell'] == False:
                        cell.dict['touching_macrophage_cell'] = True
                        print("Contact happened!")

```

## 2.13.2 Periodic Contact Events

Notice that this interaction can only happen once. If you want contact events to happen periodically, then you should limit how often the event triggers. The below code shows how to trigger a contact event every 100 MCS. Essentially, we check the previous time stamp of a contact event, `last_cell_touch_time`, to see if it was at least 100 MCS in the past.

```

def start(self):
    #Set the property to -100 so that the interaction can still happen on MCS=0.
    for cell in self.cell_list:
        cell.dict['last_cell_touch_time'] = -100

def step(self, mcs):
    ...
        if neighbor.type == self.MACROPHAGE:
            if cell.dict['last_cell_touch_time'] - mcs >= 100:
                print("Contact happened!")
                cell.dict['last_cell_touch_time'] = mcs
                #From here, you could change the cell's type or kill the cell
                #if you wish to stop future interactions.

```

## 2.13.3 Using Contact to Transmit Cell Signals

Cells that need to maintain contact for an extended period would use similar code. For example, a CD4 T cell receptor requires time for an interaction with an antigen-presenting cell (Also called the MHC class II and TCR cell-cell interaction).

```

def start(self):
    #Now -1 is a placeholder value to show that there
    for cell in self.cell_list:
        cell.dict['signal_received'] = 0

def step(self, mcs):

```

(continues on next page)

(continued from previous page)

```

for cell in self.cell_list_by_type(self.CD4_T_CELL):
    for neighbor, common_surface_area in self.get_cell_neighbor_data_list(cell):
        if neighbor and neighbor.type == self.AP_CELL:
            if cell.dict['signal_received'] < 50:
                cell.dict['signal_received'] += 1
            elif cell.dict['signal_received'] == 50:
                print("Signal sufficient!")

```

## 2.14 Examples: Mitosis

Related: [Mitosis](#)

The folder containing this simulation is

*Demos/CompuCellPythonTutorial/steppableBasedMitosis.*

File:

*Demos/CompuCellPythonTutorial/steppableBasedMitosis/Simulation/steppableBasedMitosisSteppables.py*

```

from PySteppables import *
from PySteppablesExamples import MitosisSteppableBase
import CompuCell

class VolumeParamSteppable(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):
        for cell in self.cellList:
            cell.targetVolume += 1

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

        # 0 - parent child position will be randomized between mitosis event
        # negative integer - parent appears on the 'left' of the child
        # positive integer - parent appears on the 'right' of the child
        self.setParentChildPositionFlag(-1)

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

```

(continues on next page)

(continued from previous page)

```

for cell in cells_to_divide:
    # to change mitosis mode leave one of the below lines uncommented
    self.divideCellRandomOrientation(cell)

def updateAttributes(self):
    self.parentCell.targetVolume /= 2.0 # reducing parent target volume
    self.cloneParent2Child()

    if self.parentCell.type == self.CONDENSING:
        self.childCell.type = self.NONCONDENSING
    else:
        self.childCell.type = self.CONDENSING

```

Two steppables: VolumeParamSteppable and MitosisSteppable are the the essence of the above simulation. The first steppable initializes the volume constraint for all the cells present at T=0 MCS (only one cell) and then every 10 MCS (see the frequency with which VolumeParamSteppable is initialized to run - *Demos/CompuCellPythonTutorial/steppableBasedMitosis/Simulation/steppableBasedMitosis.py*) it increases the target volume of cells, effectively causing cells to grow.

```

from steppableBasedMitosisSteppables import VolumeParamSteppable
volumeParamSteppable=VolumeParamSteppable(sim ,10)
steppableRegistry.registerSteppable(volumeParamSteppable)

from steppableBasedMitosisSteppables import MitosisSteppable
mitosisSteppable=MitosisSteppable(sim, 10)
steppableRegistry.registerSteppable(mitosisSteppable)

```

The second steppable checks every 10 MCS (we can, of course, run it every MCS) if a cell has reached its doubling volume of 50. If it did, that cell is added to the list `cells_to_divide`. After construction of `cells_to_divide` is complete, we iterate over this list and divide all the cells within it.

### Warning

It is important to divide cells outside the loop where we iterate over the entire cell inventory. If we keep dividing cells in this loop we are adding elements to the list over which we iterate and this might have unwanted side effects. The solution is to use a list of cells to divide as we did in the example.

Notice that we call `self.divideCellRandomOrientation(cell)` function to divide cells. Other modes of division are available as well and they are as follows:

```

self.divideCellOrientationVectorBased(cell, 1,0,0)
self.divideCellAlongMajorAxis(cell)
self.divideCellAlongMinorAxis(cell)

```

## 2.15 Custom Cell Attributes in Python

As you have already seen, each cell object has a several attributes describing properties of model cell (e.g. volume, surface, target surface, type, id *etc...*). However, in almost every simulation that you develop, you need to associate additional attributes with the cell objects. For example, you may want every cell to have a countdown clock that will be recharged once its value reaches zero. One way to accomplish this task is to add a line:

**int clock**

to Cell.h file and recompile entire CompuCell3D package. Cell.h is a C++ header file that defines basic properties of the CompuCell3D cells and it happened so that almost every C++ file in the CC3D source code depends on it. Consequently, any modification of this file will mean that you would need to recompile almost entire CC3D from scratch. This is inefficient. Even worse, you will not be able to share your simulation using this extra attribute, unless the other person also recompiled her/his code using your tweak. Fortunately CC3D let's you easily attach any type of Python object as cell attribute. Each cell, by default, has a Python dictionary attached to it. This allows you to store any object that has Python interface as a cell attribute. Let's take a look at the following implementation of the step function:

```
def step(self,mcs):

    for cell in self.cell_list:
        cell.dict["Double_MCS_ID"] = mcs*2*cell.id

    for cell in self.cell_list:
        print('cell.id=', cell.id, ' dict=', cell.dict)
```

We have two loops that iterate over list of all cells. In the first loop we access dictionary that is attached to each cell:

```
cell.dict
```

and then insert into a dictionary a product of 2, mcs and cell id:

```
cell.dict["Double_MCS_ID"] = mcs*2*cell.id
```

In the second loop we access the dictionary and print its content to the screen. The result will look something like:

Figure 14 Simple simulation demonstrating the usage of custom cell attributes.

If you would like attach a Python list to the cell all you do it insert Python list as one of the elements of the dictionary e.g.:

```
for cell in self.cell_list:
    cell.dict["MyList"] = list()
```

Thus all you really need to store additional cell attributes is the dictionary.

## 2.16 Adding and managing extra fields for visualization purposes

Quite often in your simulation you will want to label cells using scalar field, vector fields or simply create your own scalar or vector fields which are fully managed by you from the Python level. CC3D allows you to create four kinds of fields:

1. Scalar Field – to display scalar quantities associated with single pixels
2. Cell Level Scalar Field – to display scalar quantities associated with cells

3. Vector Field - to display vector quantities associated with single pixels
4. Cell Level Vector Field - to display vector quantities associated with cells

You can take look at `CompuCellPythonTutorial/ExtraFields` to see an example of a simulation that uses all four kinds of fields. The Python syntax used to create and manipulate custom fields is relatively simple but quite hard to memorize. Fortunately Twedit++ has CC3DPython->Extra Fields menu that inserts code snippets to create/manage fields.

### 2.16.1 Scalar Field – pixel based

Let's look at the steppable that creates and manipulates scalar cell field. This field is implemented as Numpy float array and you can use Numpy functions to manipulate this field.

```
from cc3d.core.PySteppables import *
from random import random
from math import sin

class ExtraFieldVisualizationSteppable(SteppableBasePy):
    def __init__(self, frequency=10):
        SteppableBasePy.__init__(self, frequency)
        self.create_scalar_field_py("ExtraField")

    def step(self, mcs):

        cell = self.field.Cell_Field[20, 20, 0]
        print('cell=', cell)

        # clear field
        self.field.ExtraField[:, :, :] = 10.0

        for x, y, z in self.every_pixel(4, 4, 1):
            if not mcs % 20:
                self.field.ExtraField[x, y, z] = x * y
            else:
                self.field.ExtraField[x, y, z] = sin(x * y)
```

The scalar field (we called it `ExtraField`) is declared in the `__init__` function of the steppable using

```
self.create_scalar_field_py("ExtraField")
```

#### Note

Ideally you would declare extra fields in the `__init__` function but if you create them elsewhere they will work as well. However in certain situations you may notice that fields declared outside `__init__` may be missing

from e.g. player menus. Normally it is not a big deal but if you want to have full functionality associated with the fields declare them inside `__init__`

In the step function we initialize `ExtraField` using slicing operation:

```
self.field.ExtraField[:, :, :] = 10.0
```

In Python slicing convention, a single colon means all indices – here we put three colons for each axis which is equivalent to selecting all pixels. Notice how we use `self.field.ExtraField` construct to access the field.

It is perfectly fine (and faster too if you acces field repeatedly) to split this int two lines:

```
extra_field = self.field.ExtraField
extra_field[:, :, :] = 10
```

Following lines in the step functions iterate over every pixel in the simulation and if MCS is divisible by 20 then `self.scalarField` is initialized with  $x^*y$  value if MCS is not divisible by 20 than we initialize scalar field with `sin(x^*y)` function. Notice, that we imported all functions from the `math` Python module so that we can get sin function to work.

`SteppableBasePy` provides convenience function called `self.every_pixel` (CC3D Python->Visit->All Lattice Pixels) that facilitates compacting triple loop to just one line:

```
for x,y,z in self.every_pixel():
    if not mcs % 20:
        self.field.ExtraField[x, y, z]=x*y
    else:
        self.field.ExtraField[x, y, z]=sin(x*y)
```

If we would like to iterate over x axis indices with step 5, over y indices with step 10 and over z axis indices with step 4 we would replace first line in the above snippet with.

```
for x, y, z in self.every_pixel(5,10,4):
```

You can still use triple loops if you like but shorter syntax leads to a cleaner code.

## 2.16.2 Vector Field – pixel based

By analogy to pixel based scalar field we can create vector field. Let's look at the example code:

```
class VectorFieldVisualizationSteppable(SteppableBasePy):
    def __init__(self, frequency=10):
        SteppableBasePy.__init__(self, frequency)
        self.create_vector_field_py("VectorField")

    def step(self, mcs):
        vec_field = self.field.VectorField

        # clear vector field
        vec_field[:, :, :, :] = 0.0

        for x, y, z in self.everyPixel(10, 10, 5):
            vec_field[x, y, z] = [x * random(), y * random(), z * random()]
```

The code is very similar to the previous steppable. In the `__init__` function we create pixel based vector field , in the step function we initialize it first to with zero vectors and later we iterate over pixels using steps 10, 10, 5 for x, y, z axes respectively and to these select lattice pixels we assign  $[x * \text{random}(), y * \text{random}(), z * \text{random}()]$  vector. Internally, `self.field.VectorField` is implemented as numpy array:

```
np.zeros(shape=(_dim.x, _dim.y, _dim.z, 3), dtype=np.float32)
```

### 2.16.3 Scalar Field – cell level

Pixel based fields are appropriate for situations where we want to assign scalar or vector to particular lattice locations. If, on the other hand, we want to label cells with a scalar or a vector we need to use cell level field (scalar or vector). It is still possible to use pixel-based fields but we assure you that the code you would write would be very ugly at best.

Internally cell-based scalar field is implemented as a map or a dictionary indexed by cell id (although in Python instead of passing cell id we pass cell object to make syntax cleaner). Let us look at an example code:

```
class IdFieldVisualizationSteppable(SteppableBasePy):
    def __init__(self, frequency=10):
        SteppableBasePy.__init__(self, frequency)

    def start(self):
        # note if you create field outside constructor this field will not be properly
        # initialized if you are using restart snapshots. It is OK as long as you are
        ↵aware of this limitation
        self.create_scalar_field_cell_level_py("IdFieldNew")

    def step(self, mcs):
        # clear id field
        try:
            id_field = self.field.IdFieldNew
            id_field.clear()
        except KeyError:
            # an exception might occur if you are using restart snapshots to restart
            ↵simulation
            # because field has been created outside constructor
            self.create_scalar_field_cell_level_py("IdFieldNew")
            id_field = self.field.IdFieldNew

        for cell in self.cell_list:
            id_field[cell] = cell.id * random()
```

As it was the case with other fields we create cell level scalar field in the `__init__` function using `self.create_scalar_field_cell_level_py`. In the step function we first clear the field – this simply removes all entries from the dictionary. If you forget to clean dictionary before putting new values you may end up with stray values from the previous step. Inside the loop over all cells we assign random value to each cell. When we plot `IdFieldNew` in the player we will see that cells have different color labels. If we used pixel-based field to accomplish same task we would have to manually assign same value to all pixels belonging to a given cell. Using cell level fields we save ourselves a lot of work and make code more readable.

### 2.16.4 Vector Field – cell level

We can also associate vectors with cells. The code below is analogous to the previous example:

Inside `__init__` function we create cell-level vector field using `self.create_vector_field_cell_level_py` function. In the step function we clear field and then iterate over all cells and assign random vector to each cell. When we plot this field on top cell borders you will see that vectors are anchored in “cells’ corners” and not at the COM. This is because such rendering is faster.

You should remember that all those 4 kinds of field discussed here are for display purposes only. They do not participate in any calculations done by C++ core code and there is no easy way to pass values of those fields to the CC3D computational core.

## 2.17 Automatic Tracking of Cells' Attributes

Sometimes you would like to color-code cells based on the value (scalar or vector) of one of the cellular attributes. You can use the techniques presented above to display cell-level scalar or vector field or you can take advantage of a very convenient shortcut that using one line of code allows you to setup up visualization field that tracks cellular attributes. Here is a simple example:

```
class DemoVisSteppable(SteppableBasePy):
    def __init__(self, _simulator, _frequency=1):
        SteppableBasePy.__init__(self, _simulator, _frequency)
        self.track_cell_level_scalar_attribute(field_name='COM_RATIO',attribute_name=
→'ratio')

    def start(self):
        for cell in self.cellList:
            cell.dict['ratio'] = cell.xCOM / cell.yCOM

    def step(self, mcs):
        for cell in self.cellList:
            cell.dict['ratio'] = cell.xCOM / cell.yCOM
```

In the start and step functions we iterate over all cells and attach a cell attribute `ratio` that is equal to the ration of x and y center-of mass coordinates for each cell. In the init function we setup automatic tracking of this attribute i.e. we create a cell-level scalar field (called `COM_RATIO`) where cells are colored according to the value of their '`ratio`' attribute:

```
self.track_cell_level_scalar_attribute (field_name='COM_RATIO',attribute_name='ratio')
```

The syntax of this function can be found in Twedit Python helper menu: CC3D Python->Extra Fields Automatic Tracking -> Track Scalar Cell

Attribute (`__init__`):

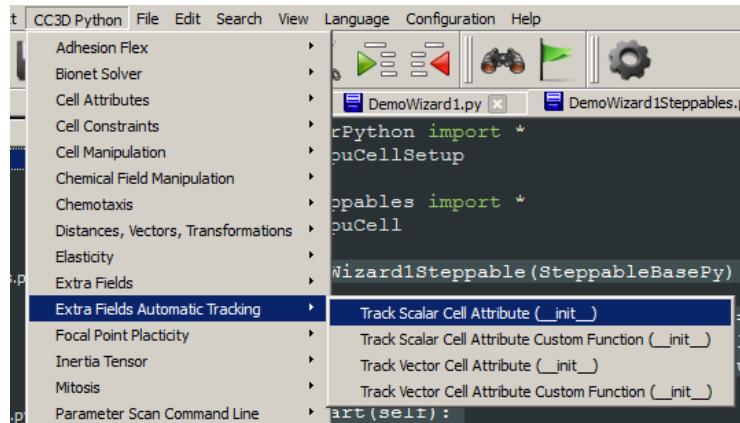


Figure 15 Setting up automatic tracking of cells' scalar attribute using Twedit++

Sometimes instead of tracking the actual attribute we would lie to color-code cells according to the user-specified function of the attribute. For example instead of color-coding cells according to ration of x and y center-of-mass coordinates we would lie to color-code them according to a sinus of the ratio:

```
class DemoVisSteppable(SteppableBasePy):
    def __init__(self, _simulator, _frequency=1):
```

(continues on next page)

(continued from previous page)

```

SteppableBasePy.__init__(self, _simulator, _frequency)
    self.track_cell_level_scalar_attribute(field_name='COM_RATIO',
                                            attribute_name='ratio')

import math
self.track_cell_level_scalar_attribute(field_name='SIN_COM_RATIO',
                                        attribute_name='ratio',
                                        function=lambda attr_val: math.sin(attr_
val))

def start(self):
    for cell in self.cellList:
        cell.dict['ratio'] = cell.xCOM / cell.yCOM

def step(self, mcs):
    for cell in self.cellList:
        cell.dict['ratio'] = cell.xCOM / cell.yCOM

```

All we did in the snippet above was to add new field SIN\_COM\_RATIO using the track\_cell\_level\_scalar\_attribute function. The call to this function almost identical as before except now we also used function argument:

```
function = lambda attr_val: math.sin(attr_val)
```

The meaning of this is the following: for each attribute ratio attached to a cell a function `math.sin(attr_val)` will be evaluated where `attr_val` will assume same value as 'ratio' cell attribute for a given cell. If you are puzzled about lambda Python key word don't be. Python lambda's are a convenient way to define inline functions For example:

```
f = lambda x: x**2
```

defines function f that takes one argument x and returns its square. Thus, `f(2)` will return 4 and `f(4)` would return 16.

Lambda function can be replaced by a regular function f as follows:

```
def f(x):
    return x**2
```

When we run the simulation above the output may look like in the figure below:

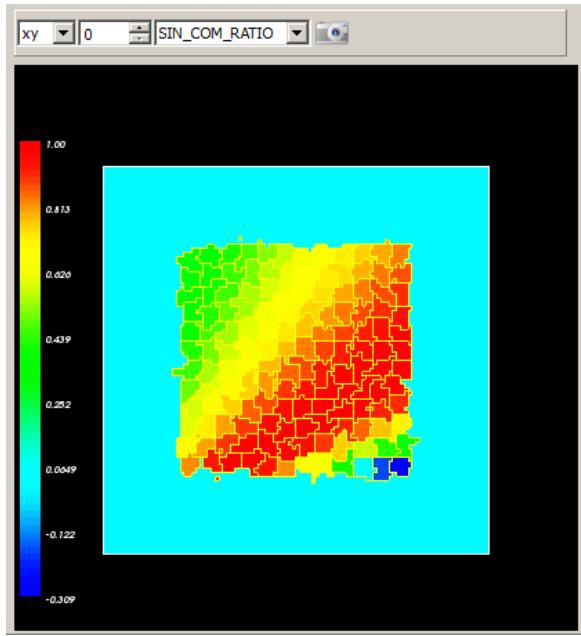


Figure 16. Automatic labeling of cells according to scala cell's attribute

Now that we learned how to color-code cells according to the custom attribute we can use analogous approach to label cells using vector attribute. **Important:** vector quantity must be a list, tuple or numpy array with 3 elements.

The steppable code below demonstrates how we can enable auto-visualization of the cell's vector attribute:

```
class DemoVisSteppable(SteppableBasePy):

    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator,_frequency)
        self.track_cell_level_vector_attribute (field_name = 'COM_VECTOR',\
        attribute_name = 'com_vector')
        import math
        self.track_cell_level_vector_attribute (field_name = 'SIN_COM_VECTOR',\
        attribute_name = 'com_vector',\
        function = lambda attr_val: [ math.sin(attr_val[0]), math.sin(attr_val[1]), 0] )

    def start(self):
        for cell in self.cellList:
            cell.dict['com_vector'] = [cell.xCOM, cell.yCOM, 0.0]

    def step(self,mcs):
        for cell in self.cellList:
            cell.dict['com_vector'] = [cell.xCOM, cell.yCOM, 0.0]
```

There are few differences as compared to the code that used scalar quantities: 1) we used `self.track_cell_level_vector_attribute` in the `__init__` constructor, 2) our attributes are vectors:

```
cell.dict['com_vector'] = [cell.xCOM, cell.yCOM, 0.0]
```

3) the lambda function we use takes a single argument which is this case is a vector (i.e. it has 3 elements) and also returns 3 element vector.

## HOW TO PROGRAMMATICALLY CONTROL EVERY ASPECT OF THE SIMULATION

 Note

The features described in this section are available as of CompuCell3D version 4.4.0.

Most new users start learning CompuCell3D by configuring and editing simulations in [Twedit GUI](#) and then running them via [Player GUI](#). Starting from version 4.4.0, CompuCell3D is shipped with an upgraded Python API that allows modelers to bypass both Player and Twedit and configure simulations - or even an ensemble of multiple simulations running in parallel and informing each other - directly from a single Python script. This way of constructing simulations gives users full control over every aspect of the simulation. The material presented in this section explains:

- how to configure the entire simulation in pure Python (*i.e.*, without XML)
- how to write and control the main CC3D loop where, at every step, we call the [Potts algorithm](#)
- how to run multiple concurrent and interacting simulations
- how to specify and control visualization
- how to use Jupyter notebook.

In a sense, this tutorial shows an alternative way of specifying and running simulations that bypasses legacy tools like Twedit++ and Player. It gives you full programmatic control of the CC3D simulation.

We recommend that you use PyCharm or VS Code if you decide to work with the API presented in this section

### 3.1 Overview

### 3.2 CC3D Projects vs. Python Projects

The [Twedit](#) and [Player](#) GUIs facilitate the generation, development, execution and sharing of simulations in a project structure that can consist of Python and [CC3DML](#) source code, auxiliary resources (*e.g.*, [PIF files](#)) and a `.cc3d` project file that describes the contents of a CC3D project. This structure is the officially supported CC3D project structure that the complete distribution of CC3D packages supports. However, arbitrary Python projects, which can consist of one or hundreds of Python source files, can also integrate CC3D to support biological modeling and simulation in integrated applications. CC3D provides basic support for such applications through an extensive Python API that makes available all built-in plugins and steppables for model specification, as well as interactive control over simulation execution through memory-safe objects that contain simulation instances.

In general, the `cc3d` python module contains the entire CC3D Python runtime API. CC3D projects typically define plugins and steppables in [CC3DML](#) and custom steppables in a single Python script, the complete project of which can be loaded in the [Player](#) GUI. In Python projects, model and simulation specification through the Python API requires

Python instructions to create a simulation object, create plugin and steppable objects, register the plugins and steppables with the simulation object, and then execute the simulation along with optional instructions for real-time visualization. As such, Python projects tend to require more startup work to accomplish tasks that are performed by the [Player](#) GUI but can develop and support more advanced and specific functionality.

### 3.3 CC3D Python Interactive Simulation

The new CC3D Python API gives users a direct access to a live, interactive CC3D simulation via `CC3DSimService` object. Executing a simulation using `CC3DSimService` consists of creating a `CC3DSimService` instance, loading it with model and simulation specification, performing simulation startup routines, executing simulation steps, performing what interactions may be appropriate for an application (*e.g.*, steering), and ultimately performing simulation shutdown routines.

```
from cc3d.CompuCellSetup.CC3DCaller import CC3DSimService

cc3d_sim = CC3DSimService()          # Create a simulation
# Load specification here...
cc3d_sim.run()                      # Start the underlying process of the simulation
cc3d_sim.init()                     # Execute the simulation initialization stage
cc3d_sim.start()                    # Execute the simulation startup stage (e.g., steppable
# start` is called)
cc3d_sim.step()                     # Execute one simulation step
cc3d_sim.finish()                   # Execute the simulation finalization stage (e.g., u
# steppable `finish` is called)
```

`CC3DSimService` supports executing an arbitrary number of custom steppables during simulation through a registration method `register_steppable`. Passing a steppable class to `register_steppable` registers a steppable to be called at specified step intervals, and the `CC3DSimService` instance will instantiate the steppable during simulation startup. Custom Python steppables with the Python API provide a straightforward way to perform manipulations to simulation data during execution of a simulation, even by simple module-level variables,

```
from cc3d.CompuCellSetup.CC3DCaller import CC3DSimService
from cc3d.core.PySteppables import SteppableBasePy

# Define volume parameters
target_volume_init = 25             # Initial target volume
lambda_volume = 2                   # Volume constraint parameter
target_volume = target_volume_init  # Current target volume; varies during simulation

class GrowthSteppable(SteppableBasePy):
    """
    A custom steppable that uniformly applies a volume constraint
    """

    def start(self):
        """Applies initial volume parameters to all cells during startup"""
        for cell in self.cell_list:
            cell.targetVolume = target_volume_init
            cell.lambdaVolume = lambda_volume

    def step(self, mcs):
        """Updates the target volume of all cells every 100 steps"""
        if mcs % 100 == 0:
```

(continues on next page)

(continued from previous page)

```

# Whatever the value of the variable `target_volume`, apply it to all cells
for cell in self.cell_list:
    cell.targetVolume = target_volume

# Launch a simulation and register GrowthSteppable
cc3d_sim = CC3DSimService()
# Load specification here...
cc3d_sim.register_steppable(steppable=GrowthSteppable, frequency=1)
cc3d_sim.run()
cc3d_sim.init()
cc3d_sim.start()

# Execute 10k steps and update the target volume along the way
num_steps = 10000
while cc3d_sim.current_step < num_steps:
    target_volume = target_volume_init * (1.0 + cc3d_sim.current_step / num_steps)
    cc3d_sim.step()

```

`CC3DSimService.register_steppable` also supports registering a steppable instance, which can be used to perform operations with both the steppable and its interface to simulation core objects and convenience features,

```

from cc3d.CompuCellSetup.CC3DCaller import CC3DSimService
from cc3d.core.PySteppables import SteppableBasePy

# Define volume parameters
target_volume_init = 25           # Initial target volume
lambda_volume = 2                # Volume constraint parameter
target_volume = target_volume_init # Current target volume; varies during simulation

# Launch a simulation and register a generic steppable instance
cc3d_sim = CC3DSimService()
# Load specification here...
steppable = SteppableBasePy()
cc3d_sim.register_steppable(steppable=steppable)
cc3d_sim.run()
cc3d_sim.init()
cc3d_sim.start()

for cell in steppable.cell_list:
    cell.targetVolume = target_volume_init
    cell.lambdaVolume = lambda_volume

# Execute 10k steps and update the target volume along the way using the steppable
→ interface
num_steps = 10000
while cc3d_sim.current_step < num_steps:
    if cc3d_sim.current_step % 100 == 0:
        target_volume = target_volume_init * (1.0 + cc3d_sim.current_step / num_steps)
        for cell in steppable.cell_list:
            cell.targetVolume = target_volume
    cc3d_sim.step()

```

## 3.4 Python Built-In Plugins and Steppables

### Note

All features described in this section can also be employed in CC3D projects. In such cases, an all-Python CC3D project can be generated in the [Twedit](#) GUI, and all plugins and steppables can be specified using the Python API described in this section.

The CC3D Python module `cc3d.core.PyCoreSpecs` provides an interactive object for using each built-in plugin and steppable in simulation. Each interactive object in the `cc3d.core.PyCoreSpecs` module contains all internal data necessary to create a corresponding built-in plugin or steppable, which can be manipulated through the interface of each interactive object. For example, a typical simulation specification consists of the [Potts](#) specification, [CellType](#), [Volume](#) and [Contact](#) plugins, and a [BlobInitializer](#) steppable to initialize a cell distribution, which can look like the following when using the Python API:

```
from cc3d.CompuCellSetup.CC3DCaller import CC3DSimService
from cc3d.core.PyCoreSpecs import PottsCore, CellTypePlugin, VolumePlugin, ContactPlugin

# Specify a two-dimensional simulation with a 100x100 lattice and second-order Potts
# neighborhood.
potts_specs = PottsCore(dim_x=100, dim_y=100, neighbor_order=2)
# Define two cell types called "Condensing" and "NonCondensing".
cell_type_specs = CellTypePlugin("Condensing", "NonCondensing")
# Assign a volume constraint to both cell types.
volume_specs = VolumePlugin()
volume_specs.param_new("Condensing", target_volume=25, lambda_volume=2)
volume_specs.param_new("NonCondensing", target_volume=25, lambda_volume=2)
# Assign adhesion between cells by type.
contact_specs = ContactPlugin(neighor_order=2)
contact_specs.param_new(type_1="Medium", type_2="Condensing", energy=20)
contact_specs.param_new(type_1="Medium", type_2="NonCondensing", energy=20)
contact_specs.param_new(type_1="Condensing", type_2="Condensing", energy=2)
contact_specs.param_new(type_1="Condensing", type_2="NonCondensing", energy=11)
contact_specs.param_new(type_1="NonCondensing", type_2="NonCondensing", energy=16)
# Initialize cells as a blob with a random distribution by type.
blob_init_specs = BlobInitializer()
blob_init_specs.region_new(width=5, radius=20, center=(50, 50, 0), cell_types=(
    "Condensing", "NonCondensing"))
```

`PottsCore` defines the global simulation properties such as `dim_x/dim_y` (the simulation boundaries) and `fluctuation_amplitude/temperature` (the cell membrane activity level).

```
CompuCellSetup.register_specs(PottsCore(dim_x=dim_x,
    dim_y=dim_y,
    steps=100000,
    neighbor_order=2,
    boundary_x="Periodic",
    boundary_y="Periodic",
    fluctuation_amplitude=10))
```

A built-in plugin or steppable specification in the Python API consists of creating an instance of its corresponding class, setting the internal data of the instance and registering the instance with a `CC3DSimService` instance through the method `register_specs`. Like in typical CC3D projects, every built-in plugin and steppable that is registered with

a CC3DSimService instance will automatically function within the underlying simulation of the CC3DSimService instance for the entire duration of the simulation.

```
# Launch a simulation and register all specifications
cc3d_sim = CC3DSimService()
cc3d_sim.register_specs([potts_specs, cell_type_specs, volume_specs, contact_specs, blob_
    _init_specs])
cc3d_sim.run()
cc3d_sim.init()
cc3d_sim.start()
# Execution proceeds here...
```

For applications using a single CC3DSimService instance, instances of classes from the cc3d.core.PyCoreSpecs module that correspond to built-in plugins and steppables that support steering provide a method `steer`. When `steer` is called on a registered cc3d.core.PyCoreSpecs instance, the underlying built-in plugin or steppable is updated according to the internal data of the cc3d.core.PyCoreSpecs instance,

```
from cc3d.CompuCellSetup.CC3DCaller import CC3DSimService
from cc3d.core.PyCoreSpecs import PottsCore, CellTypePlugin, VolumePlugin,_
    LengthConstraintPlugin

# Previous specifications for Potts, Volume, etc., here...

# Specify a length constraint for the NonCondensing cell type
length_specs = LengthConstraintPlugin()
length_specs.params_new("NonCondensing", target_length=5, lambda_length=10)
# Launch a simulation and register all specifications
cc3d_sim = CC3DSimService()
cc3d_sim.register_specs([potts_specs, cell_type_specs, volume_specs, contact_specs, blob_
    _init_specs, length_specs])
cc3d_sim.run()
cc3d_sim.init()
cc3d_sim.start()
# Execute 10k steps and update target length for the NonCondensing cell type along the_
# way
num_steps = 10000
target_length_init = length_specs["NonCondensing"].target_length
while cc3d_sim.current_step < num_steps:
    if cc3d_sim.current_step % 100 == 0:
        target_length = target_length_init * (1.0 + cc3d_sim.current_step / num_steps)
    # Calculate new length
    length_specs["NonCondensing"].target_length = target_length
    # Apply new length
    length_specs.steer()
    # Update the backend
    cc3d_sim.step()
```

### Warning

Not every built-in plugin and steppable supports steering. Calling `steer` on a cc3d.core.PyCoreSpecs module instance that does not support steering results in a cc3d.core.PyCoreSpecs.SteerableError.

CC3D projects can also use cc3d.core.PyCoreSpecs objects to specify a simulation, and in the same way. The single

difference between their deployment in CC3D and Python projects is the process of registration, which in CC3D projects is done through the `CompuCellSetup.register_specs` method in the same way as through the `CC3DSimService.register_specs` method in Python projects. Specification cannot mix `cc3d.core.PyCoreSpecs` objects and `CC3DML`. However, passing a list of `cc3d.core.PyCoreSpecs` objects to the method `cc3d.core.PyCoreSpecs.build_xml` generates CC3DML data, and likewise passing the absolute path to a `.xml` file containing a CC3DML specification, or to a `.cc3d` file of a project that uses a CC3DML specification, to the method `cc3d.core.PyCoreSpecs.from_file` generates a list of populated `cc3d.core.PyCoreSpecs` objects.

## 3.5 Visualization in Python

The CC3D Python API provides support for real-time simulation data visualization. The `CC3DSimService` method `visualize` creates a visualization frame that updates according to simulation data updates and configurable options.

```
# Launch a simulation and register all specifications
cc3d_sim = CC3DSimService()
cc3d_sim.register_specs(specs) # `specs` includes specifications for diffusion fields "F1"
                                # and "F2"
cc3d_sim.run()
cc3d_sim.init()
cc3d_sim.start()
# Show a frame of the cell field
cc3d_sim.visualize()
```

By default, `CC3DSimService.visualize` creates a frame that renders a two-dimensional view of the cell field. However, `CC3DSimService.visualize` returns a reference to the created frame that provides methods and properties to configure the frame, save an image to file, etc.,

```
# Show another frame of the field "F1" and plot every 10 steps
frame_f1 = cc3d_sim.visualize(plot_freq=10)
frame_f1.field_name = "F1"
# Show a third frame of the field "F2", limit the frames per second to 60 and label the window
frame_f2 = cc3d_sim.visualize(fps=60, name="Field F2")
frame_f2.field_name = "F2"
# Set limits on the frame for F1
frame_f1.min_range_fixed = frame_f1.max_range_fixed = True
frame_f1.min_range = 0.0
frame_f1.max_range = 1.0
# Show another cell field frame, but visualize cluster borders instead of cell borders
frame_clusters = cc3d_sim.visualize(name="Clusters")
frame_clusters.cell_borders_on = False
frame_clusters.cluster_borders_on = True
# Save an image of the initial cluster configuration
frame_clusters.save_img(file_path="clusters.png")
```

## 3.6 Concurrent Interactive Simulations

The CC3D Python API supports execution of concurrent, interactive (and interacting) simulations. While CC3D simulations are stateful in that creating two `CC3DSimService` instances in the same process results in undefined behavior, the method `service_cc3d` creates a `CC3DSimService` instance in a new, memory-isolated process using the `SimService` Python package and returns a proxy to the `CC3DSimService` instance. When using `CC3DSimService` proxies, applications can dynamically instantiate and simultaneously orchestrate an arbitrary number of simulations.

```
from cc3d.core.simservice import service_cc3d

# Proxies of CC3DSimService instances, but memory-safe
cc3d_sim1 = service_cc3d()
cc3d_sim2 = service_cc3d()
```

Proxies returned by `service_cc3d` start with the same interface as their underlying `CC3DSimService` instance and provide the same capability, though with some particularities related to support for concurrent simulation. In general, `service_cc3d` sets up a server-client architecture and relays information between a `CC3DSimService` instance (server side) and its corresponding proxy (client side) using a message passing interface. The client-side process that calls `service_cc3d` receives a proxy as the returned value, and the server-side `CC3DSimService` instance persists for as long as the proxy exists. This architecture allows multiple simulations to execute the same core specification and custom steppables, however the core specifications and custom steppables executed by a `CC3DSimService` instance are not directly accessible (*e.g.*, for steering) on the client side to prevent memory conflicts between concurrent simulation. Rather, the CC3D Python API provides alternative features to establish data pipelines with an interactive simulation launched from `service_cc3d`.

`CC3DSimService` proxies have properties `sim_input` and `sim_output` for basic data passing between the client side and the custom steppables executing in a simulation on the server side. When an object (*e.g.*, a dictionary) is set on `sim_input`, the object is copied and accessible to all custom steppables via the property `external_input`. Likewise, any custom steppable can set an object on the steppable property `external_output`, which is copied and forwarded when the `CC3DSimService` proxy instance property `sim_output` is read. For example, this data pipeline suffices to launch multiple concurrent simulations, specify the initial location of cells, and report their final location,

```
from cc3d.core.simservice import service_cc3d
from cc3d.core.PySteppables import SteppableBasePy

class TrackerSteppable(SteppableBasePy):
    """
        Simple steppable that initializes a cell at an externally specified location,
        and reports the location of the cell back to the external environment whenever the
        simulation finishes.
    """

    def __init__(self, frequency=1):
        super().__init__(frequency=frequency)
        self.cell_id = None

    def start(self):
        """Initializes a cell at an externally specified location"""
        cell_pos = self.external_input # Get data on the simulation property `sim_input`
        new_cell = self.new_cell(self.cell_type.CellType)
        self.cell_id = new_cell.id
        for i in range(5):
            for j in range(5):
                self.cell_field[cell_pos[0] + i, cell_pos[1] + j, cell_pos[2]] = new_cell

    def finish(self):
        """Reports the location of the cell back to the external environment"""
        cell = self.fetch_cell_by_id(self.cell_id)
        self.external_output = cell.xCOM, cell.yCOM, cell.zCOM # Set data on the
        # simulation property `sim_output`

    def main():
        pass
```

(continues on next page)

(continued from previous page)

```

sims = []      # Container of all running simulations
locs_init = [] # Container of all initial cell locations
for i in range(10):          # Instantiate ten concurrent simulations
    loc = i + 10, i + 10, 0   # Initial cell location
    cc3d_sim = service_cc3d() # Create the simulation instance
    cc3d_sim.register_specs(specs)
    cc3d_sim.register_steppable(steppable=TrackerSteppable)
    cc3d_sim.run()           # Run the process with the simulation; nothing is
    ↪available until after this
    cc3d_sim.sim_input = loc  # Set data on steppable property `external_input`
    cc3d_sim.init()
    cc3d_sim.start()
    locs_init.append(loc)     # Store the initial location
    sims.append(cc3d_sim)     # Store the simulation
for _ in range(10000):        # Execute 10k steps
    [cc3d_sim.step() for cc3d_sim in sims] # Execute step on each simulation
[cc3d_sim.finish() for cc3d_sim in sims] # Finish all simulations
locs_fin = [cc3d_sim.sim_output for cc3d_sim in sims] # Collect all final cell
    ↪locations

if __name__ == '__main__': # Guard for multiprocessing
    main()

```

Steppables have the property `specs` for steering capability using the CC3D Python API. In general, when a simulation is instantiated using objects from the `cc3d.core.PyCoreSpecs` module, each object is available by registered name as a property on the `specs` property and functions in the same way. The registered name of each `cc3d.core.PyCoreSpecs` is defined on the class attribute `registered_name`. For example, `LengthConstraintPlugin` from `cc3d.core.PyCoreSpecs` has the registered name `length_constraint`, and so any custom steppable executed in a simulation with `LengthConstraintPlugin` can access the `LengthConstraintPlugin` instance with `self.specs.length_constraint`.

```

from cc3d.core.simservice import service_cc3d
from cc3d.core.PySteppables import SteppableBasePy

num_steps = 10000

class LengthConstraintSteppable(SteppableBasePy):
    """A steppable that increases the target length of a length constraint during
    ↪simulation"""

    def start(self):
        """Records the initial target length for the "Noncondensing" cell type"""
        self.target_length_init = self.specs.length_constraint["NonCondensing"].target_
    ↪length

    def step(self, mcs):
        if mcs % 100 == 0:
            target_length = self.target_length_init * (1.0 + mcs / num_steps)
        ↪# Calculate new length
            self.specs.length_constraint["NonCondensing"].target_length = target_length
        ↪# Apply new length
            self.specs.length_constraint.steer()

```

(continues on next page)

(continued from previous page)

↳ # Update the backend

The CC3D Python API supports `CC3DSimService` proxy interface customization through the `SimService` service function. A service function is a simulation-specific proxy interface method that passes arguments to an underlying server-side `CC3DSimService` instance method, and returns the returned value of the server-side `CC3DSimService` instance method on the client side. Conversely, a simulation can add an internal method to its proxy when a `CC3DSimService` instance and proxy are created through `service_cc3d` by declaring a method as a service function. When a simulation declares a method as a service function, a method of the same signature is added to each proxy when a `CC3DSimService` instance and proxy are created through `service_cc3d`.

### ⚠ Warning

A service function only works when all data passed through the service function can be serialized.

A simulation can declare a method as a service function by passing it to `service_function`. For each `CC3DSimService` proxy, each service function declared by its simulation is available immediately after the simulation declares the service function and can be used on the proxy as if calling the underlying simulation method. For example, a simulation can add service functions to present an interface for steering by implementing methods that handle changes to simulation parameter values and then declaring those methods as service functions,

```
from cc3d.core.simservice import service_cc3d, service_function
from cc3d.core.PySteppables import SteppableBasePy

# Core specs initializations here, including a LengthConstraintPlugin instance...

class LengthConstraintControlSteppable(SteppableBasePy):

    def start(self):
        """Adds method `set_parameters` to simulation service interface"""
        service_function(self.set_parameters)

    def set_parameters(self, cell_type_name: str, target_length: int, lambda_length: float):
        """Updates the parameters of the length constraint on demand"""
        self.specs.length_constraint[cell_type_name].target_length = target_length
        self.specs.length_constraint[cell_type_name].lambda_length = lambda_length
        self.specs.length_constraint.steer()

def main():
    sims = []      # Container of all running simulations
    for i in range(10):          # Instantiate ten concurrent simulations
        cc3d_sim = service_cc3d()      # Create the simulation instance
        cc3d_sim.register_specs(specs) # `specs` includes a `LengthConstraintPlugin` instance
        cc3d_sim.register_steppable(steppable=LengthConstraintControlSteppable)
        cc3d_sim.init()
        cc3d_sim.start()            # Service function is added here
        cc3d_sim.set_parameters(i, 2) # Set the length constraint for this instance with the service function
        sims.append(cc3d_sim)       # Store the simulation
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__': # Guard for multiprocessing
    main()
```

## 3.7 CC3D in Jupyter Notebook

The CC3D Python API readily supports simulation work in a Jupyter Notebook. Most aforementioned functionality of the Python API works as described in a Jupyter Notebook, with a few exceptions and additions. Concurrent simulation through `service_cc3d` is not supported, and so a Jupyter Notebook can only implement a single simulation at a time. `CC3DSimService` also provides an additional method `jupyter_run_button`, which returns an `ipywidgets` toggle button that pauses and resumes a simulation.

```
from cc3d.CompuCellSetup.CC3DCaller import CC3DSimService
from IPython.display import display

cc3d_sim = CC3DSimService()
# Simulation specification here...
cc3d_sim.visualize() # Show a visualization frame
display(cc3d_sim.jupyter_run_button()) # Show a toggle button to pause/resume simulation
```

Within Jupyter Notebook, CC3D provides functionality for viewing and controlling a simulation interactively. The `CC3DSimService.visualize` function returns the visualization frame, which can be put into a `CC3DJupyterGraphicsFrameGrid` can hold any number of visualization frames (returned by the `visualize` function). This FrameGrid is useful for watching multiple different fields as the simulation runs. Set the position of frames inside FrameGrid using coordinates starting at 0,0 at the top left corner.

Frame Grid Coordinates					
0,0	0,1	0,2	...	0,n	
1,1	1,1	1,2	...	1,n	
2,0	2,1	2,2	...	2,n	
...	...	...	...	...	
n,0	n,1	n,2	...	n,n	

The FrameGrid also has a method `control_panel()`, which will display a graphical interface for controlling simulation settings during runtime.

```
from cc3d.core.GraphicsUtils.JupyterGraphicsWidget import_
    CC3DJupyterGraphicsFrameGrid

frame_field1 = cc3d_sim.visualize()
frame_field2 = cc3d_sim.visualize()
frame_field1.set_field_name('MyField1') # optional; field can also be set through the_
    control panel
frame_field2.set_field_name('MyField2') # optional; field can also be set through the_
    control panel

frame_grid = CC3DJupyterGraphicsFrameGrid(rows=1, cols=2) # 1x2 grid
frame_grid.set_frame(frame_field1, 0, 0) # left frame
frame_grid.set_frame(frame_field2, 0, 1) # right frame

frame_grid.control_panel() # optional; show graphical interface for interacting with
```

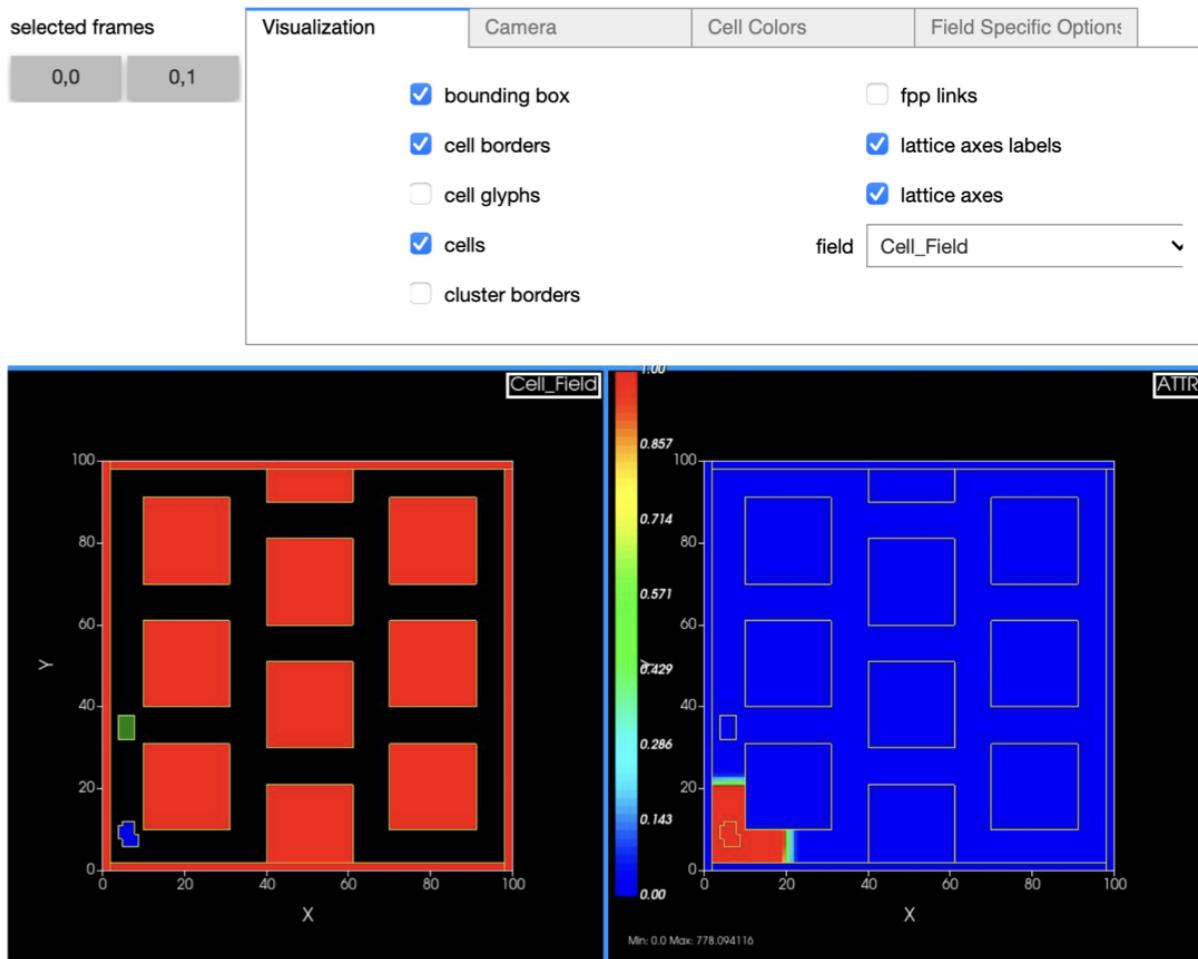
(continues on next page)

(continued from previous page)

→ [simulation](#)

```
frame_grid.show() # show the frame grid widget
```

Here is an example screenshot of the control panel and frame grid based on the `bacterium_macrophage` demo:



The settings on the control panel will only apply to active *selected frames*. Use the buttons to toggle which frames are active.



## COMMON CC3D TASKS

### 4.1 Modifying attributes of CellG object

So far, the only attributes of a cell we have been modifying were those that we attached during runtime, members of the cell dictionary. However, CC3D allows users to modify core cell attributes i.e. those which are visible to the C++ portion of the CC3D code. Those attributes are members of CellG object (see Potts3D/Cell.h in the CC3D source code) define properties of a CC3D cell. The full list of the attributes is shown in Appendix B. Here we will show a simple example how to modify some of those attributes using Python and thus alter the course of the simulation. As a matter of fact, the way to build “dynamic” simulation where cellular properties change in response to simulation events is to write a Python function/class which alters CellG object variables as simulation runs.

**CAUTION:** CC3D does not allow you to modify certain attributes, e.g. cell volume, and in case you try you will get warning and simulation will stop. Given that CC3D is under constant development with many new features being added continuously, it may happen that CC3D will let you modify attribute that should be read-only. In such a case you will most likely get cryptic error and the simulation will crash. Therefore you should be careful and double-check CC3D documentation to see which attributes can be modified.

The steppable below shows how to change `targetVolume` and `lambdaVolume` of a cell and how to implement cell differentiation (changing cell type):

```
class TypeSwitcherAndVolumeParamSteppable(SteppableBasePy):
    def __init__(self, frequency=100):
        SteppableBasePy.__init__(self, frequency)

    def start(self):
        for cell in self.cell_list:
            if cell.type == 1:
                cell.targetVolume = 25
                cell.lambdaVolume = 2.0
            elif cell.type == 2:
                cell.targetVolume = 50
                cell.lambdaVolume = 2.0

    def step(self, mcs):
        for cell in self.cell_list:
            if cell.type == 1:
                cell.type = 2
            elif cell.type == 2:
                cell.type = 1
```

As you can see in the step function we check if cell is of type 1. If it is we change it to type 2 and do analogous check/switch for cell of type 2. In the start function we initialize target volume of type 1 cells to 25 and type 2 cells

will get target volume 50. The only other thing we need to remember is to change definition of Volume plugin in the XML from:

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

to

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

to tell CC3D that volume constraint energy term will be calculated using local values (i.e. those stored in CellG object – exactly the ones we have modified using Python) rather than global settings.

Notice that we have referred to cell types using numbers. This is OK but as we have mentioned earlier using type aliases leads to much cleaner code.

## 4.2 How to Stop Steppables or Adjust Frequency

### 4.2.1 Adjust Steppable Frequency

When you create a steppable using Twedit++, the editor will write template steppable code and register it in the main Python script. By default, the steppable will run every Monte Carlo Step (MCS) as shown by `frequency=1`. You can increase the frequency of nonessential code like a chart update steppable or cell death handler to improve performance since that code could be called less often.

```
from cellsortingSteppables import cellsortingSteppable
CompuCellSetup.register_steppable(steppable=cellsortingSteppable(frequency=1))
```

We can change `frequency` argument to any non-negative value  $N$  to ensure that our steppable gets called every  $N$  MCS.

### 4.2.2 Adjust Steppable Frequency During Simulation

For some projects, it may happen that you initially want to call a steppable, say, every 50 MCS but later slow it down to every 500 MCS or not call it at all. In such a case, all you need to do is to put the following code in the step function:

```
def step(self,mcs):
    ...
    if mcs > 10000:
        self.frequency = 500
```

This will ensure that after `MCS = 10000` the steppable will be called every 500 MCS. If you want to disable steppable completely, you can always set `frequency` to a number that is greater than MCS and this would do the trick.

### 4.2.3 How to Stop a Simulation On Demand

Place the following code using Twedit++'s helper menu CC3D Python->Simulation->Stop Simulation anywhere in the steppable.

```
self.stop_simulation()
```

The inverse situation may also occur – you want to run a simulation for more MCS than originally planned.

In this case you use (CC3D Python->Simulation->SetMaxMCS)

```
self.set_max_mcs(100000)
```

to extend simulation to 100000 MCS.

## 4.3 Resizing the lattice

When you have mitosis in your simulation the numbers of cells usually grows and cells need more space. Clearly, you need a bigger lattice. CC3D lets you enlarge, shrink and shift lattice content using one simple function. There are few caveats that you have to be aware of few issues before using this functionality:

1. When resizing lattice, the new lattice is created and existing lattice is kept *alive* until all the information from old lattice is transferred to the new lattice. This might strain memory of your computer and even crash CC3D. If you have enough RAM you should be fine
2. Shrinking operation may crop portion of the lattice occupied by cells. In this case shrinking operation will be aborted.
3. When shifting lattice content, some cells might end up outside lattice boundaries. In this case operation will fail.
4. When you are using a wall of frozen cells you have to first destroy the wall, do resize/shifting operation and rebuild a wall again.

The example in CompuCellPythonTutorial/BuildWall3D demonstrates how to deal with lattice resize in the presence of wall:

```
from cc3d.core.PySteppables import *

class BuildWall3DSteppable(SteppableBasePy):

    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)

    def start(self):
        self.build_wall(self.WALL)

    def step(self, mcs):
        print('MCS=', mcs)
        if mcs == 4:
            self.destroy_wall()
            self.resize_and_shift_lattice(new_size=(80, 80, 80), shift_vec=(10, 10, 10))
        if mcs == 6:
            self.build_wall(self.WALL)
```

In the step function, during  $MCS = 4$  we first destroy the wall (we have built it in the start function), resize the lattice to dimension  $x, y, z = 80, 80, 80$  and shift content of the old lattice (but without the wall, because we have just destroyed it) by a vector  $x, y, z = 10, 10, 10$ . Finally we rebuild the wall around bigger lattice.

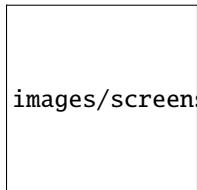
Twedit++ offers help in case you forget the syntax – go to CC3D Python->Simulation menu and choose appropriate submenu option.

The ability to dynamically resize lattice can play an important role in improving performance of your simulation. If you expect that number of cells will grow significantly during the simulation you may start with small lattice and as the number of cells increases you keep increasing lattice size in a way that “comfortably” accommodates all cells. This significantly shortens simulation run times compared to the simulation where you start with big lattice. When you work with a big lattice but have few cells CC3D will spend a lot of time probing areas occupied by Medium and this wastes machine cycles.

Along with cell field CC3D will resize all PDE fields. When lattice grows all new pixels of the PDE field are initialized with 0.0.

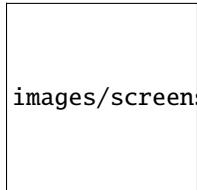
## 4.4 How to Automatically Take Screenshots

Notice how one option allows you to edit how often to save screen shots:



images/screenshot\_save\_frequency.png

And another customizes how often to save visualizations, such as line charts:



images/screenshot\_visualization\_frequency.png

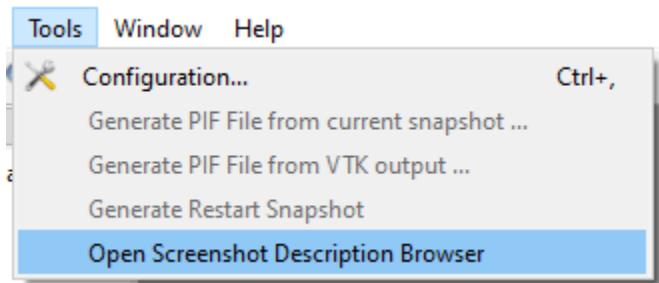
Finally, you can specify where to save the images. Checking the box for ‘=Project’ makes the screenshots save inside your CC3D project folder.

Workspace Dir  Output  =Project

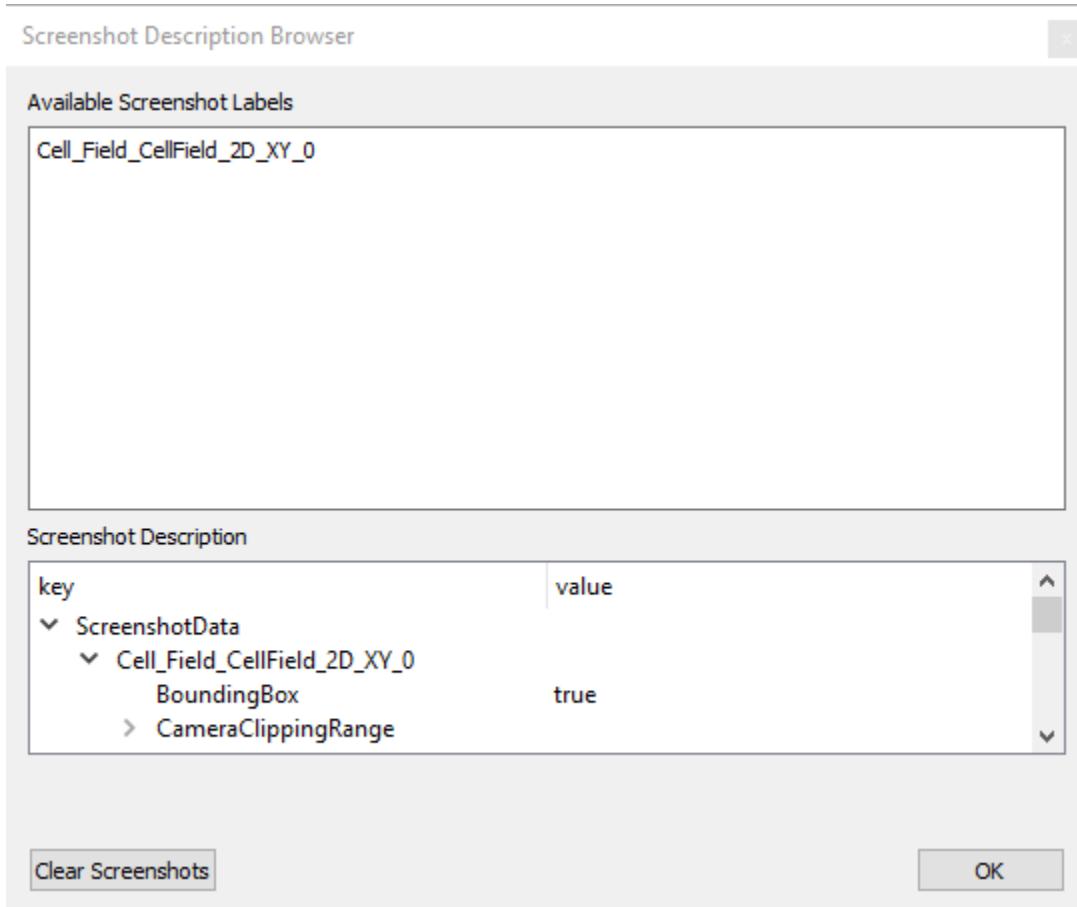
After performing these steps, Player will be set to record screen shots each time you run your simulation.

### 4.4.1 Optional: Editing the Camera Position

Once you have completed the above tutorial, you can open the settings panel like this:



In this panel, you can edit the settings by double-clicking on the values.



This data is also saved to C:/<Screenshot directory you chose>/Screenshot\_data/screenshots.json, but you should not need to edit this file manually.

#### 4.4.2 Troubleshooting

1. Ensure that your simulation is paused before editing the Configuration. (Click Step to start it.)
2. Create the directory you want to save to if it does not exist.
3. If the screen shots do not appear, you should ensure that your computer has full write permissions for the chosen directory.

### 4.5 Building a wall (it is going to be terrific. Believe me)

One of the side effects of the Cellular Potts Model occurring when lattice is filled with many cells is that some of them will stick to lattice boundaries. This happens usually when your contact energies are positive numbers. When a cell touches lattice boundaries the interface between lattice boundary and cell contributes 0 to the contact energy. Thus, when all contact energies are positive touching cell boundary is energetically favorable and as a result cell will try to lay itself along lattice boundary. To prevent this type of behavior we can create a wall of frozen cells around the lattice and ensure that contact energies between cells and the wall are very high. To build wall we first need to declare Wall cell type in the CC3DML e.g.

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

(continues on next page)

(continued from previous page)

```
<CellType TypeId="1" TypeName="A"/>
<CellType TypeId="2" TypeName="B"/>
<CellType TypeId="3" TypeName="Wall" Freeze="" />
</Plugin>
```

Notice that `Wall` type is declared as `Frozen`. Frozen cells do not participate in pixel copies but they are taken into account when calculating contact energies.

Next, in the start function we build a wall of frozen cells of type `Wall` as follows:

```
def start(self):
    self.build_wall(self.WALL)
```

If you go to CC3D Python->Simulation menu in Twedit++ you will find shortcut that will paste appropriate code snippet to build wall.

---

CHAPTER  
FIVE

---

## EXAMPLE: CONTACT-INHIBITED CELL GROWTH

Related: [Mitosis](#)

---

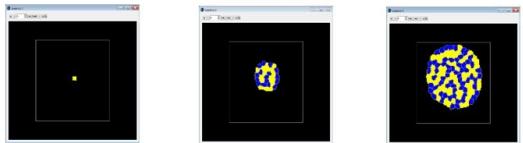
Generally speaking, cell-to-cell contact inhibits proliferation, and this is known as contact inhibition.

[Download the sample code here](#), then watch the video from the latest workshop to follow along:

[Get the slides here](#).

**CompuCell3D Workshop Module 2.3**  
**Cell Properties and Behaviors:**  
**Cell Growth and Cell Division**

Gilberto L. Thomas  
Physics Dept. - UFRGS  
Porto Alegre, RS 91501-970  
**BRAZIL**



Workshop is live-streamed, recorded and distributed

Slides, cheat-sheets and demos available at: [https://drive.google.com/drive/folders/1y4-EDktaZ6-e2F-8\\_UhtXpQlDrZW9RCp](https://drive.google.com/drive/folders/1y4-EDktaZ6-e2F-8_UhtXpQlDrZW9RCp)

**Funding:** NIH U24 EB028887, NSF 2120200, NSF 2000281, NSF 1720625

Previous Funding: NIH R01 GM1224241



---

CHAPTER  
SIX

---

## VOLUME AND CELL GROWTH

Related: [Global Volume and Surface Constraints \[Mathematics\]](#)

**Relevant Examples:**

- Example: Contact-Inhibited Cell Growth

### 6.1 Properties

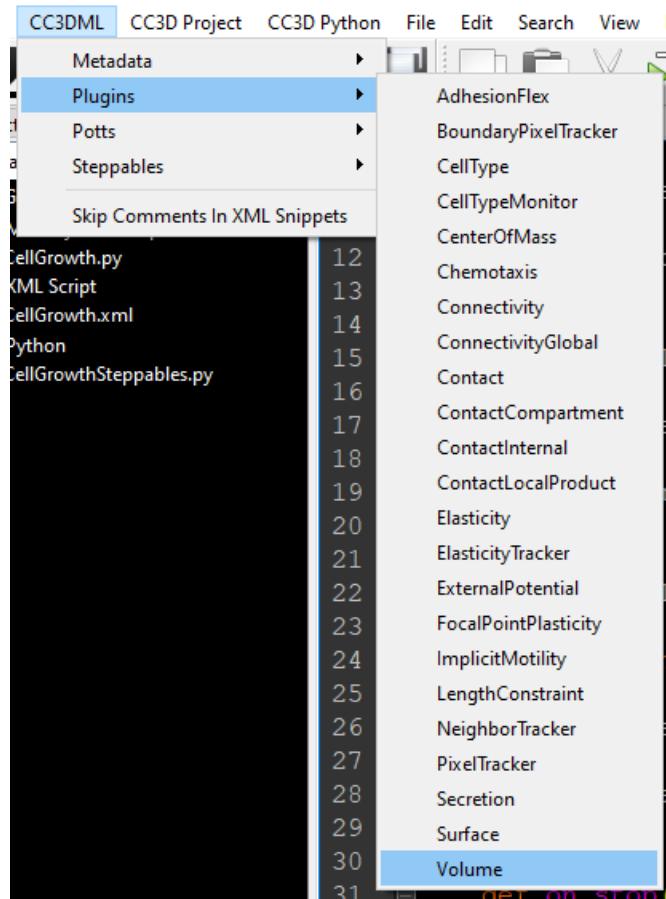
**cell.targetVolume:** the “goal” volume that a cell tries to shrink or grow to whenever possible

**cell.lambdaVolume:** the strength of the volume constraint; that is, how fast a cell will shrink/grow towards its targetVolume

---

#### How to Add the Volume Plugin in XML

- As you create your simulation, check the box for either **VolumeFlex** or **VolumeLocalFlex**.
- Otherwise, add in the XML manually with this button in **Twedit++**:



Your complete .xml file should look like this. Yours may have extra code, but that's fine.

```
<CompuCell3D Revision="3" Version="4.4.1">
    <Metadata>
        <!-- Basic properties simulation -->
        <NumberOfProcessors>1</NumberOfProcessors>
        <DebugOutputFrequency>10</DebugOutputFrequency>
    </Metadata>

    <Potts>
        <!-- Basic properties of CPM (GGH) algorithm -->
        <Dimensions x="256" y="256" z="1"/>
        <Steps>100000</Steps>
        <Temperature>10.0</Temperature>
        <NeighborOrder>1</NeighborOrder>
    </Potts>

    <Plugin Name="CellType">
        <!-- Listing all cell types in the simulation -->
        <CellType TypeId="0" TypeName="Medium"/>
        <CellType TypeId="1" TypeName="CellA"/>
        <CellType TypeId="2" TypeName="CellB"/>
    </Plugin>

```

(continues on next page)

(continued from previous page)

```

<!-- VOLUME PLUGIN -->
<Plugin Name="Volume">
    <VolumeEnergyParameters CellType="CellA" LambdaVolume="2.0" TargetVolume="50"/>
    <VolumeEnergyParameters CellType="CellB" LambdaVolume="2.0" TargetVolume="50"/>
</Plugin>

<Plugin Name="NeighborTracker">
    <!-- Module tracking neighboring cells of each cell -->
</Plugin>

<Plugin Name="Contact">
    <!-- Specification of adhesion energies goes here -->
</Plugin>

<Steppable Type="BlobInitializer">
    <!-- Initial layout of cells in the form of spherical (circular in 2D) blob -->
    <Region>
        <Center x="128" y="128" z="0"/>
        <Radius>51</Radius>
        <Gap>0</Gap>
        <Width>7</Width>
        <Types>CellA,CellB</Types>
    </Region>
</Steppable>
</CompuCell3D>

```

## 6.2 VolumeFlex vs VolumeLocalFlex

VolumeFlex is designed so that lambda volume and target volume are defined in XML. When using VolumeLocalFlex, the lambda volume and target volume must be defined in Python. (The same is true for SurfaceLocalFlex, lambda surface, and target surface).

### Example 1: VolumeFlex

XML

```

<Plugin Name="Volume">
    <VolumeEnergyParameters CellType="Somatic" LambdaVolume="2.0" TargetVolume="50"/>
    <VolumeEnergyParameters CellType="Necrotic" LambdaVolume="2.0" TargetVolume="50"/>
</Plugin>

```

### Example 2: (a separate project) VolumeLocalFlex

XML

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

Python Steppable

```

def start(self):
    for cell in self.cell_list:
        cell.targetVolume = 25
        cell.lambdaVolume = 5.0

```

## 6.3 What is Lambda?

Think of a simulation in CompuCell3D as a lazy person who wants to *minimize their energy usage* at all times, although they have a certain probability of going out of their way to do something. A lazy person may have competing interests in mind at the same time. If their “hunger” is 2 and their “sleepiness” is 100, then they will be much more likely to go to sleep than eat something. Likewise, CC3D is more likely to accept a lattice site copy attempt when the result will lower the simulation’s total energy. Essentially, **lambda is a multiplier used to control a decision about the lattice.**

## SURFACE AND CELL CONTACT

By controlling a cell's surface, you can adjust the amount of perimeter it exposes to neighbors or the Medium. We recommend to always add the Contact plugin when creating a new simulation.

### 7.1 Properties

**cell.targetSurface:** the “goal” surface that a cell tries to reshape itself to whenever possible. For roughly square or blob-shaped cells, targetSurface would be  $4 * \sqrt{\text{targetVolume}}$ .

**cell.lambdaSurface:** the strength of the surface constraint; that is, how quickly a cell will reshape itself to meet its targetSurface.

---

#### Relevant Examples:

- How to Detect Contact
- `Epithelial-Mesenchymal Transition (EMT)<example\_epithelial\_mesenchymal\_transition.html>`



---

CHAPTER  
EIGHT

---

## MITOSIS

### Related Examples

- General Mitosis Examples
  - Contact-Inhibited Cell Growth
- 

In developmental simulations, we often need to simulate cells that grow and divide. We start with a single cell and grow it. When a cell reaches a **critical volume**, it undergoes Mitosis. We **check if the cell has reached this volume threshold** at the end of every Monte Carlo Step (MCS). The folder containing this simulation is CompuCellPythonTutorial/steppableBasedMitosis

 Note

*Tip:* Be sure to turn off mitosis for dying cells. *cell\_death.rst* `See how here.

### How to Implement a Simple Mitosis Steppable

```
from cc3d.core.PySteppables import *

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

        # Optional: Customize where to place the new cell.
        # 0 - parent and child positions will be randomized between mitosis events
        # negative integer - parent appears on the 'left' of the child
        # positive integer - parent appears on the 'right' of the child
        self.set_parent_child_position_flag(-1)

    def step(self, mcs):

        cells_to_divide = []
        for cell in self.cell_list:
            if cell.volume > 50: #the critical mass, chosen arbitrarily
                cells_to_divide.append(cell)

        for cell in cells_to_divide:
            # To change mitosis mode, leave one of the below lines uncommented
```

(continues on next page)

(continued from previous page)

```

self.divide_cell_random_orientation(cell)
# Other valid options
# self.divide_cell_orientation_vector_based(cell, 1, 1, 0)
# self.divide_cell_along_major_axis(cell)
# self.divide_cell_along_minor_axis(cell)

def update_attributes(self):

    # Reduce parent target volume BEFORE cloning
    self.parent_cell.targetVolume /= 2.0

    self.clone_parent_2_child()

    # Make the cell type change every time the cell divides
    if self.parent_cell.type == self.CONDENSING:
        self.child_cell.type = self.NONCONDENSING
    else:
        self.child_cell.type = self.CONDENSING

```

**How does the `step(...)` function work?**

1. Check every cell to see if its volume is greater than 50.
2. All cells with this critical volume are placed into a list, `cells_to_divide`.
3. Then, loop over all `cells_to_divide` to perform mitosis with a built-in function such as `divide_cell_random_orientation(cell)`.
4. **`update_attributes()` is automatically called before each division. Use this to control the data of the parent and child cells.**
  - Here, `self.clone_parent_2_child()` copies over all data for you.

**Note**

It's important to use two Python loops in `step(...)` to avoid iterating over a newly-created cell. If we keep dividing cells in this loop we are adding elements to the list over which we iterate over and this might have unwanted side effects. The solution is to use use list of cells to divide as we did in the example.

We have a choice in step 3 to divide cells along a randomly-oriented plane (line in 2D), along major, minor, or user-specified axis. When using a user specified axis, you specify a vector which is perpendicular to the plane (axis in 2D) along which you want to divide the cell. This vector does not have to be normalized but it has to have length different than 0. The `update_attributes` function is called automatically each time you call any of the functions which divide cells.

**Note**

The name of the function where we update attributes after mitosis has to be exactly `update_attributes`. If it is different, CC3D will not call it automatically.

The `update_attributes` of the function is actually the heart of the mitosis module and you implement parameter adjustments for parent and child cells inside this function. It is, in general, a good practice to make sure that you update attributes of both parent and child cells. Notice that we reset target volume of the parent to 25:

```
self.parent_cell.targetVolume = 25.0
```

Had we forgotten to do that, the parent cell would keep the high target volume from before the mitosis and its actual volume would be, roughly 25 pixels. As a result, after the mitosis, the parent cell would “explode” to get its volume close to the target target volume. As a matter of fact, if we keep increasing `targetVolume` without resetting, the target volume of parent cell would be higher for each consecutive mitosis event. Therefore, you should always make sure that the attributes of parent and child cells are adjusted properly in the `update_attributes` function.

The next call in the `update_attributes` function is `self.clone_parent_2_child()`. This function is a convenience function that copies all parent cell’s attributes to the child cell. It is completely up to you to call this function or do a manual copy of select attributes from parent to child cell.

## 8.1 Deep-Copy a Cell (recommended)

`clone_parent_2_child()`: Copies all attributes of the parent cell to the child cell, including `cell.dict`.

## 8.2 Shallow-Copy a Cell

`clone_attributes(source_cell, target_cell, no_clone_key_dict_list)`: Creates a shallow copy of a cell. Parent attributes are copied, but dictionary elements, such as `cell.dict`, are skipped.

**Example:**

```
self.clone_attributes(source_cell=self.parent_cell,
                      target_cell=self.child_cell,
                      no_clone_key_dict_list=["ATTRIB_1", "ATTRIB_2"])
```

The dictionary elements ATTRIB\_1 and ATTRIB\_2

```
no_clone_key_dict_list=["ATTRIB_1", "ATTRIB_2"]
```

are not copied. Remember that you can always ignore those convenience functions and assign parent and child cell attributes manually if this gives your code the behavior you want or makes code run faster.

For example, the implementation of the `update_attributes` function where we manually set parent and child properties could look like that:

```
def update_attributes(self):
    self.child_cell.targetVolume = self.parent_cell.targetVolume
    self.child_cell.lambdaVolume = self.parent_cell.lambdaVolume
    if self.parent_cell.type == self.CONDENSING:
        self.child_cell.type = self.NONCONDENSING
    else:
        self.child_cell.type = self.CONDENSING
```

## 8.3 Remember to Grow Your Cells

You can use either one of the two XML plugins to grow your cells to the target volume of 50. Let CC3D define this for you by clicking on **CCDML -> Plugins -> Volume** in Twedit++.

```
<Plugin Name="Volume">
    <VolumeEnergyParameters CellType="Condensing" LambdaVolume="2.0" TargetVolume="50.0"/>
    <VolumeEnergyParameters CellType="NonCondensing" LambdaVolume="2.0" TargetVolume="50.0"/>
</Plugin>
```

or

```
<Plugin Name="Volume">
    <TargetVolume>50</TargetVolume>
    <LambdaVolume>2.0</LambdaVolume>
</Plugin>
```

---

### 8.3.1 Directionality of mitosis - a source of possible simulation bias

When the mitosis module divides cells (and, for simplicity, let's assume that division happens along a vertical line), then the parent cell will always remain on the same side of the line. For example, if you run have a “stem” cell that keeps dividing, all of its offspring will be created on the same side of the dividing line. What you may observe then is that, if you reassign the cell type of a child cell after mitosis, then, in certain simulations, the cell will appear to be biased to move in one direction of the lattice.

To avoid this bias, you need to call the `self.set_parent_child_position_flag` function from the `Base` class of the `Mitosis` steppable. When you call this function with argument 0, then the relative position of parent and child cells after mitosis will be randomized (this is the default behavior). When the argument is a negative integer, the child cell will always appear on the right of the parent cell. Conversely, when the argument is a positive integer, the child cell will appear always on the left-hand side of the parent cell.

## CELL DEATH

This guide covers suggested apoptosis, necrosis, and phagocytosis.

### 9.1 Death by Apoptosis

For each cell you want to kill, set its `targetVolume` to 0. The cell will begin to shrink over time. You can make this happen faster by increasing `lambdaVolume`. Once the cell's volume is small enough, you should delete it with `self.delete_cell(cell)`.

**Example step 1:** This example simulates apoptosis for B cells. In your main Steppable, write:

```
def step(self, mcs):
    for b_cell, common_surface_area in self.get_cell_neighbor_data_list(tfh_cell):
        if b_cell and b_cell.type == self.CENTROCYTE:
            b_cell.targetVolume = 0 #Starts apoptosis
            b_cell.lambdaVolume = 1000 #Optional: make the shrinkage happen very fast
```

**Example step 2:** Add this to your Main Python Script file. We use a separate Steppable to check for cell death only periodically (that is, every 100 MCS instead of every 1 step). This is optional, but it may provide a slight boost to performance.

```
from GerminalCenterMigrationSteppables import DeathSteppable

CompuCellSetup.register_steppable(steppable=DeathSteppable(frequency=100))
```

**Example step 3:** Add a new Steppable to your Main Python Script file.

```
class DeathSteppable(SteppableBasePy):
    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)

    def step(self, mcs):
        cells_to_delete = []
        for cell in self.cell_list:
            if cell.volume < 2: #Replace '2' with any arbitrary small number
                cells_to_delete.append(cell)

        for cell in cells_to_delete:
            self.delete_cell(cell)
```

**Note**

It's important to use move all cells to delete to a new list, `cells_to_delete`, so that a separate Python loop will make the calls to `self.delete_cell(cell)`. This ensures that the first loop does not lose its place as it searches for dying cells.

## 9.2 Death by Necrosis

There are many ways to achieve this, but one is to create a separate cell type, `Necrotic`, that has a very high target surface and lambda surface. Its volume can remain the same as before. This simulates the cell tearing apart, which, in real cells, creates a toxic environment for surviving neighbors. This time, there is no call to `self.delete_cell(cell)` so that the cell's detritus will remain there.

```
<Plugin Name="Volume">
    <VolumeEnergyParameters CellType="Somatic" LambdaVolume="2.0" TargetVolume="50"/>
    <VolumeEnergyParameters CellType="Necrotic" LambdaVolume="2.0" TargetVolume="50"/>
</Plugin>

<Plugin Name="Surface">
    <SurfaceEnergyParameters CellType="Somatic" LambdaSurface="2.0" TargetSurface="50"/>
    <SurfaceEnergyParameters CellType="Necrotic" LambdaSurface="2.0" TargetSurface="200"/>
</Plugin>
```

When you are ready to kill a cell, just change its type to `NECROTIC`. In this example, all cells die at Monte Carlo Step 100.

```
def step(self, mcs):
    if mcs == 100:
        for cell in self.cell_list:
            cell.type = self.NECROTIC
```

## 9.3 Death by Phagocytosis

This time, another cell will absorb the volume of the cell that dies. Think of a macrophage eating a bacterium and becoming slightly larger. This code checks every bacteria cell to see if its only neighbors are macrophage(s).

```
def step(self, mcs):
    cells_to_delete = []
    for i, bacteria in enumerate(self.cell_list_by_type(self.BACTERIA)):
        is_only_touching_macrophage = True
        macrophage = None
        for neighbor, common_surface_area in self.get_cell_neighbor_data_list(bacteria):
            if neighbor:
                if neighbor.type == self.MACROPHAGE:
                    macrophage = neighbor
                else:
                    is_only_touching_macrophage = False

        if is_only_touching_macrophage and macrophage != None:
            #Now, the macrophage eats the bacteria.
            macrophage.targetVolume += bacteria.volume
```

(continues on next page)

(continued from previous page)

```

macrophage.targetSurface += 2 * sqrt(bacteria.volume) #Try to retain volume-
→to-surface ratio
cells_to_delete.append(bacteria)

for cell in cells_to_delete:
    self.delete_cell(cell)

```

Alternative Approach 1: You could also check the length of `cell_list_by_type` to see if it is 1, but that would prevent phagocytosis from happening if the cell is touching any of the Medium.

Alternative Approach 2: Since `common_surface_area` is not used in this example, CC3D has no way to know if one cell is inside of another. You could check the shared surface area against each cell's current surface.

### Note

As in apoptosis, it's important to use move all cells to delete to a new list, `cells_to_delete`, so that a separate Python loop will make the calls to `self.delete_cell(cell)`. This ensures that the first loop does not lose its place as it searches for dying cells.

### 9.3.1 How to Turn off Mitosis for Dying Cells

If you have a separate cell type for dying cells, then just add a line like `if cell.type != self.NECROTIC`.

```

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

    def step(self, mcs):
        cells_to_divide=[]

        for cell in self.cell_list:
            if cell.type != self.NECROTIC:
                cells_to_divide.append(cell)

        for cell in cells_to_divide:
            self.divide_cell_random_orientation(cell)

    def update_attributes(self):
        # ...

```

Otherwise, for apoptosis, you could check the targetVolume:

```

for cell in self.cell_list_by_type(self.CENTROBLAST):
    if cell.targetVolume > 0:
        cells_to_divide.append(cell)

```

### 9.3.2 How to Control Division Time

If you want to divide every cell every 70 MCS, for instance, you should track each cell's last division time independently using `cell.dict`.

```
DIVISION_TIME = 70 #mcs

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

    def step(self, mcs):
        cells_to_divide=[]

        for cell in self.cell_list:
            last_div_time = cell.dict["last division time"]
            if mcs - last_div_time >= DIVISION_TIME:
                cell.dict["last division time"] = mcs
                cells_to_divide.append(cell)

        for cell in cells_to_divide:
            self.divide_cell_random_orientation(cell)

    def update_attributes(self):
        # ...
```

#### Note

You may like to implement some rules to reduce crowding, such as contact-inhibited growth or a very simple if-statement that prevents cells with too many neighbors from dividing. Using pressure tends to be more realistic.

## SECRETION PLUGIN REFERENCE

### Related:

- Secretion Guide
  - Field Secretion
  - Secretion (legacy version for pre-v3.5.0)
- 

## 10.1 FieldSecretor Methods

### 10.1.1 Secretion Modes

1. `secreteInsideCell(cell, concentration: float) -> bool` – secretion will occur on every one of the given cell’s pixels. concentration determines how much to increase the amount of the field present at that pixel.
2. `secreteInsideCellConstantConcentration(cell, concentration: float) -> bool` – secretion will occur on every one of the given cell’s pixels. The concentration will be fixed and uniform across all pixels.
3. `secreteInsideCellAtBoundary(cell, concentration: float) -> bool` – secretion takes place in pixels that are part of the cell’s boundary.
4. `secreteInsideCellAtBoundaryOnContactWith(cell, concentration: float, cell_types: list) -> bool` - secretion takes place in pixels that are part of the cell’s boundary (i.e., its outermost pixels). Secretion only happens on pixels that are in contact with cells listed in `cell_types`.
5. `secreteOutsideCellAtBoundary(cell, concentration: float) -> bool` – secretion takes place in pixels that are outside the cell but touching its boundary.
6. `secreteOutsideCellAtBoundaryOnContactWith(cell, concentration: float, cell_types: list) -> bool` - secretion takes place in pixels that are outside the cell but touching its boundary. Secretion only happens on pixels that are in contact with cells listed in `cell_types`.
7. `secreteInsideCellAtCOM(cell, concentration: float) -> bool` – secretion at the center of mass of the cell

### 10.1.2 Uptake Modes

**Important:** Uptake works as follows: when available concentration is greater than `max_amount`, then `max_amount` is subtracted from `current_concentration`, otherwise we subtract `relative_uptake*current_concentration`. Typically, `max_amount` is  $\geq 1.0$  while `relative_uptake` is between 0 and 1.0.

1. `uptakeInsideCell(cell, max_amount: float, relative_uptake: float) -> bool` – uptake will occur on every one of the given cell's pixels.
2. `uptakeInsideCellAtBoundary(cell, max_amount: float, relative_uptake: float) -> bool` – uptake takes place in pixels that are part of the cell's boundary.
3. `uptakeInsideCellAtBoundaryOnContactWith(cell, max_amount: float, relative_uptake: float, cell_types: list) -> bool` - uptake takes place in pixels that are part of the cell's boundary (i.e., its outermost pixels). Uptake only happens on pixels that are in contact with cells listed in `cell_types`.
4. `uptakeOutsideCellAtBoundary(cell, max_amount: float, relative_uptake: float) -> bool` – uptake takes place in pixels that are outside the cell but touching its boundary.
5. `uptakeOutsideCellAtBoundaryOnContactWith(cell, max_amount: float, relative_uptake: float, cell_types: list) -> bool` - uptake takes place in pixels that are outside the cell but touching its boundary. Uptake only happens on pixels that are in contact with cells listed in `cell_types`.
6. `uptakeInsideCellAtCOM(cell, max_amount: float, relative_uptake: float) -> bool` – uptake at the center of mass of the cell

### 10.1.3 Tracking Total Amount Secreted or Uptaken

The below methods can be used to get a `FieldSecretorResult` object, which will have a property called `tot_amount` that contains the summary of the secretion/uptake operation. Just add “`TotalCount`” to the name of the same method you used above. The arguments remain the same.

1. `secreteInsideCellTotalCount(...)`
2. `secreteInsideCellConstantConcentrationTotalCount(...)`
3. `secreteInsideCellAtBoundaryTotalCount(...)`
4. `secreteInsideCellAtBoundaryOnContactWithTotalCount(...)`
5. `secreteOutsideCellAtBoundaryTotalCount(...)`
6. `secreteOutsideCellAtBoundaryOnContactWithTotalCount(...)`
7. `secreteInsideCellAtCOMTotalCount(...)`

and similarly for uptake:

1. `uptakeInsideCellTotalCount(...)`
2. `uptakeInsideCellAtBoundaryTotalCount(...)`
3. `uptakeInsideCellAtBoundaryOnContactWithTotalCount(...)`
4. `uptakeOutsideCellAtBoundaryTotalCount(...)`
5. `uptakeOutsideCellAtBoundaryOnContactWithTotalCount(...)`
6. `uptakeInsideCellAtCOMTotalCount(...)`

## FIELD SECRETION | INTERACTING WITH PDE SOLVER FIELDS

### Related:

- Secretion Reference
  - Secretion Guide
  - Secretion (legacy version for pre-v3.5.0)
- 

### 11.1 Methods

1. `fieldSecretor``secreteInsideCellTotalCount``` – returns a `FieldSecretorResult`` object that contains the summary of the secretion/uptake operation. Most importantly, when we access ```total_amount` member of the `res` object we get the total amount that was added/uptaken from the chemical field e.g. :

2. `cell.lambdaVolume` –

---

Every field declared in a PDE solver is accessible by name in Python from every registered steppable using the property `field`, which allows us to retrieve and change the value of a field at a particular point by using the coordinates of the point as indices of the field. For example, if we have a PDE solver running with a field named ATTR and we would like to increment the value of ATTR at a point (10, 20, 30), we could write in a steppable,

```
self.field.ATTR[10, 20, 30] += 1
```

Likewise, we can manipulate a field using slicing operators, such as setting the value of our ATTR field to a value of 1.0 along a line,

```
self.field.ATTR[0:11, 20, 0] = 1.0
```

#### Note

The functionality described up to this point is also applicable for extra scalar and vector fields. They can also be accessed and manipulated using the `field` property of a steppable. For more on extra fields, see [Adding and managing extra fields for visualization purposes](#).

### 11.1.1 Field Secretion

PDE solvers in CC3D allow you to specify secretion properties individually for each cell type. However, there are situations where **you want only a single cell to secrete the chemical**. In this case, you have to use Secretor objects. In Twedit++, go to the CC3D Python->Secretion menu to see what options are available. Let us look at the example code to understand what kind of capabilities CC3D offers in this regard (see Demos/SteppableDemos/Secretion):

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

    def step(self, mcs):
        attr_secretor = self.get_field_secretor("ATTR")
        for cell in self.cell_list:
            if cell.type == self.WALL:
                # Choose one of the secretion methods according to your use case
                attr_secretor.secreteInsideCellAtBoundaryOnContactWith(cell, 300, [self.
                    ↵WALL])
                attr_secretor.secreteOutsideCellAtBoundaryOnContactWith(cell, 300, [self.
                    ↵MEDIUM])
                attr_secretor.secreteInsideCell(cell, 300)
                attr_secretor.secreteInsideCellAtBoundary(cell, 300)
                attr_secretor.secreteOutsideCellAtBoundary(cell, 500)
                attr_secretor.secreteInsideCellAtCOM(cell, 300)
```

#### Note

[History] As we mentioned in the introductory section, we switched capitalization conventions for Python functions. For example, we use `get_field_secretor` and not `getFieldSecretor`. However, there are function calls in the above snippet that do not follow this convention - e.g. `secreteInsideCell`. This is because those functions belong to a C++ object (here, `attr_secretor`) that is accessed through Python. We decided to keep those two conventions (snake-case for pure Python functions) and Pascal-case for C++ functions. It provides a clue for where various functions come from.

In the step function, we obtain a handle to field secretor object that operates on diffusing field ATTR. In the for loop where we go over all cells in the simulation we pick cells that are of type 3 (notice we use a numeric value here instead of an alias). Inside the loop, we use `secreteInsideCell`, `secreteInsideCellAtBoundary`, `secreteOutsideCellAtBoundary`, and `secreteInsideCellAtCOM` member functions of the secretor object to carry out secretion in the region occupied by a given cell. See the [secretion reference guide](#) for more details.

`secreteInsideCell`: increases concentration by a given amount (here 300) in every pixel occupied by a cell.

`secreteInsideCellAtBoundary` and `secreteOutsideCellAtBoundary`: increases concentration but only in pixels at the cell's boundary. The “inside” version chooses the cell's pixels (recommended) whereas the “outside” version chooses pixels touching the cell's boundary.

`secreteInsideCellAtCOM`: increases concentration for the single pixel that is closest to the cell's center of mass.

Notice that `SecretionSteppable` inherits from `SecretionBasePy`. We do this to ensure that Python-based secretion plays nicely with PDE solvers. This requires that such steppable must be called before MCS, or rather before the PDE solvers start evolving the field. If we look at the definition of `SecretionBasePy`, we will see that it inherits from `SteppableBasePy`. In the `__init__` function, it sets the `self.runBeforeMCS` flag to 1:

```
class SecretionBasePy(SteppableBasePy):
    def __init__(self, frequency=1):
```

(continues on next page)

(continued from previous page)

```
SteppableBasePy.__init__(self, frequency)
self.runBeforeMCS = 1
```

### 11.1.2 Direct (but somewhat naive) Implementation

Now, for the sake of completeness, let us implement cell secretion at the COM using alternative code:

```
field = self.field.ATTR
lmf_length = 1.0;
x_scale = 1.0
y_scale = 1.0
z_scale = 1.0
# FOR HEX LATTICE IN 2D
#     lmf_length = sqrt(2.0/(3.0*sqrt(3.0)))*sqrt(3.0)
#     x_scale = 1.0
#     y_scale = sqrt(3.0)/2.0
#     z_scale = sqrt(6.0)/3.0

for cell in self.cell_list:
    # converting from real coordinates to pixels
    x_cm = int(cell.xCOM / (lmf_length * x_scale))
    y_cm = int(cell.yCOM / (lmf_length * y_scale))

    if cell.type == 3:
        field[x_cm, y_cm, 0] = field[x_cm, y_cm, 0] + 10.0
```

As you can tell, it is significantly more work than our original solution.

### 11.1.3 Lattice Conversion Factors

In the code where we manually implement secretion at the cell's COM, we use strange-looking variables like `lmf_length`, `x_scale` and `y_scale`. CC3D allows users to run simulations on square (Cartesian) or hexagonal lattices. Under the hood, these two lattices rely on the Cartesian lattice. However, distances between neighboring pixels are different on Cartesian and hex lattices. This is what those 3 variables accomplish. The take-home message is that to convert COM coordinates on hex lattice to Cartesian lattice coordinates, we need to use converting factors. Please see writeup “**Hexagonal Lattices in CompuCell3D**” ([http://www.compcell3d.org/BinDoc/cc3d\\_binaries/Manuals/HexagonalLattice.pdf](http://www.compcell3d.org/BinDoc/cc3d_binaries/Manuals/HexagonalLattice.pdf)) for more information. To convert between hex and Cartesian lattice coordinates we can use `SteppableBasePy` built-in functions (`self.cartesian_2_hex`, and `self.hex_2_cartesian`). You can use Twedit++’s Python snippets menu: Distances → Vectors → Transformations to get code like this:

```
hex_coords = self.cartesian_2_hex(coords=[10, 20, 11])
pt = self.hex_2_cartesian(coords=[11.2, 13.1, 21.123])
```

### 11.1.4 Tracking Amount of Secreted (Uptaken) Chemical

While the ability to have fine control over how the chemicals get secreted/uptaken is a useful feature, quite often we would like to know the total amount of the chemical that was added to the simulation as a result of the call to one of the `secrete` or `uptake` functions from the secretor object.

Let us rewrite the previous example using the API that facilitates tracking the amount of chemical that was added:

```
class SecretionSteppable(SecretionBasePy):
    def __init__(self, frequency=1):
```

(continues on next page)

(continued from previous page)

```
SecretionBasePy.__init__(self, frequency)

def step(self, mcs):
    attr_secretor = self.get_field_secretor("ATTR")
    for cell in self.cell_list:
        if cell.type == 3:

            res = attr_secretor.secreteInsideCellTotalCount(cell, 300)
            print('secreted ', res.tot_amount, ' inside cell')
            res = attr_secretor.secreteInsideCellAtBoundaryTotalCount(cell, 300)
            print('secreted ', res.tot_amount, ' inside cell at the boundary')
            res = attr_secretor.secreteOutsideCellAtBoundaryTotalCount(cell, 500)
            print('secreted ', res.tot_amount, ' outside the cell at the boundary')
            res = attr_secretor.secreteInsideCellAtCOMTotalCount(cell, 300)
            print('secreted ', res.tot_amount, ' inside the cell at the COM')
```

As you can see, the calls that return the total amount of chemical added/uptaked are the same calls as we used in our previous example except we add TotalCount to the name of the function. The new function, secreteInsideCellTotalCount, returns an object called `res` that is an instance of the `FieldSecretorResult` class that contains the summary of the secretion/uptake operation. Most importantly, when we access `total_amount` member of the `res` object we get the total amount that was added/uptaken from the chemical field e.g. :

```
res = attr_secretor.secreteInsideCellTotalCount(cell, 300)
print('secreted ', res.tot_amount, ' inside cell')
```

## 11.1.5 Volume Integrals

`FieldSecretor` objects also provide convenience methods to easily and quickly compute a volume integral of a PDE solver field over a particular cell or the entire simulation domain. Say we would like to construct another steppable to be also simulated with the previously described `SecretionSteppable`, and say this additional steppable computes the volume integral of the diffusing field ATTR everywhere for each cell. Such a steppable could look like the following...

## 11.1.6 Obtaining how much chemical the cell is exposed to (sampling)

To fetch the total amount of chemical a cell is exposed to we can simply call `secretor_object.amountSeenByCell(cell)`. In more detail

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

    def step(self, mcs):
        attr_secretor = self.get_field_secretor("ATTR")
        for cell in self.cell_list:
            print('Cell exposed to ', attr_secretor.amountSeenByCell(cell), 'units of '
                  'ATTR')
```

```
class IntegralSteppable(SteppableBasePy):
    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)

    def step(self, mcs):
```

(continues on next page)

(continued from previous page)

```

attr_secretor = self.get_field_secretor("ATTR")
total_attr = attr_secretor.totalFieldIntegral()
for cell in self.cell_list:
    cell_total_attr = attr_secretor.amountSeenByCell(cell)

```

Like in `SecretionSteppable`, a field secretor object is obtained for the diffusing field ATTR. However, `IntegralSteppable` computes the volume integral of the ATTR field over the simulation domain using the field secretor method `totalFieldIntegral` (and stores it in `total_attr`). Likewise, in a loop over every cell, `IntegralSteppable` then computes the volume integral of the ATTR field over the domain of each cell using the field secretor method `amountSeenByCell` by simply passing as argument a cell of interest (and stores it in `cell_total_attr`).

### 11.1.7 Algorithmic Considerations

Note that, in the previous example, `IntegralSteppable` inherits from `SteppableBasePy` instead of from `SecretionBasePy`. This distinction is important because CC3D calls `step` on all steppables that inherit from `SteppableBasePy` *after* executing diffusion by the PDE solvers. In our case, we are then enforcing that computing volume integrals occurs *after* diffusion and secretion have been implemented for a simulation step. If we were to simulate `SecretionSteppable` and `IntegralSteppable` with a PDE solver, then the order of calls to `step` would be executed as follows,

- `SecretionSteppable` instance performs cell-based secretion for ATTR field
- PDE solver performs diffusion of ATTR field
- `IntegralSteppable` instance computes volume integrals of ATTR field



## CHEMOTAXIS ON A CELL-BY-CELL BASIS

Just as `secretion` is typically defined for cell types, the same applies to chemotaxis. And, as in the case of secretion, there is an easy way to implement chemotaxis on a cell-by-cell basis. You can find a relevant example in `Demos/PluginDemos/chemotaxis_by_cell_id`.

Related: [Chemotaxis Plugin](#)

Let us look at the code:

```
from cc3d.core.PySteppables import *

class ChemotaxisSteering(SteppableBasePy):
    def __init__(self, frequency=100):
        SteppableBasePy.__init__(self, frequency)

    def start(self):

        for cell in self.cell_list:
            if cell.type == self.cell_type.Macrophage:
                cd = self.chemotaxisPlugin.addChemotaxisData(cell, "ATTR")
                cd.setLambda(20.0)
                cd.assignChemotactTowardsVectorTypes([self.cell_type.Medium, self.cell_
→type.Bacterium])
                break

    def step(self, mcs):
        if mcs > 100 and not mcs % 100:
            for cell in self.cell_list:
                if cell.type == self.cell_type.Macrophage:

                    cd = self.chemotaxisPlugin.getChemotaxisData(cell, "ATTR")
                    if cd:
                        lm = cd.getLambda() - 3
                        cd.setLambda(lm)
                    break
```

Before we start analyzing this code let's look at CC3DML declaration of the chemotaxis plugin:

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

(continues on next page)

(continued from previous page)

```
</ChemicalField>
</Plugin>
```

As you can see we have commented out `ChemotaxisByType` but leaving information about fields so that the chemotaxis plugin can fetch pointers to the fields. Clearly, leaving such a definition of chemotaxis in the CC3DML would not affect the simulation. However, as you can see in the Python steppable code, we define chemotaxis on a cell-by-cell basis. We loop over all cells, and, when we encounter a cell of type Macrophage, we assign to it an object called `ChemotaxisData` (we use `self.chemotaxisPlugin.addChemotaxisData` function to do that). The `ChemotaxisData` object allows the definition of all chemotaxis properties available via CC3DML but here we apply them to single cells. In our example code, we set `lambda` to describe chemotaxis strength and cell types that don't inhibit chemotaxis by touching our cell (in other words, the cell experiences chemotaxis when it touches cell types listed in `assignChemotactTowardsVectorTypes` function).

Notice `break` instruction at the end of the loop. It ensures that the for loop that iterates over all cells stops after it encounters the first cell of type Macrophage.

In the step function iterate through all cells and search for the first occurrence of Macrophage cell (`break` instruction at the end of this function will ensure it). This time, however, instead of adding chemotaxis data we fetch `ChemotaxisData` object associated with a cell. We extract `lambda` and decrease it by 3 units. The net result of several operations like that are that `lambda` chemotaxis will go from a positive number to negative number, and the cell that initially chemotaxed up the concentration gradient will now start moving away from the source of the chemical.

When you want to implement chemotaxis using alternative calculations with saturation terms, all you need to do is to add `cd.setSaturationCoef` function call to enable the type of chemotaxis that corresponds to the CC3DML to the following call:

```
<ChemotaxisByType ChemotactTowards="Medium, Bacterium" Lambda="1.0" SaturationCoef="100.0"
  ↵ " Type="Macrophage"/>
```

The Python code would look like:

```
for cell in self.cell_list:
    if cell.type == self.cell_type.Macrophage:
        cd = self.chemotaxisPlugin.addChemotaxisData(cell, "ATTR")
        cd.setLambda(1.0)
        cd.setSaturationCoef(100.0)
        cd.assignChemotactTowardsVectorTypes([self.cell_type.Medium, self.cell_type.
  ↵ Bacterium])
```

If we want to replicate the following CC3DML version of chemotaxis for a single cell:

```
<ChemotaxisByType ChemotactTowards="Medium, Bacterium" Lambda="1.0" SaturationLinearCoef=
  ↵ "10.1" Type="Macrophage"/>
```

we would use the following Python snippet:

```
for cell in self.cell_list:
    if cell.type == self.cell_type.Macrophage:
        cd = self.chemotaxisPlugin.addChemotaxisData(cell, "ATTR")
        cd.setLambda(1.0)
        cd.setSaturationLinearCoef(10.1)
        cd.assignChemotactTowardsVectorTypes([self.cell_type.Medium, self.cell_type.
  ↵ Bacterium])
```

If we want to replicate the following CC3DML version of chemotaxis for a single cell:

```
<ChemotaxisByType ChemotactTowards="Medium, Bacterium" Lambda="1.0" LogScaledCoef="1.0"
    Type="Macrophage"/>
```

we would use the following Python snippet:

```
for cell in self.cell_list:
    if cell.type == self.cell_type.Macrophage:
        cd = self.chemotaxisPlugin.addChemotaxisData(cell, "ATTR")
        cd.setLambda(1.0)
        cd.setLogScaledCoef(1.0)
        cd.assignChemotactTowardsVectorTypes([self.cell_type.Medium, self.cell_type.
    ↪Bacterium])
```



---

CHAPTER  
THIRTEEN

---

## STEERING – CHANGING CC3DML PARAMETERS ON-THE-FLY.

CC3D 4.0.0 greatly simplifies modification of CC3DML parameters from Python script as the simulation runs. we call it programmatic steering.

Imagine that we would like to increase cell membrane fluctuation amplitude (in CC3D terminology Temperature) every 100 MCS by 1 unit. The Temperature is declared in the CC3DML so, unlike variables declared in Python script, we dont have a direct access to it.

Let's look at CC3DML code first:

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

The way CC3D 4.x solves this problem is very much inspired by Javascript/HTML approach. First you tag element that you wish to change using id tag as shown below:

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

Here we add id="temp\_elem". If you have multiple id tags for in your XML script we require that they are unique.

After we tagged the CC3DML elements we can easily access and modify them on-the-fly as shown below

```
class TemperatureSteering(SteppableBasePy):
    def __init__(self, frequency=100):
        SteppableBasePy.__init__(self, frequency)

    def step(self, mcs):
        temp_elem = self.get_xml_element('temp_elem')
        temp_elem.cdata = float(temp_elem.cdata) + 1
```

In the step function we first access the tagged element:

```
temp_elem = self.get_xml_element('temp_elem')
```

and then we modify its cdata portion using

```
temp_elem.cdata = float(temp_elem.cdata) + 1
```

After this last modification CC3D will take a notice that the change has been made and will update appropriate internal variables.

## 13.1 XML Element Structure

Each XML (CC3DML is also a valid XML) has the following structure

```
<MyMLElement attr_1="attr_1_value" attr_2="attr_2_value">cdata</MyMLElement>
```

For example in the following element

```
<Energy Type1="Medium" Type2="Condensing">10</Energy>
```

the element name is Energy. It has two attributes Type1 with value Medium and Type2 with value Condensing. And the value of cdata component is 10

Similarly, the element ChemotaxisByType

```
<ChemotaxisByType Type="Macrophage" Lambda="20"/>
```

has two attributes Type and Lambda with values Macrophage and 20 respectively.

## 13.2 Modifying CC3DML attributes

If we want to modify attributes of the XML element we use similar approach to the one outlined above. We tag element we want modify and then update attributes. Here is an example

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

and the Python code that modifies Lambda attribute of ChemotaxisByType

```
from cc3d.core.PySteppables import *

class ChemotaxisSteering(SteppableBasePy):
    def __init__(self, frequency=100):
        SteppableBasePy.__init__(self, frequency)

    def step(self, mcs):
        if mcs > 100 and not mcs % 100:
            macro_chem_elem = self.get_xml_element('macro_chem')
            macro_chem_elem.Lambda = float(macro_chem_elem.Lambda) - 3
```

As you can see the syntax is quite straightforward. We first fetch the reference to the XML element (we tagged it using id="macro\_chem"):

```
macro_chem_elem = self.get_xml_element('macro_chem')
```

and modify its `Lambda` attribute

```
macro_chem_elem = self.get_xml_element('macro_chem')
macro_chem_elem.Lambda = float(macro_chem_elem.Lambda) - 3
```

Two things are worth mentioning here.

1. when we access attribute we use the name of the attribute and “dot” it with reference to the XML element we fetched:

```
macro_chem_elem.Lambda
```

2. `cdata` and attributes are returned as strings so before doing any arithmetic operation we need to convert strings to appropriate Python types. Here we convert string returned by `macro_chem_elem.Lambda` (first call will return string `20`) to floating point number `float(macro_chem_elem.Lambda)` and subtract `3`. When we assign it back to the attribute we do not need to convert to string - CC3D will handle this conversion automatically

Full example of steering can be found in `Demos/Models/bacterium_makrophage_2D_steering`

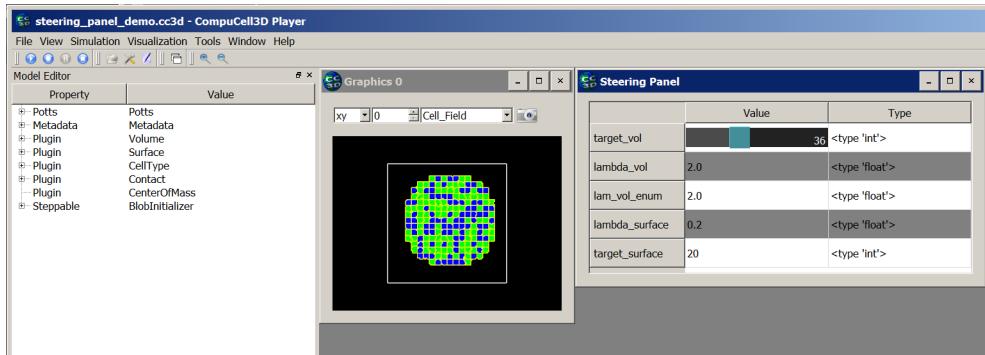


## STEERING – CHANGING PYTHON PARAMETERS USING GRAPHICAL USER INTERFACE.

In the previous section we outlined how to programmatically change CC3DML parameters. In this section we will show you how to create a graphical panel where you can use sliders and or pull down list to control parameters defined in the Python scripts. this type of interaction is very desirable because you can try different values of parameters without writing complicated procedural code that alters the parameters. It also provides you with a very convenient way to create truly interactive simulations.

In our demo suite we have included examples (e.g. Demos/SteeringPanel/steering\_panel\_demo) that demonstrate how to setup such interactive simulations. In this section we will explain all coding that is necessary to accomplish this task.

When you run open Demos/SteeringPanel/steering\_panel\_demo in CC3D and run it the screen will look as follows:



All the code that is necessary to get steering panel for Python parameters working will reside in Python steppable file. To specify steering panel we have to define `add_steering_panel` function to our steppable (the function has to be called exactly that):

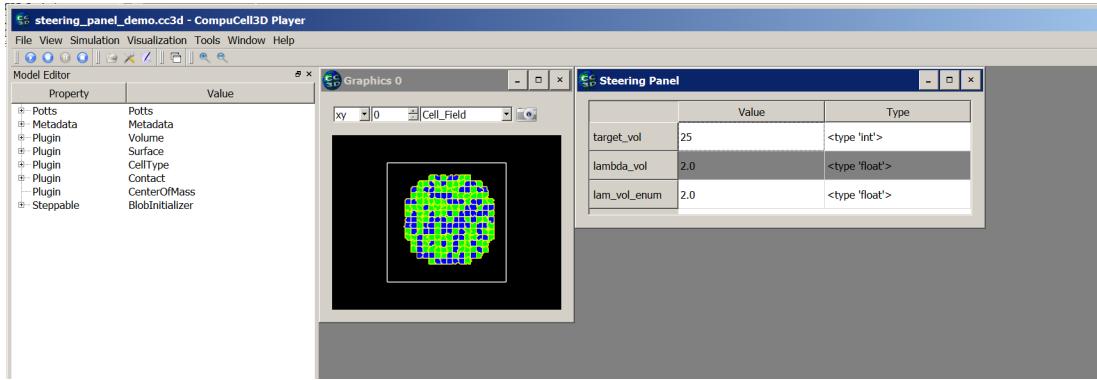
```
class VolumeSteeringSteppable(SteppableBasePy):
    def __init__(self, frequency=10):
        SteppableBasePy.__init__(self, frequency)

    def add_steering_panel(self):
        self.add_steering_param(name='target_vol', val=25, min_val=0, max_val=100, widget_name='slider')
        self.add_steering_param(name='lambda_vol', val=2.0, min_val=0, max_val=10.0, decimal_precision=2, widget_name='slider')
        self.add_steering_param(name='lam_vol_enum', val=2.0, min_val=0, max_val=10.0, decimal_precision=2, widget_name='slider')
```

The `add_steering_panel` function requires several arguments:

- `name` - specifies the label of the parameters to be displayed in the gui
- `val` - specifies current value of the python parameter
- `min_val` and `max_val` - specify range of parameters. those are optional and by default CC3D will pick some reasonable range given the `val` parameter. However, we recommend that you specify the allowable parameters range
- `widget_name` - a string that specifies the widget via which you will be changing the parameter. The allowed options are: `slider` - it wil create a slider, `combobox`, `pulldown`, `pull-down` will pull-down list for discrete values, and no argument will resort to a editable field where you have to type the parameter
- `decimal_precision` - specifies how many decimal places are to be displayed when changing parameters. Default value is 0

Once you add such function to the steppable and start the simulation the steering panel will pop-up but it will have no functionality. Panel will have just 3 entries as show below:



### Note

You can add steering panel from multiple steppables. In such a case CC3D will gather all parameters you defined in the `add_steering_panel` function and display them in a single panel. For example, if our code has two steppables and we add steering parameters in both of them:

```
class VolumeSteeringSteppable(SteppableBasePy):
    def __init__(self, frequency=10):
        SteppableBasePy.__init__(self, frequency)

    def add_steering_panel(self):
        self.add_steering_param(name='target_vol', val=25, min_val=0, max_val=100,
                               widget_name='slider')
        self.add_steering_param(name='lambda_vol', val=2.0, min_val=0, max_val=10.0,
                               decimal_precision=2, widget_name='slider')
        self.add_steering_param(name='lam_vol_enum', val=2.0, min_val=0, max_val=10.0,
                               decimal_precision=2, widget_name='slider')

class SurfaceSteeringSteppable(SteppableBasePy):
    def __init__(self, frequency=10):
        SteppableBasePy.__init__(self, frequency)
```

(continues on next page)

(continued from previous page)

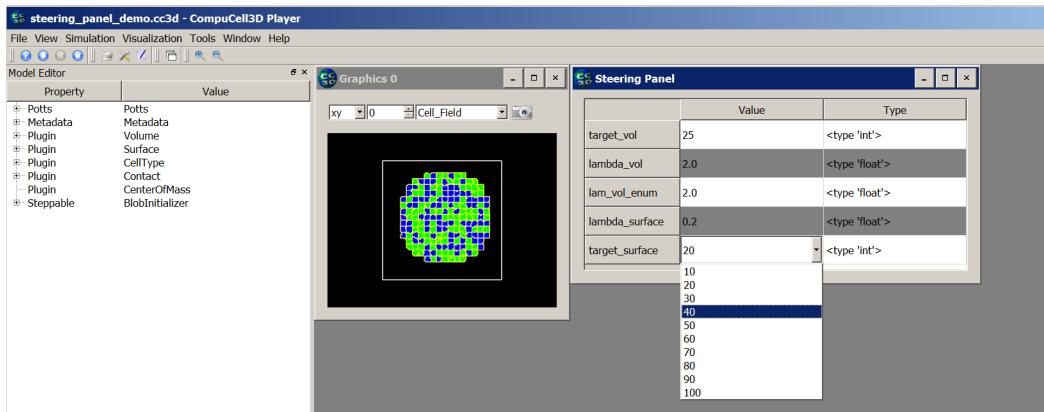
```

def add_steering_panel(self):
    #adding slider
    self.add_steering_param(name='lambda_surface', val=0.2, min_val=0, max_val=10.0,
    decimal_precision=2,
                           widget_name='slider')

    # adding combobox
    self.add_steering_param(name='target_surface', val=20, enum=[10, 20, 30, 40, 50, 60,
    ↪70, 80, 90, 100],
                           widget_name='combobox')

```

we will end up with a panel that looks as follows :



Notice that the last parameter `target_surface` uses `combobox` and appropriately the widget displayed is a pull-down-list

Now that we know how to specify steering panel let's learn how to implement interactivity. All we need to do is to implement another function in the steppable - `process_steering_panel_data`. Let's look at the example:

```

class VolumeSteeringSteppable(SteppableBasePy):
    def __init__(self, frequency=10):
        SteppableBasePy.__init__(self, frequency)

    def add_steering_panel(self):
        self.add_steering_param(name='target_vol', val=25, min_val=0, max_val=100,
        ↪widget_name='slider')
        self.add_steering_param(name='lambda_vol', val=2.0, min_val=0, max_val=10.0,
        ↪decimal_precision=2, widget_name='slider')
        self.add_steering_param(name='lam_vol_enum', val=2.0, min_val=0, max_val=10.0,
        ↪decimal_precision=2, widget_name='slider')

    def process_steering_panel_data(self):
        target_vol = self.get_steering_param('target_vol')
        lambda_vol = self.get_steering_param('lambda_vol')

        for cell in self.cell_list:

            cell.targetVolume = target_vol
            cell.lambdaVolume = lambda_vol

```

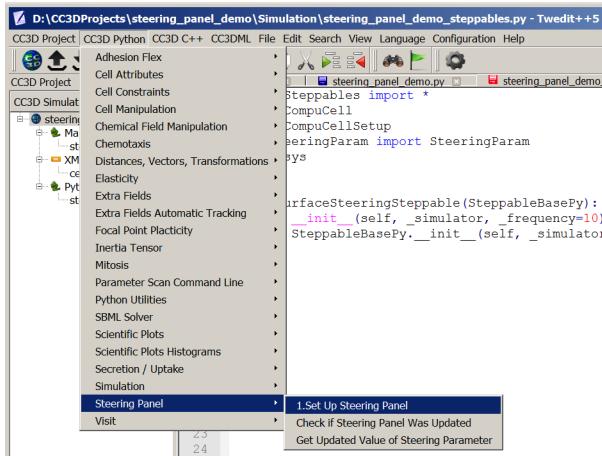
Inside `process_steering_panel_data` (the function has to be called exactly that) we read the current value indicated in the steering panel using convenience function `get_steering_param`. In our example we are reading two parameter values from the panel -`target_val` and `lambda_val`. Once we fetched the values from the panel we iterate over all cells and modify `targetVolume` and `lambdaVolume` parameters of every cell.

**Important:** `process_steering_panel_data` gets called only when the user modified the values in the steering panel by either moving a slider, changing entry in the pull-down list or changing the parameter value using text field. This means that potentially expensive loops that alter parameters are not executed every MCS but only if panel entries have changed. you can manually check if the panel values have changed by adding to your code steppable's convenience function `steering_param_dirty()`. You do not have to do that but just in case you would like to get flag indicating whether panel has change or not all that's required is simple code like that:

```
def process_steering_panel_data(self):
    print('all dirty flag=', self.steering_param_dirty())
```

As you can see by adding two functions to the steppable - `add_steering_panel` and `process_steering_panel_data` you can create truly interactive simulations where you can have a direct control over simulations. Tool like that can be especially useful in the exploratory phases of your model building where you want to quickly see what impact a given parameter has on the overall simulation.

**Important .** You can simplify setting up of interactive steering using Twedit Python helpers menu. Simply, go to CC3D Python -> Steering Panel menu and choose 1. Setup Steering Panel option:



## REPLACING CC3DML WITH EQUIVALENT PYTHON SYNTAX

Some modelers prefer using Python only and skipping XML entirely. CC3D has special Python syntax that allows users to replace CC3DML with Python code. Manual conversion is possible but as you can predict quite tedious. Fortunately Twedit++ has nice shortcuts that converts existing CC3DML (and for that matter any XML) into equivalent Python syntax that can be easily incorporated into CC3D code. In Twedit++ all you have to do is right click XML file in the project panel and you will see option Convert XML To Python. When you choose this option Twedit++ will generate Python syntax which can replace your XML:

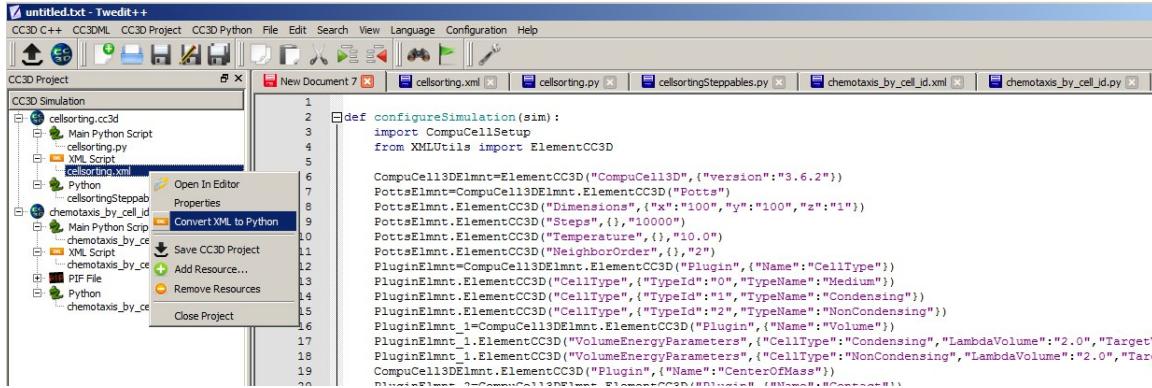


Figure 17 Generating Python code that replaces XML in Twedit++.

If we look at the XML code:

```

<CompuCell3D version="3.6.2">

    <Potts>
        <Dimensions x="100" y="100" z="1"/>
        <Steps>10000</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"/>
    </Plugin>

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

```

(continues on next page)

(continued from previous page)

```
<VolumeEnergyParameters CellType="NonCondensing"
    LambdaVolume="2.0" TargetVolume="25"/>
</Plugin>
```

And then at equivalent Python code:

```
def configureSimulation():

    from cc3d.core.XMLUtils import ElementCC3D

    CompuCell3DElmnt = ElementCC3D("CompuCell3D", {"version": "4.0.0"})
    PottsElmnt = CompuCell3DElmnt.ElementCC3D("Potts")
    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"})
    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"})
```

We can see that there is one-to-one correspondence. We begin by creating top level element CompuCell3D:

```
CompuCell3DElmnt = ElementCC3D("CompuCell3D", {"version": "4.0.0"})
```

We attach a child element (**Potts**) to CompuCell3D element and a return value of this call is object representing Potts element:

We looks at the XML and notice that **Potts** element has several child elements – e.g. **Dimensions**, **Temperature** etc... We attach all of these child elements to **Potts** element:

```
PottsElmnt.ElementCC3D("Dimensions", {"x": "100", "y": "100", "z": "1"})
PottsElmnt.ElementCC3D("Temperature", {}, "10.0")
```

We hope you see the pattern. The general rule is this. To create root element you use function **ElementCC3D** from **XMLUtils`** - see how we created ``CompuCell3D element. When you want to attach child element we call **ElementCC3D** member function of the parent element e.g.:

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

This syntax can be represented in a more general form:

```
childElementObject = parentElementObject.ElementCC3D(Name_Of_Element, {attributes},
    ↴ Element_Value)
```

Each call to **ElementCC3D** returns **ElementCC3D** object. When we call **ElementCC3D** to create root element (here **CompuCell3D**) this call will return root element object. When we call **ElementCC3D** to attach child element this call

returns child element object.

Notice that at the end of the autogenerated Python code replacing XML we have function the following line:

```
CompuCellSetup.setSimulationXMLDescription(CompuCell3DElmnt)
```

This line is actually very important and it passes root element of the CC3DML to the CompuCell3D core code for initialization. It is interesting that by passing just one node (one object representing single XML element – here CompuCell3D) we are are actually passing entire XML. As you probably can guess, this is because we are dealing with recursive data structure.

Notice as well that our code sits inside configureSimulation function, We need to call this function from Python main script to ensure that XML replacement code gets processed. See Demos/CompuCellPythonTutorial/PythonOnlySimulations for examples of a working code:

```
from cc3d import CompuCellSetup

def configure_simulation():
    from cc3d.core.XMLUtils import ElementCC3D

    cc3d = ElementCC3D("CompuCell3D")
    potts = cc3d.ElementCC3D("Potts")
    potts.ElementCC3D("Dimensions", {"x": 100, "y": 100, "z": 1})

    ...
    CompuCellSetup.setSimulationXMLDescription(cc3d)

configure_simulation()

CompuCellSetup.run()
```

The actual placement of configureSimulation function in the main script matters. It has to be called right before

```
CompuCellSetup.run()
```

**Finally, one important remark:** Twedit++ has CC3DML helper menu which pastes ready-to-use CC3DML code for all available modules. This means that when you work with XML and you want to add cell types, insert syntax for new modules etc... You can do it with a single click. When you work with Python syntax replacing XML, all modifications to the autogenerated code must be made manually.



---

CHAPTER  
SIXTEEN

---

## CELL MOTILITY. APPLYING FORCE TO CELLS.

In some CC3D simulations we need make cells move in certain direction. Sometimes we do it using chemotaxis energy term (if indeed in real system that chemotaxis is the reason for directed motion) and sometimes we simply apply energy term which simulates a force. In the CC3D manual we show how to apply constant force to all cells or on a type-by-type basis. Here let us concentrate on a situation where we apply force to individual cells and how change its value and the direction. You can check simulation code in Demos/CompuCellPythonTutorial/CellMotility. To be able to use force in our simulation ( we need to include ExternalPotential plugin in the CC3DML:

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

Let us look at the steppable code:

```
from random import uniform

class CellMotilitySteppable(SteppableBasePy):
    def __init__(self, frequency=10):
        SteppableBasePy.__init__(self, frequency)

    def start(self):

        # iterating over all cells in simulation
        for cell in self.cell_list:
            break
        # Make sure ExternalPotential plugin is loaded
        # negative lambdaVecX makes force point in the positive direction
        # force component along X axis
        cell.lambdaVecX = 10.1 * uniform(-0.5, 0.5)
        # force component along Y axis
        cell.lambdaVecY = 10.1 * uniform(-0.5, 0.5)
        #           cell.lambdaVecZ=0.0 # force component along Z axis

    def step(self, mcs):

        for cell in self.cell_list:
            # force component along X axis
            cell.lambdaVecX = 10.1 * uniform(-0.5, 0.5)
            # force component along Y axis
            cell.lambdaVecY = 10.1 * uniform(-0.5, 0.5)
```

Once ExternalPotential plugin has been loaded we assign a constant force in a given direction by initializing lambdaVecX, lambdaVecY, lambdaVecZ cell attributes.

 Note

When pushing cell along X axis toward higher X values (i.e. to the right) use `lambdaVecX` negative. When pushing to the left use positive values.

In the start function we assign random values of X and Y components of the force. The `uniform(-0.5, 0.5)` function from the Python random module picks a random number from a uniform distribution between -0.5 and 0.5.

In the step function we randomize forces applied to the cells in the same way we did it in start function.

As you can see the whole operation of applying force to any given cell in the CC3D is very simple.

The presented example is also very simple. But you can imagine more complex scenarios where the force depends on the velocity, of neighboring cells. This is however beyond the scope of this introductory

---

CHAPTER  
**SEVENTEEN**

---

## **SETTING CELL MEMBRANE FLUCTUATION ON A CELL-BY-CELL BASIS**

As you probably know the (in)famous Temperature parameter used in CPM modeling represents cell membrane fluctuation amplitude. When you increase temperature cell boundary gets jagged and if you decrease it cells may freeze. One problem with global parameter describing membrane fluctuation is that it applies to all cells. Fortunately in CC3D you may set membrane fluctuation amplitude on per-cell-type basis or individually for each cell. The code that does it is very simple:

```
cell.fluctAmpl = 50
```

From now on all calculations involving the cell for which we set membrane fluctuation amplitude will use this new value. If you want to undo the change and have global temperature parameter describe membrane fluctuation amplitude you use the following code:

```
cell.fluctAmpl = -1
```

In fact, this is how CC3D figures out whether to use local or global membrane fluctuation amplitude. If `fluctAmpl` is a negative number CC3D uses global parameter. If it is greater than or equal to zero local value takes precedence.

In Twedit++ go to CC3D Python->Cell Attributes-> Fluctuation Amplitude in case you forget the syntax.



---

CHAPTER  
EIGHTEEN

---

## DIVIDING CLUSTERS (AKA COMPARTMENTAL CELLS)

Related: Compartmentalized Cells, ContactInternal Plugin, and related XML

In compartmental models, a single cell is actually composed of several compartments. Each compartment is somewhat like an individual cell in that its behaviors can be specified independently, yet all the compartments can be treated as a single entity called a “cluster.” A cluster is a collection of compartment cells with the same `clusterId`. If you use “simple” (non-compartmentalized) cells, then you can check that each such cell has a distinct id and `clusterId`. An example simulation can be found in `CompuCellPythonTutorial/clusterMitosis`.

Let’s look at how we can divide “compact,” that is, “blob-shaped,” clusters.

```
from cc3d.core.PySteppables import *

class MitosisSteppableClusters(MitosisSteppableClustersBase):

    def __init__(self, frequency=1):
        MitosisSteppableClustersBase.__init__(self, frequency)

    def step(self, mcs):

        for cell in self.cell_list:
            cluster_cell_list = self.get_cluster_cells(cell.clusterId)
            print("DISPLAYING CELL IDS OF CLUSTER ", cell.clusterId, "CELL. ID=", cell.
id)
            for cell_local in cluster_cell_list:
                print("CLUSTER CELL ID=", cell_local.id, " type=", cell_local.type)

        mitosis_cluster_id_list = []
        for compartment_list in self.clusterList:
            # print( "cluster has size=",compartment_list.size())
            cluster_id = 0
            cluster_volume = 0
            for cell in CompartmentList(compartment_list):
                cluster_volume += cell.volume
                cluster_id = cell.clusterId

            # condition under which cluster mitosis takes place
            if cluster_volume > 250:
                # instead of doing mitosis right away we store ids for clusters which
                # should be divide.
                # This avoids modifying cluster list while we iterate through it
```

(continues on next page)

(continued from previous page)

```

mitosis_cluster_id_list.append(cluster_id)

for cluster_id in mitosis_cluster_id_list:

    self.divide_cluster_random_orientation(cluster_id)

    # # other valid options - to change mitosis mode leave one of the below
    ↵lines uncommented
    # self.divide_cluster_orientation_vector_based(cluster_id, 1, 0, 0)
    # self.divide_cluster_along_major_axis(cluster_id)
    # self.divide_cluster_along_minor_axis(cluster_id)

def update_attributes(self):
    # compartments in the parent and child clusters are
    # listed in the same order so attribute changes require simple iteration through
    ↵compartment list
    compartment_list_parent = self.get_cluster_cells(self.parent_cell.clusterId)

    for i in range(len(compartment_list_parent)):
        compartment_list_parent[i].targetVolume /= 2.0
    self.clone_parent_cluster_2_child_cluster()

```

The steppable is quite similar to the mitosis steppable which works for non-compartmental cells. This time, after mitosis happens, you have to reassign properties of children compartments and of parent compartments which usually means iterating over a list of compartments. Conveniently, this iteration is quite simple, and SteppableBasePy class has a convenience function called `get_cluster_cells` which returns a list of cells belonging to a cluster with a given `clusterId`:

```
compartment_list_parent = self.get_cluster_cells(self.parent_cell.clusterId)
```

The call above returns a list of cells in a cluster with `clusterId` specified by `self.parent_cell.clusterId`. In the subsequent for-loop, we iterate over a list of cells in the parent cluster and assign appropriate values of volume constraint parameters. Notice that `compartment_list_parent` is indexable (ie. we can access directly any element of the list provided our index is not out of bounds).

```
for i in range(len(compartment_list_parent)):
    compartment_list_parent[i].targetVolume /= 2.0
```

Notice that nowhere in the update attribute function we have modified cell types. This is because, by default, the cluster mitosis module assigns cell types to all the cells of the child cluster and it does it in such a way that the child cell looks like a quasi-clone of the parent cell.

The next call in the `update_attributes` function is `self.clone_parent_cluster_2_child_cluster()`. This copies all the attributes of the cells in the parent cluster to the corresponding cells in the child cluster. If you would like to copy attributes from parent to child cell skipping select ones you may use the following code:

```
compartment_list_parent = self.get_cluster_cells(self.parent_cell.clusterId)

compartment_list_child = self.get_cluster_cells(self.child_cell.clusterId)

self.clone_cluster_attributes(source_cell_cluster=compartment_list_parent,
                               target_cell_cluster=compartment_list_child,
                               no_clone_key_dict_list=['ATTR_NAME_1', 'ATTR_NAME_2'])
```

where `clone_cluster_attributes` function allows specification of this attributes are not to be copied (in our case `cell.dict` members `ATTR_NAME_1` and `ATTR_NAME_2` will not be copied).

Finally, if you prefer manually setting the parent and child cells, you would use the following code:

```
class MitosisSteppableClusters(MitosisSteppableClustersBase):

    def __init__(self, frequency=1):
        MitosisSteppableClustersBase.__init__(self, frequency)

    def step(self, mcs):

        for cell in self.cell_list:
            cluster_cell_list = self.get_cluster_cells(cell.clusterId)
            print("DISPLAYING CELL IDS OF CLUSTER ", cell.clusterId, "CELL. ID=", cell.
id)
            for cell_local in cluster_cell_list:
                print("CLUSTER CELL ID=", cell_local.id, " type=", cell_local.type)

        mitosis_cluster_id_list = []
        for compartment_list in self.clusterList:
            # print( "cluster has size=",compartment_list.size())
            cluster_id = 0
            cluster_volume = 0
            for cell in CompartmentList(compartment_list):
                cluster_volume += cell.volume
                cluster_id = cell.clusterId

            # condition under which cluster mitosis takes place
            if cluster_volume > 250:
                # instead of doing mitosis right away we store ids for clusters which
                # should be divide.
                # This avoids modifying cluster list while we iterate through it
                mitosis_cluster_id_list.append(cluster_id)

        for cluster_id in mitosis_cluster_id_list:
            self.divide_cluster_random_orientation(cluster_id)

            # # other valid options - to change mitosis mode leave one of the below_
            # lines uncommented
            # self.divide_cluster_orientation_vector_based(cluster_id, 1, 0, 0)
            # self.divide_cluster_along_major_axis(cluster_id)
            # self.divide_cluster_along_minor_axis(cluster_id)

    def updateAttributes(self):

        parent_cell = self.mitosisSteppable.parentCell
        child_cell = self.mitosisSteppable.childCell

        compartment_list_child = self.get_cluster_cells(child_cell.clusterId)
        compartment_list_parent = self.get_cluster_cells(parent_cell.clusterId)

        for i in range(len(compartment_list_child)):
```

(continues on next page)

(continued from previous page)

```

compartment_list_parent[i].targetVolume /= 2.0

compartment_list_child[i].targetVolume = compartment_list_parent[i].
    ↪targetVolume
compartment_list_child[i].lambdaVolume = compartment_list_parent[i].
    ↪lambdaVolume

```

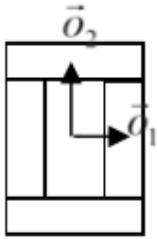
A Python helper for mitosis is available from Twedit++'s code snippets: CC3D\_Python->Mitosis.

## 18.1 How It Works

While dividing non-clustered cells is straightforward, doing the same for clustered cells is more challenging. To divide non-clustered cells using the directional mitosis algorithm, we construct a line or a plane passing through the center of mass of a cell and pixels of the cell on one side of the line/plane end up in the child cell and the rest stays in the parent cell. Be sure to use the [PixelTracker plugin](#) with mitosis to enable this pixel manipulation. The orientation of the line/plane can be either specified by the user, or we can use CC3D's built-in feature to calculate the orientation of the principal axes and divide either along the minor or major axis.

With compartmental cells, things get more complicated because: 1) Compartmental cells are composed of many sub-cells. 2) There can be different topologies of clusters. Some clusters may look “snake-like” and some might be compactly packed blobs of subcells. The algorithm which we implemented in CC3D works in the following way:

- 1) We first construct a set of pixels containing every pixel belonging to a cluster cell. You may think of it as a single “regular” cell.
- 2) We store volumes of compartments so that we know how big compartments should be after mitosis (they will be half of the original volume)
- 3) We calculate the center of mass of the entire cluster and calculate the vector offsets between the center of mass of a cluster and the center of mass of particular compartments as in the figure below:

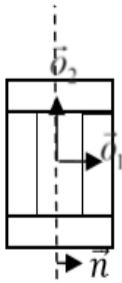


**Figure 7.** Vectors  $\vec{o}_1$  and  $\vec{o}_2$  show offsets between center of mass of a cluster and center of mass particular compartments.

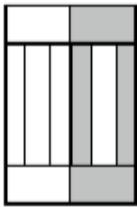
- 4) We pick a division line/plane for parent and child cells with offsets between cluster center of mass (after mitosis) and center of masses of clusters. We do it according to the formula:

$$\vec{p} = \vec{o} - \frac{1}{2}(\vec{o} \cdot \vec{n})\vec{n} \quad (18.1)$$

where  $\vec{p}$  denotes offset after mitosis from the center of mass of child (parent) clusters,  $\vec{o}$  is the orientation vector before mitosis (see picture above), and  $\vec{n}$  is a normalized vector perpendicular to division line/plane. If we try to divide the cluster along a dashed line as in the picture below:



**Figure 8.** Division of cell along dashed line. Notice the orientation of  $\vec{n}$ . The offsets after the mitosis for child and parent cell will be  $\vec{p}_1 = \frac{1}{2}\vec{o}_1$  and  $\vec{p}_2 = \vec{o}_2$  as expected because both parent and child cells will retain their heights, but these cells will also become twice as narrow.



**Figure 9.** Child and parent cells after mitosis. The parent cell is the one with gray outer compartments.

The formula given above is heuristic. It gives a fairly simple way of assigning pixels of child/parent clusters to cellular compartments. It is not perfect, but the idea is to get the approximate shape of the cell after the mitosis, and, as the simulation runs, the cell shape will readjust based on constraints such as adhesion of [focal point plasticity](#). Before continuing with mitosis, we check if the center of masses of the compartments belong to child/parent clusters. If the center of masses are outside their target pixels, we abandon mitosis and wait for readjustment of the cell shape, at which point the mitosis algorithm will pass this sanity check. For certain “exotic” shapes of cluster shapes, this mitosis algorithm may not work well or at all. In this case, we would have to write a specialized mitosis algorithm.

5) We divide clusters and knowing offsets from the child/parent cluster center of mass we assign pixels to particular compartments. The assignment is based on the distance of the particular pixel to the center of masses of clusters. The pixel is assigned to the compartment if its distance to the center of mass of the compartment is the smallest as compared to distances between centroids of other compartments. If the given compartment has reached its target volume and other compartments are underpopulated, we would assign pixels to other compartments based on the closest distance criterion. Although this method may result in some deviation from perfect 50-50 division of compartment volume, the cells will typically readjust their volume after a few MCS.



**Figure 10.** CC3D example of compartmental cell division. See also [Demos/CompuCellPythonTutorial/clusterMitosis](#).



---

CHAPTER  
**NINETEEN**

---

## **CHANGING CLUSTER ID OF A CELL.**

Quite often when working with mitosis you may want to reassign cell's cluster id i.e. to make a given cell belong to a different cluster than it currently does. You might think that statement like:

```
cell.clusterId = 550
```

is a good way of accomplishing it. This could have worked with CC3D versions prior to 3.4.2 . However, this is not the case anymore and in fact this is an easy recipe for hard to find bugs that will crash your simulation and display very enigmatic messages. So what is wrong here? First of all you need to realize that all the cells (strictly speaking pointers to CellG objects) in the CompuCell3D are stored in a sorted container called inventory. The ordering of the cells in the inventory is based on cluster id and cell id. Thus when a cell is created it is inserted to inventory and positioned according to cluster id and cell id. When you iterate inventory cells with lowest cluster id will be listed first. Within cells of the same cluster id cells with lowest cell id will be listed first. In any case if the cell is in the inventory and you do brute force cluster id reassignment the position of the cell in the inventory will not be changed. Why should it be? However when this cell is deleted CompuCell3D will first try to remove the cell from inventory based on cell id and cluster id and it will not find the cell because you have altered cluster id so it will ignore the request however it will delete underlying cell object so the net outcome is that you will end up with an entry in the inventory which has pointer to a cell that has been deleted. Next time you iterate through inventory and try to perform any operation on the cell the CC3D will crash because it will try to perform something with a cell that has been deleted. To avoid such situations always use the following construct to change clusterId of the cell:

```
reassignIdFlag = self.reassign_cluster_id(cell, 550)
```



## SBML SOLVER

When you study biology, sooner or later, you encounter pathway diagrams, gene expression networks, **Physiologically Based Pharmacokinetics (PBPK)** whole body diagrams, *etc...* Often, these can be mathematically represented in the form of Ordinary Differential Equations (ODEs). There are many ODE solvers available and you can write your own. However the solution we like most is called SBML Solver. Before going any further let us explain briefly what **SBML** itself is. **SBML** stands for **Systems Biology Markup Language**. It was proposed around year 2000 by a few scientists from Caltech (Mike Hucka, Herbert Sauro, Andrew Finney). According to Wikipedia, SBML is a representation format, based on XML, for communicating and storing computational models of biological processes. In practice SBML focuses on reaction kinetics models but can also be used to code these models that can be described in the form of ODEs such as e.g. PBPK, population models etc...

Being a multi-cell modeling platform CC3D allows users to associate multiple SBML model solvers with a single cell or create “free floating” SBML model solvers. The CC3D Python syntax that deals with the SBML models is referred to as “SBML Solver”. Internally SBML Solver relies on C++ libRoadRunner developed by Herbert Sauro’s team. libRoadRunner in turn is based on the C# code written by Frank Bergmann. CC3D uses libRoadRunner as the engine to solve systems of ODEs. All SBML Solver functionality is available via SteppableBasePy member functions. Twedit++ provides nice shortcuts that help users write valid code while tapping into SBML Solver functionality. See CC3DPython->SBML Solver menu for options available.

Let us look at the example steppable that uses SBML Solver:

```
from cc3d.core.PySteppables import *

class SBMLSolverSteppable(SteppableBasePy):
    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)

    def start(self):
        # adding options that setup SBML solver integrator
        # these are optional but useful when encountering integration instabilities

        options = {'relative': 1e-10, 'absolute': 1e-12}
        self.set_sbml_global_options(options)

        model_file = 'Simulation/test_1.xml'

        initial_conditions = {}
        initial_conditions['S1'] = 0.00020
        initial_conditions['S2'] = 0.000002

        self.add_sbml_to_cell_ids(model_file=model_file, model_name='dp',
```

(continues on next page)

(continued from previous page)

```

    cell_ids=list(range(1, 11)), step_size=0.5, initial_conditions=initial_
    ↵conditions)

        self.add_free_floating_sbml(model_file=model_file, model_name='Medium_dp', step_
    ↵size=0.5,
                                initial_conditions=initial_conditions)
        self.add_free_floating_sbml(model_file=model_file, model_name='Medium_dp1', step_
    ↵size=0.5,
                                initial_conditions=initial_conditions)

        self.add_free_floating_sbml(model_file=model_file, model_name='Medium_dp2')
        self.add_free_floating_sbml(model_file=model_file, model_name='Medium_dp3')
        self.add_free_floating_sbml(model_file=model_file, model_name='Medium_dp4')

cell_20 = self.fetch_cell_by_id(20)

self.add_sbml_to_cell(model_file=model_file, model_name='dp', cell=cell_20,_
    ↵integrator='gillespie')

def step(self, mcs):
    self.timestep_sbml()

cell_20 = self.fetch_cell_by_id(20)
print('cell_20, dp=', cell_20.sbml.dp.values())

print('Free Floating Medium_dp2', self.sbml.Medium_dp2.values())
if mcs == 3:
    Medium_dp2 = self.sbml.Medium_dp2
    Medium_dp2['S1'] = 10
    Medium_dp2['S2'] = 0.5

if mcs == 5:
    self.delete_sbml_from_cell_ids(model_name='dp', cell_ids=list(range(1, 11)))

if mcs == 7:
    cell_25 = self.fetch_cell_by_id(25)
    self.copy_sbml_simulators(from_cell=cell_20, to_cell=cell_25)

```

In the start function we specify the path to the SBML model (here we use partial path `Simulation/test_1.xml` because `test_1.xml` is in our CC3D Simulation project directory) and also create a python dictionary that has initial conditions for the SBML model. This particular model has two floating species : S1 and S2 and our dictionary – `initialConditions` - stores the initial concentration of these species to 0.0002 and 0.000002 respectively:

```

model_file = 'Simulation/test_1.xml'
initial_conditions = {}
initial_conditions['S1'] = 0.00020
initial_conditions['S2'] = 0.000002

```

### Note

We can initialize each SBML Solver using different initial conditions. When we forget to specify initial conditions

the SBML code usually has initial conditions defined and they will be used as starting values.

Before we discuss `add_sbml_to_cell_ids` function let us focus on statements that open the start function:

```
options = {'relative': 1e-10, 'absolute': 1e-12}
self.set_sbml_global_options(options)
```

We set here SBML integrator options. These statements are optional, however when your SBML model crashes with e.g. CVODE error, it often means that your numerical tolerances (relative and absolute) or number of integration steps in each integration interval (steps) should be changed. Additionally you may want to enable stiff ODE solver by setting `stiff` to True:

```
options = {'relative': 1e-10, 'absolute': 1e-12, 'stiff': False}
self.set_sbml_global_options(options)
```

After defining the options dictionary we inform CC3D to use these settings. We do it by using as shown above. A thing to remember that new options will apply to all SBML model that were added after calling `set_sbml_global_options`. This means that usually you want to ensure that SBML integration options setting should be the first thing you do in your Python steppable file. If you want to retrieve options simply type:

```
options = self.get_sbml_global_options()
```

Notice that options can be None indicating that options have not been set (this is fine) and the default SBML integrator options will be applied.

Let us see how we associate SBML model with several cells using `add_sbml_to_cell_ids`:

```
self.add_sbml_to_cell_ids(model_file=model_file, model_name='dp',
cell_ids=list(range(1, 11)), step_size=0.5, initial_conditions=initial_conditions)
```

This function looks relatively simple but it does quite a lot if you look under the hood. The first argument is the path to the SBML models file. The second one is the model alias - it is a name you choose for the model. It is an arbitrary model identifier that you use to access the model. The third argument is a Python list that contains cell ids to which CC3D will attach an instance of the SBML Solver.

### Note

Each cell will get a separate SBML solver object. SBML Solver objects are associated with cells, while free floating SBML Solvers are independent.

The fourth argument specifies the size of the integration step – here we use a value of 0.5 time unit. The fifth argument passes the optional initial conditions dictionary.

Each SBML Solver function that associates models with a cell or adds a free floating model calls libRoadRunner functions that parse SBML and translate it to very fast LLVM code. Everything happens automatically and produces optimized solvers which are much faster than solvers that rely on some kind of interpreters.

The next five function calls to `self.add_free_floating_sbml` create instances of SBML Solvers which are not associated with cells but, as you can see, have distinct names. Unique naming of free floating models is required because when we want to refer to such a solver to extract model values we will do so using the model name. The reason all models attached to cells have same name is that when we refer to such a model we pass the cell object and a name, which uniquely identify the model. Notice that the last three calls to `self.add_free_floating_sbml` do specify neither step size (we use the default step size of 1.0 time unit) nor initial conditions (we use whatever defaults are in the SBML code).

Finally, the last two lines of the start function demonstrates how to add an SBML Solver object to a single cell and select a non-default SBML Solver integrator:

```
cell_20 = self.fetch_cell_by_id(20)
self.add_sbml_to_cell(model_file=model_file, model_name='dp', cell=cell_20, integrator=
    ↪'gillespie')
```

Instead of a passing list of cell ids we pass a cell object (cell\_20). All integrators supported by libRoadRunner are available using the keyword `integrator`, which includes CVODE (default), Gillespie (`integrator='gillespie'`), Euler (`integrator='euler'`), Runge-Kutta (`integrator='rk4'`) and Gillespie Direct Method (`integrator='rk45'`). For more information on the details of available integrators, visit the [libRoad-Runner online documentation](#). The `integrator` keyword argument is optional to, and available in, all functions that create a SBML Solver instance.

We can also associate SBML model with certain cell types using the following syntax:

```
self.add_sbml_to_cell_types(model_file=model_file, model_name='dp', cell_types=[self.
    ↪NONCONDENSING],
                                step_size=step_size, initial_conditions=initial_
    ↪conditions)
```

This time instead of passing a list of cell ids we pass list of cell types.

Let us move on to the step function. The first call we see there is `self.timestep_sbml`. This function carries out integration of all SBML Solver instances defined in the simulation. The integration step can be different for different SBML Solver instances (as shown in our example).

To check the values of model species after the integration step we can call e.g.

```
print('Free Floating Medium_dp2', self.sbml.Medium_dp2.values())
```

These functions check and print model variables for the free floating model called `Medium_dp2`.

The next set of function calls:

```
if mcs == 3:
    Medium_dp2 = self.sbml.Medium_dp2
    Medium_dp2['S1'] = 10
    Medium_dp2['S2'] = 0.5
```

set a new state for the free floating model called `Medium_dp2`. If we wanted to print the state of the model `dp` belonging to the cell object called `cell_20` we would use the following syntax:

```
print('cell_20, dp=', cell_20.sbml.dp.values())
```

To assign new values to `dp` model variables for `cell_20` we use the following syntax:

```
cell_20.sbml.dp['S1'] = 10
cell_20.sbml.dp['S2'] = 0.5
```

### Note

We access a free-floating SBML Solver via `self.sbml.MODEL_ALIAS` syntax whereas SBML Solvers associated with a particular cell are accessed using a reference to the cell objects e.g. `cell_20.sbml.MODEL_ALIAS`

Another useful operation within SBML Solver capabilities is deletion of models. This is handy when at a certain point in your simulation you no longer need to solve ODEs described in the SBML model. This is the syntax that deletes a named SBML Solver model from specific cell ids:

```
self.delete_sbml_from_cell_ids(model_name='dp', cell_ids=list(range(1, 11)))
```

As you probably suspect, we can delete a named SBML Solver model from cell types:

```
self.delete_sbml_from_cell_types(model_name='dp', cell_types=range[self.A, self.B])
```

from a single cell:

```
self.delete_sbml_from_cell(model_name='dp', cell=cell_20)
```

or delete a free floating SBML Solver object:

```
self.delete_free_floating_sbml(model_name='Medium_dp2')
```

### Note

When cells get deleted all SBML Solver models attached to them are deleted automatically. You do not need to call `delete_sbml` functions in such a case.

Sometimes you may encounter a need to clone all SBML models from one cell to another (e.g. in the mitosis `updateAttributes` function where you clone SBML Solver objects from a parent cell to a child cell). SBML Solver lets you do that very easily:

```
cell_10 = self.fetch_cell_by_id(10)
cell_25 = self.fetch_cell_by_id(25)
self.copy_sbml_simulators(from_cell=cell_10, to_cell=cell_25)
```

What happens here is that source cell (`from_cell`) provides SBML Solver object templates and based on these templates new SBML Solver objects are gets created and CC3D assigns them to target cell (`to_cell`). All the state variables in the target SBML Solver objects are the same as values in the source objects.

If you want to copy only select models you would use the following syntax:

```
cell_10 = self.fetch_cell_by_id(10)
cell_25 = self.fetch_cell_by_id(25)
self.copy_sbml_simulators(from_cell=cell_10, to_cell=cell_25, sbml_names=['dp'])
```

As you can see there is a third argument - a Python list that specifies which models to copy by name. Here we are copying only dp models. All other models associated with the parent cells will not be copied.

This example demonstrates most important capabilities of SBML Solver. The next example shows a slightly more complex simulation where we reset initial condition of the SBML model before each integration step ([Demos/SBMLSolverExamples/DeltaNotch](#)).

A full description of the Delta-Notch simulation is in the introduction to CompuCell3D Manual. The Delta-Notch example demonstrates multi-cellular implementation of Delta-Notch mutual inhibitory coupling. In this juxtracrine 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 sub-cellular networks with inter-cellular networks and cell behaviors, it already shows how different such behaviors can be from those of their non-spatial simplifications.

We begin with the ODE Delta-Notch patterning model of Collier 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:

$$\frac{dD}{dt} = \left( \nu \times \frac{1}{1 + bN^h} - D \right) \quad (20.1)$$

$$\frac{dN}{dt} = \frac{\bar{D}^k}{a + \bar{D}^k} - N \quad (20.2)$$

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

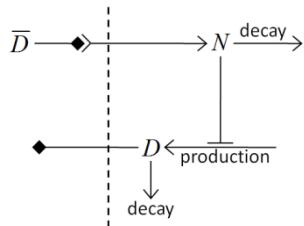


Figure 18 Diagram of Delta-Notch feedback regulation between and within cells.

For the sake of simplicity let us assume that we downloaded the SBML model implementing the Delta-Notch ODEs. How do we use such SBML model in CC3D? Here is the code:

```

from random import uniform
from cc3d.core.PySteppables import *

class DeltaNotchClass(SteppableBasePy):
    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)

    def start(self):

        # adding options that setup SBML solver integrator
        # these are optional but useful when encountering integration instabilities
        options = {'relative': 1e-10, 'absolute': 1e-12}
        self.set_sbml_global_options(options)

        model_file = 'Simulation/DN_Collier.sbml'
        self.add_sbml_to_cell_types(model_file=model_file, model_name='DN', cell_
        types=[self.TYPEA], step_size=0.2)

        for cell in self.cell_list:
            dn_model = cell.sbml.DN

            dn_model['D'] = uniform(0.9, 1.0)
            dn_model['N'] = uniform(0.9, 1.0)

            cell.dict['D'] = dn_model['D']
            cell.dict['N'] = dn_model['N']

    def step(self, mcs):
  
```

(continues on next page)

(continued from previous page)

```

for cell in self.cell_list:
    delta_tot = 0.0
    nn = 0
    for neighbor, commonSurfaceArea in self.get_cell_neighbor_data_list(cell):
        if neighbor:
            nn += 1

            delta_tot += neighbor.sbml.DN['D']
    if nn > 0:
        D_avg = delta_tot / nn

    cell.sbml.DN['Davg'] = D_avg
    cell.dict['D'] = D_avg
    cell.dict['N'] = cell.sbml.DN['N']

self.timestep_sbml()

```

In the `start` function we add SBML model (`Simulation/DN_Collier.sbml`) to all cells of type A (it is the only cell type in this simulation besides `Medium`). Later in the for loop we initialize D and N species from the SBML model using random values so that each cell has a different SBML model starting state. We also store the initial SBML model in a cell dictionary for visualization purposes – see the full code in `Demos/SBMLSolverExamples/DeltaNotch`. In the `step` function for each cell we visit its neighbors and sum the value of Delta in the neighboring cells. We divide this value by the number of neighbors (this gives the average Delta concentration in the neighboring cells - `D_avg`). We pass `D_avg` to the SBML Solver for each cell and then carry out integration for the next time step. Before calling `self.timestep_sbml` we store the values of Delta and Notch concentrations in the cell dictionary, but we do it for visualization purposes only. As you can see from this example SBML Solver programming interface is convenient to use, not to mention SBML Solver itself is a very powerful tool which allows coupling cell-level and sub-cellular scales.



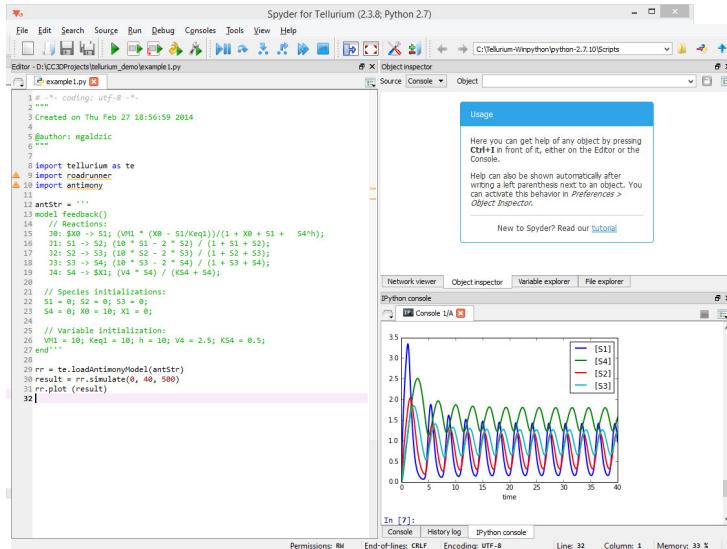
---

## CHAPTER TWENTYONE

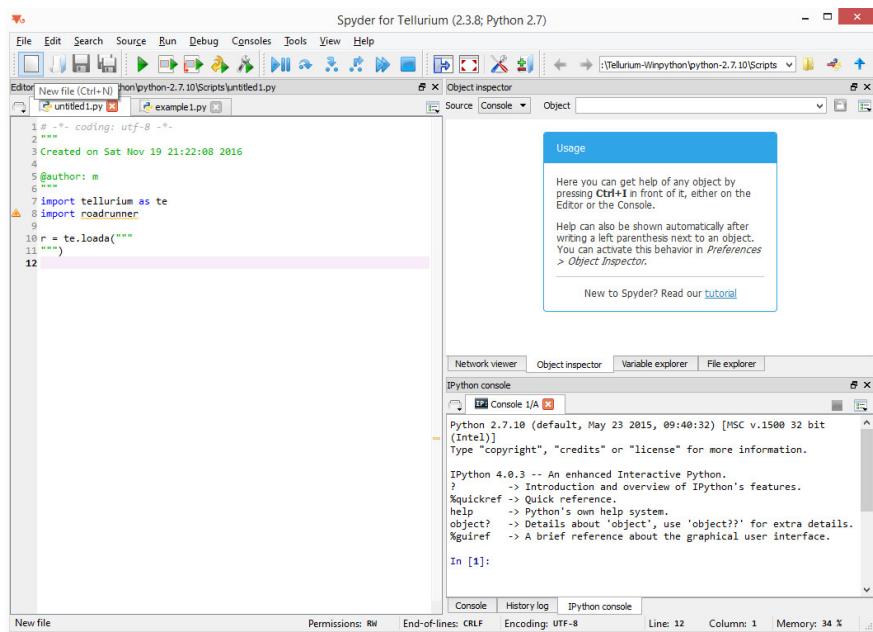
---

### BUILDING SBML MODELS USING TELLURIUM

In the previous section we showed you how to attach reaction-kinetics models to each cell and how to obtain solve them on-the-fly inside CC3D simulation. Once we have up-to-date solution to these models we could parameterize cell behavior in terms of the underlying chemical species. But, how do we construct those reaction-kinetics models and how do we save them in the SBML format. There are many packages that will do this for you – Cell Designer, Copasi, Virtual Cell, Systems Biology Workbench (that includes Jarnac, and JDesigner apps), Pathway Designer and many more. All of those excellent apps will do the job. Some are easier to use than others but in most cases some training will be required. For this reason we decided to focus on Tellurium which appears to be the easiest to use. Tellurium, similarly to CC3D uses Python to describe reaction-kinetics models which is an added bonus for CC3D users who are already well versed in Python. First thing you need to do is to install tellurium on your machine – go to their download site <https://sourceforge.net/projects/pytellurium/> and get the installer package. The installation is straight-forward. Once installed, open up Tellurium and it will display ready-to-run example. Hit play button (green triangle at the top toolbar) and the results of the simulation will be displayed in the bottom right subwindow:

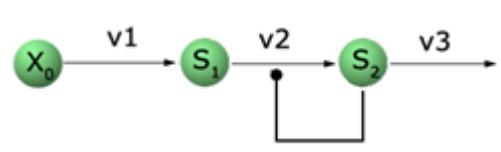


Hopefully this was easy. Now let's build our own model. We start creating our new reaction-kinetics model by pressing “New File” button - the first button on the left in the main toolbar:



Pressing **New File** button opens up a Tellurium template for reaction-kinetics model. All we need to do is to define the model and add few statements that will export it in the SBML format.

Let's start by defining the model. The model definition uses language called Antimony. To learn more about Antimony please visit its tutorial page <http://tellurium.analogmachine.org/documentation/antimony-tutorial/>. Antimony is quite intuitive and allows to specify system of chemical reaction in a way that is very natural to anybody who has basic understanding of chemical reaction notation. In our case we will define a simple relaxation oscillator that consists of two floating species ( $S_1$  and  $S_2$ ) and two boundary species ( $X_0$ ,  $X_3$ ). Boundary species serve as sources/sinks of the reaction and their concentrations remain constant. Concentration of floating species change with time. The presented example has been developed by Herbert Sauro (University of Washington) - one of the three “founding fathers” of SBML, author of many books on reaction kinetics and lead contributor to the Tellurium and Systems Biology Workbench packages. Let's look at the reaction schematics we are about to implement:



Qualitatively,  $X_0$  is being kept at a constant concentration of 1.0 (we assume arbitrary units here) and  $S_1$  accumulates at the constant rate  $v_1$ .  $S_1$  also gets “transformed” to  $S_2$  at the rate  $v_2$  which depends on concentration of  $S_1$  and  $S_2$  in such a way that when  $S_1$  and  $S_2$  are at relatively high values the reaction “speeds up” depleting quickly concentration of  $S_1$ . Once  $S_1$  concentration is low reaction  $S_1 \rightarrow S_2$  slows down allowing  $S_1$  to build up again. As you may suspect, when properly tuned this type of reaction produces oscillations. Let us formalize the description and present rate laws and parameters that result in oscillatory behavior. Using Antimony we would write the following set of reactions corresponding to the reaction schematics above:

```

$X0 -> S1 ; k1*X0;
S1 -> S2 ; k2*S1*S2^h/(10 + S2^h) + k3*S1;
S2 -> $X3 ; k4*S2;

```

$\$X0$  and  $\$X4$  are “boundary species”.  $\$X0$  serves as a source and  $X3$  is a sink .

In the line

```
$X0 -> S1 ; k1*X0;
```

part  $\text{S}0 \rightarrow \text{S}1$  represents reaction and  $k1 * \text{S}0$  is a rate law that describes the speed at which species  $\text{S}0$  transform to  $\text{S}1$ .

The interesting part that leads to oscillatory behavior is the rate law in the second reaction. It involves Hill-type kinetics. Finally the third reaction defines first order kinetics that transfers  $\text{S}2$  into  $\text{S}3$ . We are not going to discuss the theory of reaction kinetics here but rather focus on the mechanics of how to define and solve RK models using Tellurium. If you are interested in learning more about modeling of biochemical pathways please see books by Herbert Sauro.

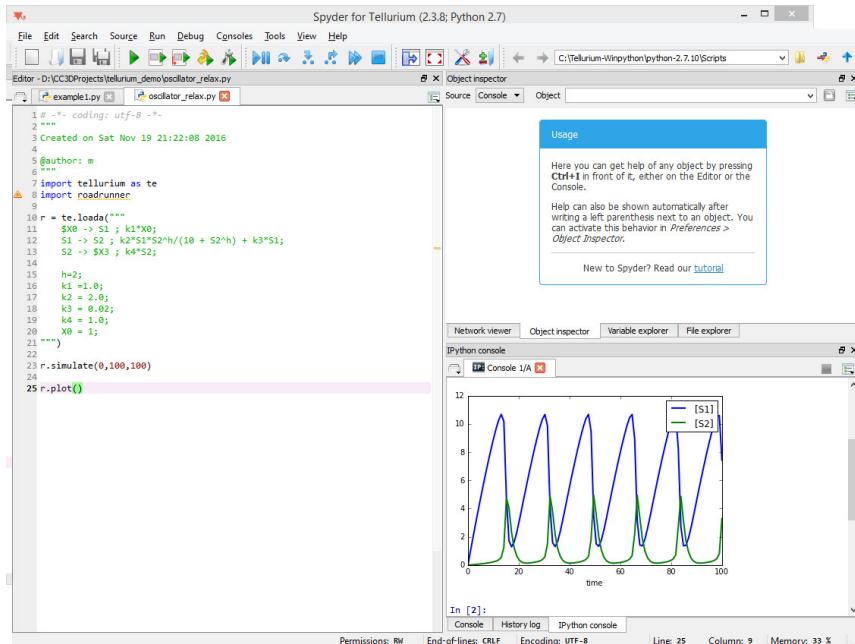
Assuming we have our Antimony definition of RK model we put it into Tellurium's Python code:

```
import tellurium as te
import roadrunner

r = te.loada("""
$X0 -> S1 ; k1*X0;
S1 -> S2 ; k2*S1*S2^h/(10 + S2^h) + k3*S1;
S2 -> $X3 ; k4*S2;

h=2;
k1 =1.0;
k2 = 2.0;
k3 = 0.02;
k4 = 1.0;
X0 = 1;
""")
```

As you can see, all we need to do is to copy the description of reactions and define constants that are used in the rate laws. When we run this model inside tellurium we will get the following result:



Notice that we need to add two lines of code - one to actually solve the model

```
r.simulate(0,100,100)
```

And another one to plot the results

```
r.plot()
```

The general philosophy of Tellurium is that you define reaction-kinetics model that is represented as object r inside Python code. To solve the model we invoke r's function simulate and to plot the results we call r's function plot.

Finally, to export our model defined in Antimony/Tellurium as SBML we write simple export code at the end of our code:

```
sbml_file = open('d:\CC3DProjects\oscillator_relax.sbml','w')
print(sbml_file,r.getSBML())
sbml_file.close()
```

The function that “does the trick” is getSBML function belonging to object r. At this point you can take the SBML model and use SBML solver described in the section above to link reaction-kinetics to cellular behaviors

You may also find the following youtube video by Herbert Sauro useful.

<https://www.youtube.com/watch?v=pEWxfIKE18c>

---

CHAPTER  
TWENTYTWO

---

## BUILDING SBML MODELS EFFICIENTLY WITH ANTIMONY AND CELLML

Now that we know how to write an Antimony model in Tellurium and use it with SBML solver, let's explore some built-in functionality to efficiently write and use reaction-kinetics models in Antimony and CellML<sup>1</sup> model definition languages. Antimony and CellML models can be attached to simulation objects exactly like SBML models, using variations in function naming conventions corresponding to the choice in model specification (*e.g.*, for SBML model functions `add_sbml_to_cell` and `add_sbml_to_link` there are corresponding functions `add_antimony_to_cell`, `add_cellml_to_cell`, `add_antimony_to_link` and `add_cellml_to_link`). For Antimony and CellML model specification, models are translated to SBML using the libAntimony library developed by Lucian Smith and Herbert Sauro. Furthermore, models can be specified in external files and imported by CC3D steppables *or* within CC3D steppables as Python multi-line strings.

Say we want to implement our previous Antimony model definition of an RK model:

```
$X0 -> S1 ; k1*X0;
S1 -> S2 ; k2*S1*S2^h/(10 + S2^h) + k3*S1;
S2 -> $X3 ; k4*S2;

h=2;
k1 = 1.0;
k2 = 2.0;
k3 = 0.02;
k4 = 1.0;
X0 = 1;
```

Now say we want to attach the Antimony model definition to a cell *without* leaving Twedit++, but rather all within a steppable `AntimonySolverSteppable`. This can be accomplished using a multi-line string using proper Antimony syntax:

```
class AntimonySolverSteppable(SteppableBasePy):

    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)

    def start(self):

        antimony_model_string = """model rkModel()
$X0 -> S1 ; k1*X0;
S1 -> S2 ; k2*S1*S2^h/(10 + S2^h) + k3*S1;"""

(continues on next page)
```

<sup>1</sup> Cuellar, A.A., Lloyd, C.M., Nielsen, P.F., Bullivant, D.P., Nickerson, D.P. and Hunter, P.J. An overview of CellML 1.1, a biological model description language. SIMULATION: Transactions of The Society for Modeling and Simulation International. 2003 Dec;79(12):740-747.

(continued from previous page)

```

S2 -> $X3 ; k4*S2;

h = 2;
k1 = 1.0;
k2 = 2.0;
k3 = 0.02;
k4 = 1.0;
X0 = 1;
end"""

options = {'relative': 1e-10, 'absolute': 1e-12}
self.set_sbml_global_options(options)
step_size = 1e-2

for cell in self.cell_list:
    self.add_antimony_to_cell(model_string=antimony_model_string,
                               model_name='dp',
                               cell=cell,
                               step_size=step_size)

def step(self):
    self.timestep_sbml()

```

Compared to attaching a SBML model to cells, we see that not much has changed procedurally: we define a model (in this case, using Antimony instead of SBML), set the requisite settings for SBML Solver, and attach our model to CC3D objects using built-in functions (in this case, using `add_antimony_to_cell` instead of `add_sbml_to_cell`). The major differences here are that we have defined our model using Antimony model specification within the steppable using built-in functions, and also that we defined our model initial conditions with Antimony. In this case, SBML Solver retrieved our initial conditions from the translation of our Antimony model, rather than from explicit arguments. Were we to also pass initial conditions to SBML Solver as an argument through `add_antimony_to_cell`, the explicit calls would take precedence over those declared in the Antimony model string.

Now say that we've obtained a shared CellML model that we'd like to use in CC3D. We can load the model specification from file:

```

class CellMLSolverSteppable(SteppableBasePy):

    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)

    def start(self):

        cellml_model_file = 'Simulation/shared_cellml_model.txt'

        options = {'relative': 1e-10, 'absolute': 1e-12}
        self.set_sbml_global_options(options)
        step_size = 1e-2

        self.add_cellml_to_cell_ids(model_file=cellml_model_file,
                                    model_name='dp',
                                    cell_ids=list(range(1,11)),
                                    step_size=step_size)

```

(continues on next page)

(continued from previous page)

```
def step(self):
    self.timestep_sbml()
```

Here the shared CellML model is stored in the same Simulation directory as the CC3D project steppables, in the file `shared_cellml_model.txt`. We pass the path of the model file to `add_cellml_to_cell_ids` to attach the shared CellML model to all cells with ids in the list `list(range(1,11))`, just like passing a SBML model file to `add_sbml_to_cell_ids`. Note that if the CellML model does not specify initial conditions, then we must explicitly pass them to `add_cellml_to_cell_ids`.

We see that there are two ways of passing an Antimony or CellML model to SBML Solver, either as a Python multi-line string, or as a path to a file containing the model. One of these must be accomplished, where a multi-line string is passed to the keyword argument `model_string`, or a path to a file is passed to the keyword argument `model_file`. These are true for SBML, Antimony, and CellML built-in functions.

For attaching Antimony and CellML models to links, steppables have the functions `add_antimony_to_link` and `add_cellml_to_link`. Their usage is the exact same as their cell-based counterparts, with the exception that a link is passed to the keyword argument `link`, rather than passing a cell to the keyword argument `cell`,

```
self.add_antimony_to_link(link=link, ...)
self.add_cellml_to_link(link=link, ...)
```



---

CHAPTER  
**TWENTYTHREE**

---

## BUILDING AND USING MABOSS MODELS

As of v4.2.5., CC3D supports simulating boolean models using MaBoSS (Markovian Boolean Stochastic Simulator)<sup>1</sup> and attaching them to cells. MaBoSS enables representing intracellular mechanisms like gene regulatory networks and chemical kinetics as Boolean networks, on which MaBoSS simulates continuous or discrete time Markov processes. CC3D provides an interactive version of MaBoSS simulations with select MaBoSS simulation capabilities, so that individual networks can be selectively integrated in time, modified, and reconfigured according to an overall CC3D simulation. With combined capabilities, MaBoSS models in CC3D can be interconnected and coupled with cellular properties, local conditions, and ODE models. For a detailed description of MaBoSS capabilities, examples of MaBoSS simulations and downloading the full, standalone MaBoSS simulation package, visit the [MaBoSS homepage](#).

 **Note**

To cite MaBoSS usage in CC3D, use the following,

Stoll, Gautier, et al. “MaBoSS 2.0: an environment for stochastic Boolean modeling.” *Bioinformatics* 33.14 (2017): 2226-2228.

### 23.1 Building a MaBoSS Model in CC3D

MaBoSS simulation specification is divided into two components: a network description, and a simulation configuration. A MaBoSS network description describes a network on the basis of nodes, including the name, logic, and transition rates of each node of the network. A MaBoSS simulation configuration describes the network parameters and initial conditions. Typical MaBoSS usage includes creating a network description file with the extension .bnd and a configuration file with the extension .cfg, which are both passed to MaBoSS using a terminal. In CC3D, MaBoSS simulations can be created using any combination of network and configuration files *or* multiline strings.

A very simple network description of two nodes A and B in a multiline string could look like the following,

```
#####
node A {
    logic = $B_ext && !B;
    rate_up = @logic ? $node_rate : 0.0;
    rate_down = @logic ? 0.0 : $node_rate;
}
node B {
    logic = A;
    rate_up = @logic ? $node_rate : 0.0;
    rate_down = $node_rate / 10;
}
#####
```

<sup>1</sup> Stoll, Gautier, et al. “MaBoSS 2.0: an environment for stochastic Boolean modeling.” *Bioinformatics* 33.14 (2017): 2226-2228.

The network description describes the logic of each node (*i.e.*, `logic`), and an expression for the rate of each node going from 0 to 1 (*i.e.*, `rate_up`) and going from 1 to 0 (*i.e.*, `rate_down`). Rate expressions can be constant (*e.g.*, `rate_down` for node B), or conditional (*e.g.*, `rate_down` for node B). For conditional expressions, the syntax `X ? Y : Z` reads Y if X is true, otherwise Z. The network also uses two external variables `$B_ext` and `$node_rate`, the values of which are specified in the configuration file and, along with everything else described in this network except the network itself, can be accessed and modified in CC3D for individual simulation instances of this network.

A corresponding configuration for this network in a multiline string could look like the following,

```
"""
A.istate = FALSE;
$B_ext = FALSE;
$node_rate = 10.0;
"""
```

The configuration declares the initial state of node A, and declares values for the external variables `$B_ext` and `$node_rate`. Note that, since the configuration does not declare the initial state of node B, each network instance will initialize node B with a randomly selected state.

### Note

MaBoSS provides additional specification capabilities for the configuration file (*e.g.*, declaring a random generator seed). The network and configuration specification relevant to MaBoSS simulations in CC3D are described in the previous example. The remaining inputs relevant to specifying a MaBoSS simulation in CC3D are described in subsequent text, and are declared through an interface provided by CC3D.

A MaBoSS simulation can be created and attached to a cell in CC3D using MaBoSS network description and configuration files from within a steppable,

```
self.add_maboss_to_cell(cell=cell,
                        model_name=model_name,
                        bnd_file=bnd_file,
                        cfg_file=cfg_file,
                        time_step=time_step,
                        time_tick=time_tick)
```

Here `cell` is the CC3D cell to which the MaBoSS simulation should be attached, `model_name` is a string that declares an alias name by which the MaBoSS simulation can be accessed after the call, `bnd_file` and `cfg_file` are the absolute path to the location of a network description and configuration file, respectively, `time_step` is a float equal to the MaBoSS simulation time that corresponds to one simulation step in CC3D, and `time_tick` is the MaBoSS simulation time that corresponds to considering the transition of every node in the network one time (*i.e.*, the MaBoSS *time window*).

MaBoSS model and simulation specification declared in Python multiline strings can also be used to create a MaBoSS simulation and attach it to a cell,

```
self.add_maboss_to_cell(cell=cell,
                        model_name=model_name,
                        bnd_str=bnd_str,
                        cfg_str=cfg_str,
                        time_step=time_step,
                        time_tick=time_tick)
```

Here `bnd_str` and `cfg_str` are network description and configuration multiline strings, respectively (*i.e.*, the contents of a file, but as a multiline string). Any combination of file path and string inputs can be used for passing the network

description and configuration, so long as each is passed.

MaBoSS simulations can also be created from within a steppable without being attached to a cell,

```
mm = self.maboss_model(model_name=model_name,
                       bnd_file=bnd_file,
                       cfg_file=cfg_file,
                       time_step=time_step,
                       time_tick=time_tick)
```

The same can be accomplished from outside a steppable using a function of the same name and arguments from the `cc3d.core.MaBoSSCC3D` module,

```
from cc3d.core import MaBoSSCC3D
mm = MaBoSSCC3D.maboss_model(model_name=model_name,
                               bnd_file=bnd_file,
                               cfg_file=cfg_file,
                               time_step=time_step,
                               time_tick=time_tick)
```

Both implementations of `maboss_model` take the same optional arguments as the steppable method `add_maboss_to_cell` (*e.g.*, the keyword arguments `bnd_str` and `seed`).

MaBoSS simulations are automatically destroyed when a cell is destroyed unless the MaBoSS simulation is also stored elsewhere (*e.g.*, as an attribute on a steppable). MaBoSS simulations can also be manually removed from a cell,

```
self.delete_maboss_from_cell(cell=cell, model_name=model_name)
```

All provided functions to create MaBoSS simulations in CC3D can also take optional keyword arguments,

- `discrete_time` takes a Boolean value (default is `False`). When passing `discrete_time=True`, a MaBoSS simulation will perform time integration with fixed time intervals equal to `time_tick` until an amount of time equal to `time_step` has elapsed for one integration step. By default, a MaBoSS simulation will integrate using the Gillespie algorithm.
- `seed` takes an integer value (default is 0) as the seed for the random generator of the MaBoSS simulation.
- `istate` a dictionary of string names and corresponding initial state values.

### Note

A `time_step` value less than a `time_tick` value is only valid when using the default Gillespie algorithm. Otherwise, the `time_step` value must be greater than the `time_tick` value for anything to occur.

## 23.2 Interacting with a MaBoSS Model

All MaBoSS simulations attached to each cell can be accessed using the cell property `maboss` and referring to the alias of the model as passed to the keyword argument `model_name` when the simulation was created,

```
self.add_maboss_to_cell(cell=cell, model_name='MyMaBoSSModel', ...)
...
mm = cell.maboss.MyMaBoSSModel
```

Every MaBoSS simulation attached to every cell in simulation can be integrated one step in time from a steppable,

```
self.timestep_maboss()
```

Likewise, a MaBoSS simulation instance can be individually integrated one step in time,

```
cell.maboss.MyMaBoSSModel.step()
```

The values passed to the keyword arguments `time_step`, `time_tick`, `discrete_time` and `seed` when creating a MaBoSS simulation can all be overwritten at any time by interacting with the MaBoSS simulation instance and its attached CC3DRunConfig object,

```
mm = cell.maboss.MyMaBoSSModel
mm.step_size *= 2.0 # Double value passed to time_step
mm.run_config.time_tick /= 2.0 # Half value passed to time_tick
mm.run_config.discrete_time = False # Enable Gillespie algorithm
mm.run_config.seed = mm.run_config.seed + 1 # Increment value passed to seed
```

Each node of a MaBoSS simulation network can be accessed as if interacting with a Python dictionary. For example, to get node A in a network attached to a cell with alias `MyMaBoSSModel`,

```
node_a = cell.maboss.MyMaBoSSModel['A']
```

Data about a node can be accessed and, where appropriate, set using properties and functions of a node. For example, the state of a node can be accessed and set using the property `state` such that networks of individual cells and/or multiple networks in the same cell can be coupled,

```
node_1a = cell1.maboss.MyMaBoSSModel['A']
node_1ao = cell1.maboss.MyOtherMaBoSSModel['A']
node_2b = cell2.maboss.MyMaBoSSModel['B']
node_1a.state = node_2b.state or node_1ao.state
```

Likewise, the value of external variables can be accessed and set using the MaBoSS Network attached to a MaBoSS simulation. The network of a MaBoSS simulation can be accessed with the property `network`, and the external variables of a network can be accessed and set using the property `symbol_table` of the network. For example, an external variable declared in a MaBoSS simulation specification as `$cellVolume` can be coupled to the current volume of a cell (e.g., to use the cell volume as a MaBoSS model parameter),

```
cell.maboss.MyMaBoSSModel.network.symbol_table['cellVolume'] = cell.volume
```

### 23.3 CC3D MaBoSS API

For brevity and generality, the following APIs are presented as relevant to modeling and simulation using CC3D and MaBoSS.

The module `cc3d.cpp.MaBoSSCC3DPy` contains the following relevant API,

```
# Network node
class CC3DMaBoSSNode:
    # Description; read-only
    description: str
    # Input node flag; read-only
    is_input: bool
    # Internal node flag
    is_internal: bool
```

(continues on next page)

(continued from previous page)

```

# Reference node flag
is_reference: bool
# Initial state
istate: bool
# Current rate down; read-only
rate_down: float
# Current rate up; read-only
rate_up: float
# Reference state
ref_state: bool
# Current state
state: bool
# MaBoSS Node "mutate" method
def mutate(self, value: float) -> None

# External variable value container
class SymbolTable:
    # List of symbol names; read-only
    names: List[str]
    # Gets a symbol value by name
    def __getitem__(self, item: str) -> Union[bool, int, float]
    # Sets a symbol value by name
    def __setitem__(self, item: str, value: Union[bool, int, float]) -> None

# Simulation network
class Network:
    # List of nodes; read-only
    nodes: List[CC3DMaBoSSNode]
    # Symbol table; read-only
    symbol_table: SymbolTable

# Simulation configuration
class CC3DRunConfig:
    # Current random generator seed
    seed: int
    # Simulation time tick
    time_tick: float
    # Discrete time flag
    discrete_time: bool

# The main MaBoSS simulation class in CC3D
class CC3DMaBoSSEngine:
    # Network of the simulation; read-only
    network: Network
    # Configuration of the simulation; read-only
    run_config: CC3DRunConfig
    # Current simulation time; read-only
    time: float
    # Current default step size
    step_size: float
    # Integrates the simulation one step; passing no argument integrates over current
    # value of step_size

```

(continues on next page)

(continued from previous page)

```
def step(self, _stepSize=-1.0) -> None
# Gets the network state
def getNetworkState(self) -> NetworkState
# Loads an existing network state
def loadNetworkState(self, _networkState: NetworkState) -> None
# Get a node by node name
def __getitem__(self, key: str) -> CC3DMaBoSSNode
# Set a node state by node name
def __setitem__(self, key: str, value: bool) -> None
```

The module `cc3d.core.MaBoSSCC3D` contains the following relevant API,

```
# Instantiates and returns a MaBoSS simulation instance from files and/or strings.
def maboss_model(bnd_file: str = None,
                  bnd_str: str = None,
                  cfg_file: str = None,
                  cfg_str: str = None,
                  time_step: float = 1.0,
                  time_tick: float = 1.0,
                  discrete_time: bool = False,
                  seed: int = None,
                  istate: Dict[str, bool] = None) -> MaBoSSCC3DPy.CC3DMaBoSSEngine
```

The `SteppableBasePy` class contains the following relevant API,

```
class SteppableBasePy:
    # Adds a MaBoSS simulation instance to a cell
    @staticmethod
    def add_maboss_to_cell(cell: CellG,
                           model_name: str,
                           bnd_file: str = None,
                           bnd_str: str = None,
                           cfg_file: str = None,
                           cfg_str: str = None,
                           time_step: float = 1.0,
                           time_tick: float = 1.0,
                           discrete_time: bool = False,
                           seed: int = 0,
                           istate: Dict[str, bool] = None) -> None
    # Removes a MaBoSS simulation instance from a cell
    @staticmethod
    def delete_maboss_from_cell(cell: CellG, model_name: str) -> None
    # Instantiates and returns a MaBoSS simulation instance from files and/or strings.
    @staticmethod
    def maboss_model(bnd_file: str = None,
                     bnd_str: str = None,
                     cfg_file: str = None,
                     cfg_str: str = None,
                     time_step: float = 1.0,
                     time_tick: float = 1.0,
                     discrete_time: bool = False,
                     seed: int = 0,
```

(continues on next page)

(continued from previous page)

```
        istate: Dict[str, bool] = None) -> MaBoSSCC3DPy.CC3DMaBoSSEngine
# Steps all existing MaBoSS simulations
def timestep_maboss(self) -> None:
# Returns a dictionary with summary statistics of a node of a model
def maboss_stats(self, model_name: str, node_name: str) -> dict:
```



## REAL-WORLD EXAMPLES SECTION

### 24.1 Compartments, FPP Links and Curvature - how to build elongated cells.

The goal of this tutorial is to teach you how to transform the most basic simulation involving just a single cell into a simulation involving multiple elongated cells where each cell is composed of compartments. We want the cells to stay elongated throughout the course of the simulation.

#### 24.1.1 Understanding Contact Energies - how to avoid pixelated cells

Now that we know about compartments and how they are handled by CC3D, let's see how we can translate our knowledge into building a simple simulation that involves contact energies. In particular, we want to show you a common pitfall that you may encounter in your work and how to diagnose it.

In fact this tutorial is an intro to a more sophisticated simulation where we will leverage Focal Point Plasticity Plugin and Curvature plugin.

 **Note**

Our intention is to teach you how you can start building complex simulation from grounds up by starting with a single cell understand the behavior of the single cell under different set of parameters and gradually adding complexity to your simulation. We strongly believe that in order to build robust and complex simulations you first must master simple cases and build confidence needed to bring your modeling skills to the “next level”. It is very much like playing the piano, in general it is advised to learn how to play “Chopsticks” <https://www.youtube.com/watch?v=JM5fjgiFrXg> before attempting to play the “Flight of the Bumblebee” <https://www.youtube.com/watch?v=M93qXQWaBdE>

Let's start. Our first simulation will involve single cell of type “Top” and two plugins - Volume and Contact. The goal is to make the cell look non-pixelized and do not disappear.

The code for this simulation can be found in `Demos/CompuCellPythonTutorial/ElongatedCellsTutorial/Tutorial_01`

 **Note**

The Demos can be open from Player using `File -> Demo Browser` menu. Alternatively you may navigate to <https://github.com/CompuCell3D/CompuCell3D/tree/master/CompuCell3D/core/Demos>

Here is the XML

```

<CompuCell3D>
  <Potts>
    <Dimensions x="100" y="100" z="1"/>
    <Steps>10000</Steps>
    <Temperature>10</Temperature>
    <Flip2DimRatio>1</Flip2DimRatio>
    <NeighborOrder>2</NeighborOrder>
  </Potts>

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

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

  <Plugin Name="Contact">
    <Energy Type1="Medium" Type2="Medium">0</Energy>
    <Energy Type1="Top" Type2="Top">0</Energy>
    <Energy Type1="Top" Type2="Medium">0</Energy>
    <NeighborOrder>4</NeighborOrder>
  </Plugin>

</CompuCell3D>

```

Note that while we are using Contact Energy all the coefficients there are set to 0. As you can expect

Main Python script is simple

```

from cc3d import CompuCellSetup

from ElongatedCellsSteppables import ElongatedCellsSteppable

CompuCellSetup.register_steppable(steppable=ElongatedCellsSteppable(frequency=1))

CompuCellSetup.run()

```

and Python file with steppables is also not too complex:

```

from cc3d.core.PySteppables import *

class ElongatedCellsSteppable(SteppableBasePy):
  def __init__(self, frequency=1):

    SteppableBasePy.__init__(self, frequency)

```

(continues on next page)

(continued from previous page)

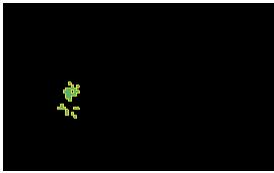
```
def start(self):
    """
    any code in the start function runs before MCS=0
    """

    top = self.new_cell(cell_type=1)
    self.cell_field[45:50, 25:30, 0] = top
```

In the steppable class `ElongatedCellsSteppable` we create a cell of type 1 (this is cell of type Top - see XML above). The syntax `self.cell_field[45:50, 25:30, 0] = top` assigns pointer to cell `top` to all locations of the field enclosed between pixels 45 to 50 along x-axis, pixels 45 to 50 along y-axis and pixels for which z=0. It follows the numpy (<https://numpy.org/>) convention.

The XML is also very simple. We defined 3 cell types there and set `TargetVolume` and `LambdaVolume` to 25 and 2.0 All contact energy coefficients are 0 - effectively stating that contact energy included in the actual simulation is always 0.

If we run this simulation we will get the following:



A partially pixelated cell is not particularly interesting but we should expect this. We created a square cell - see Steppable code above and after few MCS it disintegrated into few pieces. Because we have only volume energy there is nothing to prevent cell pixelization and any cell shape as long as the total number of pixel in the single cell is roughly 25 is perfectly fine.

Let's try using contact energy to see if we can make the cell non-pixelized - [Demos/CompuCellPythonTutorial/ElongatedCellsTutorial/Tutorial\\_02](#) The rationale is as follows: Volume energy will assure the number of pixel in the cell is roughly 25 and the contact energy's task will be to keep cell from pixelizing by penalizing cell-Medium interface. As you recall, CC3D minimizes energy so if we use positive contact coefficient between cell and the Medium, the simulation the pixelated cell will have quite a high energy - because many single pixels are surrounded by Medium and each such pixel will bring up total energy by multiples of contact energy coefficient.

The actual number of interfaces between single pixel and Medium is control by `<NeighborOrder>` input in Contact Plugin. In our case, we are including interfaces up to 4th nearest neighbor - `<NeighborOrder>4</NeighborOrder>`.

Let's look at the new specification of Contact energy:

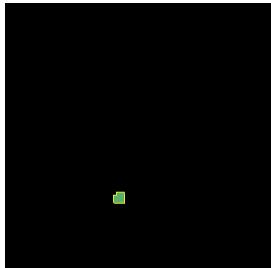
```
<Plugin Name="Contact">
    <Energy Type1="Medium" Type2="Medium">0</Energy>
    <Energy Type1="Top" Type2="Top">0</Energy>
    <Energy Type1="Top" Type2="Medium">15</Energy>
    <NeighborOrder>4</NeighborOrder>
</Plugin>
```

By changing contact energy coefficient between Top cells and Medium to a positive number CC3D will work to minimize Top-Medium interfaces while maintaining total number of pixels of the cell (due to Volume energy term).

It turns out that the cell disappears. Why? This is because Volume energy term was not “strong enough” to overcome minimization of energy coming from Contact energy. Simply put when we get to one-pixel cell and we try to overwrite this pixel by Medium the Volume energy plugin will contribute positive term to change of energy and Contact energy will contribute negative term (because loosing cell medium interfaces leads to a negative change energy).

Let's try fixing it by “strengthening” Volume energy term

```
<Plugin Name="Volume">
  <TargetVolume>25</TargetVolume>
  <LambdaVolume>4.0</LambdaVolume>
</Plugin>
```



This time we get the desired result.

Let's add few more cells (including of type Center).

```
class ElongatedCellsSteppable(SteppableBasePy):
    def __init__(self, frequency=1):

        SteppableBasePy.__init__(self, frequency)

    def start(self):
        """
        any code in the start function runs before MCS=0
        """

        top = self.new_cell(cell_type=1)
        self.cell_field[45:50, 25:30, 0] = top

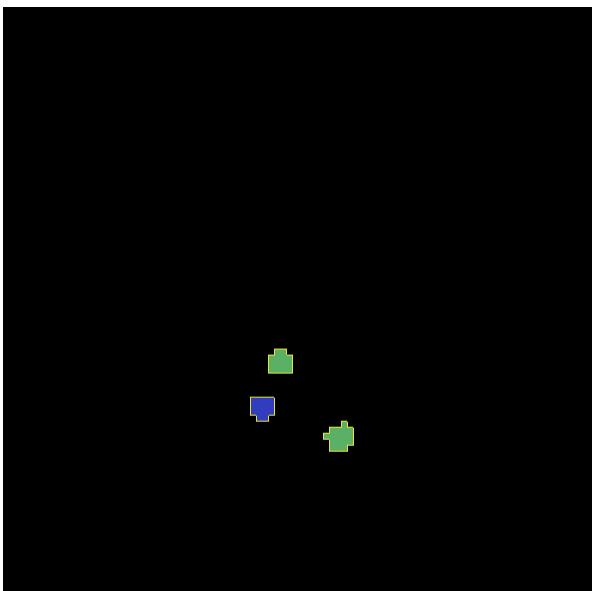
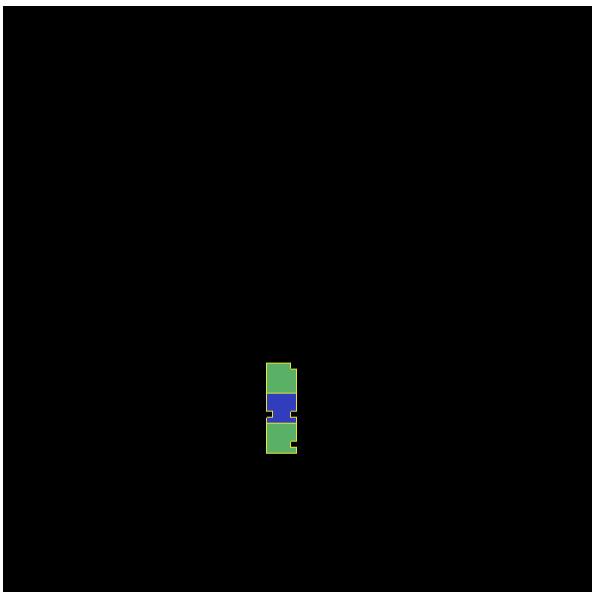
        center_1 = self.new_cell(cell_type=2)
        self.cell_field[45:50, 30:35, 0] = center_1

        top_1 = self.new_cell(cell_type=1)
        self.cell_field[45:50, 35:40, 0] = top_1
```

and let's create a situation where cells “prefer” to be surrounded by Medium and not touch each other. This means contact energy coefficient between cell and Medium is lower than contact energy between two cells:

```
<Plugin Name="Contact">
  <Energy Type1="Medium" Type2="Medium">0</Energy>
  <Energy Type1="Top" Type2="Top">30</Energy>
  <Energy Type1="Top" Type2="Medium">15</Energy>
  <Energy Type1="Center" Type2="Medium">15</Energy>
  <Energy Type1="Center" Type2="Center">30</Energy>
  <Energy Type1="Center" Type2="Top">30</Energy>
  <NeighborOrder>4</NeighborOrder>
</Plugin>
```

When we run this new simulation (Demos/CompuCellPythonTutorial/ElongatedCellsTutorial/Tutorial\_03) we get the following:



Cells that initially stick to each other after few steps are separated but each cell is in a non-pixelated form.

### Writing Convenience function to create elongated cell

When we look at the Python code above where we created 3 cells we can see that it would be nice to have a function that would create entire cell for us. Here is a prototype of such function:

The entire code can be found here: Demos/CompuCellPythonTutorial/ElongatedCellsTutorial/Tutorial\_04

```
class ElongatedCellsSteppable(SteppableBasePy):
    def __init__(self, frequency=1):

        SteppableBasePy.__init__(self, frequency)

    def start(self):
```

(continues on next page)

(continued from previous page)

```

    self.create_arranged_cells(x_s=25, y_s=25, size=5, cell_type_ids=[1, 2, 2, 2, 2, ↵
→ 1])
    for cell in self.cell_list:
        print("cell id=", cell.id, " cluster_id=", cell.clusterId)

def create_arranged_cells(self, x_s, y_s, size, cell_type_ids=None):
    """
    this function creates vertically arranged cells.

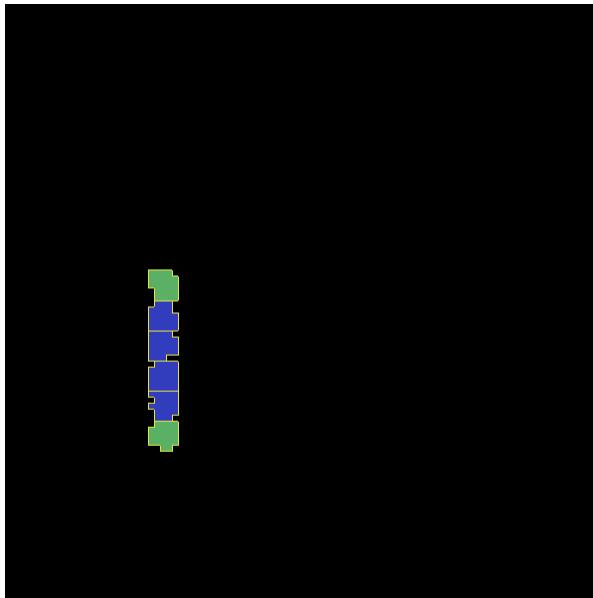
    x_s, ys - coordinates of bottom_left corner of the cell arrangement
    size - size of the cell arrangement
    cell_type_ids - list of cell type ids

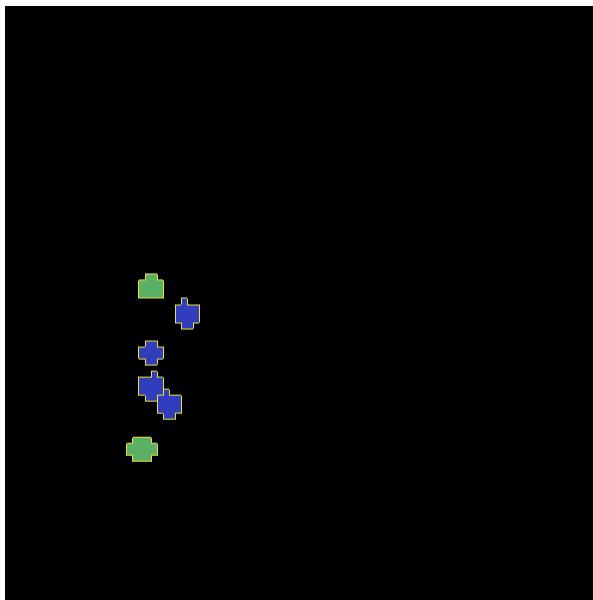
    """
    for i, cell_type_id in enumerate(cell_type_ids):
        cell = self.new_cell(cell_type=cell_type_id)
        self.cell_field[x_s : x_s + size, y_s + i * size : y_s + (i + 1) * size, 0] = cell

```

This function iterates over a list of `cell_type_ids` and for each new cell type listed it creates a new cell of this type that is placed 5 pixel above previous cell. Note that we want to have cell types of “first” and “last” cell to be different because they will play a special role as we start using FocalPointPlasticity Links and ExternalPotential plugins later in the chapter

When we run the simulation we will see the following initial configuration (after first MCS and after several MCS):





Adding convenience functions to your steppables will make your code easier to read and maintain

### 24.1.2 Making Compartmentalized Cell

To understand better the concept of compartmentalized cell we added a printout in the `start` function that prints cells' `cell.id` and `clusterId`. When we dont have compartmentalized cells in our simulation, no two cells share the same `clusterId` or each cell's `id` is paired with a distinct `clusterId` - as we can see from the printout below:

```
cell id= 1 clusterId= 1
cell id= 2 clusterId= 2
cell id= 3 clusterId= 3
cell id= 4 clusterId= 4
cell id= 5 clusterId= 5
cell id= 6 clusterId= 6
```

Let's change it. Instead of creating 6 independent cells let's assign them to the same cluster so that our six cells will now be turned into compartments of a single cell with `clusterId` 1. From the modeling point of view this corresponds to a situation where you would like your biological cells be represented with more level of internal details. Perhaps you would like to simulate polarized cells, or perhaps you may want to better control shape of cells.

Here the code ([Demos/CompuCellPythonTutorial/ElongatedCellsTutorial/Tutorial\\_05](#)) that turns six independent cells in to six compartments of a single compartmentalized cell:

```
def create_arranged_cells(self, x_s, y_s, size, cell_type_ids=None):
    """
    this function creates vertically arranged cells.

    x_s, ys - coordinates of bottom_left corner of the cell arrangement
    size - size of the cell arrangement
    cell_type_ids - list of cell type ids

    """
    cluster_id = None
    for i, cell_type_id in enumerate(cell_type_ids):
        cell = self.new_cell(cell_type=cell_type_id)
```

(continues on next page)

(continued from previous page)

```

if i == 0:
    cluster_id = cell.clusterId
else:
    # to make all cells created by this function, we must reassign clusterId
    # of all the cells created by this function except the first one
    # When the first cell gets created, it gets reassigned clusterId by
    # CompuCell3D and we will use this clusterId to assign it to all other cells
    # created by this function
    self.reassign_cluster_id(cell=cell, cluster_id=cluster_id)
    self.cell_field[x_s : x_s + size, y_s + i * size : y_s + (i + 1) * size, 0] = cell

```

and here is the printout of cell ``id``s and ``clusterId``s:

```

cell id= 1 clusterId= 1
cell id= 2 clusterId= 1
cell id= 3 clusterId= 1
cell id= 4 clusterId= 1
cell id= 5 clusterId= 1
cell id= 6 clusterId= 1

```

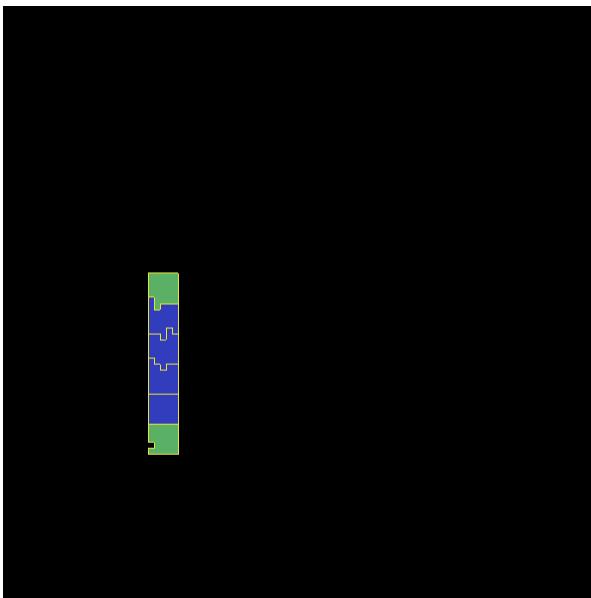
As we can tell all 6 cells share the same `clusterId` which means they represent single compartmentalized cells that is composed of 6 compartments.

It is interesting how we have accomplished this assignment. It is worth pointing out that naive assignment of `cell.id` or `cell.clusterId`, for example `cell.clusterId = 20` will lead to an error:

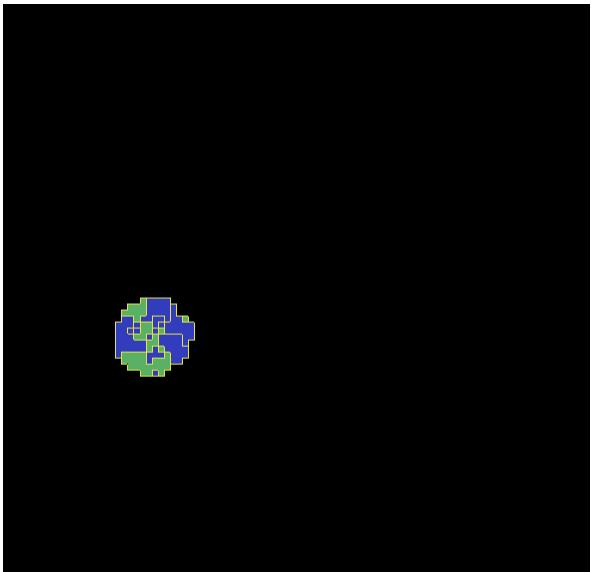
```
AttributeError: ASSIGNMENT cell. clusterId=1 is illegal. clusterId is read only variable
```

Instead you must use steppable built-in function called `self.reassign_cluster_id` to reassign `clusterId` of a cell

If we look at the code - `create_arranged_cells` - notice that when we iterate over list of `cell_type_ids` we first create cell and then we keep track of the `clusterId` of the first cell that was created inside the `for` loop. For each subsequently created cell we reassign its `clusterId` attribute to match the `clusterId` of the cell that was created first. Normally when new cell gets created CC3D will bump both `cell.id` and `cell.clusterId` but by reassigning we are correcting CC3D default behavior so all six cells end up with `clusterId` 1. Here is the initial configuration of the cell field:



But if we run simulation a bit longer we will get the following:



This is not what we expected. In the previous simulation all cells were nicely separated, but now, with the same energy parameters we are getting a completely different simulation where cells are pixelated and intermixed with each other. The only change we did was reassigning ``clusterId``s. The explanation is simple but not obvious. The Contact energy plugin that controls whether cells like to stick to each other or like to be surrounded by Medium works only between cells that are members of different clusters. Contact energy plugin computes adhesion energy between cells that are members of different clusters but when two cells that belong to the same cluster touch each other such junctions are ignored by contact energy. In other words, if we have 5 compartments, i.e. 5 cells that are members of the same cluster the contribution of Contact energy will be 0 regardless if those cells are far apart or clustered together. Those cell-cell interfaces within a single cluster will not contribute anything to the change of energy. Therefore, in order to minimize energy CC3D will bundle cells together, in order to minimize cell-Medium interfaces because Medium is considered a separate cluster so in our simulation we will have two clusters - Medium and cluster composed of 5 cells. Each such interface between those two clusters contributes 15 units of energy and CC3D minimizes those contributions by bundling cells together into a single domain

To make sure this is indeed the case, go back to Demos/CompuCellPythonTutorial/ElongatedCellsTutorial/

Tutorial\_04 and change definition of Contact energy to looks as follows:

```
<Plugin Name="Contact">
  <Energy Type1="Medium" Type2="Medium">0</Energy>
  <Energy Type1="Top" Type2="Top">0</Energy>
  <Energy Type1="Top" Type2="Medium">15</Energy>
  <Energy Type1="Center" Type2="Medium">15</Energy>
  <Energy Type1="Center" Type2="Center">0</Energy>
  <Energy Type1="Center" Type2="Top">0</Energy>
  <NeighborOrder>4</NeighborOrder>
</Plugin>
```

and you will get exactly the same cell shape as we did in the current simulation with compartmentalized cells.

### ContactInternal Plugin

To restore the expected behavior (where we have 6 cells that are members of the same cluster but are not intermingled with each other) we need to add a plugin that will count energy contributions coming from interfaces between cells that belong to the same cluster. Here is the XML code we need to include - Demos/CompuCellPythonTutorial/ElongatedCellsTutorial/Tutorial\_06:

```
<Plugin Name="Contact">
  <Energy Type1="Medium" Type2="Medium">0</Energy>
  <Energy Type1="Top" Type2="Top">30</Energy>
  <Energy Type1="Top" Type2="Medium">15</Energy>
  <Energy Type1="Center" Type2="Center">30</Energy>
  <Energy Type1="Center" Type2="Top">30</Energy>
  <Energy Type1="Center" Type2="Medium">15</Energy>

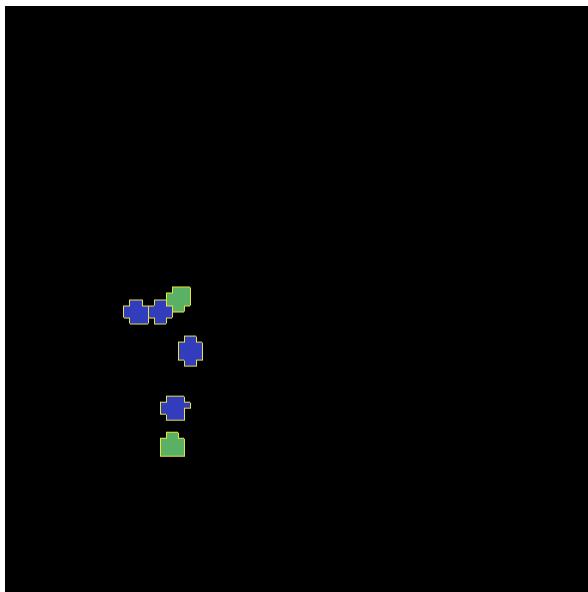
  <NeighborOrder>4</NeighborOrder>
</Plugin>

<Plugin Name="ContactInternal">

  <Energy Type1="Top" Type2="Top">30</Energy>
  <Energy Type1="Center" Type2="Center">30</Energy>
  <Energy Type1="Center" Type2="Top">30</Energy>

  <NeighborOrder>4</NeighborOrder>
</Plugin>
```

Contact and ContactInternal work in tandem. Contact takes care of interfaces between cells that belong to different clusters while ContactInternal computes energies coming from interfaces between cells belonging to the same cluster. Now each cell-cell interface coming from same cluster will add 30 units of energy which combined with 15 units between cell and Medium will cause cells to avoid intermingling.



### FocalPointPlasticity Plugin - constraining intercellular distances

Now that we understand how to handle Contact and ContactInternal plugins, let us focus attention on energy terms that will allow us to constrain distances between cells. *FocalPointPlasticity Plugin* (**FPP**) is one of the solutions.

This plugin implements energy term that penalizes deviations from target distance between two cells that are connected by FPP link. This plugin is described in details in [FocalPointPlasticity Plugin](#), but it is worth mentioning that this plugin has separate mechanisms for handling links between cells that are part of the same cluster and cells that are part of different clusters. The simulation code we will use in this section is in [Demos/CompuCellPythonTutorial/ElongatedCellsTutorial/Tutorial\\_07](#)

In order to add spring-like links between members of the same cluster we need to add the following section to the XML

```
<Plugin Name="FocalPointPlasticity">

    <InternalParameters Type1="Top" Type2="Center">
        <Lambda>100.0</Lambda>
        <ActivationEnergy>-50.0</ActivationEnergy>
        <TargetDistance>5</TargetDistance>
        <MaxDistance>10.0</MaxDistance>
        <MaxNumberOfJunctions>1</MaxNumberOfJunctions>
    </InternalParameters>

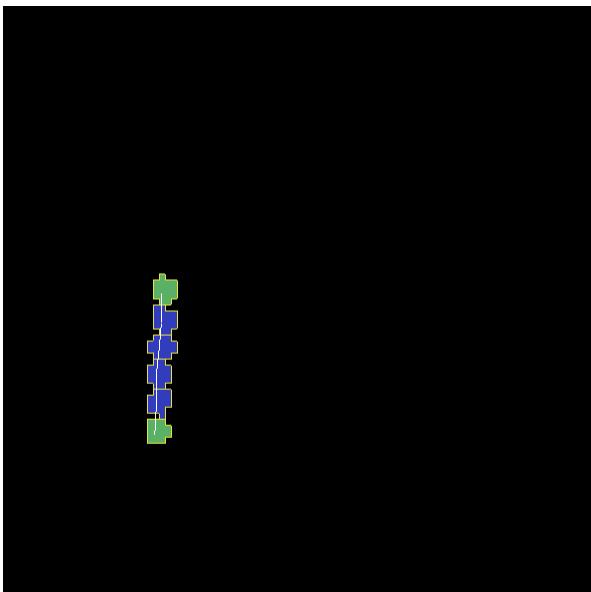
    <InternalParameters Type1="Center" Type2="Center">
        <Lambda>100.0</Lambda>
        <ActivationEnergy>-50.0</ActivationEnergy>
        <TargetDistance>5</TargetDistance>
        <MaxDistance>10.0</MaxDistance>
        <MaxNumberOfJunctions>2</MaxNumberOfJunctions>
    </InternalParameters>

</Plugin>
```

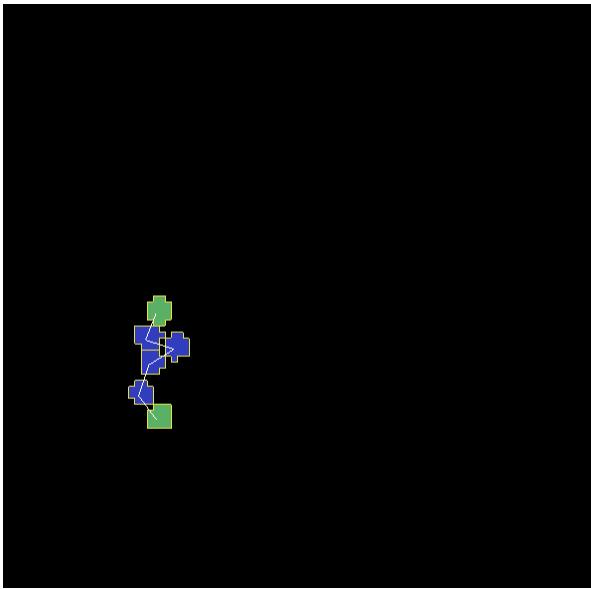
Because in our initial compartment arrangement we have two Top cells capping the “ends” of the cluster we want to allow only a single internal (i.e. between compartments) link between Top and Center cells. However for Center cells we will allow two internal links. Let’s run the simulation and turn on visualization of the links (Visualization->

FPP Links):

After few MCS (FPP links might take few MCS to form because there is stochasticity involved in establishing links between cells) we will see the following picture



If we let the simulation run for a while, however, we will see that while the distance between cells is maintained, the cells do not become elongated.



Additionally, if we lower FPP Lambdas from 100 to 10:

```
<Plugin Name="FocalPointPlasticity">  
  
<InternalParameters Type1="Top" Type2="Center">  
  <Lambda>10.0</Lambda>  
  <ActivationEnergy>-50.0</ActivationEnergy>  
  <TargetDistance>5</TargetDistance>  
  <MaxDistance>10.0</MaxDistance>
```

(continues on next page)

(continued from previous page)

```

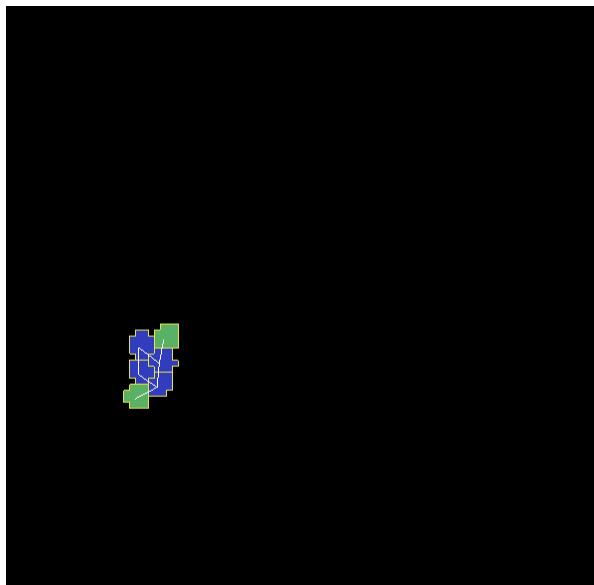
<MaxNumberOfJunctions>1</MaxNumberOfJunctions>
</InternalParameters>

<InternalParameters Type1="Center" Type2="Center">
    <Lambda>10.0</Lambda>
    <ActivationEnergy>-50.0</ActivationEnergy>
    <TargetDistance>5</TargetDistance>
    <MaxDistance>10.0</MaxDistance>
    <MaxNumberOfJunctions>2</MaxNumberOfJunctions>
</InternalParameters>

</Plugin>

```

We will see that Center cells that initially touched Top cell form additional links between themselves. This happens because those two Center cells can form two links between Center themselves. The first link is formed at the beginning of the simulation but during the course of the simulation , when those two Center cells come together (e.g. due to weak FPP Lambda) there is nothing keeping them from forming another link.



### Note

To prevent this situation where Center cells form a “triangle of links” you may add an override in the FocalPoint-Plasticity Plugin that will cap number of total links that Center cells can form to 2 links:

```
<Plugin Name="FocalPointPlasticity">
```

```

<InternalParameters Type1="Top" Type2="Center">
    <Lambda>100.0</Lambda>
    <ActivationEnergy>-50.0</ActivationEnergy>
    <TargetDistance>5</TargetDistance>
    <MaxDistance>10.0</MaxDistance>
    <MaxNumberOfJunctions>1</MaxNumberOfJunctions>
</InternalParameters>

<InternalParameters Type1="Center" Type2="Center">

```

```
<Lambda>100.0</Lambda>
<ActivationEnergy>-50.0</ActivationEnergy>
<TargetDistance>5</TargetDistance>
<MaxDistance>10.0</MaxDistance>
<MaxNumberOfJunctions>2</MaxNumberOfJunctions>
</InternalParameters>

<InternalMaxTotalNumberOfLinks CellType="Center">2</
→InternalMaxTotalNumberOfLinks>

</Plugin>
```

## Curvature Plugin

Let us now put everything together and implement elongated compartmentalized cell. The solution that will prevent two Center cells (the ones that initially were touching Top cell), from forming an extra FPP link, is to use Curvature Plugin. The *Curvature Plugin* constrains the angle at which two adjacent links can form. By using high value of Curvature lambda you may constrain two adjacent links to form a straight line and by adiabatically lowering the lambda you can control how much elongated cell can bend. The code for this section is in [Demos/CompuCellPythonTutorial/ElongatedCellsTutorial/Tutorial\\_08](#)

Here is the code that we add to the XML to enable Curvature energy calculations:

```
<Plugin Name="Curvature">

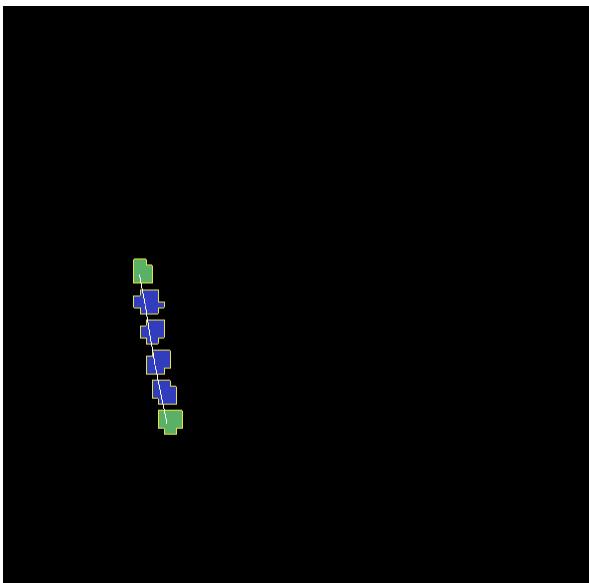
<InternalParameters Type1="Top" Type2="Center">
    <Lambda>1000.0</Lambda>
    <ActivationEnergy>-50.0</ActivationEnergy>
</InternalParameters>

<InternalParameters Type1="Center" Type2="Center">
    <Lambda>1000.0</Lambda>
    <ActivationEnergy>-50.0</ActivationEnergy>
</InternalParameters>

<InternalTypeSpecificParameters>
    <Parameters TypeName="Top" MaxNumberOfJunctions="1" NeighborOrder="1"/>
    <Parameters TypeName="Center" MaxNumberOfJunctions="2" NeighborOrder="1"/>
</InternalTypeSpecificParameters>

</Plugin>
```

With this extra addition the compartments will form a line even if we let the simulation run for a very long time:



As you probably have noticed, the syntax for this plugin resembles the syntax of the FPP plugin - we have `<Lambda>`, `<ActivationEnergy>`, `MaxNumberOfJunctions`. This is because Curvature plugin establishes its own set of “links” between cells but those links are not used to penalize intercellular distance but rather to penalize the deviation from straight line arrangement of compartment cells

### Adding persistent motion to cells

Let us add a bit more code to make this simulation more interesting. First, we will create more cells. We will use our convenience function `create_arranged_cells` and as a result all of those cells will be arranged vertically - this will not be a problem though because, next, we will be applying random force to the “first” cell of each cluster i.e. to the cell that is created first in each cluster. We will store a list of “first” cells inside member variable `self.list_of_leading_cells = []` which is a list. Before we apply any force, we will give simulation a generous 300 MCS for all the FPP links to get established. If we applied force before links are established it is likely that some Top cell could have moved away from the cluster before links had a chance to form. Next, every 500 MCS we will reassign random forces applied to “first” cells.

The simulation code can be found in `Demos/CompuCellPythonTutorial/ElongatedCellsTutorial/Tutorial_09`

In terms of XML modification, we only need to add a one-liner that enables `ExternalPotential` plugin that simulates external force:

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

Notice that we do not specify any parameters because we will use Python to set force vectors applied to “first” cells

We also have to be careful to ensure that cells stored in that list do not disappear because if they do disappear and we try to reference them we will get Segmentation Fault Error. We will show later how we could avoid this issue in the code, just to show you how to handle situation of that type.

The `ElongatedCellsSteppables.py` is more interesting:

```
from cc3d.core.PySteppables import *
import random

class ElongatedCellsSteppable(SteppableBasePy):
```

(continues on next page)

(continued from previous page)

```

def __init__(self, frequency=1):

    SteppableBasePy.__init__(self, frequency)
    self.list_of_leading_cells = []
    self.maxAbsLambdaX = 10

    def start(self):
        # creating 5 "vertical" compartmental cells that are separated from each other
        # in the x direction
        # each compartment consists of 5 compartments that have cell type ids: 1, 2, 2,
        # 2, 1 respectively
        self.create_arranged_cells(x_s=25, y_s=25, size=5, cell_type_ids=[1, 2, 2, 2, 2,
        ↵1])
        self.create_arranged_cells(x_s=40, y_s=25, size=5, cell_type_ids=[1, 2, 2, 2, 2,
        ↵1])
        self.create_arranged_cells(x_s=50, y_s=25, size=5, cell_type_ids=[1, 2, 2, 2, 2,
        ↵1])
        self.create_arranged_cells(x_s=60, y_s=40, size=5, cell_type_ids=[1, 2, 2, 2, 2,
        ↵1])
        self.create_arranged_cells(x_s=70, y_s=60, size=5, cell_type_ids=[1, 2, 2, 2, 2,
        ↵1])

        for cell in self.cell_list:
            print("cell id=", cell.id, " clusterId=", cell.clusterId)

    def create_arranged_cells(self, x_s, y_s, size, cell_type_ids=None):
        """
        this function creates vertically arranged cells.

        x_s, ys - coordinates of bottom_left corner of the cell arrangement
        size - size of the cell arrangement
        cell_type_ids - list of cell type ids
        """

        cluster_id = None
        for i, cell_type_id in enumerate(cell_type_ids):
            cell = self.new_cell(cell_type=cell_type_id)

            if i == 0:
                cluster_id = cell.clusterId
                self.list_of_leading_cells.append(cell)
            else:
                # to make all cells created by this function, we must reassign clusterId
                # of all the cells created by this function except the first one
                # When the first cell gets created, it gets reassigned clusterId by
                # CompuCell3D and we will use this clusterId to assign it to all other
                # cells created by this function
                self.reassign_cluster_id(cell=cell, cluster_id=cluster_id)
                self.cell_field[x_s : x_s + size, y_s + i * size : y_s + (i + 1) * size, 0] =
                ↵= cell

```

(continues on next page)

(continued from previous page)

```
def step(self, mcs):

    if mcs < 300:
        return

    if not mcs % 500:
        # randomize force applied to leading cell
        for cell in self.list_of_leading_cells:
            cell.lambdaVecX = random.randint(-self.maxAbsLambdaX, self.maxAbsLambdaX)
            cell.lambdaVecY = random.randint(-self.maxAbsLambdaX, self.maxAbsLambdaX)
```

Inside the `step` method we create not one but several linear clusters - notice how we vary location of bottom left coordinates of each cluster.

Inside constructor:

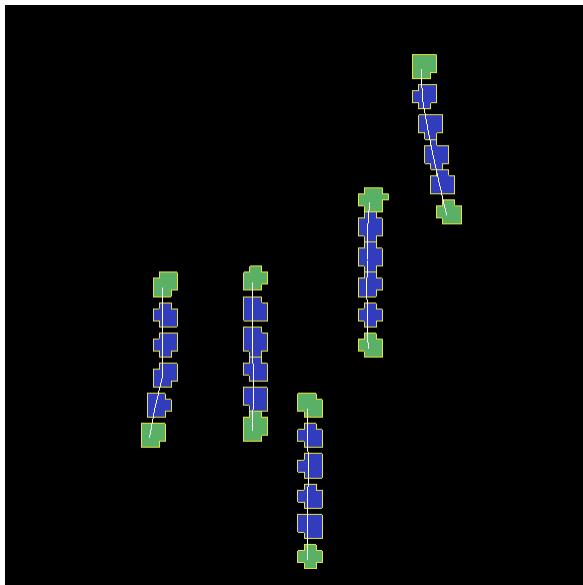
```
def __init__(self, frequency=1):

    SteppableBasePy.__init__(self, frequency)
    self.list_of_leading_cells = []
    self.maxAbsLambdaX = 10
```

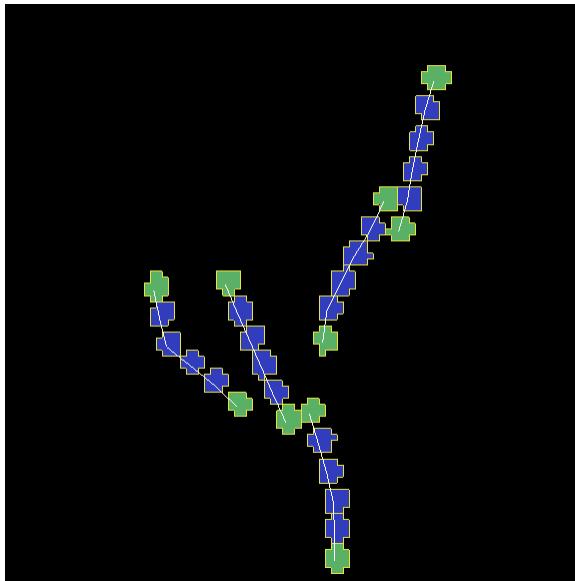
we create `self.list_of_leading_cells` that holds cell objects representing “first” cells of each cluster. Storing of the “first” cell of each cluster takes place inside `self.create_arranged_cells` method. We also add a convenience variable `self.maxAbsLambdaX = 10` that determines absolute value of force component - in x or y directions. We also introduce `step(self, mcs)` which “does nothing” for first 300 MCS and after 300 mcs it assigns a random force to each cell in the `self.list_of_leading_cells` every 500 MCS - we use `if not mcs % 500:` to execute code every 500 MCS.

Here are few screenshots of the simulation:

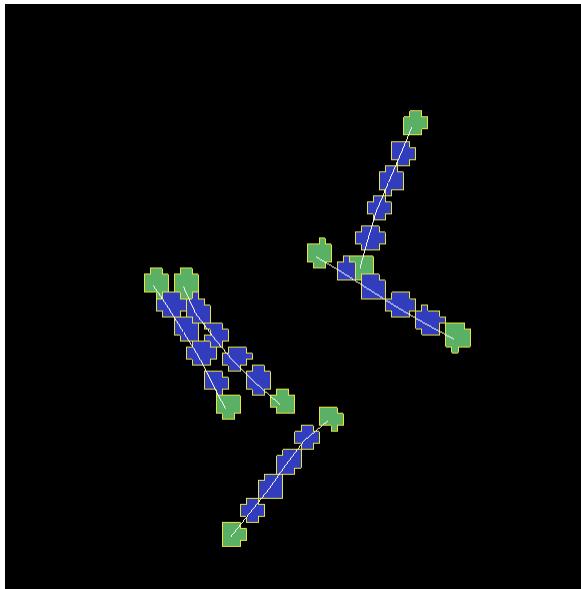
MCS=447:



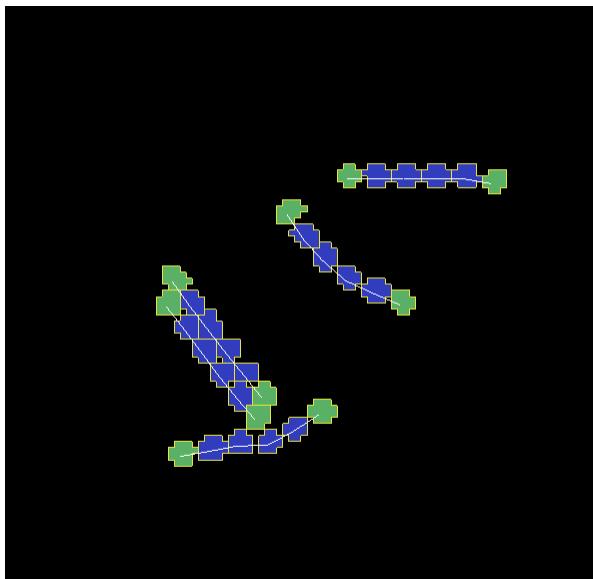
MCS=1125:



MCS=4006:



MCS=8289:



Notice, how cells belonging to a different clusters in general to not “mix with each other”. We can control this behavior by adjusting Contact energy plugin coefficients - because they govern interactions between cells belonging to different clusters

### Note

It is possible that you may apply a force that is too large and FPP links may break. To handle situations like this you should run simulation many times and observe issues and write a code that addresses them

Let us write a more robust code is better prepared to handle cells that may disappear (Top cells to which we apply the force)

```
class ElongatedCellsSteppable(SteppableBasePy):
    def __init__(self, frequency=1):

        SteppableBasePy.__init__(self, frequency)
        self.leading_cells_ids = set()
        self.maxAbsLambdaX = 10

    def start(self):
        self.create_arranged_cells(x_s=25, y_s=25, size=5, cell_type_ids=[1, 2, 2, 2, 2, ↵1])
        self.create_arranged_cells(x_s=40, y_s=25, size=5, cell_type_ids=[1, 2, 2, 2, 2, ↵1])

        self.create_arranged_cells(x_s=50, y_s=5, size=5, cell_type_ids=[1, 2, 2, 2, 2, ↵1])
        self.create_arranged_cells(x_s=60, y_s=40, size=5, cell_type_ids=[1, 2, 2, 2, 2, ↵1])
        self.create_arranged_cells(x_s=70, y_s=60, size=5, cell_type_ids=[1, 2, 2, 2, 2, ↵1])

    for cell in self.cell_list:
        print("cell id=", cell.id, " clusterId=", cell.clusterId)
```

(continues on next page)

(continued from previous page)

```

def create_arranged_cells(self, x_s, y_s, size, cell_type_ids=None):
    """
    this function creates vertically arranged cells.

    x_s, ys - coordinates of bottom_left corner of the cell arrangement
    size - size of the cell arrangement
    cell_type_ids - list of cell type ids

    """
    cluster_id = None
    for i, cell_type_id in enumerate(cell_type_ids):
        cell = self.new_cell(cell_type=cell_type_id)

        if i == 0:
            cluster_id = cell.clusterId
            self.leading_cells_ids.add(cell.id)
        else:
            # to make all cells created by this function, we must reassign clusterId
            # of all the cells created by this function except the first one
            # When the first cell gets created, it gets reassigned clusterId by
            # CompuCell3D and we will use this clusterId to assign it to all other
            # cells created by this function
            self.reassign_cluster_id(cell=cell, cluster_id=cluster_id)
        self.cell_field[x_s : x_s + size, y_s + i * size : y_s + (i + 1) * size, 0] = cell

    def step(self, mcs):

        if mcs < 300:
            return

        if not mcs % 500:
            # randomize force applied to leading cell
            for cell in self.cell_list:
                if cell.id in self.leading_cells_ids:
                    cell.lambdaVecX = random.randint(-self.maxAbsLambdaX, self.
                        maxAbsLambdaX)
                    cell.lambdaVecY = random.randint(-self.maxAbsLambdaX, self.
                        maxAbsLambdaX)

```

Let us outline the changes we made

1. Instead of using `self.list_of_leading_cells` to store cell objects, we use a set `self.leading_cells_ids` to store cell ids (integers). Python set has this nice property that lookups are instantaneous.
2. Instead of iterating the list with cell objects we iterate over each cell in the simulation (yes, a bit inefficient by we can speed it up by iterating over cells of type Top) and we check if `cell.id` is in `self.leading_cells_ids` and only then we apply the force.

This change will avoid accessing Top cell object that was deleted in the course fo the simulation.

In summary, in this case study we have demonstrated how to turn a very simple simulation involving just a single cell into a not-so-trivial simulation that involves multiple motile, elongated compartmental cells.

## 24.2 Examples: Cell Sorting

Try a live demo here: <https://nanohub.org/tools/cc3dcellsort>

Related: [Contact Plugin](#)



In a randomly mixed aggregate of two embryonic cell types, cells of the same type form clusters, which gradually merge until one cell type forms a sphere in the center of the aggregate, with the other type surrounding it.

We call this phenomenon cell sorting, and we say that the outer cell type engulfs the inner cell type. Cell sorting enables each cell to slide into place according to how easily it maintains contact with surrounding cells.

Steinberg's Differential Adhesion Hypothesis states:

1. Cell types have a consistent hierarchy of adhesion energies per unit of surface area when they come into contact with their own and other cell types.
2. The cell's adhesivity depends on the number and type of cell adhesion molecules in its membrane. In our simulation, this can be abstracted into a scalar interface value
3. Cells fluctuate randomly within an aggregate
4. Cells don't grow or change their properties during sorting
5. Less adhesive cell types engulf more adhesive cell types

**Tip:** It's important to note that cells can only interact with their immediate vicinity (based on NeighborOrder). A cell cannot "see" far away to know that it needs to move closer to a cell to adhere to it. Thus, it's possible to get unwanted pockets from cell sorting where some cells become trapped (that is, engulfed by the other cell type).

Here is one possible set of values for contact energy that will achieve cell sorting:

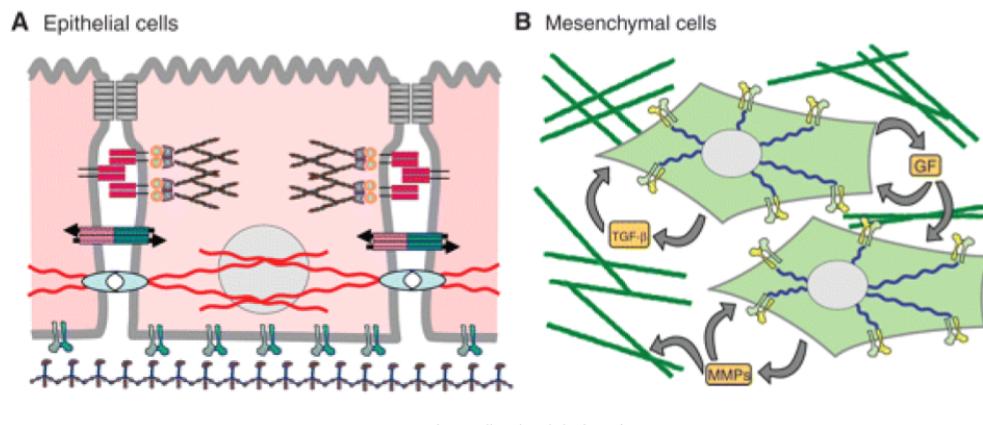
```
<Plugin Name="Contact">
  <Plugin Name="Contact">
    <Energy Type1="Medium" Type2="Medium">0.0</Energy>
    <Energy Type1="CellA" Type2="CellA">16.0</Energy>
    <Energy Type1="CellB" Type2="CellB">2.0</Energy>
    <Energy Type1="CellA" Type2="CellB">11.0</Energy>
    <Energy Type1="CellA" Type2="Medium">16.0</Energy>
    <Energy Type1="CellB" Type2="Medium">16.0</Energy>
  <NeighborOrder>2</NeighborOrder>
</Plugin>
```

Notice that CellB cells will be most likely to adhere because of the value 2.0. Meanwhile, CellA has an equal affinity with other CellA cells and the Medium; this helps it form a thin membrane around the exterior. Finally, the CellA-to-CellB affinity is somewhere between 2 and 16. We picked 11.0. Choosing contact energy often comes down to trial and error to get the behavior you want.

## 24.3 Example: Epithelial-Mesenchymal Transition (EMT)

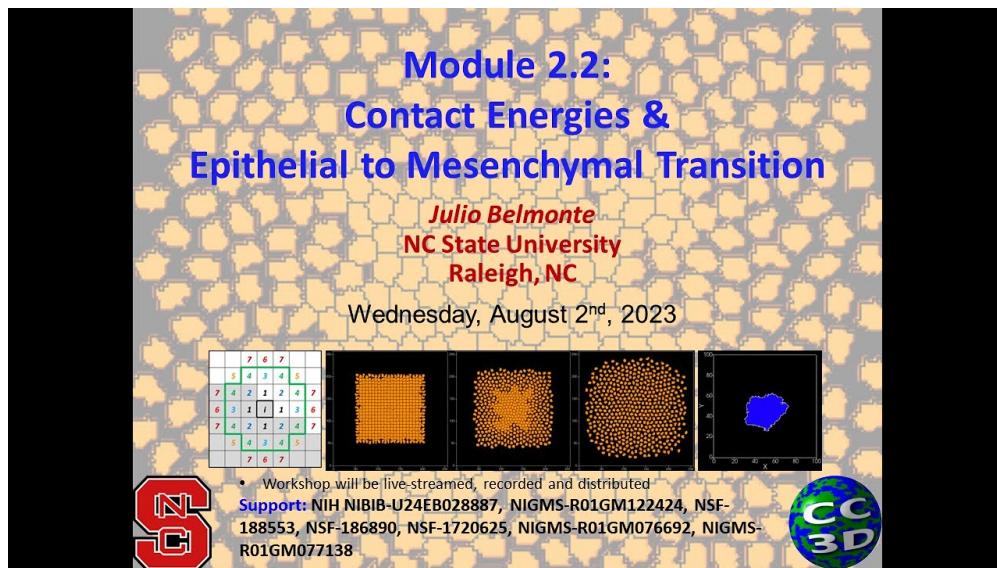
When a cell transitions from an epithelial cell to a mesenchymal cell, it is a hallmark of cancer. The cell loses its orderly, cube-like shape from its epithelial phenotype to a mesenchymal cell that has increased migrational capacity. The mesenchymal cell may be elongated and fragment away from its neighbors to prepare for intravasation.

### Epithelial vs. Mesenchymal cells



Download the sample code [here](#), then watch the video from the latest workshop to follow along:

[Get the slides here.](#)



### Key Takeaways:

- Increasing contact energy between cells of the same type will move the cells apart
- Decreasing contact energy between the mesenchymal cell and the medium will also help those cells move apart
- Very high or low contact energy may cause cells to tear apart



---

## CHAPTER TWENTYFIVE

---

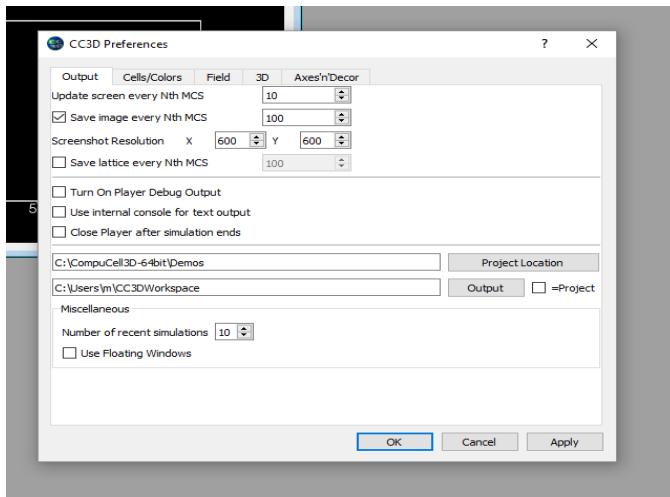
### CONFIGURING MULTIPLE SCREENSHOTS

Starting with CompuCell3d version 3.7.9 users have an option to save multiple screenshots directly from simulation running in GUI or GUI-less mode. Keep in mind that there is already another way of producing simulation screenshots that requires users to first save complete snapshots (VTK-files) and then replaying them in the player and at that time users would take screenshots.

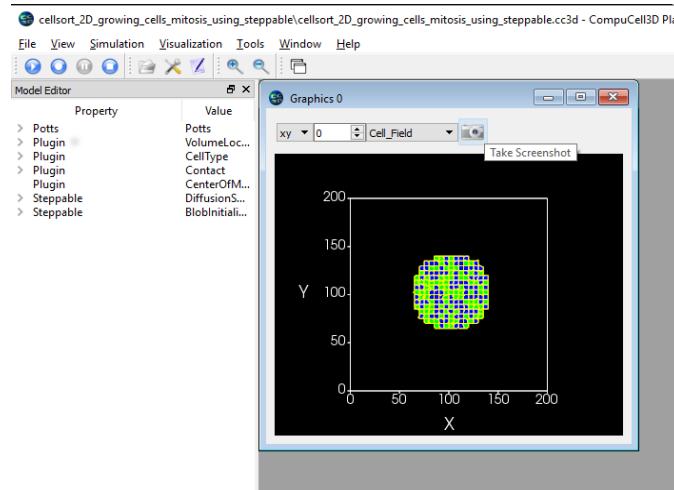
The feature we present here is a very straightforward way to generate multiple screenshots with, literally, few clicks.

The process is very simple - you open up a simulation in the Player and use “camera button” on lattice configurations you want to save. In doing so CompuCell3D will generate .json screenshot description file that will be saved with along the simulation code so that from now on every run of the simulation will generate the same set of screenshots. Obviously we can delete this file if we no longer wish to generate the screenshots.

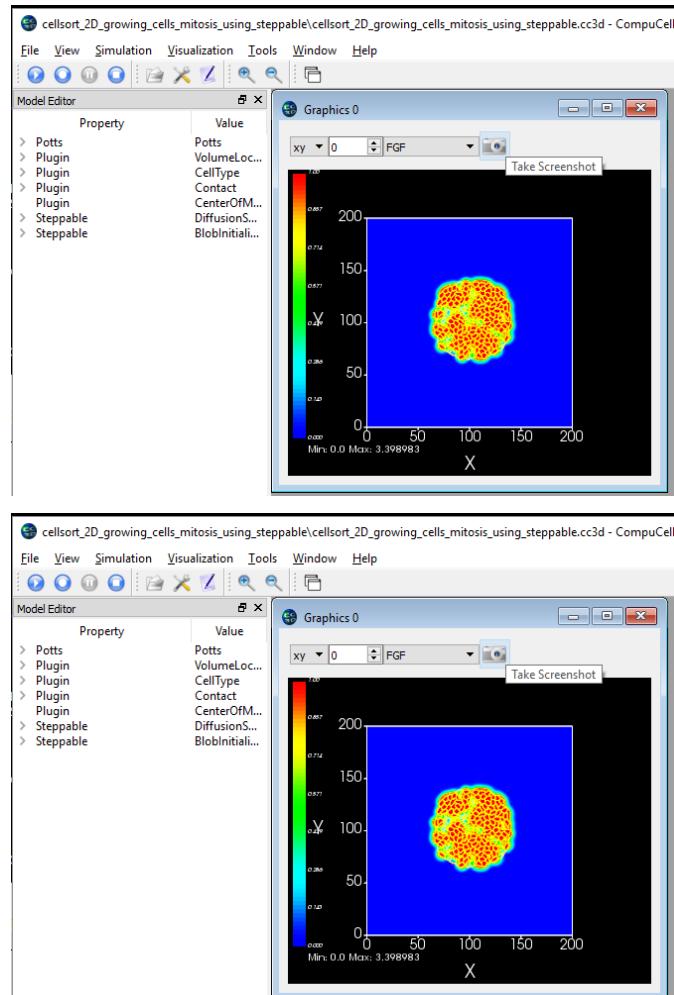
Let's review all the steps necessary to configure multiple screenshots. First we need to enable screenshot output from the configuration page:



**Fig 1.** Enable screenshot output - check box next to Save image every Nth MCS and choose screenshot output frequency

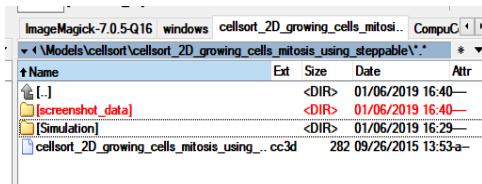


**Fig 2.** Open up simulation and start running it. Press Pause and click camera button (the button next to Take screenshot tool-tip) on the graphics configuration you would like to save.



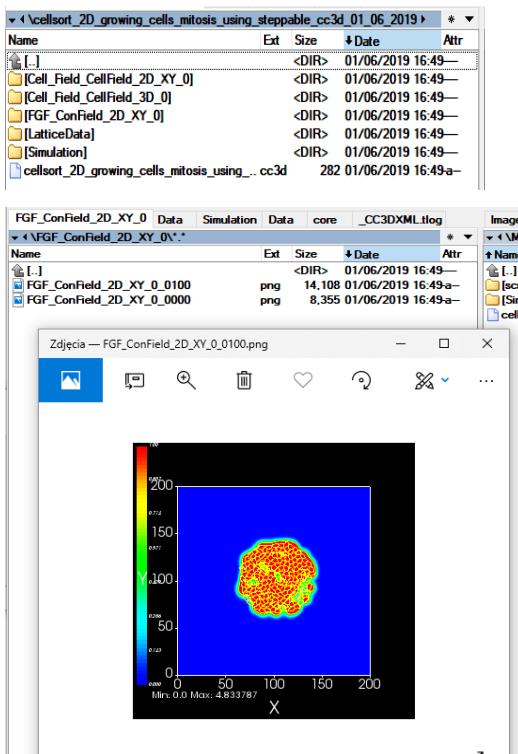
**Fig 3.** Repeat the sam process on other graphics configurations you would like to output as screenshots. Here we are adding screenshots for FGF field and for the cell field in 3D. See pictures above

The screenshot configuration data folder is stored along the simulation code in the original .cc3d project location:



**Fig 4.** When you click camera button , CC3D will store screenshot configuration data in the `screenshot_data` folder and it will become integral part od .cc3d project. Every time you run a simulation screenshots described there will be output to the CC3DWorkspace folder - unless you disable taking of the screenshots via configuration dialog or by removing the `screenshot_data` folder

The screenshots are written in the CC3DWorkspace folder. Simpy go the the subfolder of the CC3DWorkspace directory and search for folders with screenshots. In our case there are 3 folders that have the screenshots we configured: `Cell_Field_CellField_2D_XY_0`, `Cell_Field_CellField_3D_0`, `FGF_ConField_2D_XY_0` - see figures below:



**Fig 5.** Screenshots are written to simulation output folder (*i.e.* subfolder of CC3DWorkspace)



---

## CHAPTER TWENTYSIX

---

### PARAMETER SCANS

#### Note

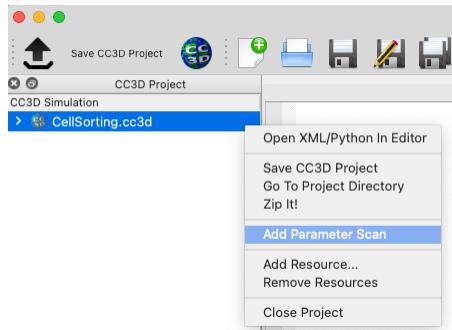
##### Specification of parameter scans in version 4.x.x is different than in earlier versions.

In particular old parameter scan simulations will not run in 4.x but as you will see the new way of specifying parameter scans is much simpler and less laborious than in previous implementations.

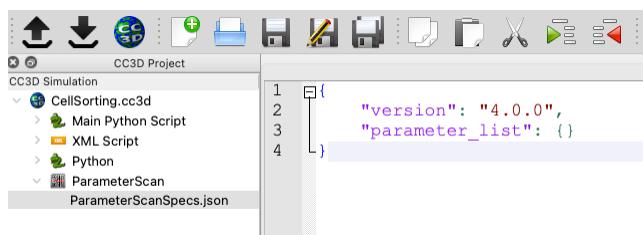
When building biomedical simulations it is a common practice to explore parameter space to search for optimal solution or to study the robustness of parameter set at hand. In the past researchers have used (or abused) Python to run multiple replicas of the same simulation with different parameter set for each run. Because this approach usually involved writing some kind of Python wrapper on top of existing CC3D code, more often than not it led to hard-to-understand codes which were difficult to share and were hard to access by non-programmers.

Current version of CC3D attempts to solve these issues by offering users ability to create and run parameter scans directly from CC3D GUI's or from command line. The way in which parameter scan simulation is run is exactly the same as for “regular”, single-run simulation.

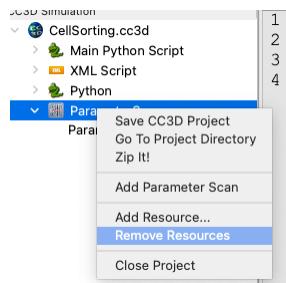
Adding parameter scan to your .cc3d project is best accomplished using Twedit++. Let's open arbitrary project in Twedit++ by navigating to CC3D Project->Open CC3D Project.... To add Parameter Scan, right-click on the top-level project stub in the tree view in the left panel and pick Add Parameter Scan



When we expand the top level project stub, expend the newly added ParameterScan stub and double-click on ParameterScanSpecs.json we will see a pre-formatted template of the parameter scan file:



You may also find it useful to learn how to remove parameter scan from CC3D project. All you need to do is to right-click on the ParameterScan stub and pick Remove Resources



Implementation of parameter scans requires users to populate JSON file with parameter scan specification and replacing actual values in the CC3DML or Python scripts with template markers.

You may find examples of ready-to-run Parameter Scans in Demos/ParameterScan folder in your CC3D installation folder

Let us look at the example simulation in Demos/ParameterScan/CellSorting. The parameter scan specification file `ParameterScanSpecs.json` looks as follows:

```
{
  "version": "4.0.0",
  "parameter_list": {
    "y_dim": {
      "values": [65, 110, 120]
    },
    "steps": {
      "values": [2, 3, 4, 5, 6]
    },
    "MYVAR": {
      "values": [0, 1, 2]
    },
    "MYVAR1": {
      "values": ["'abc1,abc2'", "'abc'"]
    }
  }
}
```

the syntax is fairly simple and if you look closely it is essentially syntax of nested Python dictionaries At the top-level we specify `version` and `parameter_list` entries. The latter one stores several entries each for the parameter we wish to scan.. in our example we will be changing parameter `y_dim` - assigning values from the following list: [65, 110, 120], parameter `steps` with values specified by [2,3,4,5,6], `MYVAR`, that will take values from list [0,1,2] and `MYVAR1`, taking values from ["'abc1,abc2'", "'abc'"]. As you can see, the values we assign can be either numbers or strings.

Next, we need to indicate which parameters in the CC3DML and Python files are to be replaced with values specified in `ParameterScanSpecs.json`. Let's start with analysing CC3DML script:

```
<CompuCell3D version="4.0.0">

<Potts>
  <Dimensions x="100" y="{{y_dim}}" z="1"/>
  <Steps>{{steps}}</Steps>
  <Temperature>10.0</Temperature>
```

(continues on next page)

(continued from previous page)

```
<NeighborOrder>2</NeighborOrder>
</Potts>

...
</CompuCell3D>
```

Here in the `Potts` section we can see two labels that appeared in `ParameterScanSpecs.json` - `{}{y_dim}` and `{}{steps}`. they are surrounded in double curly braces to allow templating engine to make substitutions i.e. `{}{y_dim}` will be replaced with appropriate value from [65, 110, 120] list and, similarly, `{}{steps}` will take values from [2, 3, 4, 5, 6].

The remaining two parameters `MYVAR` and `MYVAR1` will be used to make substitutions in Python steppable script:

```
from cc3d.core.PySteppables import *

MYVAR={{"MYVAR"}}
MYVAR1={{"MYVAR1"}}

class CellSortingSteppable(SteppableBasePy):

    def __init__(self,frequency=1):
        SteppableBasePy.__init__(self,frequency)

    def step(self,mcs):
        #type here the code that will run every _frequency MCS
        global MYVAR

        print ('MYVAR=',MYVAR)
        for cell in self.cell_list:
            if cell.type==self.DARK:
                # Make sure ExternalPotential plugin is loaded
                cell.lambdaVecX=-0.5 # force component pointing along X axis - towards
                ↪positive X's
```

When the parameter scan runs CC3D keeps track of which combinations of parameters to apply at a given moment.

## 26.1 Running Parameter Scans

To run parameter scans you typically need to execute a command that calls `paramScan` script. Those commands are not the easiest things to type because they can be lengthy. For this reason Player has a very convenient tool that lets you open simulation with parameter scan directly from the Player and then you can either copy command and run it in a separate terminal or simply run the scan directly from the Player.

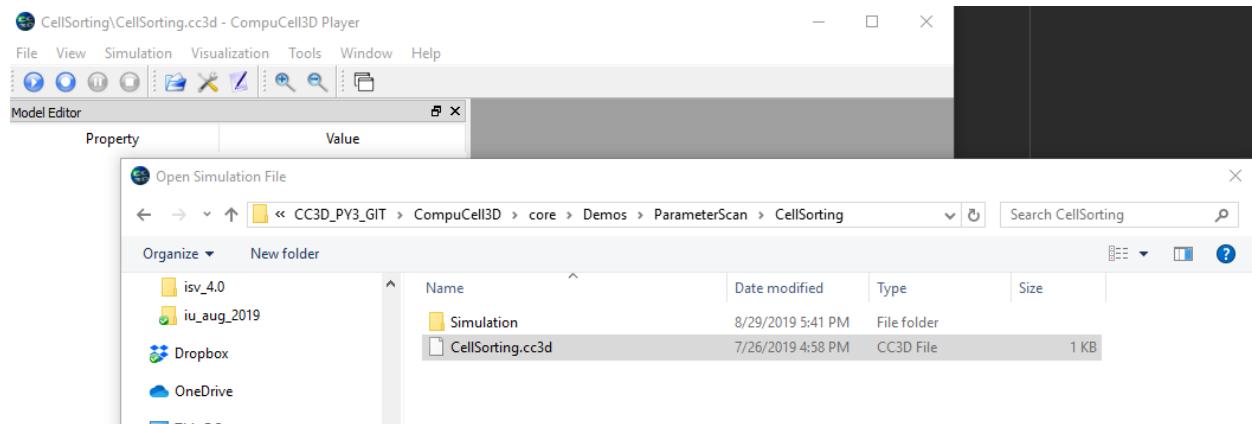
### Note

Parameter scan can execute in parallel. To do so open multiple terminals and execute THE SAME parameter scan command in all of them. `paramScan` script will take care of distributing runs tasks properly.

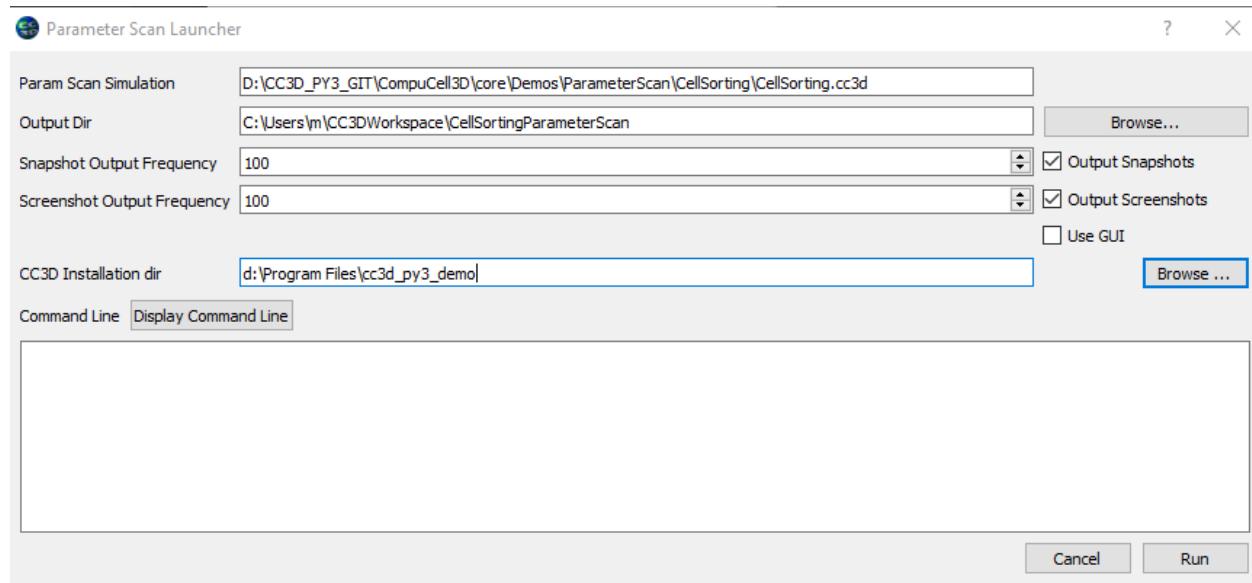
### Note

We recommend that you always run parameter scan from a separate terminal. This is because it is easier to kill it (by closing terminal) than accomplishing the same task from Player.

To run a parameter scan you open up a parameter scan .cc3d project in the Player:



Next, when you click “Play” or “Step” buttons on the Player’s tool bar you will get the following po-pup dialog:



This dialog gives you options to configure how parameter scan gets executed:

**Param Scan Simulation** - here you specify the full path to the .cc3d project that is in fact a parameter scan

**Output Dir** - you can manually select a folder where the output of the parameter scan gets written. By default CC3D will choose a path that is based on globally-configured simulation output folder and the name of the parameter scan .cc3d project

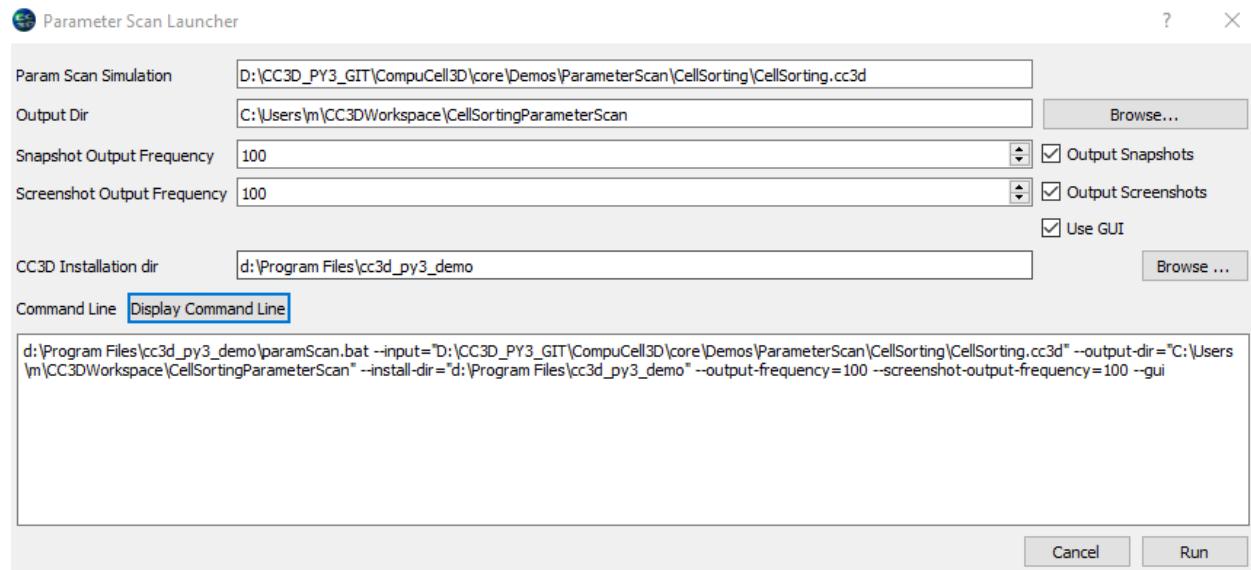
**Snapshot Output Frequency** - specifies how often snapshots (vtk files that you can replay in the Player later) will be taken. The check box next to the spin-box disables snapshot taking altogether.

**Screenshot Output Frequency** - specifies how often screenshots will be taken (provided you configured screenshots for your param scan project). The check box next to the spin-box disables screenshot taking altogether.

**Use Gui** - this checkbox will cause that every simulation that is part of the parameter scan will be executed in the Player.

**CC3D Installation Dir** - specifies where CC3D is installed. this field is populated by CC3D but you can modify it if you really want to use a different installation folder for CC3D

Once you are happy with your configurations you press “Display Command Line” button and in the text box below you will see the command line text for paramScan script.



At this point you have two options

1. Copy the command line text and paste it in the terminal
2. Press Run button at the bottom of the dialog

In both cases parameter scan will start running

### **⚠ Warning**

Pasting long command lines on Windows may not work as expected. For historical reasons some terminals on windows limit the total size of the pasted text to 255 characters. In this situation you probably want to run parameter scan from the Player or try to find console application on windows that does not have such limitation. For example if you install Miniconda or Anaconda on windows and use Anaconda Prompt it will open a console that will behave correctly

Although it is easiest to use Player to launch parameter scans, we also present the options that paramScan script takes. Just remember to use appropriate script ending for your operating system - paramScan.bat (windows), paramScan.sh (linux) or paramScan.command (osx):

```
paramScan.command --input=<path to the CC3D project file (*.cc3d)> --output-dir=<path to the output folder to store parameter scan results> --output-frequency=<simulation snapshot output frequency> --snapshot-output-frequency=<screenshot output frequency> --gui --install-dir=<CC3D install directory>
```

for example to run above simulation on OSX one could type

```
./paramScan.command --input=/Users/m/Demo2/CC3D_4.0.0/Demos/ParameterScan/CellSorting/
    ↵CellSorting.cc3d --output-dir=/Users/m/CC3DWorkspace/ParameterScanOutput --output-
    ↵frequency=2 --Screenshot-output-frequency=2 --gui --install-dir=/Users/m/Demo2/CC3D_4.
    ↵0.0
```

 Note

You may easily run parameter scans in parallel. Simply execute above command from different terminals and CC3D will synchronize multiple instances of paramScan scripts and as a result you will run several simulations in parallel which will come handy once you are scanning many values of parameters

## 26.2 Using numpy To Specify Parameter Lists

In the above example we used simple Python list syntax to specify list of parameters. this works for simple cases but

when you are dealing with a more sophisticated cases when you require e.g. points to be distributed logarithmically then you would need to pregenerate such list in external program (e.g. Python console) and copy/paste values into parameter scan file. Fortunately CC3D allows you to use numpy syntax directly in parameter scan specification file:

```
{
  "version": "4.0.0",
  "parameter_list": {
    "y_dim": {
      "code": "np.arange(165,220,3, dtype=int)"
    },
    "steps": {
      "code": "list(range(5,11,1))"
    },
    "MYVAR": {
      "code": "np.linspace(0,2.3, 10)"
    },
    "MYVAR1": {
      "values": ["'abc1,abc2'", "'abc'"]
    }
  }
}
```

The structure of the file looks the same but when we replace values with code we can type actual numpy statement and it will be evaluated by CC3D. Clearly, as shown above, you can mix-and-match which parameters are specified using numpy statement and which ones are specified using simple Python lists.

One particularly useful type parameter scan you should run once your model is mature is the one where you vary <RandomSeed> in the <Potts> section and keep all other parameters intact. By doing repeat runs of your simulation with different random seed you can quantify how stochasticity affects simulation outcomes.

Here is a minimal setup for such simulation (also shown in Demos/ParameterScan/CellSortingRandomSeedScan)

ParameterScanSpecs.json:

```
{
  "version": "4.0.0",
```

(continues on next page)

(continued from previous page)

```
"parameter_list": {  
    "random_seed":{  
        "code":"np.random.randint(10, 99999, size=100)"  
    }  
}
```

The statement `np.random.randint(10, 99999, size=100)` generates an array of 100 random numbers between 10 and 99999

CellSorting.xml:

```
<CompuCell3D version="3.6.2">  
  
<Potts>  
    <Dimensions x="100" y="100" z="1"/>  
    <Steps>110</Steps>  
    <Temperature>10.0</Temperature>  
    <NeighborOrder>2</NeighborOrder>  
    <RandomSeed>{{random_seed}}</RandomSeed>  
</Potts>  
  
...
```



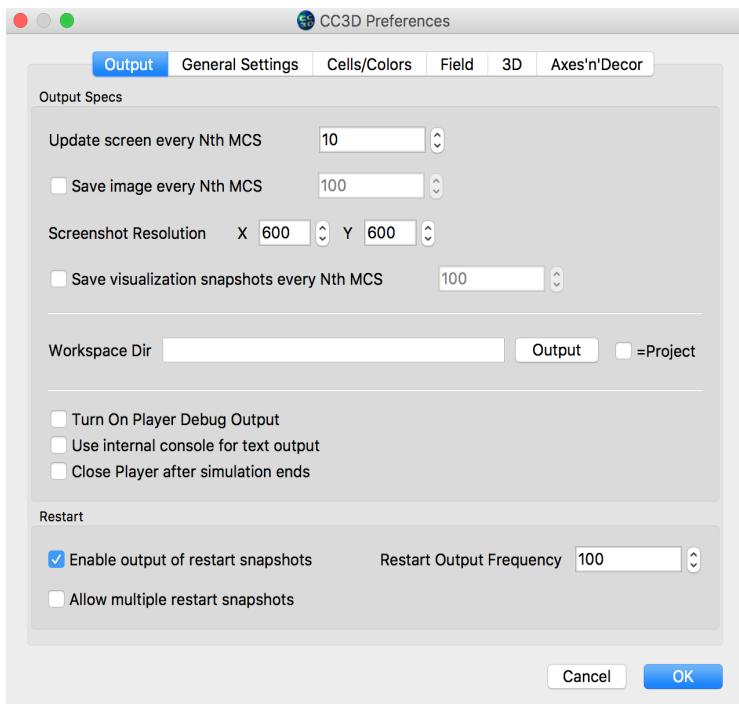
---

## CHAPTER TWENTYSEVEN

---

### RESTARTING SIMULATIONS

Very often when you run CC3D simulations you would like to save the state of the simulation and later restart it from the place where you interrupted it. For example let's say that you are running vascularized tumor simulation on a  $500 \times 500 \times 500$  lattice. After the simulated tumor reaches certain size or mass you may want to simulate various treatment strategies. Instead of running full simulations from the beginning and turning on treatment at specific MCS you may run a simulation up to the point when tumor reaches required mass and then start new set of simulations. This would save you a lot of computational time. Another advantage of the ability to restart simulations at arbitrary time point is when you run your job on a cluster or cloud and there is a risk that your simulation may get interrupted. In this case you could periodically save state of the simulation and then restart it from the latest snapshot. CompuCell3D makes all such tasks very easy. The only thing you need to do is to click **Enable output of restart snapshots**.



Alternatively if you run simulation using command line script `runScript` you need to pass the following argument to your command line:

```
--restart-snapshot-frequency=<restart snapshot output frequency>
```

CompuCell3D will write simulation snapshots to the restart folder. You can navigate to the output folder and load such simulation as if it were a regular one

 **Warning**

When you run simulation from restart snapshot `start` functions of steppables are not called. Therefore you need to take steps to ensure that any type of initialization that might still be required gets executed in the step function. Let's study example below:

Supose that in our original simulation we create plots inside start function, clearly the plots are not created during the restart and since `step` function refers to plot window object the error arises. Take a look at our original code and see if you can follow what I explained here:

```
class SBMLSolverOscilatorDemoSteppable(SteppableBasePy):

    def __init__(self, frequency=10):
        SteppableBasePy.__init__(self, frequency)
        self.pW = None

    def start(self):
        self.pW = self.add_new_plot_window(title='S1 concentration', x_axis_title=
        ←'MonteCarlo Step (MCS)', y_axis_title='Variables')
        self.pW.addPlot('S1', _style='Dots', _color='red', _size=5)

        # iterating over all cells in simulation
        for cell in self.cell_list:
            # you can access/manipulate cell properties here
            cell.targetVolume = 25
            cell.lambdaVolume = 2.0
        ...

    def step(self, mcs):
        ...
        self.timestep_sbml()
```

A simple fix (not necessarily optimal one) would be to introduce a new function `initialize_plots` that first checks if `self.pW` plot window object is `None` and if so it creates a plot otherwise it exits. Of course for this to work `self.pW` will need to be declared in the steppable constructor `__init__` (constructors of steppables are called during restart initialization)

Here is the code – changes are highlighted using bold code:

```
class SBMLSolverOscilatorDemoSteppable(SteppableBasePy):

    def __init__(self, frequency=10):
        SteppableBasePy.__init__(self, frequency)
        self.pW = None

    def initialize_plots(self):
        if self.pW:
            return

        self.pW = self.add_new_plot_window(title='S1 concentration', x_axis_title=
        ←'MonteCarlo Step (MCS)',
```

(continues on next page)

(continued from previous page)

```
y_axis_title='Variables')
self.pW.addPlot('S1', _style='Dots', _color='red', _size=5)

def start(self):
    self.initialize_plots()

    # iterating over all cells in simulation
    for cell in self.cell_list:
        # you can access/manipulate cell properties here
        cell.targetVolume = 25
        cell.lambdaVolume = 2.0

    ...

def step(self, mcs):
    ...
    self.initialize_plots()
    self.timestep_sbml()
```

To wrap up, setting up simulation restart is quite easy in CC3D. Making sure that simulation restarts properly may require you to slightly modify your code to account for the fact that start functions of steppables are not called during restart.



---

CHAPTER  
**TWENTYEIGHT**

---

## STEPPABLE SECTION

Steppables are CompuCell modules that are called every Monte Carlo Step (MCS). More precisely, they are called after all the pixel copy attempts in a given MCS have been carried out. Steppables may have various functions - for example solving PDEs, checking if critical concentration threshold have been reached, updating target volume or target surface given the concentration of some growth factor, initializing cell field, writing numerical results to a file, *etc...*. In general, steppables perform all functions that need to be done every MCS. In the remainder of this section we will present steppables currently available in the CompuCell3D and describe their usage.

 **Tip**

It is most convenient to implement Steppables in Python. However, in certain situations where code performance is an issue users can implement steppables in C++

This section presents “off-the-shelf” steppables that are available in CC3D and were implemented using C++. Those are the steppables that are pre-programmed for you and you have little control over them (except specifying input parameters). If you are interested in how to write your own Steppable in Python please see [Python Tutorials Section](#)

For more advanced users who want to write Steppables in C++ to improve the performance we recommend getting familiar with [Developers' Manual](#)

### 28.1 BoxWatcher Steppable

 **Warning**

Functionality of this module has been reduced in CC3D versions that support parallel computations (3.6.0 and up). Main motivation for this module was to speed up computations but with parallel version the need for this module is somewhat smaller.

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

The syntax is as follows:

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

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

## 28.2 BlobInitializer Steppable

`BlobInitializer` steppable is used to lay out circular blob of cells on the lattice.

An example syntax where we create one circular region of cells in the lattice is presented below:

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

Similarly as for the `UniformFieldInitializer` users can define many regions each of which is a blob of a particular center point, radius and list of cell types that will be assigned to cells forming the blob. Listing types in the `<Types>` tag follows same rules as in the `UniformInitializer`.

### Note

Original (**and deprecated**) syntax of this plugin looks as follows:

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

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

## 28.3 PIF Initializer

To initialize the configuration of the simulation lattice we can write custom **lattice initialization file**. Our experience suggests that you will probably have to write your own initialization files rather than relying on built-in initializers. The reason is simple: the built-in initializers implement very simple cell layouts, and if you want to study more complicated cell arrangements, the built-in initializers will not be very helpful. Therefore, we encourage you to learn how to prepare lattice initialization files. We have developed `CellDraw` tool which is a part of CC3D suite and it allows users to draw

initial cell layout in a very intuitive way. We encourage you to read “Introduction to CellDraw” to familiarize yourself with this tool.

To import custom cell layouts, CompuCell3D uses very simple **Potts Initial File** PIF file format. It tells CompuCell3D how to lay out assign the simulation lattice pixels to cells.

The PIF consists of multiple lines of the following format:

```
cell# celltype x1 x2 y1 y2 z1 z2
```

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

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

Let’s look at a PIF example for foams:

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

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

You can write the PIF file manually, but using a script or program that will write PIF file for you in the language of your choice (Perl, Python, Matlab, Mathematica, C, C++, Java or any other programming language) will save a great deal of typing.

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

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

For example:

```
Include Clusters
1 1 Side1 23 25 47 56 10 14
1 2 Center 26 30 50 54 10 14
1 3 Side2 31 33 47 56 10 14
1 4 Top 26 30 55 59 10 14
1 5 Bottom 26 30 45 49 10 14
2 6 Side1 35 37 47 56 10 14
2 7 Center 38 42 50 54 10 14
2 8 Side2 43 45 47 56 10 14
2 9 Top 38 42 55 59 10 14
2 10 Bottom 38 42 45 49 10 14
```

**Tip**

An easy way to generate PIF file from the current simulation snapshot is to use Player Tools->Generate PIF file from current snapshot... menu option. Alternatively, we can use the PIFDumper steppable, which will be discussed next.

## 28.4 PIFDumper Steppable

This steppable does the opposite to PIFIinitialize``r - it writes PIF file of current lattice configuration. The syntax similar to the syntax of ``PIFInitializer:

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

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

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

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

**Tip**

Restarting simulation from a given configuration is actually even easier in the recent versions of CC3D. All you have to do is to create .``cc3d`` project where you add serialization option CC3D will be saving complete snapshots of the simulation (including PIF files) and you can easily restart the simulation from a given end-point of the previous run. For more details see [Restarting Simulations](#).

**Tip**

You can also generate PIF file from the current simulation snapshot by using Player tool: Tools->Generate PIF file from current snapshot...

---

CHAPTER  
TWENTYNINE

---

## PLUGINS SECTION

In this section we overview CC3DML syntax for all the plugins available in CompuCell3D. Plugins are either energy functions, lattice monitors or store user assigned data that CompuCell3D uses internally to configure simulation before it is run.

### 29.1 CellType Plugin

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

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

Here we have defined three cell types that will be present in the simulation: **Medium**, **Fluid**, **Wall**. Notice that we assign a number – **TypeId** – to every cell type. It is strongly recommended that **TypeId**'s are consecutive positive integers (e.g. **0, 1, 2, 3...**). **Medium** is traditionally given **TypeId=0** and we recommend that you keep this convention.

 Note

**Important:** Every CC3D simulation must define **CellType Plugin** and include at least **Medium** specification.

Notice that in the example above cell type **Wall** has extra attribute **Freeze=""**. This attribute tells CompuCell that cells of **frozen** type will not be altered by pixel copies. Freezing certain cell types is a very useful technique in constructing different geometries for simulations or for restricting ways in which cells can move.

### 29.2 Global Volume and Surface Constraints

Related: [Volume and Cell Growth \[Reference\]](#) and [Surface and Cell Contact \[Reference\]](#)

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

```
<Plugin Name="Volume">
    <VolumeEnergyParameters CellType="CellA" LambdaVolume="2.0" TargetVolume="25"/>
</Plugin>
```

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

```
<Plugin Name="Surface">
  <SurfaceEnergyParameters CellType="CellA" LambdaSurface="1.5" TargetSurface="20.0"/>
</Plugin>
```

These two plugins inform CompuCell that the Hamiltonian will have two additional terms associated with volume and surface conservation. That is, when a pixel copy is attempted, one cell will increase its volume and another cell will decrease. Thus, the overall energy of the system changes. Volume constraint essentially ensures that cells maintain the volume which close to (this depends on thermal fluctuations) its target volume. The role of the surface plugin is analogous to volume; it “preserves” the total length of the cell’s surface.

The examples shown below apply volume and surface constraints to all cells in the simulation. You can, however, apply volume and surface constraints to individual cell types or individual cells.

Energy terms for volume and surface constraints have the form:

*to*

$$E_{volume} = \lambda_{volume} (V_{cell} - V_{target})^2 \quad (29.1)$$

*to*

$$E_{surface} = \lambda_{surface} (S_{cell} - S_{target})^2 \quad (29.2)$$

### Note

Copying a single pixel may cause surface change in more than two cells – this is especially true in 3D.

### Note

The Volume and Surface plugins handle their respective behaviors automatically as long as they are added to the XML. However, if your particular use case needs to avoid the built-in constraints brought by these plugins, you can still [track the volume and surface manually](#).

## 29.3 VolumeTracker and SurfaceTracker plugins

Related: [Global Volume and Surface Constraints](#)

These two plugins monitor the lattice and update the volume and surface of the cells once a pixel copy occurs. In most cases, users will not call those plugins directly. They will be called automatically when either Volume (together with VolumeTracker) or Surface or CenterOfMass (calls VolumeTracker) plugins are requested. However, one should be aware that in some situations, for example when doing foam coarsening, where neither Volume nor Surface plugins are called, one may still want to track changes in surface or volume of cells. In such situations, we explicitly invoke the VolumeTracker or Surface plugins with the following syntax:

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

As of version 4.6.0, all you have to do to the Surface plugin to enable this behavior is add NeighborOrder.

```
<Plugin Name="Surface">
  <TargetSurface>120</TargetSurface>
  <LambdaSurface>0.5</LambdaSurface>
  <NeighborOrder>4</NeighborOrder>
</Plugin>
```

This will enable you to access the current volume and surface in Python with `cell.volume` and `cell.surface`.

### Note

#### Legacy Version (Pre 4.6.0)

Previously, this arrangement of plugins was required. SurfaceTracker is now deprecated.

```
<Plugin Name="SurfaceTracker">
  <NeighborOrder>4</NeighborOrder>
</Plugin>

<Plugin Name="Surface">
  <TargetSurface>120</TargetSurface>
  <LambdaSurface>0.5</LambdaSurface>
</Plugin>
```

## 29.4 VolumeFlex Plugin

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

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

### Note

Almost all CompuCell3D modules which have options Flex or LocalFlex are implemented as a single C++ module and CC3D, based on CC3DML syntax used, figures out which functionality to load at the run time. As a result for the reminder of this reference manual we will stick to the convention that all Flex and LocalFlex modules will be invoked using core name of the module only.

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

Using VolumeFlex Plugin you can effectively freeze certain cell types. All you need to do is to put very high lambda coefficient for the cell type you wish to freeze. You have to be careful though, because if initial volume of the cell of a given type is different from target volume for this cell type the cells will either shrink or expand to match target volume and only after this initial volume adjustment will they remain frozen provided `LambdaVolume` is high enough. Since rapid changes in the cell volume are uncontrolled (e.g. they can destroy many neighboring cells) you should opt for more gradual changes. In any case, we do not recommend this way of freezing cells because it is difficult to use, and also not efficient in terms of speed of simulation run.

## 29.5 SurfaceFlex Plugin

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

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

## 29.6 VolumeLocalFlex Plugin

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

Example syntax:

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

## 29.7 SurfaceLocalFlex Plugin

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

Example syntax:

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

## 29.8 NeighborTracker Plugin

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

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

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

To include this plugin in your simulation , add the following code to the CC3DML

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

This plugin is used as a helper module by other plugins and steppables e.g. FocalPointPlasticity plugin uses NeighborTracker plugin.

## 29.9 Chemotaxis

Chemotaxis plugin, is used to simulate the chemotaxis of cells. Put simply, a cell will follow the highest concentration of a chemical field. Note that a cell can only move towards or away from a chemical field if it can detect any concentration above 0 at its surface. The cell cannot “see” far away to know where to move.

Related: [Chemotaxis Examples in Python](#).

### 29.9.1 Properties

Assume `self` is an instance of `SteppableBasePy`.

`self.chemotaxisPlugin.addChemotaxisData(cell, fieldName: string)`: adds a chemotaxis behavior that will cause the given `cell` to respond to the chemical field specified by `fieldName`. `fieldName` must match the name of the chemical field exactly. Returns a chemotaxis data object, which has the following methods:

- `chemotaxisData.setLambda(lambda: float)`: assigns the lambda of the given chemotaxis behavior, which controls the intensity of the chemotaxis

(see definition below).

- `chemotaxisData.assignChemotactTowardsVectorTypes([cellType1, cellType2, ...])`: causes chemotaxis to trigger when the cell touches

one or more of the other cell types in the provided list.

- **chemotaxisData.setLogScaledCoef(coef: float)**: assign the log-scaled

coefficient of the given chemotaxis behavior, which helps to mitigate excessive orders of magnitude in chemotaxis decisions (see definition below).

**self.chemotaxisPlugin.getChemotaxisData(cell, fieldName: string)**: finds a chemotaxis behavior that is already assigned to the cell and returns it. `fieldName` must match the name of the chemical field exactly.

---

## 29.9.2 How It Works

For every pixel copy, this plugin calculates the change of energy associated with the pixel move. There are several methods to define a change in energy due to chemotaxis. By default, we define a chemotaxis using the following formula:

$$\Delta E_{chem} = -\lambda (c(x_{destination}) - c(x_{source})) \quad (29.3)$$

where  $c(x_{source})$  and  $c(x_{destination})$  denote chemical concentration at the pixel-copy-source and pixel-copy-destination pixel, respectively.

The body of the `Chemotaxis` plugin description contains sections called `ChemicalField`. In this section, we tell CompuCell3D which module contains a chemical field that we wish to use for chemotaxis. In our case it is `FlexibleDiffusionSolverFE`. Next, we specify the name of the field - FGF. Subsequently, we specify `lambda` for each cell type so that cells of different types may respond differently to a given chemical. In particular, types not listed will not respond to chemotaxis at all.

**Lambda**: Controls how quickly a cell will move in response to a chemical field. More precisely, it affects the *decision* of whether or not a cell's pixel will be copied so that the cell can move. If a cell has multiple chemicals that it is exposed to, it is more inclined to move towards the field that its lambda is higher for. If lambda is positive, the cell will move toward the field. Conversely, if it is negative, the cell will move away. Note that the *absolute value* controls the intensity, so -100 and +100 will each have a similar effect on the cell.

---

Occasionally, we may want to use a different formula for the chemotaxis than the one presented above. The current version of CompCell3D supports the following definitions of change in chemotaxis energy (`Saturation` and `SaturationLinear` respectively):

$$\Delta E_{chem} = -\lambda \left[ \frac{c(x_{destination})}{s + c(x_{destination})} - \frac{c(x_{source})}{s + c(x_{source})} \right] \quad (29.4)$$

or

$$\Delta E_{chem} = -\lambda \left[ \frac{c(x_{destination})}{sc(x_{destination}) + 1} - \frac{c(x_{source})}{sc(x_{source}) + 1} \right] \quad (29.5)$$

where `s` denotes saturation constant. To use the first of the above formulas, we set the value of the saturation coefficient:

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

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

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

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

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

The `lambda` value specified for each cell type can also be scaled using the `LogScaled` formula according to the concentration of the field at the center of mass of the chemotaxing cell  $c_{CM}$ ,

$$\Delta E_{chem} = -\frac{\lambda}{s + c_{CM}} (c(x_{destination}) - c(x_{source})) \quad (29.6)$$

The `LogScaled` formula is commonly used to mitigate excessive forces on cells in fields that vary over several orders of magnitude, and can be selected by setting the value of `s` with the attribute `LogScaledCoef` like as follows,

```
<ChemotaxisByType Type="Amoeba" Lambda="100" LogScaledCoef="1"/>
```

Sometimes it is desirable to have chemotaxis **at the interface between** only certain types of cells **and not between** other cell-type-pairs. In such a case we augment `ChemotaxisByType` element with the following attribute:

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

This will cause the change in chemotaxis energy to be non-zero only for those pixel copy attempts that happen between pixels belonging to `Amoeba` and `Medium`. Essentially, the amoeba will follow the highest concentration of the medium it can find.

### Note

The term `ChemotactTowards` means “chemotax at the interface between”

CC3D supports slight modifications of the above formulas in the `Chemotaxis` plugin where  $\Delta E$  is non-zero only if the cell located at  $x_{source}$  *after* the pixel copy is non-medium. To enable this mode users need to include

```
<Algorithm>"Regular"</Algorithm>
```

tag in the body of CC3DML plugin. Additionally, `Chemotaxis` plugin can apply the above formulas using the parameters and formulas of both the cell located at  $x_{source}$  (if any) *and* the cell located at  $x_{destination}$  (if any). To enable this mode users need to include

```
<Algorithm>"Reciprocated"</Algorithm>
```

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

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

The definitions of chemotaxis presented so far do not allow specification of chemotaxis parameters individually for each cell. To do this we will use Python scripting. We still need to specify in the CC3DML which fields are important from chemotaxis stand point. Only fields listed in the CC3DML will be used to calculate chemotaxis energy:

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

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

...
```

In the above excerpt from the CC3DML configuration file, we see that cells of type Macrophage will chemotax in response to ATTR gradient.

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

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

    def start(self):

        for cell in self.cellList:
            if cell.type == self.cell_type.Macrophage:
                cd = self.chemotaxisPlugin.addChemotaxisData(cell, "ATTR")
                cd.setLambda(20.0)
                cd.assignChemotactTowardsVectorTypes([self.cell_type.Medium, self.
→cell_type.Bacterium])
                break

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

In the `start` function for the first encountered cell of type Macrophage (`type==self.cell_type.Macrophage`), we insert a `ChemotaxisData` object (it determines chemotaxing properties) and initialize parameter to 20. We also initialize a vector of cell types towards which Macrophage cells will chemotax (it will chemotax towards Medium and Bacterium cells). Notice the `break` statement inside the `if` statement, inside the loop. It ensures that only first

encountered Macrophage cell will have chemotaxing properties altered.

In the step function we decrease lambda chemotaxis by 3 units every 100 MCS. In effect we turn a cell from chemotaxing up ATTR gradient to being chemorepelled.

In the above example we have more than one macrophage but only one of them has altered chemotaxing properties. The other macrophages have chemotaxing properties set in the CC3DML section. CompuCell3D first checks if local definitions of chemotaxis are available (i.e. for individual cells) and if so it uses those. Otherwise it will use definitions from the CC3DML.

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

```
def start(self):
    for cell in self.cellList:
        if cell.type == self.cell_type.Macrophage:
            cd = self.chemotaxisPlugin.addChemotaxisData(cell, "ATTR")
            cd.setLambda(20.0)
            cd.setSaturationCoef(200.0)
            cd.assignChemotactTowardsVectorTypes([self.cell_type.Medium, self.cell_type.
            ↵Bacterium])
            break
```

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

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

To activate the LogScaled formula for a cell, we would use:

```
cd.setLogScaledCoef(3.0)
```

### ⚠ Warning

When you use chemotaxis plugin you have to make sure that fields that you refer to and module that contains this fields are declared in the CC3DML file. Otherwise you will most likely cause either program crash (which is not as bad as it sounds) or unpredicted behavior (much worse scenario, although unlikely as we made sure that in the case of undefined symbols, CompuCell3D exits)

## 29.10 ExternalPotential Plugin

Chemotaxis plugin is used to cause directional cell movement in response to chemical gradient. Another way to achieve directional movement is to use ExternalPotential plugin. This plugin is responsible for imposing a directed pressure (or rather force) on cells. It is used to implement persistent motion of cells and its applications can be very diverse.

Example usage of this plugin looks as follows:

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

Lambda is a vector quantity and determines components of force along three axes. In this case we apply force along x pointing in the positive direction.

**Note**

Positive component of Lambda vector pushes cell in the negative direction and negative component pushes cell in the positive direction

We can also apply external potential to specific cell types:

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

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

We can also apply external potential to individual cells. In that case, in the CC3DML section we only need to specify:

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

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

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

Calculations done by ExternalPotential Plugin are by default based on direction of pixel copy (similarly as in chemotaxis plugin). One can, however, force CC3D to do calculations based on movement of center of mass of cell. To use algorithm based on center of mass movement we use the following CC3DML syntax:

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

**Note**

In the pixel-based algorithm the typical value of pixel displacement used in calculations is of the order of 1 (pixel) whereas typical displacement of center of mass of cell due to single pixel copy is of the order of 1/cell volume (pixels) – ~ 0.1 pixel. This implies that to achieve compatible behavior of cells when using center of mass algorithm we need to multiply Lambda's by appropriate factor, typically of the order of 10.

## 29.11 CenterOfMass Plugin

CenterOfMass plugin monitors changes in the lattice and updates centroids of the cell:

$$\begin{aligned}x_{CM}^{centroid} &= \sum_i x_i \\y_{CM}^{centroid} &= \sum_i y_i \\z_{CM}^{centroid} &= \sum_i z_i\end{aligned}$$

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

$$\begin{aligned}x_{CM} &= \frac{x_{CM}^{centroid}}{V} \\y_{CM} &= \frac{y_{CM}^{centroid}}{V} \\z_{CM} &= \frac{z_{CM}^{centroid}}{V}\end{aligned}$$

This plugin is aware of boundary conditions and centroids are calculated properly regardless which boundary conditions are used. The CC3DML syntax is very simple:

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

To access center of mass coordinates from Python we use the following syntax:

```
print('x-component of COM is:', cell.xCOM)
print ('y-component of COM is:', cell.yCOM)
print ('z-component of COM is:', cell.zCOM)
```

### Warning

Center of mass parameters in Python are read only. Any attempt to modify them will likely mess up the simulation.

## 29.12 Contact Plugin

### Relevant Examples:

- Cell Sorting

The contact plugin implements computations of adhesion energy between neighboring cells.

Together with volume constraint, contact energy is one of the most commonly used plugins. In essence, it describes how cells “stick” to each other.

Contact energy is contrived. It is merely a way to replicate the properties of a cell’s membrane, the bindings of the nano-structures on its surface, and its environment (the Medium).

Two cell types that have *high* contact energy will not “want to” adhere to each other. If possible, those cells may separate, effectively reducing the total energy to stay in that position. Conversely, *low* contact energy “encourages” cell types to bind. As contact energy is lowered, it also increases the surface of the contact.

Contact energy is constantly re-calculated each time a cell's surface changes.

We define contact energy as a matrix that compares each cell type against each other cell type. In CC3DML, try changing the contact energy between CellA and CellB to 2. Keep everything else the same, then click Play  to see the result.

```
<Plugin Name="Contact">
  <Plugin Name="Contact">
    <Energy Type1="CellA" Type2="CellA">10</Energy>
    <Energy Type1="CellB" Type2="CellB">10</Energy>
    <Energy Type1="CellA" Type2="CellB" >2</Energy> <!--Here!-->
    <Energy Type1="CellA" Type2="Medium">10</Energy>
    <Energy Type1="CellB" Type2="Medium">10</Energy>
    <Energy Type1="Medium" Type2="Medium">10</Energy> <!--Medium-Medium energy doesn't
  really matter-->
    <NeighborOrder>2</NeighborOrder>
  </Plugin>
```

This should stimulate cell mixing, the opposite of [cell Sorting](#).

## 29.13 What Does Energy Mean?

When tweaking your parameters, it's most important to pay attention to their **relative values**.

However, the actual value of each parameter—whether you set 10 or 1—affects the contact behaviors, too, because of *temperature*.

Each time a cell flexes its surface, a decision is made about which pixels in the lattice will be copied over to new lattice sites. We discussed this in the [Potts](#) page.

The probability of accepting a lattice site copy is:

$$P = e^{-\Delta H/T} \quad (29.7)$$

where  $\Delta H = H_f - H_i$ .  $H_f$  is the new energy after a potential lattice site copy.  $H_i$  is the current energy of the system at step  $i$ . The new energy,  $H_i$ , may be higher, lower, or the same after a site copy. Meanwhile,  $T$  is the temperature (or fluctuation amplitude *or* membrane fluctuation level *or* noise level). A high temperature lowers the probability that site copy attempts will be made.

## 29.14 Calculating Contact Energy

The explicit formula for contact energy is:

$$E_{contact} = \sum_{i,j,neighbors} J(\tau_{\sigma(i)}, \tau_{\sigma(j)}) (1 - \delta_{\sigma(i), \sigma(j)}) \quad (29.8)$$

where  $i$  and  $j$  label two neighboring lattice sites  $\sigma$ 's denote cell IDs, and each  $\tau$  denotes a cell type. Note that  $i$  and  $j$  are at separate coordinates; for example,  $i = (0, 4)$  and  $j = (1, 4)$ .

In the above formula, we need to differentiate between cell types and cell IDs. This formula shows that cell types and cell IDs **are not the same**. The Contact plugin in the .xml file, defines the energy per unit area of contact between cells of different types ( $J(\tau_{\sigma(i)}, \tau_{\sigma(j)})$ ) and the interaction range NeighborOrder of the contact:

```
<Plugin Name="Contact">
  <Energy Type1="Foam" Type2="Foam">3</Energy>
  <Energy Type1="Medium" Type2="Medium">0</Energy>
```

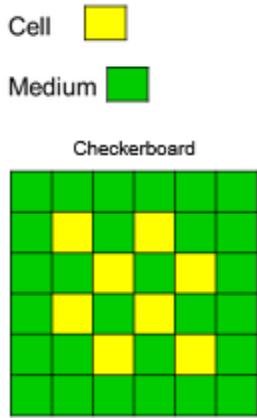
(continues on next page)

(continued from previous page)

```
<Energy Type1="Medium" Type2="Foam">0</Energy>
<NeighborOrder>2</NeighborOrder>
</Plugin>
```

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

Suppose we have **8 cells** on the checkboard and **4 sides each**. If the cells are in a checkerboard pattern, they have the most possible contact energy with the Medium.

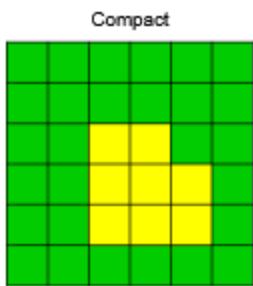


So, the total contact energy is:

$$H_{contact} = 4 \times 8 \times J_{Cell-to-Medium} \quad (29.9)$$

where  $J_{Cell-to-Medium}$  denotes the contact energy between the base cell type, Cell, and the Medium.

How would you calculate the contact energy if all the cells were touching the medium as little as possible?



$$H_{contact} = 12 \times J_{Cell-to-Medium} + 10 \times J_{Cell-to-Cell} \quad (29.10)$$

Again, count the number of contact surfaces and multiply them by the respective contact energies for those cell types.

## 29.15 AdhesionFlex Plugin

AdhesionFlex offers a very flexible way to define adhesion between cells. It allows setting individual adhesivity properties for each cell. Users can use either CC3DML syntax or Python scripting to initialize adhesion molecule

density for each cell. In addition, Medium can also carry its own adhesion molecules. We use the following formula to calculate adhesion energy in AdhesionFlex plugin:

$$E_{adhesion} = \sum_{i,j,neighbors} \left( - \sum_{m,n} k_{mn} F(N_m(i), N_n(j)) \right) (1 - \delta_{\sigma(i), \sigma(j)}) \quad (29.11)$$

where indexes i, j label pixels,  $-\sum_{m,n} k_{mn} F(N_m(i), N_n(j))$  denotes contact energy between cell types  $\sigma(i)$  and  $\sigma(j)$  and indexes m, n label adhesion molecules in cells composed of pixels i and j respectively. F denotes user-defined function of  $N_m$  and  $N_n$ . Although this may look a bit complex, the basic idea is simple: each cell has certain number of adhesion molecules on its surface. When cells touch each other the resultant energy is simply a “product of interactions” of adhesion molecules from one cell with adhesion molecules from another cell. The CC3DML syntax for this plugin is given below:

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

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

    <NeighborOrder>2</NeighborOrder>
</Plugin>
```

$k_{mn}$  matrix is specified within the AdhesionInteractionMatrix tag – the elements are listed using BindingParameter tags. The AdhesionMoleculeDensity tag specifies initial concentration of adhesion molecules. Even if you are going to modify those from Python you are still required to specify the names of adhesion molecules and associate them with appropriate cell types. Failure to do so may result in simulation crash or undefined behaviors. The user-defined function \*F\* is specified using Formula tag where the arguments of the function are called Molecule1 and Molecule2 . The syntax has to follow syntax of the muParser - <https://beltoforion.de/en/muparser/features.php#idDef1>

### Note

Using more complex formulas with muParser requires special CDATA syntax. Please check [XML Expression Evaluator - muParser](#) for more details.

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

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

allows to access adhesion molecule concentration using its name (as given in the CC3DML above using AdhesionMoleculeDensity tag).

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

allows to access adhesion molecule concentration using its index in the adhesion molecule density vector. The order of the adhesion molecule densities in the vector is the same as the order in which they were declared in the CC3DML above - AdhesionMoleculeDensity tags.

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

allows access to entire adhesion molecule density vector. Each of these functions has its corresponding function which operates on Medium . In this case we do not give cell as first argument:

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

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

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

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

```
self.adhesionFlexPlugin.setAdhesionMoleculeDensity(cell, 'NCad', 0.1)
```

```
self.adhesionFlexPlugin.setAdhesionMoleculeDensityByIndex(cell, 1, 1.02)
```

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

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

#### Note

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

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

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

#### Note

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

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

```
self.adhesionFlexPlugin.setMediumAdhesionMoleculeDensity("NCam", 2.8)  
self.adhesionFlexPlugin.setMediumAdhesionMoleculeDensityByIndex(2, 16.8)  
self.adhesionFlexPlugin.setMediumAdhesionMoleculeDensityVector([1.4, 3.1, 18.1])  
self.adhesionFlexPlugin.assignNewMediumAdhesionMoleculeDensityVector([1.4, 3.1, 18.1])
```

## 29.16 Compartmentalized Cells | ContactInternal Plugin

Related: [Dividing Clusters \(aka compartmental cells\)](#) and [Python examples](#)

**Contact Plugin:** controls how easily cells adhere to one another or themselves.

**ContactInternal Plugin:** controls how easily sub-cells within the same compartment adhere to each other.

In both cases, low energy values “encourage” pixels to stick together whereas high values cause them to spread apart. The ContactInternal Plugin can help to control the shape and arrangement of a compartmentalized cell. Usually, the standard Contact Plugin should still be included to control how clusters interact with one another.

---

Calculating contact energies between compartmentalized cells is analogous to the non-compartmentalized case. The energy expression takes the following form:

$$E_{adhesion} = \sum_{i,j,neighbors} J(\sigma(\mu_i, \nu_i), \sigma(\mu_j, \nu_j)) \quad (29.12)$$

where  $i$  and  $j$  denote pixels,  $(\mu, \nu)$  denotes a cell type of a cell with cluster id  $\mu$  and cell id  $\nu$ . In compartmental cell models, a cell is a collection of subcells. Each subcell has a unique id (cell id). In addition to that, each subcell will have an additional attribute, a cluster id ( $\mu$ ) that determines to which cluster of subcells a given subcell belongs. Think of a cluster as a cell with a non-homogenous cytoskeleton. So a cluster corresponds to a biological cell and subcells (or compartments) represent parts of a cell, e.g., a nucleus. The core idea here is to have different contact energies between subcells belonging to the same cluster and different energies for cells belonging to different clusters. Technically subcells of a cluster are “regular” CompuCell3D cells. By giving them an extra attribute, cluster id ( $\mu$ ), we can introduce the concept of compartmental cells. In our convention,  $(0,0)$  denotes medium.

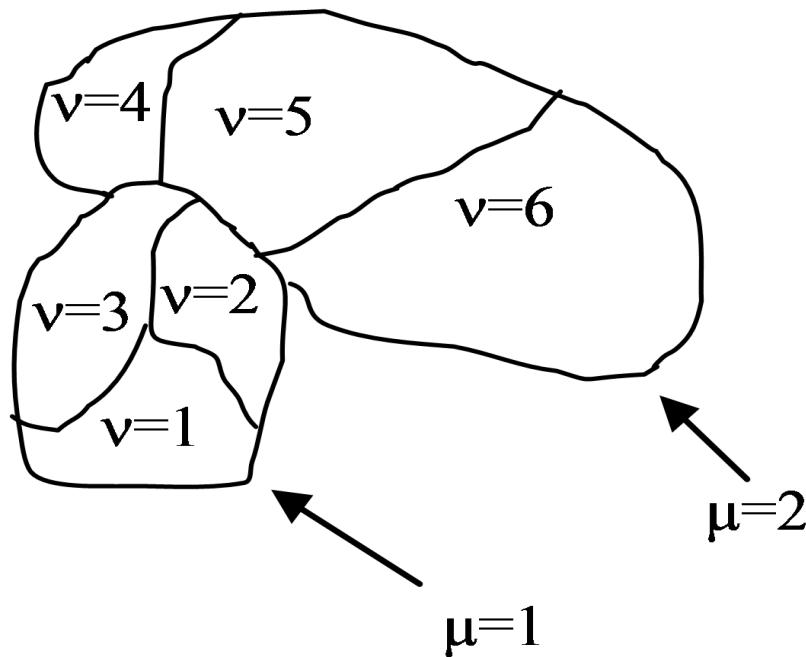


Figure 2. Two compartmentalized cells (cluster id  $\mu = 1$  and cluster id  $\mu = 2$ ). Compartmentalized cell  $\mu = 1$  consists of subcells with cell id  $\nu = 1, 2, 3$  and compartmentalized cell  $\mu = 2$  consists of subcells with cell id  $\nu = 4, 5, 6$

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

Indeed: 
$$\begin{cases} J(\sigma(\mu_i, \nu_i), \sigma(\mu_j, \nu_j)) = J^{external}(\sigma(\mu_i, \nu_i), \sigma(\mu_j, \nu_j)) & \text{if } \mu_i \neq \mu_j \\ J(\sigma(\mu_i, \nu_i), \sigma(\mu_j, \nu_j)) = J^{internal}(\sigma(\mu_i, \nu_i), \sigma(\mu_j, \nu_j)) & \text{if } \mu_i = \mu_j \end{cases}$$
 As we can see above, there are two hierarchies of contact energies – external and internal. To describe adhesive interactions between different compartmentalized cells, we use two plugins: Contact and ContactInternal. Contact plugin calculates energy between two cells belonging to different clusters and ContactInternal calculates energies between cells belonging to the same cluster. An example syntax is shown below

```
<Plugin Name="Contact">

<Energy Type1="Base" Type2="Base">0</Energy>
<Energy Type1="Top" Type2="Base">25</Energy>
<Energy Type1="Center" Type2="Base">30</Energy>
<Energy Type1="Bottom" Type2="Base">-2</Energy>
<Energy Type1="Side1" Type2="Base">25</Energy>
<Energy Type1="Side2" Type2="Base">25</Energy>
<Energy Type1="Medium" Type2="Base">0</Energy>

<Energy Type1="Medium" Type2="Medium">0</Energy>
<Energy Type1="Top" Type2="Medium">30</Energy>
<Energy Type1="Bottom" Type2="Medium">20</Energy>
<Energy Type1="Side1" Type2="Medium">30</Energy>
```

(continues on next page)

(continued from previous page)

```

<Energy Type1="Side2" Type2="Medium">30</Energy>
<Energy Type1="Center" Type2="Medium">45</Energy>

<Energy Type1="Top" Type2="Top">2</Energy>
<Energy Type1="Top" Type2="Bottom">100</Energy>
<Energy Type1="Top" Type2="Side1">25</Energy>
<Energy Type1="Top" Type2="Side2">25</Energy>
<Energy Type1="Top" Type2="Center">35</Energy>

<Energy Type1="Bottom" Type2="Bottom">10</Energy>
<Energy Type1="Bottom" Type2="Side1">25</Energy>
<Energy Type1="Bottom" Type2="Side2">25</Energy>
<Energy Type1="Bottom" Type2="Center">35</Energy>

<Energy Type1="Side1" Type2="Side1">25</Energy>
<Energy Type1="Side1" Type2="Center">25</Energy>
<Energy Type1="Side2" Type2="Side2">25</Energy>
<Energy Type1="Side2" Type2="Center">25</Energy>
<Energy Type1="Side1" Type2="Side2">15</Energy>

<Energy Type1="Center" Type2="Center">20</Energy>

<NeighborOrder>2</NeighborOrder>
</Plugin>

```

and

```

<Plugin Name="ContactInternal">

<Energy Type1="Base" Type2="Base">0</Energy>
<Energy Type1="Base" Type2="Bottom">0</Energy>
<Energy Type1="Base" Type2="Side1">0</Energy>
<Energy Type1="Base" Type2="Side2">0</Energy>
<Energy Type1="Base" Type2="Center">0</Energy>

<Energy Type1="Top" Type2="Top">4</Energy>
<Energy Type1="Top" Type2="Bottom">25</Energy>
<Energy Type1="Top" Type2="Side1">22</Energy>
<Energy Type1="Top" Type2="Side2">22</Energy>
<Energy Type1="Top" Type2="Center">15</Energy>

<Energy Type1="Bottom" Type2="Bottom">4</Energy>
<Energy Type1="Bottom" Type2="Side1">15</Energy>
<Energy Type1="Bottom" Type2="Side2">15</Energy>
<Energy Type1="Bottom" Type2="Center">10</Energy>

<Energy Type1="Side1" Type2="Side1">11</Energy>
<Energy Type1="Side2" Type2="Side2">11</Energy>
<Energy Type1="Side1" Type2="Side2">11</Energy>

<Energy Type1="Side2" Type2="Center">10</Energy>
<Energy Type1="Side1" Type2="Center">10</Energy>

```

(continues on next page)

(continued from previous page)

```
<Energy Type1="Center" Type2="Center">2</Energy>
<NeighborOrder>2</NeighborOrder>
</Plugin>
```

Depending on whether or not the pixels for which we calculate contact energies belong to the same cluster, we will use internal or external contact energies, respectively.

## 29.17 LengthConstraint Plugin

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

The syntax is as follows:

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

LambdaLength determines the degree of cell length oscillation around TargetLength parameter. The higher LambdaLength the less freedom a cell will have to deviate from TargetLength.

In the 3D case, we use the following syntax:

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

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

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

To control length constraint individually for each cell we may use Python scripting to assign LambdaLength, TargetLength and in 3D MinorTargetLength. In Python steppable we typically would write the following code:

```
self.lengthConstraintPlugin.setLengthConstraintData(cell, 10, 20)
```

which sets length constraint for cell by setting LambdaLength=10 and TargetLength=20. In 3D we may specify ``MinorTargetLength`` (we set it to 5) by adding 4<sup>th</sup> parameter to the above call:

```
self.lengthConstraintPlugin.setLengthConstraintData(cell, 10, 20, 5)
```

### Note

If we use CC3DML specification of length constraint for certain cell types and in Python we set this constraint individually for a single cell then the local definition of the constraint has priority over definitions for the cell type.

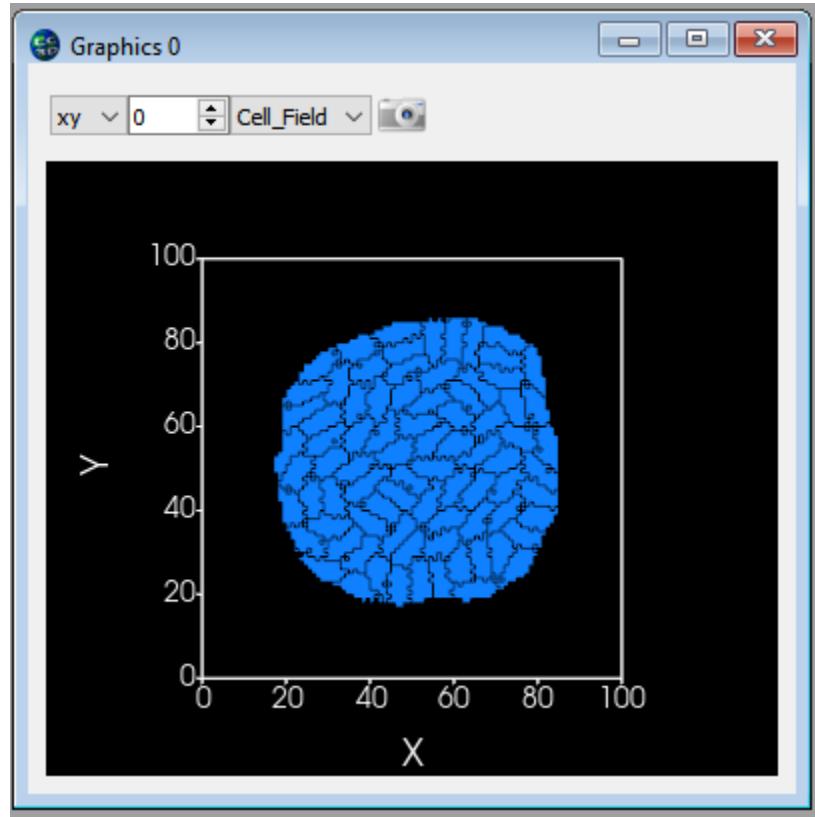
If, in the simulation, we will be setting length constraints for only a few individual cells then it is best to manipulate the constraint parameters from the Python script. In this case, in the CC3DML we only have to declare that we will use a length constraint plugin and we may skip the definition by-type definitions:

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

### Note

**IMPORTANT:** When using target length, plugins it is important to use the connectivity constraint. This constraint will check if a given pixel copy breaks cell connectivity. If it does, the plugin will add a large energy penalty (defined by a user) to change energy effectively, prohibiting such pixel copy. In the case of 2D on square lattice checking cell connectivity can be done locally and thus is very fast. In 3D the connectivity algorithm is a bit slower but its performance is acceptable. Therefore if you need large cell elongations you should always use connectivity in order to prevent cell fragmentation

Here's how a cluster of elongated cells might look:



## 29.18 Connectivity Plugins

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

The syntax of the plugin is straightforward:

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

#### Note

The value of `Penalty` is irrelevant because all `Connectivity` plugins have a special status. Namely, CC3D will call connectivity plugin to check if the a given pixel copy would lead to cell fragmentation and if so it will reject the pixel copy without computing energy-based pixel-copy acceptance function. thus the only thing that matters here is that penalty parameter is a positive number. Any number

As we mentioned, earlier 2D connectivity algorithm is particularly fast but works only on Cartesian lattice nad in 2D only. If you are on hex lattice or are working with 3 dimensions You should use `ConnectivityGlobal` plugin and there specify so called `FastAlgorithm` to get decent performance.

For example (see *Demos/PluginDemos/connectivity\_global\_fast*):

```
<Plugin Name="ConnectivityGlobal">
    <FastAlgorithm/>
    <ConnectivityOn Type="NonCondensing"/>
    <ConnectivityOn Type="Condensing"/>
</Plugin>
```

will enforce connectivity of cells of type Condensing and NonCondensing and will use “fast algorithm”.

If you want to enforce connectivity for individual cells cells (see *Demos/PluginDemos/connectivity\_elongation\_fast*) you would use the following CC3DML code:

```
<Plugin Name="ConnectivityGlobal">
    <FastAlgorithm/>
</Plugin>
```

and couple it with the following Python steppable:

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

    def start(self):
        for cell in self.cellList:
            if cell.type==1:
                cell.connectivityOn = True

            elif cell.type==2:
                cell.connectivityOn = True
```

Below we describe a slower version of `ConnectivityGlobal` plugin that is still supported but has much slower performance and for that reason we encourage you to try faster implementation described above

#### Note

**DEPRECATED**

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

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

As we mentioned before the actual value of `Penalty` parameter does not matter as long it is a positive number

In certain types of simulation it may happen that at some point cells change cell types. If a cell that was not subject to connectivity constraint, changes type to the cell that is constrained by global connectivity and this cell is fragmented before type change this situation normally would result in simulation freeze. However, CompuCell3D, first before applying constraint it will check if the cell is fragmented. If it is, there is no constraint. Global connectivity constraint is only applied when cell is non-fragmented.

Quite often in the simulation we don't need to impose connectivity constraint on all cells or on all cells of given type. Usually only select cell types or select cells are elongated and therefore need connectivity constraint. In such a case we simply declare `ConnectivityGlobal` with no further specifications taking place in CC3DML. The actual connectivity assignments to particular cells take place in Python

In CC3DML we only declare:

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

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

```
class ElongationFlexSteppable(SteppableBasePy):
    def __init__(self,_simulator,_frequency=10):
        SteppableBasePy.__init__(self, _simulator, _frequency)
        # self.lengthConstraintPlugin=CompuCell.getLengthConstraintPlugin()

    def start(self):
        pass

    def step(self,mcs):
        for cell in self.cellList:
            if cell.type==1:
                self.lengthConstraintPlugin.setLengthConstraintData(cell,20,
                ↵20) # cell , lambdaLength, targetLength
                self.connectivityGlobalPlugin.setConnectivityStrength(cell,
                ↵10000000) #cell, strength

            elif cell.type==2:
                self.lengthConstraintPlugin.setLengthConstraintData(cell,20,
                ↵30) # cell , lambdaLength, targetLength
                self.connectivityGlobalPlugin.setConnectivityStrength(cell,
                ↵10000000) #cell, strength
```

See also example in *Demos/PluginDemos/elongationFlexTest*.

If you are in 2D and on Cartesian lattice you may instead use `ConnectivityLocalFlex`

In this case

In CC3DML we only declare:

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

and in Python:

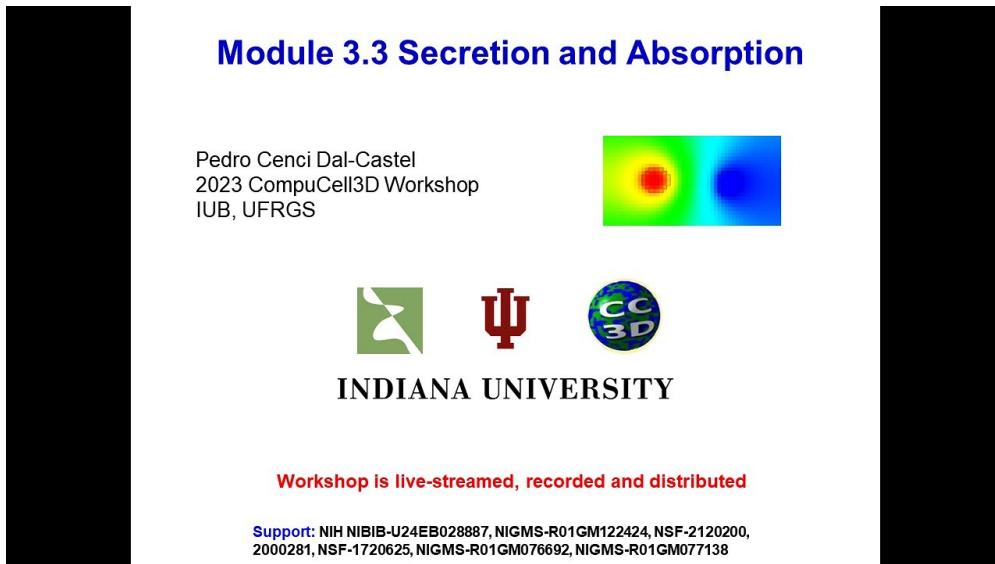
```
self.connectivityLocalFlexPlugin.setConnectivityStrength(cell, 20.7)
self.connectivityLocalFlexPlugin.getConnectivityStrength(cell)
```

## 29.19 Secretion / SecretionLocalFlex Plugin

### Related:

- Secretion Reference
- Field Secretion
- Secretion (legacy version for pre-v3.5.0)

Download the sample code here, then watch the video from the latest workshop to follow along:



Secretion “by cell type” can and should be handled by the appropriate PDE solver. To implement secretion from individual cells using Python, we first add the secretion plugin in CC3DML:

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

or as:

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

The inclusion of the above code in the CC3DML will allow users to implement secretion for individual cells from Python.

**Note**

Secretion for individual cells invoked via Python will be called only once per MCS.

**Warning**

Although the secretion plugin can be used to implement secretion by cell type, **we strongly advise against doing so**. Defining secretion by cell type in the Secretion plugin will lead to performance degradation on multi-core machines. Please see the section below for more information if you are still interested in using secretion by cell type inside the Secretion plugin.

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

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

    def step(self, mcs):
        attrSecretor = self.get_field_secretor("ATTR")
        for cell in self.cellList:
            if cell.type == 3:
                attrSecretor.secreteInsideCell(cell, 300)
                attrSecretor.secreteInsideCellAtBoundary(cell, 300)
                attrSecretor.secreteOutsideCellAtBoundary(cell, 500)
                attrSecretor.secreteInsideCellAtCOM(cell, 300)
            elif cell.type == 2:
                attrSecretor.secreteInsideCellConstantConcentration(cell, 300)
```

**Note**

Instead of using SteppableBasePy class we are using SecretionBasePy class. This ensures that the secretion plugin will be performed before diffusion by calling the Python secretion steppable *before* each Monte Carlo Step.

There is no magic to SecretionBasePy - if you still want to use SteppableBasePy as a base class for secretion do so, but remember that you need to set flag:

```
self.runBeforeMCS=1
```

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

```
class SecretionSteppable(SteppableBasePy):
    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator, _frequency)
        self.runBeforeMCS=1
    def step(self,mcs):
        attrSecretor=self.get_field_secretor("ATTR")
        for cell in self.cellList:
            if cell.type==3:
                attrSecretor.secreteInsideCell(cell,300)
```

(continues on next page)

(continued from previous page)

```

attrSecretor.secreteInsideCellAtBoundary(cell, 300)
attrSecretor.secreteOutsideCellAtBoundary(cell, 500)
attrSecretor.secreteOutsideCellAtBoundaryOnContactWith(cell, 500, [2, 3])
attrSecretor.secreteInsideCellAtCOM(cell, 300)
attrSecretor.uptakeInsideCellAtCOM(cell, 300, 0.2)
elif cell.type==2:
    attrSecretor.secreteInsideCellConstantConcentration(cell, 300)

```

The secretion of individual cells is handled through `FieldSecretor` objects. `FieldSecretor` concept is quite convenient because the amount of Python coding is quite small. To secrete a chemical from a cell, we first create a field secretor object:

```
attrSecretor = self.get_field_secretor("ATTR")
```

which allows us to manipulate how much which cells secrete into the ``ATTR`` field.

Then, we pick a cell, and using this field secretor, we simulate secretion of chemical ATTR by a cell:

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

Secretion functions use the following syntax:

```

secrete*(cell, amount)
#or...
secrete*(cell, amount, list_of_cell_types)

```

#### Note

The `list_of_cell_types` is used only for functions which implement such functionality *i.e.* `secreteInsideCellAtBoundaryOnContactWith` and `secreteOutsideCellAtBoundaryOnContactWith`

Uptake functions use the following syntax:

```

uptake*(cell, max_amount, relative_uptake, list_of_cell_types)
#or...
uptake*(cell, max_amount, relative_uptake)

```

#### Note

The `list_of_cell_types` is used only for functions which implement such functionality *i.e.* `uptakeInsideCellAtBoundaryOnContactWith` and `uptakeOutsideCellAtBoundaryOnContactWith`

#### Note

**Important:** The uptake works as follows: when available concentration is greater than `max_amount`, then `max_amount` is subtracted from `current_concentration`, otherwise we subtract `relative_uptake*current_concentration`.

As you may infer from above, the modes 1-5 require tracking of pixels belonging to cell and pixels belonging to cell boundary. If you are not using those secretion modes you may disable pixel tracking by including:

```
<DisablePixelTracker/>
```

or

```
<DisableBoundaryPixelTracker/>
```

as shown in the example below:

```
<Plugin Name="Secretion">

    <DisablePixelTracker/>
    <DisableBoundaryPixelTracker/>

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

#### Note

Make sure that fields into which you will be secreting chemicals exist. They are usually fields defined in PDE solvers. When using secretion plugin you do not need to specify SecretionData section for the PDE solvers.

When implementing e.g. secretion inside cell when the cell is in contact with other cell we use neighbor tracker and a short script in the spirit of the below snippet:

```
for cell in self.cellList:
    attrSecretor = self.get_field_secretor("ATTR")
    for neighbor, commonSurfaceArea in self.getCellNeighborDataList(cell):
        if neighbor.type in [self.WALL]:
            attrSecretor.secreteInsideCell(cell, 300)
```

## 29.20 PDESolverCaller Plugin

#### Warning

In most cases you can specify extra calls to PDE solvers in the solver itself. Thus this plugin is being deprecated. We mention it here for backward compatibility reasons as some of the older CC3D simulations may still be using this plugin.

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

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

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

## 29.21 FocalPointPlasticity Plugin

`FocalPointPlasticity` puts constraints on the distance between cells' center of masses. A key feature of this plugin is that the list of “focal point plasticity neighbors” can change as the simulation evolves and user has to specifies the maximum number of “focal point plasticity neighbors” a given cell can have. Let's look at relatively simple CC3DML syntax of `FocalPointPlasticityPlugin` (see `Demos/PluginDemos/FocalPointPlasticity/FocalPointPlasticity` example and we will show more complex examples later):

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

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

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

Once cells are linked the energy calculation is carried out according to the formula:

$$E = \sum_{i,j, \text{cell neighbors}} \lambda_{ij} (l_{ij} - L_{ij})^2 \quad (29.13)$$

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

$\lambda_{ij}$  and  $L_{ij}$  between different cell types are specified using `Lambda` and `TargetDistance` tags. The `MaxDistance` determines the distance between cells' center of masses past which the link between those cells break. When the link

breaks, then in order for the two cells to reconnect they would need to come in contact again. However it is usually more likely that there will be other cells in the vicinity of separated cells so it is more likely to establish new link than restore broken one.

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

```
<Plugin Name="FocalPointPlasticity">

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

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

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

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

<NeighborOrder>1</NeighborOrder>

</Plugin>
```

The total number of links a given cell can form is computed as a sum of <MaxNumberOfJunctions> values coming from <Parameters> or <InternalParameters> sections where a given cell type appears. In the example above Center can form maximum of 2 internal links (between compartments)

CompuCell3D gives you ability to override this default algorithm by adding additional tag that control the total number

of links:

```
<Plugin Name="FocalPointPlasticity">

    <MaxTotalNumberOfLinks CellType="Center">1</MaxTotalNumberOfLinks>
    <InternalMaxTotalNumberOfLinks CellType="Center">1</InternalMaxTotalNumberOfLinks>

    <Parameters Type1="Top" Type2="Top">
        <Lambda>10.0</Lambda>
    ...

```

In this example we are limiting the maximum number of links Center cells can form with Center cells of another cluster to 1 `<MaxTotalNumberOfLinks CellType="Center">1</MaxTotalNumberOfLinks>` and do the same for number of links the Center cells can form with cells that are members of the same cluster: `<InternalMaxTotalNumberOfLinks CellType="Center">1</InternalMaxTotalNumberOfLinks>`

where we can override this default algorithm. This feature can be useful when working with “elongated” compartmental cells where you do not want Center cells to form more than two links

We can also specify link constituent law and change it to different form that “spring relation”. To do this we use the following syntax inside FocalPointPlasticity CC3DML plugin:

```
<LinkConstituentLaw>
    <!--The following variables are defined by default: Lambda,Length,TargetLength-->

    <Variable Name='LambdaExtra' Value='1.0' />
    <Expression>LambdaExtra*Lambda*(Length-TargetLength)^2</Expression>

</LinkConstituentLaw>
```

By default CC3D defines 3 variables (Lambda, Length, TargetLength) which correspond to  $\lambda_{ij}$ ,  $l_{ij}$  and  $L_{ij}$  from the formula above. We can also define extra variables in the CC3DML (e.g. LambdaExtra). The actual link constituent law obeys `muParser` syntax convention. Once link constituent law is defined it is applied to all focal point plasticity links. The example demonstrating the use of custom link constituent law can be found in *Demos/PluginDemos/FocalPointPlasticityCustom*.

Sometimes it is necessary to modify link parameters individually for every cell pair. In this case we would manipulate FocalPointPlasticity links using Python scripting. Example *Demos/PluginDemos/FocalPointPlasticity/FocalPointPlasticityCompartments* demonstrates exactly this situation. You still need to include CC3DML section as the one shown above for compartmental cells, because we need to tell CC3D how to link cells. The only notable difference is that in the CC3DML we have to include `<Local/>` tag to signal that we will set link parameters (Lambda, TargetDistance, MaxDistance) individually for each cell pair:

```
<Plugin Name="FocalPointPlasticity">
    <Local/>
    <Parameters Type1="Top" Type2="Top">
        <Lambda>10.0</Lambda>
        <ActivationEnergy>-50.0</ActivationEnergy>
        <TargetDistance>7</TargetDistance>
        <MaxDistance>20.0</MaxDistance>
        <MaxNumberOfJunctions NeighborOrder="1">1</MaxNumberOfJunctions>
    </Parameters>
    ...
</Plugin>
```

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

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

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

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

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

```
setInternalFocalPointPlasticityParameters(cell1, cell2, lambda, targetDistance, ↪
    ↪maxDistance)
```

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

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

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

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

### Note

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

```
deleteFocalPointPlasticityLink(cell1, cell2)

deleteInternalFocalPointPlasticityLink(cell1, cell2)

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

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

## 29.22 Working on the Basis of Links

### Note

All functionality described in this section is relevant for CC3D versions 4.2.4+.

CC3D performs all link calculations on the basis of link objects. That is, every link as described so far is an object with properties and functions that, much like CellG objects, can be created, destroyed and manipulated. As such, CC3DML specification tells CC3D to simulate links and what types of links should be automatically created, but links can also be individually accessed, manipulated, created and destroyed in Python. FocalPointPlasticity Plugin always uses the basis of links for link calculations, and so the <Local/> tag in CC3DML FocalPointPlasticity Plugin specification is no longer necessary.

CC3D describes three types of links

1. FocalPointPlasticityLink: a link between two cells
2. FocalPointPlasticityInternalLink: a link between two cells of the same cluster
3. FocalPointPlasticityAnchor: a link between a cell and a point

FocalPointPlasticityLink objects are automatically created from the CC3DML FocalPointPlasticity Plugin specification in the tag Parameters, FocalPointPlasticityInternalLink objects are automatically created from CC3DML specification in the tag InternalParameters, and FocalPointPlasticityAnchor objects are only created in Python. *CompuCell3D/core/Demos/PluginDemos/FocalPointPlasticityLinks* demonstrates basic usage of FocalPointPlasticityLink, the pattern of which is mostly the same for FocalPointPlasticityInternalLink and FocalPointPlasticityAnchor except for naming conventions of certain properties, functions and objects.

CC3D adopts the convention that for every link with a cell pair (*i.e.*, FocalPointPlasticityLink and FocalPointPlasticityInternalLink), one cell is the *initiator* cell (*i.e.*, the cell that initiated the link), and the other cell is the *initiated* cell. Using this convention, every link has the following API for manipulating link properties,

```
# -----
# / Properties /
# -----
# Link length; automatically updated by CC3D
length
# Link tension = 2 * lambda * (distance - target_distance); automatically updated by CC3D
tension
# A general python dictionary
dict
# SBML solvers (as on CellG)
sbml
# -----
# / Methods /
# -----
# Given one cell, returns the other cell of a link
getOtherCell(self, _cell: CellG) -> CellG
# Returns True if the cell is the initiator
isInitiator(self, _cell: CellG) -> bool
# Get lambda distance
getLambdaDistance(self) -> float
# Set lambda distance
setLambdaDistance(self, _lm: float) -> None
# Get target distance
```

(continues on next page)

(continued from previous page)

```

getTargetDistance(self) -> float
# Set target distance
setTargetDistance(self, _td: float) -> None
# Get maximum distance
getMaxDistance(self) -> float
# Set maximum distance
setMaxDistance(self, _md: float) -> None
# Get maximum number of junctions
getMaxNumberOfJunctions(self) -> int
# Set maximum number of junctions
setMaxNumberOfJunctions(self, _mnj: int) -> None
# Get activation energy
getActivationEnergy(self) -> float
# Set activation energy
setActivationEnergy(self, _ae: float) -> None
# Get neighbor order
getNeighborOrder(self) -> int
# Set neighbor order
setNeighborOrder(self, _no: int) -> None
# Get initialization step; automatically recorded at instantiation
getInitMCS(self) -> int

```

So, for example, the value of  $\lambda_{ij}$  for a link can be retrieved with `link.getLambdaDistance()`, and can be set with `link.setLambdaDistance(lambda_ij)` for some float-valued variable `lambda_ij`. Links automatically created by CC3D according to CC3DML specification are initialized with properties accordingly. Additionally, `FocalPointPlasticityLink` and `FocalPointPlasticityInternalLink` objects have the property `cellPair`, which contains, in order, the initiator and initiated cells of the link, while each `FocalPointPlasticityAnchor` has the property `cell` (*i.e.*, the linked cell) and additional methods related to its anchor point,

Steppables have built-in method for creating and destroying each type of link,

```

# Create a link between two cells
new_fpp_link(self, initiator: CellG, initiated: CellG, lambda_distance: float, target_
->distance: float,
             max_distance: float) -> FocalPointPlasticityLink
# Create an internal link between two cells of a cluster
new_fpp_internal_link(self, initiator: CellG, initiated: CellG, lambda_distance: float,
                      target_distance: float, max_distance: float) ->_
->FocalPointPlasticityInternalLink
# Create an anchor
# Anchor point can be specified by individual components x, y and z, or by Point3D pt
new_fpp_anchor(self, cell: CellG, lambda_distance: float, target_distance: float,
                max_distance: float, x: float = 0.0, y: float = 0.0, z: float = 0.0,
                pt: Point3D = None) -> FocalPointPlasticityAnchor
# Destroy a link type FocalPointPlasticityLink, FocalPointPlasticityInternalLink or_
->FocalPointPlasticityAnchor
delete_fpp_link(self, _link) -> None
# Destroy all links attached to a cell by link type
# links, internal_links and anchors selects which type, or all types if not specified
remove_all_cell_fpp_links(self, _cell: CellG, links: bool = False, internal_links: bool_-
->= False,
                           anchors: bool = False) -> None

```

Steppables also have built-in methods for retrieving information about links in simulation, by cell, and by cell pair. These methods are as follows,

```
# Get number of links
get_number_of_fpp_links(self) -> int
# Get number of internal links
get_number_of_fpp_internal_links(self) -> int
# Get number of anchors
get_number_of_fpp_anchors(self) -> int
# Get link associated with two cells
get_fpp_link_by_cells(self, cell1: CellG, cell2: CellG) -> FocalPointPlasticityLink
# Get internal link associated with two cells
get_fpp_internal_link_by_cells(self, cell1: CellG, cell2: CellG) ->
    ↪FocalPointPlasticityInternalLink
# Get anchor assicated with a cell and anchor id
get_fpp_anchor_by_cell_and_id(self, cell: CompuCell.CellG, anchor_id: int) ->
    ↪FocalPointPlasticityAnchor
# Get list of links by cell
get_fpp_links_by_cell(self, _cell: CellG) -> FPPLinkList
# Get list of internal links by cell
get_fpp_internal_links_by_cell(self, _cell: CellG) -> FPPInternalLinkList
# Get list of anchors by cell
get_fpp_anchors_by_cell(self, _cell: CellG) -> FPPAnchorList
# Get list of cells linked to a cell
get_fpp_linked_cells(self, _cell: CompuCell.CellG) -> mvectorCellGPtr
# Get list of cells internally linked to a cell
get_fpp_internal_linked_cells(self, _cell: CellG) -> mvectorCellGPtr
# Get number of link junctions by type for a cell
get_number_of_fpp_junctions_by_type(self, _cell: CellG, _type: int) -> int
# Get number of internal link junctions by type for a cell
get_number_of_fpp_internal_junctions_by_type(self, _cell: CompuCell.CellG, _type: int) ->
    ↪ int
```

## 29.23 Curvature Plugin

The Curvature plugin implements energy term for compartmental cells. The mathematics and mechanics between the plugin have been described in “A New Mechanism for Collective Migration in *Myxococcus xanthus*”, J. Starruß, Th. Bley, L. Søgaard-Andersen and A. Deutsch, *Journal of Statistical Physics*, DOI: **10.1007/s10955-007-9298-9**, (2007). For a “long” compartmental cell composed of many subcells (think of a snake-like elongated sequence of compartments) it imposes a constraint on curvature of cells. The syntax is slightly complex:

```
<Plugin Name="Curvature">

<InternalParameters Type1="Top" Type2="Center">
    <Lambda>100.0</Lambda>
    <ActivationEnergy>-50.0</ActivationEnergy>
</InternalParameters>

<InternalParameters Type1="Center" Type2="Center">
    <Lambda>100.0</Lambda>
    <ActivationEnergy>-50.0</ActivationEnergy>
</InternalParameters>
```

(continues on next page)

(continued from previous page)

```

<InternalParameters Type1="Bottom" Type2="Center">
    <Lambda>100.0</Lambda>
    <ActivationEnergy>-50.0</ActivationEnergy>
</InternalParameters>

<InternalTypeSpecificParameters>
    <Parameters TypeName="Top" MaxNumberOfJunctions="1" NeighborOrder="1"/>
    <Parameters TypeName="Center" MaxNumberOfJunctions="2" NeighborOrder="1"/>
    <Parameters TypeName="Bottom" MaxNumberOfJunctions="1" NeighborOrder="1"/>
</InternalTypeSpecificParameters>

</Plugin>

```

The InternalTypeSpecificParameter informs Curvature Plugin how many neighbors a cell of given type will have. In the case of “snake-shaped” cell the numbers that make sense are 1 and 2. The middle segment will have 2 connections and head and tail segments will have only one connection with neighboring segments (subcells). The connections are established dynamically. The way it happens is that during simulation CC3D constantly monitors pixel copies and during pixel copy between two neighboring cells/subcells it checks if those cells are already “connected” using curvature constraint. If they are not, CC3D will check if connection can be made (e.g. Center cells can have up to two connections and Top and Bottom only one connection). Usually establishing connections takes place at the beginning of the simulation and often happens within first Monte Carlo Step (depending on actual initial configuration, of course, but if segments touch each other connections are established almost immediately). The ActivationEnergy parameter is added to overall energy in order to increase the odds of pixel copy which would lead to new connection. Lambda tag/parameter determines “the strength” of curvature constraint. The higher the Lambda the more stiff cells will be *i.e.* they will tend to align along straight line.

## 29.24 BoundaryPixelTracker Plugin

BoundaryPixelTracker plugin keeps a list of boundary pixels for each cell. The syntax is as follows:

```

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

```

This plugin is also used by other plugins as a helper module. Examples use of this plugin is found in *Demos/PluginDemos/BoundaryPixelTracker\_xxx*.

## 29.25 GlobalBoundaryPixelTracker Plugin

GlobalBoundaryPixelTracker plugin tracks boundary pixels of **all** the cells including Medium. It is used in a Boundary Walker algorithm where instead of blindly picking pixel copy candidate we pick it from the set of pixels comprising boundaries of non frozen cells. In situations when lattice is large and there are not that many cells it makes sense to use BoundaryWalker algorithm to limit number of pixel picks that would not lead to actual pixel copy.

### Note

BoundaryWalkerAlgorithm does not really work with OpenMP version of CC3D which includes all versions starting with 3.6.0.

Take a look at the following example:

```

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

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

```

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

## 29.26 PixelTracker Plugin

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

```

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

```

This plugin is also used by other modules (e.g. `Mitosis`) as a helper module. Simple example can be found in `Demos/PluginDemos/PixelTrackerExample`. Beginning with 4.1.2, medium pixels can be optionally tracked using `TrackMedium`. This feature is automatically enabled by attaching a Fluctuation Compensator to a PDE solver.

## 29.27 MomentOfInertia Plugin

Related: [Calculating Inertia Tensor in CC3D](#) and [Calculating the Shape Constraint of a Cell – the Elongation Term](#)

The `MomentOfInertia` plugin keeps an up-to-date tensor of inertia for every cell. Internally, it uses the parallel axis theorem to calculate the most up-to-date tensor of inertia. Although the plugin can be added directly with `<Plugin Name="MomentOfInertia"/>`, it is most commonly called indirectly by other plugins like `LengthConstraint`.

`MomentOfInertia` plugin gives users access (via Python scripting) to current lengths of each cell’s semiaxes. Examples in `Demos/PluginDemos/MomentOfInertia` demonstrate how to get lengths of semiaxes. For example, to get semiaxes lengths for a given cell, in Python we would type:

```
axes = self.momentOfInertiaPlugin.getSemiaxes(cell)
```

`axes` is a 3-component vector with 0<sup>th</sup> element being length of minor axis, 1<sup>st</sup> element being the length of the median axis (which is set to 0 in 2D) and 2<sup>nd</sup> element indicating the length of major semiaxis.

### Note

**Important:** Since calculating lengths of semiaxes involves many floating point operations, it may happen (usually on hexagonal lattice) that for cells composed of 1, 2, or 3 pixels, one moment the square of one of the semiaxes may end up being slightly negative leading to NaN (not a number) length. This is due to round-off error and whenever

CC3D detects a very small absolute value of the square of the length of any of the semiaxes ( $10^{-6}$ ), it sets the length of the semiaxis to  $0.0$  regardless of whether the squared value is positive or negative. However, it is a good practice to test whether the length of semiaxis is sane by adding a simple `if` statement as shown below (here we show how to test for a `NaN`):

```
if length != length:  
    print("length is NaN")  
else:  
    print("length is a proper floating point number")
```

## 29.28 ConvergentExtension plugin

### Note

This is very specialized plugin that is currently is in Tier 2 plugins in terms of support. It attempts to implement energy term described in “Simulating Convergent Extension by Way of Anisotropic Differential Adhesion”, *Zajac M, Jones GL, and Glazier JA*, Journal of Theoretical Biology **222** (2), 2003. However due to certain ambiguities in the plugin description we had difficulties to getting it to work properly.

### Note

A better way to implement convergent extension is to follow the simulations described in “Filopodial-Tension Model of Convergent-Extension of Tissues”, *Julio M. Belmonte , Maciej H. Swat, James A. Glazier*, PLoS Comp Bio <https://doi.org/10.1371/journal.pcbi.1004952>

ConvergentExtension plugin presented here is a somewhat simplified version of energy term described *Mark Zajac's* paper.

This plugin uses the following syntax:

```
<Plugin Name="ConvergentExtension">  
    <Alpha Type="Condensing" >0.99</Alpha>  
    <Alpha Type="NonCondensing" >0.99</Alpha>  
    <NeighborOrder>2</NeighborOrder>  
</Plugin>
```

The `Alpha` tag represents numerical value of parameter from the paper.

## XML EXPRESSION EVALUATOR - MUPARSER

CC3D uses muParser to allow users specify simple mathematical expressions in the XML (or XML-equivalent Python scripts). The following link points to full specification of the muParser: [http://muparser.sourceforge.net/mup\\_features.html#idDef2](http://muparser.sourceforge.net/mup_features.html#idDef2). The general guideline to using muParser syntax inside XML is to enclose muParser expression between `<![CDATA[ and ]]>`:

```
<XML_ELEMENT_WITH_MUPARSER_EXPRESSION>
<![CDATA[
    MUPARSER EXPRESSION
]]
</XML_ELEMENT_WITH_MUPARSER_EXPRESSION>
```

For example:

```
<AdditionalTerm>
<![CDATA[
    CellType<1 ? 0.01*F : 0.15*F
]]
</AdditionalTerm>
```

The reason for enclosing muParser expression between `<![CDATA[ and ]]>` is to prevent XML parser from interpreting `<` or `>` as beginning or end of the XML elements

Alternatively you may replace XML with equivalent Python syntax in which case things will look a bit simpler:

```
DiffusionDataElmnt_2.ElementCC3D("AdditionalTerm", {}, " CellType<1 ? 0.01*F : 0.15*F ")
```



## DIFFUSION (PDE SOLVERS) IN COMPUCELL3D

One of the most important and time-consuming parts of the CC3D simulation is the process of solving Partial Differential Equations, which describe behavior of certain simulation objects (usually chemical fields). Most of the CC3D PDE solvers solve PDEs with diffusive terms. Since we are dealing with moving boundary condition problems, it was easiest and probably most practical to use explicit scheme of Finite Difference method. Most of CC3D PDE solvers run on multi core architectures and we also have GPU solvers which run on high performance GPU's and they also provide biggest speedups in terms of performance. Because CC3D solvers were implemented at different stages of the CC3D life cycle and often in response to particular user requests, CC3DML specification may differ from solver to solver. However, the basic structure of CC3DML PDE solver code follows a pattern.

For most use cases, we recommend Diffusion Solver [FE](#). For the highest level of customization, we recommend Reaction Diffusion Solver with Finite Volume Method ([FVM](#)). Please see the below reference to find the right PDE solver for your use case.

### General Reference

## 31.1 Accessing concentration fields managed by PDE solvers

Many of CC3D simulations have at least one diffusing field which represents some sort of chemical it can be growth factor, toxin or nutrient. The concentration fields are created by CC3D PDE solvers. One of the most common tasks that modelers implement is modifying cellular behaviours based on the chemical concentration at the center of mass of a cell (or for that matter any other point belonging to a given cell).

In this example, we will show you how to extract and modify values of the concentration fields.

You can take a look at [CompuCellPythonTutorial/diffusion](#) example if you want a quick preview of code that deals with diffusion fields. The task here is quite simple. We first have to get a handle to the field and then using Numpy-like syntax either read or modify field values.

The example that we will implement here is the following. We will start with “regular” cell-sorting cell layout where condensing and non-condensing cells are mixed together. In the corner of the lattice, we will place the pulse of the chemical and will let the chemical diffuse. We will monitor the values of the concentration at the center of mass of each cell and if this value is greater than `0.1` we will change cell type to Condensing. Assuming that the concentration pulse is big enough all cells will become Condensing after some time. Let’s take a look at the code:

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

    def start(self):
        field = self.getConcentrationField("FGF")
        field[0, 0, 0] = 2000
```

(continues on next page)

(continued from previous page)

```
def step(self, mcs):
    field = self.getConcentrationField("FGF")

    for cell in self.cellList:
        if field[cell.xCOM, cell.yCOM, cell.zCOM] > 0.1:
            cell.type = self.CONDENSING
```

In the start function we get a handle to a diffusion field FGF. This field is defined in the XML using the following code:

```
<Steppable Type="FlexibleDiffusionSolverFE">
<DiffusionField>
    <DiffusionData>
        <FieldName>FGF</FieldName>
        <DiffusionConstant>0.1</DiffusionConstant>
        <DecayConstant>1e-05</DecayConstant>
    </DiffusionData>
</DiffusionField>
</Steppable>
```

The XML code above defines diffusion and decay constants but says nothing about initial conditions. We could have defined initial conditions in the XML but we chose to do this in the start function of the Python-based steppable.

The content of the start function is almost self-explanatory. In the first line of the function we get a handle to concentration field. Notice that we pass “FGF” to self.getConcentrationField function. It is exactly the same name as we declared in XML. If you use field name that is undefined in the XML you will get None object in return. In the second line we initialize field to have concentration 2000 units at location  $x = 0, y = 0, z = 0$ . If we wanted to extend the area of initial concentration, we could have used the following Numpy slicing operation:

```
field[0:5:1, 0:5:1, 0] = 2000
```

and this would put 2000 units of concentration in the  $5 \times 5$  square at the origin of the lattice. Like range() in Python, slicing works as follows: the first number specifies the lower bound, second specifies upper bound (the maximum index is upper bound minus one), and third specifies step. In our example,  $0:5:1$  will select indices  $0, 1, 2, 3, 4$ .

When we type  $0:6:2$  then we will select indices  $0, 2, 4$  – again 6 being upper bound is not selected. For more information about Numpy slicing please see numpy tutorial online:

[http://wiki.scipy.org/Tentative\\_NumPy\\_Tutorial](http://wiki.scipy.org/Tentative_NumPy_Tutorial)

In the start function we first get a handle to the FGF field, and then we iterate over each cell in the simulation. We check if FGF concentration at the center of mass of each cell is greater than 0.1:

```
if field[cell.xCOM, cell.yCOM, cell.zCOM] > 0.1:
    ...
```

and if so we change the cell type to Condensing. Notice that to access center of mass of a cell we need to include the CenterOfMass plugin in the XML using the following code:

```
<Plugin Name="CenterOfMass">
```

All Twedit++ -generated templates put this plugin by default, but if you type XML manually you need to remember about this module. CenterOfMass plugin tracks and keeps an up-to-date center of mass of each cell. To access COM value from Python, we use the following syntax:

```
cell.xCOM
cell.yCOM
cell.zCOM
```

When you run the simulation you will notice that gradually all of the cells will turn into Condensing.

### 31.1.1 Min/Max field values

To access min or max of concentration fields (i.e. defined in the PDE solver) you simply type

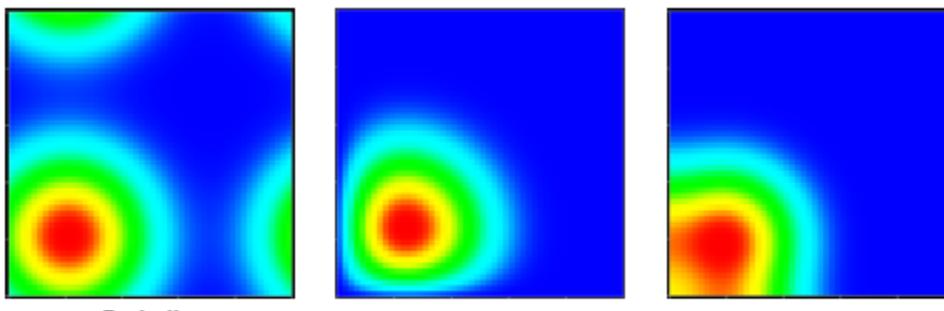
```
minVal = field.min()
```

or

```
maxVal=field.max()
```

## 31.2 Boundary Conditions for Diffusion

Boundary conditions not only control what happens when a chemical field attempts to cross the edge of the screen (simulation area); they can also serve as an infinite source from which your chemical may diffuse.



#### Related:

- Diffusion Solver Settings
- Boundary Conditions for Cells

### 31.2.1 ConstantValue

When Constant Value Diffusion is enabled, the chemical field will emit from the simulation boundary at a constant rate. The chemical field then spreads according to a Dirichlet distribution. *Max* controls the maximum concentration of the chemical on any given voxel. It also controls the maximum concentration that will generate from the source.

#### Example:

```
<Steppable Type="ReactionDiffusionSolverFVM">
  <DeltaT unit="s">1</DeltaT>
  <AutoTimeSubStep/>
  <FluctuationCompensator/>
  <DiffusionField Name="C1">
    <DiffusionData>
```

(continues on next page)

(continued from previous page)

```

<DiffusionConstant>0.1</DiffusionConstant>
<DiffusionCoefficient CellType="Type1">0.0</DiffusionCoefficient>
<DiffusivityByType/>
</DiffusionData>
<BoundaryConditions>
<Plane Axis="Y">
<ConstantValue PlanePosition="Min" Value="0.0"/>
<ConstantValue PlanePosition="Max" Value="3.0"/>
</Plane>
</BoundaryConditions>
</DiffusionField>
</Steppable>

```

**Expected Result:** The chemical C1 reaches a maximum concentration of 3.0.

### 31.2.2 ConstantDerivative

The primary use of Constant Derivative boundary conditions is to implement zero flux across the simulation boundary. This ensures that the chemical does not flow out of the simulation domain. One could use ConstantDerivative to simulate inflow/outflow of a chemical to or from the simulation domain. Alternatively, it can be used to create curvature or other nonlinear patterns of the substrate.

To use this feature, you should have one Plane tag that uses ConstantDerivative diffusion and another that uses ConstantValue.

**Example:**

```

<Steppable Type="ReactionDiffusionSolverFVM">
<DeltaT unit="s">1</DeltaT>
<AutoTimeSubStep/>
<FluctuationCompensator/>
<DiffusionField Name="C1">
<DiffusionData>
<DiffusionConstant>0.1</DiffusionConstant>
<DiffusionCoefficient CellType="Type1">0.0</DiffusionCoefficient>
<DiffusivityByType/>
</DiffusionData>
<BoundaryConditions>
<Plane Axis="X">
<ConstantValue PlanePosition="Min" Value="0.0"/>
<ConstantValue PlanePosition="Max" Value="1.0"/>
</Plane>
<Plane Axis="Y">
<ConstantDerivative PlanePosition="Min" Value="1.0"/>
<ConstantDerivative PlanePosition="Max" Value="2.0"/>
</Plane>
</BoundaryConditions>
</DiffusionField>
</Steppable>

```

**Expected Result:** We attain a semicircle pattern.

### 31.2.3 NoFlux

This is the default. It confers no special behaviors to the chemical field.

### 31.2.4 Periodic

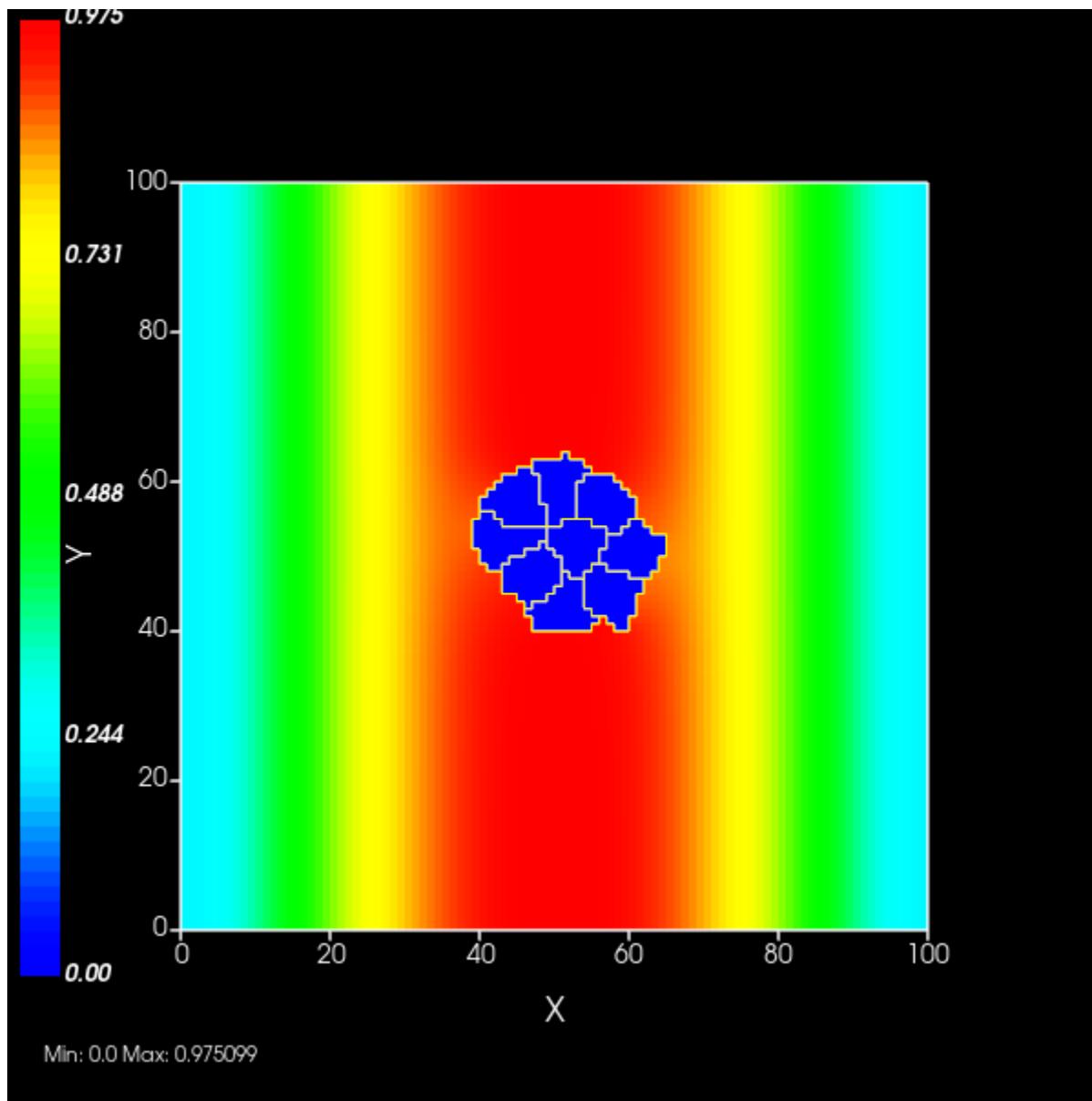
This allows the chemical field to cross over into the opposite boundary as soon as it reaches the edge of the screen.

Periodic boundary conditions are useful for large areas of tissue wherein cells near the edges need to have neighbors. This provides performance benefits since a smaller simulation area can be used.

**Example:**

```
<Steppable Type="DiffusionSolverFE">
  <FluctuationCompensator/>
  <DiffusionField Name="F1">
    <DiffusionData>
      <FieldName>F1</FieldName>
      <GlobalDiffusionConstant>0.1</GlobalDiffusionConstant>
      <InitialConcentrationExpression>x/100*(1-x/100)*4</
      InitialConcentrationExpression>
      <DiffusionCoefficient CellType="Type1">0.0</DiffusionCoefficient>
    </DiffusionData>
    <BoundaryConditions>
      <Plane Axis="X">
        <Periodic/>
      </Plane>
      <Plane Axis="Y">
        <Periodic/>
      </Plane>
    </BoundaryConditions>
  </DiffusionField>
</Steppable>
```

**Expected Result:** We attain a chemical field that looks symmetrical.



### 31.3 Diffusion Solver Settings

#### Related:

- [DiffusionSolverFE](#) (main PDE solver)
- [Flexible](#)
- [Fast](#)
- [ReactionDiffusion](#)

### 31.3.1 Instabilities of the Forward Euler Method

Most of the PDE solvers in CC3D use Forward Euler explicit numerical scheme. This method is unstable for large diffusion constant. As a matter of fact using D=0.25 with pulse initial condition will lead to instabilities in 2D. To deal with this you would normally use implicit solvers however due to moving boundary conditions that we have to deal with in CC3D simulations, memory requirements, performance and the fact that most diffusion constants encountered in biology are quite low (unfortunately this is not for all chemicals e.g. oxygen) we decided to use explicit scheme. If you have to use large diffusion constants with explicit solvers you need to do rescaling:

- 1) Set D, t, x according to your model
- 2) If

$$D \frac{\Delta t}{\Delta x^2} > 0.16 \ln 3D \quad (31.1)$$

you will need to call solver multiple times per MCS.

- 3) Set `<ExtraTimesPerMCS>` to N-1 where:

$$ND' = D \quad (31.2)$$

and

$$D \frac{\Delta t/N}{\Delta x^2} < 0.16 \ln 3D \quad (31.3)$$

### 31.3.2 Initial Conditions

We can specify initial concentration as a function of x, y, z coordinates using `<InitialConcentrationExpression>` tag we use muParser syntax to type the expression. Alternatively we may use `ConcentrationFileName` tag to specify a text file that contains values of concentration for every pixel:

`<ConcentrationFileName>initialConcentration2D.txt</ConcentrationFileName>`

The value of concentration of the last pixel read for a given cell becomes an overall value of concentration for a cell. That is if cell has, say 8 pixels, and you specify different concentration at every pixel, then cell concentration will be the last one read from the file.

**Concentration file format** is as follows:

`*x y z c*`

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

**Example:**

```
0 0 0 1.2
0 0 1 1.4
...

```

The initial concentration can also be input from the Python script (typically in the start function of the steppable) but often it is more convenient to type one line of the CC3DML script than few lines in Python.

### 31.3.3 Boundary Conditions

Related: [Boundary Conditions Reference](#)

All standard solvers (`DiffusionSolverFE`, `Flexible`, `Fast`, and `ReactionDiffusion`) by default use the same boundary conditions as the GGH simulation (and those are specified in the Potts section of the CC3DML script). Users can, however, override those defaults and use customized boundary conditions for each field individually. Currently CompuCell3D supports the following boundary conditions for the diffusing fields: periodic, constant value (Dirichlet) and constant derivative (von Neumann). To specify custom boundary condition we include `<BoundaryCondition>` section inside `<DiffusionField>` tags.

The `<BoundaryCondition>` section describes boundary conditions along particular axes. For example:

```
<Plane Axis="X">
    <ConstantValue PlanePosition="Min" Value="10.0"/>
    <ConstantValue PlanePosition="Max" Value="10.0"/>
</Plane>
```

specifies boundary conditions along the x axis. They are Dirichlet-type boundary conditions. `PlanePosition='Min'` denotes plane parallel to yz plane passing through  $x=0$ . Similarly `PlanePosition='Min'` denotes plane parallel to yz plane passing through  $x=\text{fieldDimX}-1$  where `fieldDimX` is x dimension of the lattice.

By analogy we specify constant derivative boundary conditions:

```
<Plane Axis="Y">
    <ConstantDerivative PlanePosition="Min" Value="10.0"/>
    <ConstantDerivative PlanePosition="Max" Value="10.0"/>
</Plane>
```

We can also mix types of boundary conditions along single axis:

```
<Plane Axis="Y">
    <ConstantDerivative PlanePosition="Min" Value="10.0"/>
    <ConstantValue PlanePosition="Max" Value="0.0"/>
</Plane>
```

Here in the xz plane at  $y=0$  we have von Neumann boundary conditions but at  $y=\text{fieldDimY}-1$  we have dirichlet boundary condition.

To specify periodic boundary conditions along, say, x axis we use the following syntax:

```
<Plane Axis="X">
    <Periodic/>
</Plane>
```

Notice, that `<Periodic>` boundary condition specification applies to both “ends” of the axis *i.e.* we cannot have periodic boundary conditions at  $x=0$  and constant derivative at  $x=\text{fieldDimX}-1$ .

#### Solvers

## 31.4 AdvectionDiffusionSolver

### Note

This is an experimental module and was not fully curated.

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

The equation solved by this steppable is of the type:

$$\frac{\partial c}{\partial t} = D \nabla^2 c - kc + \vec{v} \cdot \vec{\nabla} c + \text{secretion} \quad (31.4)$$

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

In addition to just solving advection-diffusion equation this module allows users to specify secretion rates of the cells as well as different secretion modes. The syntax for this module follows same pattern as `FlexibleDiffusionSolverFE`.

Example syntax:

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

## 31.5 DiffusionSolverFE

`DiffusionSolverFE` is the main PDE solver in CompuCell3D. It eliminates several limitations and inconveniences of its predecessor, `FlexibleDiffusionSolverFE`. and provides new features such as GPU implementation or cell type dependent diffusion/decay coefficients. In addition it also eliminates the need to rescale diffusion/decay/secretion constants. It checks stability condition of the PDE and then rescales appropriately all coefficients and computes how many extra times per MCS the solver has to be called. It makes those extra calls automatically.

### ⚠ Warning

One of the key differences between `FlexibleDiffusionSolverFE` and `DiffusionSolverFE` is the way in which secretion is treated. In `FlexibleDiffusionSolverFE` all secretion amount is done once followed by possibly multiple diffusion calls to diffusion (to avoid numerical instabilities). In `DiffusionSolverFE` the default mode of operation is such that multiple secretion and diffusion calls are interleaved. This means that instead of secreting full amount for a given MCS and diffusing it, the `DiffusionSolverFE` secretes substance gradually so that there is equal amount of secretion before each call of the diffusion. One can change this behavior by adding `<DoNotScaleSecretion/>` to definition of the diffusion solver e.g.

```
<Steppable Type="DiffusionSolverFE">
  <DoNotScaleSecretion/>
  <DiffusionField Name="ATTR">
    <DiffusionData>

  ...

```

With such definition the DiffusionSolverFE will behave like FlexibleDiffusionSolverFE as far as computation.

### Note

DiffusionSolverFE autoscales diffusion discretization depending on the lattice so that `<AutoscaleDiffusion/>` we used in FlexibleDiffusionSolverFE is unnecessary. This may result in slow performance so users have to be aware that those extra calls to the solver may be the cause.

Typical syntax for the DiffusionSolverFE may look like example below:

```
<Steppable Type="DiffusionSolverFE">
  <DiffusionField Name="ATTR">
    <DiffusionData>
      <FieldName>ATTR</FieldName>
      <GlobalDiffusionConstant>0.1</GlobalDiffusionConstant>
      <GlobalDecayConstant>5e-05</GlobalDecayConstant>
      <DiffusionCoefficient CellType="Red">0.0</DiffusionCoefficient>
    </DiffusionData>
    <SecretionData>
      <Secretion Type="Bacterium">100</Secretion>
    </SecretionData>
    <BoundaryConditions>
      <Plane Axis="X">
        <Periodic/>
      </Plane>
      <Plane Axis="Y">
        <Periodic/>
      </Plane>
    </BoundaryConditions>
  </DiffusionField>
</Steppable>
```

The syntax resembles the syntax for FlexibleDiffusionSolverFE. We specify global diffusion constant by using `<GlobalDiffusionConstant>` tag. This specifies diffusion coefficient which applies to entire region of the simulation. We can override this specification for regions occupied by certain cell types by using the following syntax:

```
<DiffusionCoefficient CellType="Red">0.0</DiffusionCoefficient>
```

Similar principles apply to decay constant and we use `<GlobalDecayConstant>` tag to specify global decay coefficient and

```
<DecayCoefficient CellType="Red">0.0</DecayCoefficient>
```

to override global definition for regions occupied by Red cells.

We do not support <DeltaX>, <DeltaT> or <ExtraTimesPerMCS> tags.

**Note**

DiffusionSolverFE autoscales diffusion discretization depending on the lattice so that <AutoscaleDiffusion/> we used in FlexibleDiffusionSolverFE is unnecessary.

### 31.5.1 Running DiffusionSolver on GPU

To run DiffusionSolverFE on GPU all we have to do (besides having OpenCL compatible GPU and correct drives installed) to replace first line of solver specification:

```
<Steppable Type="DiffusionSolverFE">
```

with

```
<Steppable Type="DiffusionSolverFE_OpenCL">
```

**Note**

Depending on your computer hardware you may or may not be able to take advantage of GPU capabilities.

## 31.6 FastDiffusionSolver2D

**Note**

The current implementation of DiffusionSolverFE may actually be faster than this legacy module. So before committing to it please verify the performance of teh DiffusionSolverFE vs FastDiffusionSolverFE

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

```
<Steppable Type="FastDiffusionSolver2DFE">
  <DiffusionField Name="FGF">
    <DiffusionData>
      <UseBoxWatcher/>
      <FieldName>FGF</FieldName>
      <DiffusionConstant>0.010</DiffusionConstant>
      <DecayConstant>0.003</DecayConstant>
      <ExtraTimesPerMCS>2</ExtraTimesPerMCS>
      <DoNotDecayIn>Wall</DoNotDecay>
      <ConcentrationFileName>Demos/diffusion/2D_fast_box.pulse.txt
      </ConcentrationFileName>
    </DiffusionData>
  </DiffusionField>
</Steppable>
```

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

For explanation on how <ExtraTimesPerMCS> tag works see section on FlexibleDiffusionSolverFE.

## 31.7 FlexibleDiffusionSolver

This steppable is one of the basic and most important modules in CompuCell3D simulations.

### Tip

Starting from version 3.6.2 we developed DiffusionSolverFE which eliminates several inconveniences of FlexibleDiffusionSolver.

As the name suggests it is responsible for solving diffusion equation but in addition to this it also handles chemical secretion which maybe thought of as being part of general diffusion equation:

$$\frac{\partial c}{\partial t} = D \nabla^2 c - kc + \text{secretion} \quad (31.5)$$

where  $k$  is a decay constant of concentration  $c$  and  $D$  is the diffusion constant. The term called `secretion` has the meaning as described below.

Example syntax for FlexibleDiffusionSolverFE looks as follows:

```
<Steppable Type="FlexibleDiffusionSolverFE">
    <AutoscaleDiffusion/>
    <DiffusionField Name="FGF8">
        <DiffusionData>
            <FieldName>FGF8</FieldName>
            <DiffusionConstant>0.1</DiffusionConstant>
            <DecayConstant>0.002</DecayConstant>
            <ExtraTimesPerMCS>5</ExtraTimesPerMCS>
            <DeltaT>0.1</DeltaT>
            <DeltaX>1.0</DeltaX>
            <DoNotDiffuseTo>Bacteria</DoNotDiffuseTo>
            <InitialConcentrationExpression>x*y
            </InitialConcentrationExpression>
        </DiffusionData>

        <SecretionData>
            <Secretion Type="Amoeba">0.1</Secretion>
        </SecretionData>

        <BoundaryConditions>
            <Plane Axis="X">
                <ConstantValue PlanePosition="Min" Value="10.0"/>
                <ConstantValue PlanePosition="Max" Value="10.0"/>
            </Plane>

            <Plane Axis="Y">
                <ConstantDerivative PlanePosition="Min" Value="10.0"/>
                <ConstantDerivative PlanePosition="Max" Value="10.0"/>
            </Plane>
        </BoundaryConditions>
    </DiffusionField>
    <DiffusionField Name="FGF">
```

(continues on next page)

(continued from previous page)

```

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

```

We define sections that describe a field on which the steppable is to operate. In our case we declare just two diffusion fields - FGF8 and FGF.

### ⚠ Warning

When you want to solve more than one diffusive field using given diffusion you should declare those multiple fields within a single definition of the diffusion solver. You cannot include two diffusion solver definitions - one with e.g. FGF8 and another with FGF. In other words you can only define e.g. `FlexibleDiffusionSolverFE` once per simulation. You can however use `FlexibleDiffusionSolverFE` for e.g. FGF and `DiffusionSolverFE` for e.g. FGF8

Inside the diffusion field we specify sections describing diffusion and secretion. Let's take a look at `DiffusionData` section first:

```

<DiffusionField Name="FGF8">
<DiffusionData>
    <FieldName>FGF8</FieldName>
    <DiffusionConstant>0.1</DiffusionConstant>
    <DecayConstant>0.002</DecayConstant>
    <ExtraTimesPerMCS>5</ExtraTimesPerMCS>
    <DeltaT>0.1</DeltaT>
    <DeltaX>1.0</DeltaX>
    <DoNotDiffuseTo>Bacteria</DoNotDiffuseTo>

    <InitialConcentrationExpression>x*y
    </InitialConcentrationExpression>

</DiffusionData>

```

We give a name (FGF8) to the diffusion field – this is required as we will refer to this field in other modules.

Next we specify diffusion constant and decay constant.

### ⚠ Warning

We use Forward Euler Method to solve these equations. This is not a stable method for solving diffusion equation

and we do not perform stability checks. If you enter too high diffusion constant for example you may end up with unstable (wrong) solution. Always test your parameters to make sure you are not in the unstable region.

We may also specify cells that will not participate in the diffusion. You do it using `<DoNotDiffuseTo>` tag. In this example we do not let any FGF diffuse into Bacteria cells. You may of course use as many as necessary `<DoNotDiffuseTo>` tags. To prevent decay of a chemical in certain cells we use syntax:

```
<DoNotDecayIn>Medium</DoNotDecayIn>
```

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

```
<SecretionData>
  <SecretionOnContact Type="Medium" SecreteOnContactWith="Amoeba">0.1</
  ->SecretionOnContact>
  <Secretion Type="Amoeba">0.1</Secretion>
</SecretionData>
```

Here we have a definition two major secretion modes. Line:

```
<Secretion Type="Amoeba">0.1</Secretion>
```

ensures that every cell of type Amoeba will get 0.1 increase in concentration every MCS. Line:

```
<SecretionOnContact Type="Medium" SecreteOnContactWith="Amoeba">0.1</SecretionOnContact>
```

means that cells of type Medium will get additional 0.1 increase in concentration **but only when** they touch cell of type Amoeba. This mode of `secretion` is called `SecretionOnContact`.

We can also see new CC3DML tags `<DeltaT>` and `<DeltaX>`. Their values determine the correspondence between MCS and actual time and between lattice spacing and actual spacing size. In this example for the first diffusion field one MCS corresponds to 0.1 units of actual time and lattice spacing is equal 1 unit of actual length. What is happening here is that the diffusion constant gets multiplied by:

```
DeltaT/(DeltaX * DeltaX)
```

provided the decay constant is set to 0. If the decay constant is not zero `DeltaT` appears additionally in the term (in the explicit numerical approximation of the diffusion equation solution) containing decay constant so in this case it is more than simple diffusion constant rescaling.

`DeltaT` and `DeltaX` settings are closely related to `ExtraTimesPerMCS` setting which allows calling of diffusion (and only diffusion) more than once per MCS. The number of extra calls per MCS is specified by the user on a per-field basis using `ExtraTimesPerMCS` tag.

### ⚠ Warning

When using `ExtraTimesPerMCS`, secretion functions will called only once per MCS. This is different than using `PDESolverCaller`, where the entire module is called multiple times (this includes diffusion and secretion for all fields).

### 💡 Tip

We recommend that you stay away from redefining `DeltaX` and `DeltaT` and assume that your diffusion/decay coefficients are expressed in units of pixel (distance) and MCS (time). This way when you assign physical time and distance units to MCS and pixels you can easily obtain diffusion and decay constants. `DeltaX` and `DeltaT` introduce unnecessary complications.

The `AutoscaleDiffusion` tag tells CC3D to automatically rescale diffusion constant when switching between square and hex lattices. In previous versions of CC3D such scaling had to be done manually to ensure that solutions diffusion of equation on different lattices match. Here we introduced for user convenience a simple tag that does rescaling automatically. The rescaling factor comes from the fact that the discretization of the divergence term in the diffusion equation has factors such as unit lengths, using surface area and pixel/voxel volume in it. On a square lattice all those values have numerical value of 1.0. On hex lattice, and for that matter of non-square lattices, only pixel/voxel volume has numerical value of 1. All other quantities have values different than 1.0 which causes the necessity to rescale diffusion constant. The detail of the hex lattice derivation will be presented in the “Introduction to Hexagonal Lattices in CompuCell3D” - [http://www.compcell3d.org/BinDoc/cc3d\\_binaries/Manuals/HexagonalLattice.pdf](http://www.compcell3d.org/BinDoc/cc3d_binaries/Manuals/HexagonalLattice.pdf).

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

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

where  $m_c, m_f, n_c, n_f, p_c, p_d$  are coupling coefficients. To code the above equations in CC3DML syntax you need to use the following syntax:

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

  <DiffusionField>
    <DiffusionData>
      <FieldName>d</FieldName>
      <DiffusionConstant>0.02</DiffusionConstant>
      <DecayConstant>0.001</DecayConstant>
      <CouplingTerm InteractingFieldName="c" CouplingCoefficient="-0.1"/>
      <CouplingTerm InteractingFieldName="f" CouplingCoefficient="-0.2"/>
      <DeltaT>0.01</DeltaT>
      <DeltaX>0.1</DeltaX>
      <DoNotDiffuseTo>Bacteria</DoNotDiffuseTo>
    </DiffusionData>
  </DiffusionField>
```

(continues on next page)

(continued from previous page)

```

</DiffusionData>
<SecretionData>
    <Secretion Type="Amoeba">0.1</Secretion>
</SecretionData>
</DiffusionField>

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

```

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

```

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

```

## 31.8 KernelDiffusionSolver

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

Simply put KernelDiffusionSolver solves diffusion equation:

$$\frac{\partial c}{\partial t} = D \nabla^2 c - kc + \text{secretion} \quad (31.6)$$

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

The syntax for this solver is as follows:

```

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

```

(continues on next page)

(continued from previous page)

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

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

```
<Steppable Type="KernelDiffusionSolver">
    <DiffusionField Name="FGF">
        <Kernel>4</Kernel>
        <CoarseGrainFactor>2</CoarseGrainFactor>
        <DiffusionData>
            <FieldName>FGF</FieldName>
            <DiffusionConstant>1.0</DiffusionConstant>
            <DecayConstant>0.000</DecayConstant>
            <ConcentrationFileName>
                Demos/diffusion/diffusion_2D.pulse.txt
            </ConcentrationFileName>
        </DiffusionData>
    </DiffusionField>
</Steppable>
```

## 31.9 ReactionDiffusionSolverFE Plugin

Related: ReactionDiffusionSolverFVM (finite volume) Plugin

ReactionDiffusionSolverFE is the predecessor of ReactionDiffusionSolverFVM. Its usage is relatively simpler, and it is primarily defined in XML. Unlike the ReactionDiffusionSolverFVM, this PDE solver supports the use of an *AdditionalTerm* expression that defines the reaction diffusion equation.

For example usage, see our demos on FitzHugh-Nagumo, Schnakenberg, Gierer-Meinhardt, Gray-Scott, Oscillatory, and Thomas models.

### 31.9.1 XML Properties

- Element `<ConcentrationFileName>` (optional)
  - Contains the relative path to a text file that contains values of concentration for every pixel.
  - We recommend using either `InitialConcentrationExpression` or `ConcentrationFileName` but not both.
  - See *Diffusion Solver Settings* for more details
- Element `<DeltaX>` (optional)
  - Specifies discretization along the *x* dimension.
  - If only `<DeltaX>` is specified, then a uniform discretization is applied along all directions.
- Element `<DeltaY>` (optional)
  - Specifies discretization along the *y* dimension.
- Element `<DeltaZ>` (optional)
  - Specifies discretization along the *z* dimension.

- **Element <DeltaT> (optional)**
  - Allows calling of diffusion (and only diffusion) more than once per MCS
  - It is intended to be used with <ExtraTimesPerMCS>
  - See [DeltaT](#) and [Diffusion Solver Settings](#) for more details
- **Element <DiffusionField>**
  - Defines a diffusion field
  - **Attribute Name:** the name of the field
  - **Element <DiffusionData>**
    - \* **Element <AdditionalTerm> (optional)**
      - Contains a [muParser expression](#) that determines the reaction diffusion equation. See [here](#) for more details
      - The expression can be unique for each CellType with the use of ternary operators.
    - \* **Element <DecayCoefficient> (optional)**
      - Specifies a constant decay coefficient for a cell type.
      - Can be set per cell during simulation execution
      - **Attribute CellType:** name of the cell type
    - \* **Element <DecayConstant> (optional)**
      - Specifies a constant diffusion coefficient for the medium.
      - Note that <GlobalDecayConstant> performs the same function, despite its name.
    - \* **Element <DiffusionCoefficient> (optional)**
      - Specifies a constant diffusion coefficient for a cell type.
      - Can be set per cell during simulation execution
      - **Attribute CellType:** name of the cell type
    - \* **Element <DiffusionConstant> (optional)**
      - Specifies a constant diffusion coefficient for the medium.
      - Note that <GlobalDiffusionConstant> performs the same function, despite its name.
    - **Element <SecretionData> (optional)**
      - \* Secretion data elements, defined in the same way as for [FlexibleDiffusionSolverFE](#)
    - **Element <BoundaryConditions> (optional)**
      - \* Boundary condition elements, defined in the same as for DiffusionSolverFE.
      - \* Boundary conditions are applied at surfaces and can be manipulated at each site during simulation execution.
      - \* If a condition is not specified for a boundary, then it is assumed to be zero flux.
    - \* **BoundaryConditions options:**
      - NoFlux (default)
      - ConstantValue
      - ConstantDerivative

- Periodic
- Element **<ExtraTimesPerMCS>** (optional)
  - Allows calling of diffusion (and only diffusion) more than once per MCS
  - It is intended to be used with **<DeltaT>** and **<DeltaX>**
  - See [DeltaT](#) and [Diffusion Solver Settings](#) for more details
- Element **<FluctuationCompensator>** (optional)
  - Enables deployment of the CC3D FluctuationCompensator.
- Element **<InitialConcentrationExpression>** (optional)
  - Contains a [muParser expression](#) to define the level of concentration at each voxel on MCS 0.
  - You can use either **InitialConcentrationExpression** or **ConcentrationFileName** but not both. However, the preferred way to handle initial conditions is to use the **start** function of the Python Steppable and initialize fields there.

### 31.9.2 Example Usage

The following is a representative example of a specification for the RD Solver using two fields, *F* and *H*. *DeltaX/Y/Z*, *DeltaT*, and *ExtraTimesPerMCS* apply to all RD equations. Notice, secretion is called only once per MCS regardless of how many times you call diffuse step.

```

<Steppable Type="ReactionDiffusionSolverFE">
  <!-- <DeltaT>0.1</DeltaT>
  <DeltaX>1.0</DeltaX>
  <ExtraTimesPerMCS>9</ExtraTimesPerMCS> -->
  <DiffusionField>
    <DiffusionData>
      <FieldName>F</FieldName>
      <DecayConstant>0.005</DecayConstant>
      <DiffusionConstant>0.010</DiffusionConstant>
      <ConcentrationFileName>Simulation/diffusion_2D.pulse.txt</
      ConcentrationFileName>
      <AdditionalTerm>-0.01*H</AdditionalTerm>
    </DiffusionData>
    <BoundaryConditions>
      <Plane Axis="X">
        <Periodic/>
      </Plane>
      <Plane Axis="Y">
        <Periodic/>
      </Plane>
    </BoundaryConditions>
  </DiffusionField>
  <!-- Add more DiffusionFields here as desired -->
</Steppable>

```

You can also define diffusion and decay coefficients on a per-cell basis.

```

<DiffusionData>
  <FieldName>VEGF</FieldName>
  <DiffusionConstant>0.25</GlobalDiffusionConstant>

```

(continues on next page)

(continued from previous page)

```
<DecayConstant>0</GlobalDecayConstant>
<DiffusionCoefficient CellType="StalkCell">0.1</DiffusionCoefficient>
<DiffusionCoefficient CellType="TipCell">0.1</DiffusionCoefficient>
<DecayCoefficient CellType="StalkCell">0</DecayCoefficient>
<DecayCoefficient CellType="TipCell">0</DecayCoefficient>
<InitialConcentrationExpression>0*x/100</InitialConcentrationExpression>
<!-- As an alternative to InitialConcentrationExpression, you can use -->
<ConcentrationFileName. -->
<!-- <ConcentrationFileName>INITIAL CONCENTRATION FIELD - typically a file with path -->
<Simulation/NAME_OF_THE_FILE.txt</ConcentrationFileName> -->
</DiffusionData>
```

ReactionDiffusionSolverFE supports the use of *InitialConcentrationExpression* or *ConcentrationFileName* to define the level of chemical present at MCS 0.

```
<DiffusionData>
  ...
  <InitialConcentrationExpression>0*x/100</InitialConcentrationExpression>
  <!-- OR -->
  <ConcentrationFileName>Simulation/NAME_OF_THE_FILE.txt</ConcentrationFileName> -->
</DiffusionData>
```

### 31.9.3 How It Works

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

$$\begin{aligned} \frac{\partial c_1}{\partial t} &= D\nabla^2 c_1 - kc_1 + \text{secretion} + f_1(c_1, c_2, \dots, c_N, W) \\ \frac{\partial c_2}{\partial t} &= D\nabla^2 c_2 - kc_2 + \text{secretion} + f_2(c_1, c_2, \dots, c_N, W) \\ &\dots \\ \frac{\partial c_N}{\partial t} &= D\nabla^2 c_N - kC_N + \text{secretion} + f_N(c_1, c_2, \dots, c_N, W) \end{aligned}$$

where W denotes cell type

Let's consider a simple example of such system:

$$\begin{aligned} \frac{\partial F}{\partial t} &= 0.1\nabla^2 F - 0.1H \\ \frac{\partial H}{\partial t} &= 0.0\nabla^2 H + 0.1F \end{aligned}$$

It can be coded as follows:

```
<Steppable Type="ReactionDiffusionSolverFE">
  <AutoscaleDiffusion/>
  <DiffusionField Name="F">
    <DiffusionData>
      <FieldName>F</FieldName>
      <DiffusionConstant>0.010</DiffusionConstant>
      <ConcentrationFileName>
        Demos/diffusion/diffusion_2D.pulse.txt
      </ConcentrationFileName>
    </DiffusionData>
  </DiffusionField>
</Steppable>
```

(continues on next page)

(continued from previous page)

```

<AdditionalTerm>-0.01*H</AdditionalTerm>
</DiffusionData>
</DiffusionField>

<DiffusionField Name="H">
  <DiffusionData>
    <FieldName>H</FieldName>
    <DiffusionConstant>0.0</DiffusionConstant>
    <AdditionalTerm>0.01*F</AdditionalTerm>
  </DiffusionData>
</DiffusionField>
</Steppable>

```

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

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

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

For example, let's assume that the additional term for the second equation is the following:  $f_F = \begin{cases} 0.1F \\ \text{if } \text{CellType}=1 \\ 0.51F \\ \text{otherwise} \end{cases}$  In such a case, additional terms would be coded as follows :

```
<AdditionalTerm>CellType==1 ? 0.01*F : 0.15*F</AdditionalTerm>
```

We used a ternary operator, which functions the same as an ``if-then-else`` statement, to decide which expression to use based on whether or not the `CellType` is 1. (The syntax is similar to programming languages like C or C++)

The syntax of the ternary (aka `if-then-else` statement) is as follows:

```
condition ? expression if condition is true : expression if condition false
```

### Warning

**Important:** If change the above expression to

we will get an XML parsing error. Why? This is because the XML parser will think that `<1` is the beginning of the new XML element. To fix this, you could use two approaches:

1. Present your expression as CDATA

```

<AdditionalTerm>
  <! [CDATA[
    CellType < 1 ? 0.01*F : 0.15*F
  ]]>
</AdditionalTerm>

```

In this case, the XML parser will correctly interpret the expression enclosed between `<![CDATA[` and `]]>`.

2. Replace XML using equivalent Python syntax) in which case you would code the above XML element as the following Python statement:

```
DiffusionDataElmnt.ElementCC3D('AdditionalTerm', {}, 'CellType<1 ? 0.01*F : 0.15*F')
```

In summary, if you would like to use muParser for more flexibility in your XML, make sure to use this general syntax:

```
<AdditionalTerm>
  <![CDATA[
    YOUR EXPRESSION
  ]]>
</AdditionalTerm>
```

One thing to remember is that the computing time of the additional term depends on the level of complexity of this term. Thus, you might get some performance degradation for very complex expressions coded in muParser.

Similarly as in the case of `FlexibleDiffusionSolverFE`, we may use the `<AutoscaleDiffusion>` tag, which tells CC3D to automatically rescale the diffusion constant. See section [FlexibleDiffusionSolver](#) or the [Appendix](#) for more information.

Each diffusion field can have a localized `DiffusionConstant` and/or `DecayConstant` for `ReactionDiffusionSolverFE`. Behind the scenes, the max stable decay coefficient is a value just under 1 (approx.  $1 - 1.17549e-38$ ).

## 31.10 ReactionDiffusionSolverFVM Plugin

Related: [ReactionDiffusionSolverFE Plugin](#)

### 31.10.1 How It Works

The reaction diffusion finite volume (RDFVM) solver provides a high level of customization in both XML and Python. It uses the finite volume method to solve the following system of  $N$  reaction diffusion equations:

$$\begin{aligned} \int \iiint \frac{\partial c_1}{\partial t} dV dt &= \int \iiint (\nabla (D_1 \nabla c_1) + A_1 (c_1, c_2, \dots, c_N) c_1 + B_1 (c_1, c_2, \dots, c_N)) dV dt \\ \int \iiint \frac{\partial c_2}{\partial t} dV dt &= \int \iiint (\nabla (D_2 \nabla c_2) + A_2 (c_1, c_2, \dots, c_N) c_2 + B_2 (c_1, c_2, \dots, c_N)) dV dt \\ &\dots \\ \int \iiint \frac{\partial c_N}{\partial t} dV dt &= \int \iiint (\nabla (D_N \nabla c_N) + A_N (c_1, c_2, \dots, c_N) c_N + B_N (c_1, c_2, \dots, c_N)) dV dt \end{aligned}$$

The RDFVM solver provides a broad range of basic and detailed modeling and simulation capabilities, including the following,

- Explicit surface transport modeling and boundary conditions on the basis of location
- Dynamic diffusivity on the basis of cells and location
- Maintenance of numerical stability through adaptive time-stepping
- String-based field reaction model specification
- Secretion and uptake on the basis of cells
- Settable discretization length along each spatial direction

At its most basic level, the RDFVM solver calculates the flux across the implicit surfaces of the lattice. RDVM differs from other CC3D solvers in that it provides the ability to choose a model of the flux across each surface in the lattice. Consider a point  $\mathbf{x}$  in the lattice, and suppose  $\mathbf{x}_k$  is the coordinate of a  $k$ th neighboring site in the lattice. For each  $j$ th concentration field  $c_j(\mathbf{x}, t)$  at time  $t$ , the RDFVM solver calculates the flux  $F_{j,k}$  of material over the surface shared by the volume element at  $\mathbf{x}$  and its  $k$ th neighbor. Using the finite volume method, this formalism produces the discretized form for calculating each concentration field at time  $t + \Delta t$ ,

$$c_j(\mathbf{x}, t + \Delta t) = c_j(\mathbf{x}, t) + \Delta t \left( \sum_k F_{j,k}(c_j(\mathbf{x}, t), c_j(\mathbf{x}_k, t)) + s_j(\mathbf{x}, t) \right)$$

Transport via diffusion occurs according to the diffusivity  $D_j(\mathbf{x}, t)$  on either side of a surface and distance  $\Delta x_k$  separating  $\mathbf{x}$  and  $\mathbf{x}_k$ ,

$$F_{j,k}(\mathbf{x}, t) = \frac{1}{(\Delta x_k)^2} D_{j,k}(\mathbf{x}, t) (c_j(\mathbf{x}_k, t) - c_j(\mathbf{x}, t)),$$

where  $D_{j,k}(\mathbf{x}, t)$  is the diffusivity of the surface shared by the volumes at  $\mathbf{x}$  and  $\mathbf{x}_k$ ,

$$D_{j,k}(\mathbf{x}, t) = \frac{2D_j(\mathbf{x}, t) D_j(\mathbf{x}_k, t)}{D_j(\mathbf{x}, t) + D_j(\mathbf{x}_k, t)}$$

Transport via a simple biased permeability occurs according to the permeation coefficient  $P_j(\mathbf{x}, t)$ , bias coefficients  $a_j(\tau)$  and  $b_j(\tau)$  according to cell type  $\tau$ , and area  $A_k$  of the surface shared by the volumes at  $\mathbf{x}$  and  $\mathbf{x}_k$ ,

$$F_{j,k}(\mathbf{x}, t) = \frac{1}{\Delta x_k} (b_j(\tau(\mathbf{x}_k, t)) c_j(\mathbf{x}_k, t) - a_j(\mathbf{x}, t) c_j(\mathbf{x}, t))$$

where  $P_{j,k}(\mathbf{x}, t)$  is the permeationg coefficient of the surface shared by the volumes at  $\mathbf{x}$  and  $\mathbf{x}_k$ ,

$$P_{j,k}(\mathbf{x}, t) = \frac{2P_j(\mathbf{x}, t) P_j(\mathbf{x}_k, t)}{P_j(\mathbf{x}, t) + P_j(\mathbf{x}_k, t)}$$

### 31.10.2 Example Usage

The following is a representative example of a specification for the RDFVM solver using two fields  $U$  and  $V$  and two cell types *CellType1* and *CellType2*,

```
<Steppable Type="ReactionDiffusionSolverFVM">
  <DeltaX>1.0</DeltaX>
  <DeltaY>1.0</DeltaY>
  <DeltaZ>1.0</DeltaZ>
  <AutoTimeSubStep/>
  <FluctuationCompensator/>
  <DiffusionField Name="U">
    <DiffusionData>
      <DiffusionConstant>0.1</DiffusionConstant>
      <DiffusionCoefficient CellType="CellType1">0.1</DiffusionCoefficient>
      <DiffusionCoefficient CellType="CellType2">0.1</DiffusionCoefficient>
      <DiffusivityByType/>
      <DiffusivityFieldInMedium/>
      <InitialConcentrationExpression>x</InitialConcentrationExpression>
    </DiffusionData>
    <SecretionData>
      ...
    </SecretionData>
  </DiffusionField>
</Steppable>
```

(continues on next page)

(continued from previous page)

```

<ReactionData>
    <ExpressionSymbol>u</ExpressionSymbol>
    <ExpressionMult>(-1.0+u*v)</ExpressionMult>
    <ExpressionIndep>0.1</ExpressionIndep>
</ReactionData>
<BoundaryConditions>
    ...
</BoundaryConditions>
</DiffusionField>
<DiffusionField Name="V">
    <DiffusionData>
        <DiffusionConstant>0.1</DiffusionConstant>
        <DiffusionCoefficient CellType="CellType1">0.1</DiffusionCoefficient>
        <DiffusionCoefficient CellType="CellType2">0.1</DiffusionCoefficient>
        <DiffusivityByType/>
        <DiffusivityFieldEverywhere/>
        <PermIntCoefficient Type1="CellType1" Type2="CellType1">0.1</
        <PermIntCoefficient>
            <PermIntCoefficient Type1="CellType1" Type2="CellType2">0.1</
        <PermIntCoefficient>
            <PermIntCoefficient Type1="CellType2" Type2="CellType2">0.1</
        <PermIntCoefficient>
            <PermIntBias Type1="CellType1" Type2="CellType2">0.01</PermIntBias>
            <SimplePermInt/>
        </DiffusionData>
        <SecretionData>
            ...
        </SecretionData>
    <ReactionData>
        <ExpressionSymbol>v</ExpressionSymbol>
        <ExpressionMult>(-u^2)</ExpressionMult>
        <ExpressionIndep>0.9</ExpressionIndep>
    </ReactionData>
    <BoundaryConditions>
        ...
    </BoundaryConditions>
</DiffusionField>
</Steppable>

```

This specification implements a number of features while implementing the Schnakenberg model of the form,

$$\frac{\partial U}{\partial t} = \nabla(D_U \nabla U) + (UV - 1)U + 0.1$$

$$\frac{\partial V}{\partial t} = \nabla(D_V \nabla V) - U^2V + 0.9$$

### 31.10.3 XML Properties

- Element `<DeltaX>` (optional)
  - Specifies discretization along the  $x$  dimension.
  - If only `<DeltaX>` is specified, then a uniform discretization is applied along all directions.
- Element `<DeltaY>` (optional)

- Specifies discretization along the  $y$  dimension.
- **Element <DeltaZ> (optional)**
  - Specifies discretization along the  $z$  dimension.
- **Element <AutoTimeSubStep> (optional)**
  - Enables time sub-stepping to optimize solver performance.
  - Only reliable when all reaction expressions of a field are independent of the field.
  - When enabled, simulation steps are explicitly integrated using maximum stable sub-steps.
  - Note that the derived stability condition (Scarborough) is sufficient but not necessary, so greater time steps than those calculated may be stable, but are not guaranteed to be stable.
- **Element <FluctuationCompensator> (optional)**
  - Enables deployment of the CC3D FluctuationCompensator.
- **Element <DiffusionField>**
  - Defines a diffusion field
  - **Attribute Name:** the name of the field
  - **Element <DiffusionData>**
    - \* **Element <DiffusionConstant> (optional)**
      - Specifies a constant diffusion coefficient for the medium.
      - **Value:** value of the diffusion coefficient
    - \* **Element <DiffusionCoefficient> (optional)**
      - Specifies a constant diffusion coefficient for a cell type.
      - Can be set per cell during simulation execution
      - **Attribute CellType:** name of the cell type
      - **Value:** value of the diffusion coefficient
    - \* **Element <DiffusivityByType> (optional)**
      - Imposes diffusion coefficients according to occupying ID label
      - Each cell and the medium is initialized with a diffusivity specified by <DiffusionCoefficient> and <DiffusionConstant>, respectively.
      - Without any other diffusion mode specification, <DiffusionConstant> is everywhere applied.
    - \* **Element <DiffusivityFieldInMedium> (optional)**
      - Activates editable field diffusivity in the medium.
    - \* **Element <DiffusivityFieldEverywhere> (optional)**
      - Activates editable field diffusivity in the simulation domain.
      - The diffusivity field of a field with name “Field” can be accessed in Python as a concentration field with the name “FieldDiff”.
      - Diffusion mode precedence is <DiffusivityFieldEverywhere> over <DiffusivityFieldInMedium> over <DiffusivityByType> over constant diffusion.

- \* **Element <SimplePermInt> (optional)**
  - Activates simple interface transport flux at cell-cell and cell-medium interfaces.
- \* **Element <PermIntCoefficient> (optional)**
  - Specifies a permeation coefficient for the interface of two cell types (denoted  $P_j$ ).
  - Can be accessed per cell and modified during simulation execution.
  - Value defaults to zero.
  - **Attribute Type1:** name of the first cell type, or Medium
  - **Attribute Type2:** name of the second cell type, or Medium
  - **Value:** value of the coefficient
- \* **Element <PermIntBias> (optional)**
  - Specifies a permeability bias coefficient for the interface of two types (denoted  $b_j$ ).
  - Can be accessed per cell and modified during simulation execution.
  - Value defaults to one.
  - **Attribute Type1:** name of the first cell type, or Medium
  - **Attribute Type2:** name of the second cell type, or Medium
  - **Value:** value of the coefficient
- \* **Element <InitialConcentrationExpression> (optional)**
  - String expression for the initial concentration
  - **Value:** initial concentration expression (e.g.,  $x*y+10*z$ )
- **Element <SecretionData> (optional)**
  - \* Secretion data elements, defined in the same way as for [FlexibleDiffusionSolverFE](#)
- **Element <ReactionData> (optional)**
  - \* **Element <ExpressionSymbol> (optional)**
    - Declares a custom symbol for the field in reaction expressions.
    - Can be used to refer to a field in reactions defined for other fields.
    - Value defaults to the field name + “ExpSym” (e.g., `MyFieldExpSym`).
    - Only one can be defined per field.
    - **Value:** expression symbol (e.g., `MyField`)
  - \* **Element <ExpressionMult> (optional)**
    - Defines an expression for the field-dependent reaction (denoted  $A_j$ ).
    - **Value:** reaction expression (e.g.,  $10*MyOtherField$ )
  - \* **Element <ExpressionIndep> (optional)**
    - Defines an expression for the field-independent reaction (denoted  $B_j$ ).
    - **Value:** reaction expression (e.g., `MyOtherField-20`)
- **Element <BoundaryConditions> (optional)**
  - \* Boundary condition elements, defined in the same as for [DiffusionSolverFE](#).

- \* Boundary conditions are applied at surfaces and can be manipulated at each site during simulation execution.
  - \* If a condition is not specified for a boundary, then it is assumed to be zero flux.
- \* **BoundaryConditions options:**

- NoFlux (default)
- ConstantValue
- ConstantDerivative
- Periodic

The RDFVM solver provides a runtime interface for manipulating various model features during a simulation from a steppable. In general, the RDFVM solver is accessible during simulations that use it in Python from any steppable using the attribute `reaction_diffusion_solver_fvm`,

```
from cc3d.core.PySteppables import *
from cc3d.cpp import CompuCell

class MySteppable(SteppableBasePy):

    def start(self):
        # Reference to the reaction diffusion finite volume solver, or None if the
        # solver is not loaded
        rd_fvm = self.reaction_diffusion_solver_fvm
        # Get the diffusivity field for field with name "MyField" and set some values
        my_field_diff = self.field.MyFieldDiff
        for x in range(self.dim.x):
            my_field_diff[x, 0, 0] *= 2.0
        # Make the bottom boundary concentration of the field a linear function
        for x in range(self.dim.x):
            rd_fvm.useFixedConcentration("MyField", "MinY", x / (self.dim.x - 1), 0)
        # Use permeable membrane transport at the left boundary volume elements
        for y in range(self.dim.y):
            rd_fvm.usePermeableSurface("MyField", "MaxX", CompuCell.Point3D(0, y, 0))
        # Increase the diffusivity in cell 1
        cell_1 = self.fetch_cell_by_id(1)
        cell_diff = rd_fvm.getCellDiffusivityCoefficient(cell_1, "MyField")
        rd_fvm.setCellDiffusivityCoefficient(cell_1, "MyField", 2 * cell_diff)
        # Increase the permeation coefficient between cell 1 and cells of type "Type2"
        perm_cf, cell_type1_bias, cell_type2_bias = rd_fvm.getPermeableCoefficients(cell_
        -1,
                                         self.
        -cell_type.Type2,
        -"MyField")
        rd_fvm.setCellPermeationCoefficient(cell_1, self.cell_type.Type2, "MyField", 2 *
        -perm_cf)
```

The boundary conditions of each volume element can be set, modified and unset during simulation. In general, a volume element can be selected by location using a CC3D Point3D, and a surface of a volume can be selected using the following names,

- MinX: surface with outward-facing normal pointing towards the negative  $x$  direction.

- **MaxX**: surface with outward-facing normal pointing towards the positive  $x$  direction.
- **MinY**: surface with outward-facing normal pointing towards the negative  $y$  direction.
- **MaxY**: surface with outward-facing normal pointing towards the positive  $y$  direction.
- **MinZ**: surface with outward-facing normal pointing towards the negative  $z$  direction.
- **MaxZ**: surface with outward-facing normal pointing towards the positive  $z$  direction.

### 31.10.4 Python Reference

The RDFVM solver provides methods for setting the following mass transport laws and conditions on the basis of individual volume element surface during simulation execution.

- **useDiffusiveSurface(field\_name: string, surface\_name: string, pt: CompuCell.Point3D)**
  - Use diffusive transport on a surface of a volume
  - **field\_name**: name of the field
  - **surface\_name**: name of the surface
  - **pt**: location of the volume
- **useDiffusiveSurfaces(field\_name: string, pt: CompuCell.Point3D)**
  - Use diffusive transport on all surfaces of a volume
  - **field\_name**: name of the field
  - **pt**: location of the volume
- **usePermeableSurface(field\_name: string, surface\_name: string, pt: CompuCell.Point3D)**
  - Use permeable membrane transport on a surface of a volume
  - **field\_name**: name of the field
  - **surface\_name**: name of the surface
  - **pt**: location of the volume
- **usePermeableSurfaces(field\_name: string, pt: CompuCell.Point3D)**
  - Use permeable membrane transport on all surfaces of a volume
  - **field\_name**: name of the field
  - **pt**: location of the volume
- **useFixedFluxSurface(field\_name: string, surface\_name: string, outward\_val: float, pt: CompuCell.Point3D)**
  - Use a fixed flux condition on a surface of a volume
  - **field\_name**: name of the field
  - **surface\_name**: name of the surface
  - **outward\_val**: value of the flux, oriented outward from the volume
  - **pt**: location of the volume
- **useFixedConcentration(field\_name: string, surface\_name: string, conc\_val: float, pt: CompuCell.Point3D)**
  - Use a fixed concentration condition on a surface of a volume
  - **field\_name**: name of the field

- `surface_name`: name of the surface
- `conc_val`: value of the concentration on the surface
- `pt`: location of the volume
- **`useFixedFVConcentration(field_name: string, conc_val: float, pt: CompuCell.Point3D)`**
  - Use a fixed concentration condition in a volume
  - `field_name`: name of the field
  - `surface_name`: name of the surface
  - `conc_val`: value of the concentration in the volume
  - `pt`: location of the volume

The RDFVM solver also provides methods for setting cell-based model parameters for transport laws, which are applied according to the transport laws and boundary conditions of each volume occupied by a cell.

- **`getCellDiffusivityCoefficient(cell: CompuCell.CellG, field_name: string)`**
  - Get the diffusion coefficient of a cell for a field
  - `cell`: a cell
  - `field_name`: name of the field
  - Returns (`float`): value of diffusion coefficient
- **`setCellDiffusivityCoefficient(cell: CompuCell.CellG, field_name: string, diffusion_coefficient: float)`**
  - Set the diffusion coefficient of a cell for a field
  - `cell`: a cell
  - `field_name`: name of the field
  - `diffusion_coefficient`: value of the diffusion coefficient
- **`getPermeableCoefficients(cell: CompuCell.CellG, ncell_type: int, field_name: string)`**
  - Get the permeation coefficient and bias coefficient of a cell for permeable membrane transport with neighbor cells of a type
  - `cell`: a cell
  - `ncell_type`: type ID of the type of neighbor cells
  - `field_name`: name of the field
  - Returns (`float, float`): permeation and bias coefficients
- **`getPermeableCoefficients(cell: CompuCell.CellG, ncell: CompuCell.CellG, field_name: string)`**
  - Get the permeation coefficient of a cell and bias coefficients of a cell and a neighboring cell for permeable membrane transport with a neighboring cell
  - `cell`: a cell
  - `ncell`: a neighbor cell
  - `field_name`: name of the field
  - Returns (`float, float, float`): permeation coefficient and bias coefficients of a cell, and bias coefficient of a neighboring cell
- **`setCellPermeationCoefficient(cell: CompuCell.CellG, ncell_type: int, field_name: string, permeation_coefficient: float)`**

- Set the permeation coefficient of a cell for permeable membrane transport with neighbor cells of a type
- **cell**: a cell
- **ncell\_type** type ID of the type of neighbor cells
- **field\_name**: name of the field
- **permeation\_coefficient**: value of the permeation coefficient
- **setCellPermeableBiasCoefficient(cell: CompuCell.CellG, ncell\_type: int, field\_name: string, bias\_val: float)**
  - Set the bias coefficient of a cell for permeable membrane transport with neighbor cells of a type
  - **cell**: a cell
  - **ncell\_type**: type ID of the type of neighbor cells
  - **field\_name**: name of the field
  - **bias\_val**: value of the bias coefficient

## 31.11 Steady State diffusion solver

Often in the multi-scale simulations we have to deal with chemicals which have drastically different diffusion constants. For slow diffusion fields we can use standard explicit solvers (*e.g.* `FlexibleDiffusionSolverFE`) but once the diffusion constant becomes large the number of extra calls to explicit solvers becomes so large that solving diffusion equation using Forward-Euler based solvers is simply impractical. In situations where the diffusion constant is so large that the solution of the diffusion equation is not that much different from the asymptotic solution (*i.e.* at  $t = \infty$ ) it is often more convenient to use `SteadyStateDiffusion` solver which solves Helmholtz equation:

$$\nabla^2 c - kc = F \quad (31.7)$$

where  $F$  is a source function of the coordinates - it is an input to the equation,  $k$  is decay constant and  $c$  is the concentration. The  $F$  function in CC3D is either given implicitly by specifying cellular secretion or explicitly by specifying concentration  $c$  before solving Helmholtz equation.

The CC3D steady state diffusion solvers are stable and allow solutions for large values of diffusion constants. The example syntax for the steady-state solver is shown below:

```
<Steppable Type="SteadyStateDiffusionSolver2D">
  <DiffusionField Name="INIT">
    <DiffusionData>
      <FieldName>INIT</FieldName>
      <DiffusionConstant>1.0</DiffusionConstant>
      <DecayConstant>0.01</DecayConstant>
    </DiffusionData>
    <SecretionData>
      <Secretion Type="Body1">1.0</Secretion>
    </SecretionData>

    <BoundaryConditions>

      <Plane Axis="X">
        <ConstantValue PlanePosition="Min" Value="10.0"/>
        <ConstantValue PlanePosition="Max" Value="5.0"/>
      </Plane>
    </BoundaryConditions>
  </DiffusionField>
</Steppable>
```

(continues on next page)

(continued from previous page)

```

<Plane Axis="Y">
    <ConstantDerivative PlanePosition="Min" Value="0.0"/>
    <ConstantDerivative PlanePosition="Max" Value="0.0"/>
</Plane>

</BoundaryConditions>

</DiffusionField>

</Steppable>

```

The syntax is similar (actually, almost identical) to the syntax of the `FlexibleDiffusionSolverFE`. The only difference is that while `FlexibleDiffusionSolverFE` works in both 2D and 3D users need to specify the dimensionality of the steady state solver. We use

```
<Steppable Type="SteadyStateDiffusionSolver2D">
```

for 2D simulations when all the cells lie in the `xy` plane and

```
<Steppable Type="SteadyStateDiffusionSolver">
```

for simulations in 3D.

### Note

We can use Python to control `secretion` in the steady state solvers but it requires a little bit of low level coding. Implementing secretion in steady state diffusion solver is different from “regular” Forward Euler solvers. Steady state solver takes secretion rate that is specified at  $t=0$  and returns the solution at  $t=\infty$ . For a large diffusion constants we approximate solution to the PDE during one MCS by using solution at  $t=\infty$ . However, this means that if at each MCS secretion changes we have to do three things 1) zero entire field, 2) set secretion rate 3) solve steady state solver. The reason we need to zero entire field is because any value left in the field at  $mcs=N$  is interpreted by the solver as a secretion constant at this location at  $mcs=N+1$ . **Moreover, the the secretion constant needs to have negative value if we want to secrete positive amount of substance - this weird requirements comes from the fact that we re using 3:sup:`rd` party solver which inverts signs of the secretion constants.**

An example below demonstrates how we control secretion of the steady state in Python. First we need to include tag `<ManageSecretionInPython/>` in the XML definition of the solver:

```

<Steppable Type="SteadyStateDiffusionSolver2D">
    <DiffusionField>
        <ManageSecretionInPython/>
        <DiffusionData>
            <FieldName>FGF</FieldName>
            <DiffusionConstant>1.00</DiffusionConstant>
            <DecayConstant>0.00001</DecayConstant>
        </DiffusionData>
    </DiffusionField>
</Steppable>

```

In Python the code to control the secretion involves iteration over every pixel and adjusting concentration (which as we mentioned will be interpreted by the solver as a secretion constant) and we have to make sure that we inherit from

SecretionBasePy not SteppableBasePy to ensure proper ordering of calls to Python module and the C++ diffusion solver.

### Note

Make sure you inherit from `SecretionBasePy` when you try to manage secretion in the steady state solver using Python. This will ensure proper ordering of calls to steppable and to C++ solver code.

### Note

Once you use `<ManageSecretionInPython/>` tag in the XML all secretion tags in the `SecretionData` will be ignored. In other words, for this solver you cannot mix secretion specification in Python and secretion specification in the XML.

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

def start(self):
    self.field = CompuCell.getConcentrationField \
        (_simulator, "FGF")

    secrConst = 10
    for x, y, z in self.everyPixel(1, 1, 1):
        cell = self.cellField[x, y, z]
        if cell and cell.type == 1:
            self.field[x, y, z] = -secrConst
        else:
            self.field[x, y, z] = 0.0

def step(self, mcs):
    secrConst = mcs
    for x, y, z in self.everyPixel(1, 1, 1):
        cell = self.cellField[x, y, z]
        if cell and cell.type == 1:
            self.field[x, y, z] = -secrConst
        else:
            self.field[x, y, z] = 0.0
```

### Warning

Notice that all the pixels that do not secrete **have to be 0.0** as mentioned above. **If you don't initialize field values in the non-secreting pixels to 0.0 you will get wrong results.** The above code, with comments, is available in our Demo suite (Demos/SteppableDemos/SecretionSteadyState or Demos/SteppableDemos/SteadyStateDiffusionSolver).

## 31.12 Fluctuation Compensator Solver Add-On

Fluctuation Compensator is an optional PDE solver add-on introduced in v4.1.2 to account for Metropolis surface fluctuations in field solutions in both a feasible and sensible way. The algorithm is based on that which is described by Marée et. al<sup>1</sup> and imposes total mass conservation in all cellular domains and the medium over the spin flips of a Monte Carlo step. The algorithm does not impose advection by domain deformation, but rather homogeneously applies a correction factor in each subdomain to all field solutions of a solver such that the total amount of each simulated species in each subdomain is unchanged over all spin flips.

The Fluctuation Compensator algorithm consists of the following three rules.

1. **Rule 1 (mass conservation):** In each subdomain, the total amount of each species is unchanged over an arbitrary number of spin flips.
2. **Rule 2 (uniform correction):** Corrections applied to values in each subdomain so to impose mass conservation are uniformly applied.
3. **Rule 3 (Neumann condition):** The amount of species in the copying site of a spin flip are exactly copied to the site of the flip.

### Note

Fluctuation Compensator is supported in DiffusionSolverFE, ReactionDiffusionSolverFE and ReactionDiffusion-SolverFVM.

Exactly one Fluctuation Compensator can be attached to each supported solver instance. An attached Fluctuation Compensator performs corrections on *all* fields of the solver to which it is attached. A Fluctuation Compensator can be attached to a solver in CC3DML using the tag `<FluctuationCompensator>` at the level of field specification. For example, to attach a Fluctuation Compensator to a simple use-case with DiffusionSolverFE might look like the following.

```
<Steppable Type="DiffusionSolverFE">
  <FluctuationCompensator/>
  <DiffusionField Name="ATTR"/>
</Steppable>
```

Demos showing basic usage and comparison are available in [Demos/SteppableDemos/FluctuationCompensator](#).

### Note

PDE solution field values can be modified outside of the solver routines without invalidating the correction factors of Fluctuation Compensators *so long as* they are notified that field values have been modified. The CompuCell3D library has a convenience method to do exactly this: `updateFluctuationCompensators`. Call this method after modifying field values and before the next PDE solution step to refresh Fluctuation Compensators according to your changes.

<sup>1</sup> Marée, Athanasius FM, Verônica A. Grieneisen, and Leah Edelstein-Keshet. "How cells integrate complex stimuli: the effect of feedback from phosphoinositides and cell shape on cell polarization and motility." PLoS Computational Biology 8.3 (2012).



---

CHAPTER  
THIRTYTWO

---

## POTTS AND LATTICE SECTION

### 32.1 Potts Algorithm: How It Works

#### Learning Objectives:

- Understand how the Potts algorithm determines cell shape by controlling their pixels
- 

The first section of the .xml file defines the global parameters of the lattice and the simulation. Your file may look much simpler than this.

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

This section appears at the beginning of the configuration file. The line reading `<Dimensions x="101" y="101" z="1"/>` declares the dimensions of the lattice to be 101 by 101 by 1 pixels. Since z is small, this lattice is two-dimensional. Lattice sites are 0-indexed, which means that we count them from 0 to 100 when accessing them in Python.

`<Steps>1000</Steps>` tells CompuCell how long the simulation lasts in Monte Carlos Steps (MCS). After executing this number of steps, CompuCell can simulate with zero temperature for an additional period. In our case, it will run for `<Anneal>10</Anneal>` extra steps.

### 32.1.1 Fluctuation Amplitude (Temperature)

The `FluctuationAmplitude`/`Temperature` parameter determines the intrinsic fluctuation or motility of each cell membrane. (Either key word works in XML) Try setting a high `Temperature` value in XML. You should see much more activity from pixel changes. Conversely, a low `Temperature` will decrease cell membrane activity.

#### Note

`FluctuationAmplitude` is a `Temperature` parameter in classical GGH Model formulation. We have decided to use `FluctuationAmplitude` term instead of temperature because using the word `Temperature` to describe the intrinsic motility of cell membranes was quite confusing.

In the above example, fluctuation amplitude applies to all cells in the simulation. To define fluctuation amplitude separately for each cell type, we use the following syntax. Each value must be non-negative.

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

When CompuCell3D encounters the expanded definition of `FluctuationAmplitude`, it will use it in place of a global definition like this one:

```
<FluctuationAmplitude>5</FluctuationAmplitude>
```

Alternatively, you can use Python to set the fluctuation amplitude individually for each cell:

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

When determining which value of fluctuation amplitude to use, CompuCell prioritizes Python definitions. Otherwise, if `fluctAmpl` was not set by Python, it will try to use the CC3DML for fluctuation amplitude by cell types. Lastly, it will resort to a globally defined fluctuation amplitude (`Temperature`). If none of these are defined, the default value is used. Thus, it is perfectly fine to use a combination of these techniques.

#### Note

Default Value of Fluctuation Amplitude

For XML-based simulations, the **default value** of `FluctuationAmplitude`/`Temperature` is **0**. For Python-only (PyCoreSpecs) simulations, the **default value** of `fluctuation_amplitude` is **10**.

Note that a value of 0 does not turn off activity; there will still be slight fluctuations in cell membranes.

In the Glazier-Graner-Hogeweg (GGH) Model, the fluctuation amplitude is determined by taking into account the fluctuation amplitude of a “*source*” (expanding) cell and a “*destination*” cell (the one that will be overwritten).

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

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

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

The above concepts are best illustrated by the following example:

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

Where in the CC3DML section we define global fluctuation amplitude and we also use `ArithmeticAverage` function to determine resultant fluctuation amplitude for the pixel copy.

Try this Python script to see how fluctuation amplitude affects the membranes of cells. This code assigns a different `fluctAmpl` value depending on which of 4 quadrants each cell is located in.

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

        self.quarters = [[0, 0, 50, 50], [0, 50, 50, 100], [50, 50, 100, 100], [50, 0, 100, 50]]

        self.stepableCallCounter = 0

    def step(self, mcs):

        quarterIndex = self.stepableCallCounter % 4
        quarter = self.quarters[quarterIndex]

        for cell in self.cellList:

            if cell.xCOM >= quarter[0] and cell.yCOM >= quarter[1] and cell.xCOM < quarter[2] and cell.yCOM < quarter[3]:
                cell.fluctAmpl = 50
            else:
                # this means CompuCell3D will use globally defined FluctuationAmplitude
                cell.fluctAmpl = -1

        self.stepableCallCounter += 1
```

Similarly, `fluctuation_amplitude` can be set in a Python-only simulation:

```
from cc3d import CompuCellSetup
from cc3d.core.PyCoreSpecs import Metadata, PottsCore
```

(continues on next page)

(continued from previous page)

```
spec_potts = PottsCore()
spec_potts.dim_x, spec_potts.dim_y = 100, 100
spec_potts.steps = 100000
spec_potts.neighbor_order = 2
spec_potts.fluctuation_amplitude = 50
```

OR

```
from cc3d import CompuCellSetup
from cc3d.core.PyCoreSpecs import Metadata, PottsCore

CompuCellSetup.register_specs(PottsCore(dim_x=dim_x,
                                       dim_y=dim_y,
                                       steps=100000,
                                       neighbor_order=2,
                                       boundary_x="Periodic",
                                       boundary_y="Periodic",
                                       fluctuation_amplitude=50))
```

Remember, negative values of `fluctuationAmplitude` are ignored. Here, `cell.fluctAmpl = -1` is a hint to CC3D to use fluctuation amplitude defined in the CC3DML.

Let us revisit our original example of the `Potts` section CC3DML:

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

Based on our discussion about the difference between pixel-flip attempts and MCS (see “Introduction to CompuCell3D”), we can specify how many pixel copies should be attempted in every MCS. We specify this number indirectly by specifying the `Flip2DimRatio` by using

```
<Flip2DimRatio>1</Flip2DimRatio>
```

which tells CompuCell that it should make 1 times number of lattice sites attempts per MCS – in our case one MCS is 101x101x1 pixel-copy attempts. To set 2.5 x 101 x 101 x 1 pixel-copy attempts per MCS you would write:

**<Flip2DimRatio>2.5</Flip2DimRatio>**

The line beginning with `<NeighborOrder>2</NeighborOrder>` specifies the neighbor order. Neighbor order controls how many nearby pixels the Potts algorithm will check each time it needs to do a pixel copy attempt. Think of the neighbors as a circular area around each pixel. If you set a higher neighbor order, you may have smoother cells but less performance.

In the previous example, the pixel neighbors are ranked according to their distance from a reference pixel (*i.e.* the one you are measuring a distance from). Thus, we can group the 1<sup>st</sup>, 2<sup>nd</sup>, and 3<sup>rd</sup> nearest neighbors for every pixel in the lattice. Using 1<sup>st</sup> nearest neighbor interactions may cause unwanted artifacts due to lattice anisotropy. The longer the interaction range, (*i.e.* 2<sup>nd</sup>, 3<sup>rd</sup> or higher `NeighborOrder`), the more isotropic the simulation and the slower it runs. In addition, if the interaction range is comparable to the cell size, you may generate unexpected effects, since non-adjacent cells will contact each other.

On a hex lattice, those problems seem to be less severe and there 1<sup>st</sup> or 2<sup>nd</sup> nearest neighbor usually are sufficient.

5	4	3	4	5
4	2	1	2	4
3	1	X	1	3
4	2	1	2	4
5	4	3	4	5

Fig. 1: Ranking of pixel neighbors on square 2D lattice

The Potts section also contains tags called `<Boundary_y>` and `<Boundary_x>`. These tags impose boundary conditions on the lattice. In this case, the x and y axes are **periodic**.

**Periodic Boundary Conditions:** cause the edges of the simulation area to “wrap around.” For example, a pixel at ( $x=0$ ,  $y=1$ ,  $z=1$ ) will neighbor the pixel at ( $x=100$ ,  $y=1$ ,  $z=1$ ). We recommend using periodic boundaries when you want to simulate a large area of tissue while keeping your lattice small.

**<Boundary\_x>Periodic</Boundary\_x>**

**‘NoFlux’ Boundary Conditions:** is the opposite of periodic, so the lattice remains a finite area. This is the default.

**<Boundary\_x>NoFlux</Boundary\_x>**

Boundary conditions are independent in each XYZ direction, so you can specify any combination of them you like.

See [Boundary Conditions](#) for more details.

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

**RandomSeed:** is used to initialize the random number generator. You do not need this tag unless you want every simulation to behave exactly the same, which is not recommended. See [Stochasticity and RandomSeed](#) for more details.

**EnergyFunctionCalculator:** allows you to output statistical data, such as the changes in energy from the simulation, to text files for further analysis. See [How to Output Energy Changes](#) for more details.

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

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

This ensures that pixel copy attempts that increase the energy of the system are accepted with probability

$$P = e^{(-\Delta E - \delta)/kT} \quad (32.1)$$

where and  $k$  are specified by Offset and KBoltzman tags, respectively. By default,  $= 0$  and  $k = 1$ . (That is, Offset is 0 and KBoltzman is 1).

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

$$P = 1 - \frac{E - \delta}{kT} \quad (32.2)$$

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

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

## 32.2 Lattice Type

Early versions of CompuCell3D allowed users to use only square lattice. Most recent versions allow the simulation to be run on hexagonal lattice as well.

Full description of hexagonal lattice including detailed derivations can be found in “Introduction to Hexagonal Lattices” available from [http://www.compcell3d.org/BinDoc/cc3d\\_binaries/Manuals/HexagonalLattice.pdf](http://www.compcell3d.org/BinDoc/cc3d_binaries/Manuals/HexagonalLattice.pdf)

To enable hexagonal lattice you need to put

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

in the Potts section of the CC3DML configuration file.

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

$$\text{sition: } \left\{ \begin{array}{l} [x_{hex}, y_{hex}, z_{hex}] = \left[ (x_{cart} + \frac{1}{2})L, \frac{\sqrt{3}}{2}y_{cart}L, \frac{\sqrt{6}}{3}z_{cart}L \right] \text{ for } y \bmod 2 = 0 \text{ and } z \bmod 3 = 0 \\ [x_{hex}, y_{hex}, z_{hex}] = \left[ x_{cart}L, \frac{\sqrt{3}}{2}y_{cart}L, \frac{\sqrt{6}}{3}z_{cart}L \right] \text{ for } y \bmod 2 = 1 \text{ and } z \bmod 3 = 0 \\ [x_{hex}, y_{hex}, z_{hex}] = \left[ x_{cart}L, \left( \frac{\sqrt{3}}{2}y_{cart} + \frac{\sqrt{3}}{6} \right)L, \frac{\sqrt{6}}{3}z_{cart}L \right] \text{ for } y \bmod 2 = 0 \text{ and } z \bmod 3 = 1 \\ [x_{hex}, y_{hex}, z_{hex}] = \left[ (x_{cart} + \frac{1}{2})L, \left( \frac{\sqrt{3}}{2}y_{cart} + \frac{\sqrt{3}}{6} \right)L, \frac{\sqrt{6}}{3}z_{cart}L \right] \text{ for } y \bmod 2 = 1 \text{ and } z \bmod 3 = 1 \\ [x_{hex}, y_{hex}, z_{hex}] = \left[ x_{cart}L, \left( \frac{\sqrt{3}}{2}y_{cart} - \frac{\sqrt{3}}{6} \right)L, \frac{\sqrt{6}}{3}z_{cart}L \right] \text{ for } y \bmod 2 = 0 \text{ and } z \bmod 3 = 2 \\ [x_{hex}, y_{hex}, z_{hex}] = \left[ (x_{cart} + \frac{1}{2})L, \left( \frac{\sqrt{3}}{2}y_{cart} - \frac{\sqrt{3}}{6} \right)L, \frac{\sqrt{6}}{3}z_{cart}L \right] \text{ for } y \bmod 2 = 1 \text{ and } z \bmod 3 = 2 \end{array} \right.$$

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

$$S_{hex-unit} = \sqrt{\frac{2}{3\sqrt{3}}} \approx 0.6204 \quad (32.3)$$

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

$$L_{hex-unit} = \sqrt{\frac{2}{\sqrt{3}}} \approx 1.075 \quad (32.4)$$

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

$$S = 6\frac{\sqrt{3}}{4}a^2 \quad (32.5)$$

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

$$\begin{aligned} V &= \frac{16}{9}\sqrt{3}a^3 \\ S &= 8\sqrt{2}a^2 \end{aligned}$$

where  $a$  denotes length of dodecahedron edge.

Constraining the volume to be 1 we get:

$$a = \sqrt[3]{\frac{9V}{16\sqrt{3}}} \quad (32.6)$$

and thus unit surface is given by:

$$S_{unit-hex} = \frac{S}{12} = \frac{8\sqrt{2}}{12} \sqrt[3]{\frac{9V}{16\sqrt{3}}} \approx 0.445 \quad (32.7)$$

and unit length by:

$$L_{unit-hex} = 2\frac{\sqrt{2}}{\sqrt{3}}a = 2\frac{\sqrt{2}}{\sqrt{3}}\sqrt[3]{\frac{9V}{16\sqrt{3}}} \approx 1.122 \quad (32.8)$$

## 32.3 Stochasticity and RandomSeed

A CC3DML file may contain a `<RandomSeed>` tag like this:

```
<Potts>
  <Dimensions x="101" y="101" z="1"/>
  <Steps>100000</Steps>
  <Temperature>10.0</Temperature>
  <NeighborOrder>2</NeighborOrder>
  <RandomSeed>167473</RandomSeed>
</Potts>
```

If you *do not include this tag* and a value, then a *different* random seed will be chosen for each run by the operating system and each simulation will use a *different set of random numbers*, meaning that each simulation will produce a **different result**. In general, this is the behavior you want. CompuCell3D simulations are meant to be stochastic, and a simulation represents a plausible trajectory of the system through time. So, generally, you will **not** include a <RandomSeed> element.

However, in some cases, you want to force CompuCell3D to use the same random seed for multiple runs so that it will produce the same output every time. This is often useful for debugging or for when you are doing parameter estimation or data fitting. In these cases, you include the <RandomSeed> tag with an integer of your choice. Every simulation with the same seed will produce the same results. Note though that if you also use a random number generator in your Python code then you will need to specify a random seed there as well. The two seeds do not have to be the same. For example, in your Python Steppables file you might include:

```
import random
random.seed(54321)
```

This makes your Python code use the same series of random numbers for every run, eliminating the stochasticity that would otherwise occur between simulations.

## 32.4 How to Output Energy Changes

**EnergyFunctionCalculator:** allows you to output statistical data from the simulation for further analysis.

You can define this in XML like so:

```
<Potts>
  <Dimensions x="256" y="256" z="1"/>
  <Steps>100000</Steps>
  <Temperature>5</Temperature>
  <NeighborOrder>2</NeighborOrder>
  <EnergyFunctionCalculator Type="Statistics">
    <OutputFileName Frequency="10">statData.txt</OutputFileName>
    <OutputCoreFileNameSpinFlips Frequency="1" GatherResults="" OutputAccepted="" OutputRejected="" OutputTotal="" />
  </EnergyFunctionCalculator>
</Potts>
```

### Note

CC3D has the option to run in parallel mode, but output from the energy calculator will only work when running in single-CPU mode.

The **OutputFileName** tag is used to specify the name of the file to which CompuCell3D will write average changes in energies returned by each plugin along with the corresponding standard deviations. **Frequency** controls how often to record this data. In the above example, we would write data every 10 MCS.

Furthermore, the **OutputCoreFileNameSpinFlips** tag is used to tell CompuCell3D to output the energy change for every plugin's pixel-copy operations.

Finally, **GatherResults=""** will ensure that there is only one file written for accepted (**OutputAccepted**), rejected (**OutputRejected**), and both accepted and rejected (**OutputTotal**) pixel copies. If you do not specify **GatherResults**, CompuCell3D will output separate files for different MCS's, and depending on the **Frequency**, you may end up with many files in your directory.

---

CHAPTER  
THIRTYTHREE

---

## ALGORITHMS & HOW THEY WORK SECTION

This section covers some of the computational algorithms that we use in CC3D.

### 33.1 Calculating Inertia Tensor in CC3D

Related: [Calculating the Shape Constraint of a Cell – the Elongation Term](#) and [MomentOfInertia Plugin](#)

**Learning Objectives:**

- Learn how CC3D calculates diagonal and off-diagonal inertia tensors

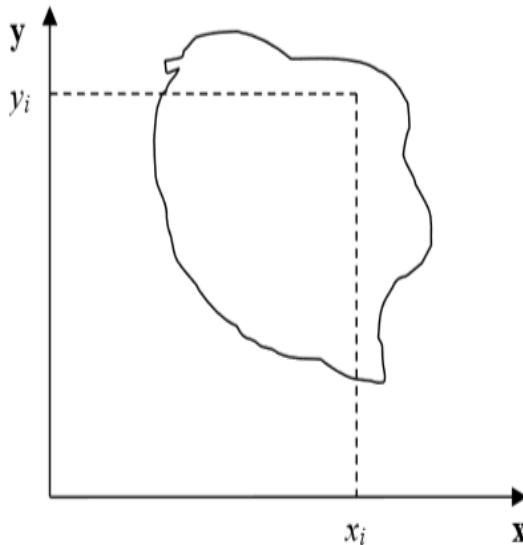
Prerequisite: [Wikipedia: What is a Moment of Inertia?](#)

---

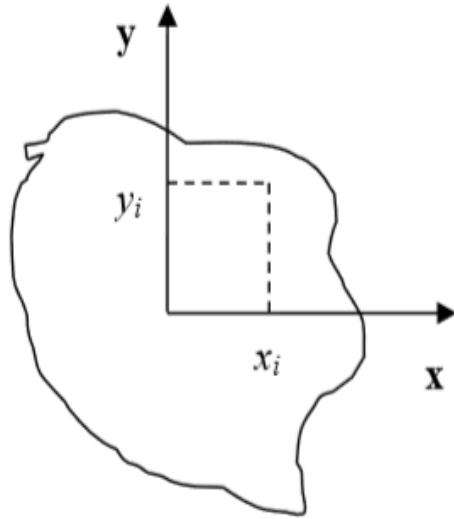
For each cell, the inertia tensor is defined as follows:

$$I = \begin{bmatrix} \sum_i y_i^2 + z_i^2 & -\sum_i x_i y_i & -\sum_i x_i z_i \\ -\sum_i x_i y_i & \sum_i x_i^2 + z_i^2 & -\sum_i y_i z_i \\ -\sum_i x_i z_i & -\sum_i y_i z_i & \sum_i x_i^2 + y_i^2 \end{bmatrix} \quad (33.1)$$

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



**Figure 3:** Cell and coordinate system passing through the center of mass of a cell. Notice that as the cell changes shape, the position of the center of mass moves.



**Figure 4:** Cell and its coordinate frame in which we calculate inertia tensor

In Figure 4, we show one possible coordinate frame in which one can calculate the inertia tensor. If the coordinate frame is fixed calculating components of inertia tensor for cell gaining or losing one pixel is quite easy. We will be adding and subtracting terms like  $y_i^2 + z_i^2$  or  $x_i z_i$ .

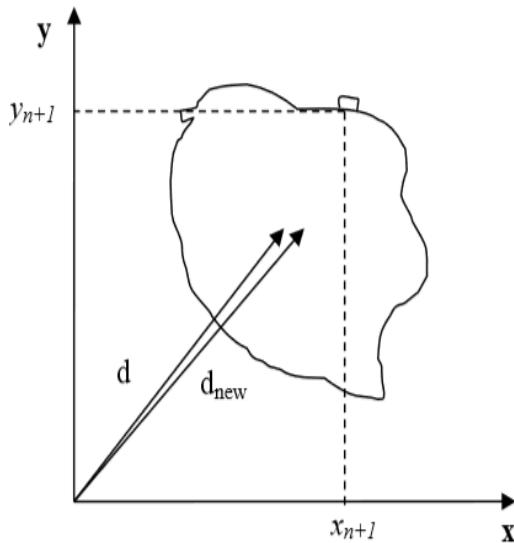
However, in CompuCell3D, we are mostly interested in knowing the tensor of inertia of a cell with respect to the xyz coordinate frame with origin at the center of mass (*COM*) of a given cell as shown in Figure 3. Now, to calculate this tensor, we cannot simply add or subtract terms  $y_i^2 + z_i^2$  or  $x_i z_i$  to account for a lost or gained pixel. If a cell gains or loses a pixel, its COM coordinates change. If so, then all the  $x_i, y_i, z_i$  coordinates that appear in the inertia tensor expression will have different values. Thus, for each change in cell shape (gain or loss of pixel) we would have to recalculate the inertia tensor from scratch. This would be quite time consuming and would require us to keep track of all the pixels belonging to a given cell. It turns out, however, that there is a better way of keeping track of inertia tensor for cells.

We will be using the parallel axis theorem to do the calculations. The parallel axis theorem states that if  $I_{COM}$  is a moment of inertia with respect to the axis passing through the center of mass, then we can calculate moment of inertia with respect to any parallel axis to the one passing through the COM by using the following formula:

$$I_{x'x'} = I_{xx} + M d^2 \quad (33.2)$$

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

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



**Figure 5:** Cell gaining one pixel denotes a distance from the origin of a fixed frame of reference to the center of mass of a cell before the cell gains a new pixel.  $d_{new}$  denotes the same distance but after the cell gains a new pixel

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

$$I_{xx}^{new} = I_{x'x'}^{new} - (V + 1)d_{new}^2 \quad (33.3)$$

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

$$I_{xx}^{new} = I_{x'x'}^{old} + y_{n+1}^2 + z_{n+1}^2 - Vd^2 + Vd^2(V + 1)d_{new}^2 \quad (33.4)$$

$$= I_{x'x'}^{old} - Vd^2 + y_{n+1}^2 + z_{n+1}^2 + Vd^2(V + 1)d_{new}^2 \quad (33.5)$$

$$I_{xx}^{old} - Vd^2 + y_{n+1}^2 + z_{n+1}^2 + Vd^2(V + 1)d_{new}^2 \quad (33.6)$$

Therefore, we have found an expression for the moment of inertia passing through the center of mass of the cell with the additional pixel. Note that this expression involves a moment of inertia for the old cell (*i.e.* the original cell, not the one with extra pixel). When we add a new pixel, we know its coordinates and we can also easily calculate  $d_{new}$ . Thus, when we need to calculate the moment of inertia for a new cell, instead of performing summation as given in the definition of the inertia tensor, we can use a much simpler expression.

---

This was a diagonal term of the inertia tensor. What about off-diagonal terms? Let us write an explicit expression for  $I_{xy}$ :

$$I_{xy} = - \sum_i^N (x_i - x_{com})(y_i - y_{com}) = - \sum_i^N x_i y_i + x_{COM} \sum_i^N y_i + y_{COM} \sum_i^N x_i - x_{COM} y_{COM} \sum_i^N \quad (33.7)$$

$$= - \sum_i^N x_i y_i + x_{COM} V y_{COM} + y_{COM} V x_{COM} - x_{COM} y_{COM} V \quad (33.8)$$

$$= - \sum_i^N x_i y_i + V x_{COM} y_{COM} \quad (33.9)$$

where  $x_{COM}$ ,  $y_{COM}$  denote x and y center of mass positions of the cell,  $V$  denotes cell volume. In the above formula, we have used the fact that:

$$x_{COM} = \frac{\sum_i x_i}{V} \implies \sum_i x_i = x_{COM} V \quad (33.10)$$

and similarly for the y coordinate.

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

$$I_{xy}^{new} = - \sum_i^{N+1} x_i y_i + (V + 1) x_{COM}^{new} y_{COM}^{new} \quad (33.11)$$

$$= - \sum_i^N x_i y_i + V x_{COM} y_{COM} - x_{COM} V y_{COM} + (V + 1) x_{COM}^{new} y_{COM}^{new} - x_{N+1} y_{n+1} \quad (33.12)$$

$$= I_{xy}^{old} - V x_{COM} y_{COM} + (V + 1) x_{COM}^{new} y_{COM}^{new} - x_{N+1} y_{n+1} \quad (33.13)$$

where we have added and subtracted  $V x_{COM} y_{COM}$  to be able to form  $I_{xy}^{old} - \sum_i^N x_i y_i + V x_{COM} y_{COM}$  on the right hand side of the expression for  $I_{xy}^{new}$ . As was the case for the diagonal element, calculating an off-diagonal of the inertia tensor involves  $\text{L}_{\{xy\}}^{\{old\}}$  and the center of mass of the cell before and after gaining a new pixel. All those quantities are either known *a priori* ( $\text{L}_{\{xy\}}^{\{old\}}$ ) or can be easily calculated (center of mass position after gaining one pixel).

We have shown how we can calculate the tensor of inertia for a given cell with respect to a coordinate frame with origin at a cell's center of mass, without evaluating full sums. Such "local" calculations greatly speed up simulations.

## 33.2 Calculating the Shape Constraint of a Cell – the Elongation Term

Related: [Calculating Inertia Tensor in CC3D](#) and [MomentOfInertia Plugin](#)

### Learning Objectives:

- Learn how CC3D leverages inertia tensors to elongate cells
- 

The shape of a single cell immersed in medium and not subject to too drastic surface or surface constraints will be spherical (circular in 2D). However, in certain situations, we may want to use cells that are elongated along one of their body axes. To facilitate, this we can place constraints on the principal lengths of the cell. In 2D, it is sufficient to constrain one of the principal lengths of the cell. However, in 3D, we need to constrain 2 out of 3 principal lengths.

Our first task is to diagonalize the inertia tensor (i.e. find a coordinate frame transformation which brings the inertia tensor to a diagonal form)

### 33.2.1 Diagonalizing inertia tensor

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

In order to diagonalize the inertia tensor, we need to solve the following eigenvalue equation:

$$\det(I - \lambda) = 0 \quad (33.14)$$

or in full form

$$0 = \begin{vmatrix} \sum_i y_i^2 + z_i^2 - \lambda & -\sum_i x_i y_i & -\sum_i x_i z_i \\ -\sum_i x_i y_i & \sum_i x_i^2 + z_i^2 - \lambda & -\sum_i y_i z_i \\ -\sum_i x_i z_i & -\sum_i y_i z_i & \sum_i x_i^2 + y_i^2 - \lambda \end{vmatrix} = \begin{vmatrix} I_{xx} - \lambda & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} - \lambda & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} - \lambda \end{vmatrix} \quad (33.15)$$

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

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

so in our case, the eigenvalue equation takes the form:

$$-L^2 + L^2(I_{xx} + I_{yy} + I_{zz}) + L(I_{xy}^2 + I_{yz}^2 + I_{xz}^2 - I_{xx}I_{yy} - I_{yy}I_{zz} - I_{xx}I_{zz}) \quad (33.16)$$

$$+ I_{xx}I_{yy}I_{zz} - I_{xx}I_{yz}^2 - I_{yy}I_{zz}^2I_{xz}^2 + 2I_{xy}I_{yz}I_{xz} \quad (33.17)$$

This equation can be solved analytically, again we may use Wikipedia ([http://en.wikipedia.org/wiki/Cubic\\_function](http://en.wikipedia.org/wiki/Cubic_function))

Now, the eigenvalues found that way are the principal moments of inertia of a cell. That is they are components of the inertia tensor in a coordinate frame rotated in such a way that off-diagonal elements of the inertia tensor are 0:  $I =$

$$\begin{vmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{vmatrix} \quad (33.18)$$

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

$$\begin{vmatrix} \frac{1}{5}(b^2 + c^2) & 0 & 0 \\ 0 & \frac{1}{5}(a^2 + c^2) & 0 \\ 0 & 0 & \frac{1}{5}(a^2 + b^2) \end{vmatrix} \quad (33.19)$$

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

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1 \quad (33.20)$$

In other words  $a, b, c$  are half lengths of principal axes (they are analogues of the circle's radius).

Now we can determine semi axes lengths in terms of principal moments of inertia by inverting the following set of equations:

$$I_{xx} = \frac{1}{5}(b^2 + c^2) \quad (33.21)$$

$$I_{yy} = \frac{1}{5}(a^2 + c^2) \quad (33.22)$$

$$I_{zz} = \frac{1}{5}(a^2 + b^2) \quad (33.23)$$

Once we have calculated semiaxes lengths in terms of moments of inertia, we can plug in actual numbers for moment of inertia (the ones for actual cell) and obtain lengths of semiaxes.

Next, we apply a quadratic constraint on largest (semimajor) and smallest (semiminor axes). This is what the elongation plugin does.

### 33.3 Forward Euler method for solving PDE's in CompuCell3D.

 Note

We present more complete derivations of explicit finite difference scheme for diffusion solver in “Introduction to Hexagonal Lattices in CompuCell3D” ([http://www.compcell3d.org/BinDoc/cc3d\\_binaries/Manuals/HexagonalLattice.pdf](http://www.compcell3d.org/BinDoc/cc3d_binaries/Manuals/HexagonalLattice.pdf)).

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

$$\frac{\partial c}{\partial t} = \frac{\partial^2 c}{\partial^2 x} + \frac{\partial^2 c}{\partial^2 y} + \frac{\partial^2 c}{\partial^2 z} \quad (33.24)$$

In a discretized form we may write:

$$\frac{c(x, t + \delta t) - c(x, t)}{\delta t} = \quad (33.25)$$

$$\frac{c(x + \delta x, t) - 2c(x, t) + c(x - \delta x, t)}{\delta x^2} \quad (33.26)$$

$$+ \frac{c(y + \delta y, t) - 2c(y, t) + c(y - \delta y, t)}{\delta y^2} \quad (33.27)$$

$$+ \frac{c(z + \delta z, t) - 2c(z, t) + c(z - \delta z, t)}{\delta z^2} \quad (33.28)$$

where to save space we used shorthand notation:

$$c(x + \Delta x, y, z, t) \equiv c(x + \Delta x, , t)) \quad (33.29)$$

$$c(x, y, z, t) \equiv c(x, t) \quad (33.30)$$

and similarly for other coordinates.

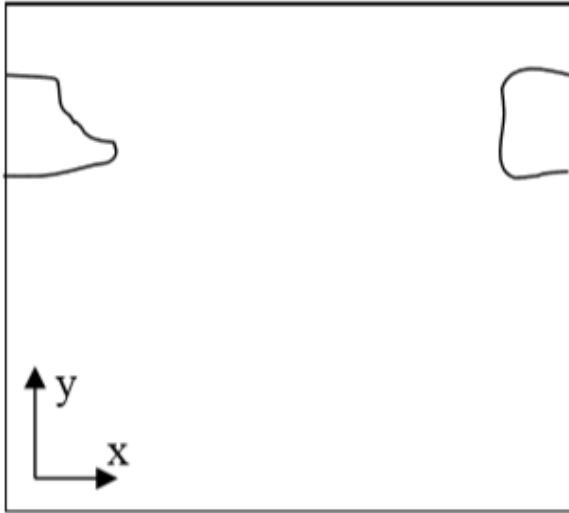
After rearranging terms we get the following expression:

$$c(x, t + \delta t) = \left[ \frac{\delta t}{\delta x^2} \sum_{i=neighbors} (c(i, t) - c(x, t)) \right] - c(x, t) \quad (33.31)$$

where the sum over index  $i$  goes over neighbors of point  $(x, y, z)$  and the neighbors will have the following concentrations:  $c(x + \delta x, t), c(y + \delta y, t), c(z + \delta z, t)$ .

### 33.4 Calculating center of mass when using periodic boundary conditions.

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



**Figure 6** A connected cell in the lattice edge area – periodic boundary conditions are applied

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

$$x_{COM} = \frac{\sum_i x_i}{V} \quad (33.32)$$

or in vector form:

$$\vec{r}_{COM} = \frac{\sum_i \vec{r}_i}{V} \quad (33.33)$$

would result in being somewhere in the middle of the lattice and obviously outside the cell. A better procedure could be as follows:

Before calculating center of mass when new pixel is added or lost we “shift” a cell and new pixel (gained or lost) to the middle of the lattice do calculations “in the middle of the lattice” and shift back. Now if after shifting back it turns out that center of mass of a cell lies outside lattice position it in the center of mass by applying a shift equal to the length of the lattice and whose direction should be such that the center of mass of the cell ends up inside the lattice (there is only one such shift and it might be equal to zero vector).

This is how we do it using mathematical formulas:

$$\vec{s} = \vec{r}_{COM} - \vec{c} \quad (33.34)$$

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

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

$$\vec{r}'_{COM} = \vec{r}_{COM} - \vec{s} \quad (33.35)$$

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

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

$$\vec{r}'_{COM}^{new} = \frac{\vec{r}'_{COM} V + \vec{r}_i}{V + 1} \quad (33.36)$$

Above we have assumed that we are adding one pixel.

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

$$\vec{r}_{COM}^{new} = \vec{r}_{COM}^{new} + \vec{s} \quad (33.37)$$

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

$$\vec{P} = (x_{max}, -y_{max}, 0) \quad (33.38)$$

where  $x_{max}$ ,  $y_{max}$ ,  $z_{max}$  are dimensions of the lattice.

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

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

## 33.5 Command line options of CompuCell3D

Although most users run CC3D using Player GUI sometimes it is very convenient to run CC3D using command line options. CC3D allows to invoke Player directly from command line which is convenient because it saves several clicks and if you run many simulations this might be quite convenient.

**Remark:** On Windows we use .bat extension for run scripts and on Linux/OSX it is .sh. Otherwise all the material in this section applies to all the platforms.

### 33.5.1 CompuCell3D Player Command Line Options

The command line options for running simulation with the player are as follows:

```
compucell3d.bat [options]
```

Options are:

-i <simulation file> - users specify .cc3d simulation file they want to run.

-s <screenshotDescriptionFileName> - name of the file containing description of screenshots to be taken with the simulation. Usually this file is prepared using Player by switching to different views, clickin camera button and saving screenshot description file from the Player File menu.

-o <customScreenshotDirectoryName> - allows users to specify where screenshots will be written. Overrides default settings.

--noOutput - instructs CC3D not to store any screenshots. Overrides Player settings.

--exitWhenDone - instructs CC3D to exit at the end of simulation. Overrides Player settings.

-h, --help - prints command line usage on the screen

Example command may look like (windows):

```
compucell3d.bat -i Demos\Models\cellsor\t\cellsor_2D\cellsor_2d.cc3d --noOutput
```

or on linux:

```
compucell3d.sh -i Demos/Models/cellsor/cellsor_2D/cellsor_2d.cc3d --noOutput
```

and OSX:

```
compuCell3d.command -i Demos/Models/cellsort/cellsort_2D/cellsort_2d.cc3d --noOutput
```

### 33.5.2 Running CompuCell3D in a GUI-Less Mode - Command Line Options.

Sometimes when you want to run CC3D on a cluster you will have to use runScript.bat which allows running CC3D simulations without invoking GUI. However, all the screenshots will be still stored.

**Remark:** current version of this script does not handle properly relative paths so it has to be run from the installation directory of CC3D i.e. you have to cd into this directory prior to runnig runScript.bat. Another solution is to use full paths.

The output of this script is in the form of vtk files which can be subsequently replayed in the Player (and one can take screenshots then). By default all fields present in the simulation are stored in the vtk file. If users want to remove some of the fields from being stored in the vtk format they have to pass this information in the Python script:

```
CompuCellSetup.doNotOutputField(_fieldName)
```

Storing entire fields (as opposed to storing screenshots) preserves exact snapshots of the simulation and allows result postprocessing. In addition to the vtk files runScript stores lattice description file with .dml` extension which users open in the Player (File->Open Lattice Description Summary File...) if they want to reply generated vtk files.

The format of the command (windows):

```
runScript.bat [options]
```

linux:

```
runScript.sh [options]
```

OSX:

```
runScript.command [options]
```

The command line options for runScript.bat are as follows:

-i <simulation file> - users specify .cc3d simulation file they want to run.

-c <outputFileCoreName> - allows users to specify core name for the vtk files. The default name for vtk files is Step

-o <customVtkDirectoryName> - allows users to specify where vtk files and the .dml file will be written. Overrides default settings

-f <frequency> or -outputFrequency=<frequency> - allows to specify how often vtk files are stored to the disk. Those files tend to be quite large for bigger simulations so storing them every single MCS (default setting) slows down simulation considerably and also uses a lot of disk space.

--noOutput - instructs CC3D not to store any output. This option makes little sense in most cases.

-h, --help - prints command line usage on the screen

Example command may look as follows(windows):

```
runScript.bat -i Demos\CompuCellPythonTutorial\InfoPrinter\cellsrt_2D_info_printer.cc3d
  ↵-f 10 -o Demos\CompuCellPythonTutorial\InfoPrinter\screenshots -c infoPrinter
```

linux:

```
runScript.sh -i Demos/CompuCellPythonTutorial/InfoPrinter/cellsort_2D_info_printer.cc3d
 ↵-f 10 -o Demos/CompuCellPythonTutorial/InfoPrinter/screenshots -c infoPrinter
```

osx:

```
runScript.command -i Demos/CompuCellPythonTutorial/InfoPrinter/cellsort_2D_info_printer.
 ↵cc3d -f 10 -o Demos/CompuCellPythonTutorial/InfoPrinter/screenshots -c infoPrinter
```

## 33.6 Managing CompuCell3D simulations (CC3D project files)

Until version 3.6.0 CompuCell3D simulations were stored as a combination of Python, CC3DML (XML), and PIF files. Starting with version 3.6.0 we introduced new way of managing CC3D simulations by enforcing that a single CC3D simulation is stored in a folder containing .cc3d project file describing simulation resources (.cc3d is in fact XML), such as CC3DML configuration file, Python scripts, PIF files, Concentration files *etc...*\* and a directory called **Simulation** where all the resources reside. The structure of the new-style CC3D simulation is presented in the diagram below:

### ->CellsorDemo

CellsorDemo.cc3d

### ->Simulation

Cellsor.xml

Cellsor.py

CellsorSteppables.py

Cellsor.piff

FGF.txt

Bold fonts denote folders. The benefit of using CC3D project files instead of loosely related files are as follows:

- 1) Previously users had to guess which file needs to be open in CC3D – CC3DML or Python. While in a well written simulation one can link the files together in a way that when user opens either one the simulation would work but, nevertheless, such approach was clumsy and unreliable. Starting with 3.6.0 users open .cc3d file and they don't have to stress out that CompUCell3D will complain with error message.

### **⚠ Warning**

The only way to load simulation in CompuCell3D is to use .cc3d project. We no longer support previous ways of opening simulations

- 2) All the files specified in the .cc3d project files are copied to the result output directory along with simulation results (unless you explicitly specify otherwise). Thus, when you run multiple simulations each one with different parameters, the copies of all CC3DML and Python files are stored eliminating guessing which parameters were associated with particular simulations.
- 3) All file paths appearing in the simulation files are relative paths with respect to main simulation folder. This makes simulations portable because all simulation resources are contained within single folder. In the example above when referring to Cellsor.piff file from Cellsor.xml you use Simulation/Cellsor.piff. This makes simulations easily exchangeable between collaborators
- 4) New style of storing CC3D simulations has also another advantage – it makes graphical management of simulation content and simulation generation very easy. As a matter of fact new component of CC3D suite – Twedit++

- CC3D edition has a graphical tool that allows for easy project file management and it also has new simulation wizard which allows users to build template of CC3D simulation within less than a minute.

Let's now look in detail at the structure of .cc3d files (we are using XML syntax here):

```
<Simulation version="3.6.0">
  <XMLScript>Simulation/Cellsrt.xml</XMLScript>
  <PythonScript>Simulation/Cellsrt.py</PythonScript>
  <Resource Type="Python">Simulation/CellsrtSteppables.py</Resource>
  <PIFFile>Simulation/Cellsrt.piff</PIFFile>
  <Resource Type="Field" Copy="No">Simulation/FGF.txt</Resource>
</Simulation>
```

As you can see the structure of the file is quite flat. All that we are storing there is names of files that are used in the simulation. Two files have special tags <XMLFile> which specifies name of the CC3DML file storing “CC3DML portion” of the simulation and <PythonScript> which specifies main Python script. We have also <PIFFile> tag which is used to designate PIF files. All other files used in the simulation are referred to as Resources. For example Python steppable file is a resource of type “Python” - <Resource Type="Python">Simulation/CellsrtSteppables.py</Resource>. FGF.txt is a resource of type “Field” - <Resource Type="Field" Copy="No">Simulation/FGF.txt</Resource>. Notice that all the files are specified using paths relative to main simulation directory i.e. w.r.t to the dir in which .cc3d file resides

As we mentioned before, when you run .cc3d simulation all the files listed in the project file are copied to result folder. If for some reason you want to avoid coping of some of the files, simply add `Copy="No"` attribute in the tag with file name specification for example”

```
<PIFFile Copy="No">Simulation/Cellsrt.piff</PIFFile>
<Resource Type="Field" Copy="No">Simulation/FGF.txt</Resource>
```



---

CHAPTER  
THIRTYFOUR

---

## TRANSITIONING FROM CC3D 3.X TO CC3D 4.X

New CC3D 4.x switches from Python 2.x to python 3.6+. This offered us an opportunity to reorganize code and simplify the way you describe your simulations. The good news is that the new way of specifying simulations in Python is much simpler than before. However we did make some changes that will require you to update your existing code to work with CC3D 4.x. We tried as much as possible to keep old API to limit the number of changes you need to make.

**As a translation method we recommend starting a new CompuCell3D model with twedit++ and copying over parts of your old 3.x code into this new CompuCell3D 4.x model.**

The list below summarizes key API and coding convention changes. Later we are providing step-by-step guide on how to port your simulations to CC3D 4.x .

1. Introduction of cc3d python module. Most of your cc3d imports will begin from `import cc3d.xxx`
2. Switching from Pascal-case to more Pythonic snake-case and dropped leading underscore from function arguments `self.addSBMLToCell(_cell) -> self.add_sbml_to_cell(cell)`.
3. Changes how we declare Steppable class
4. Changes in main Python script

### 34.1 Main Python Script

Old Python script was quite verbose and contained a lot of “boiler-plate” code. We fixed it and now instead of typing

```
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()

from bacterium_macrophage_2D_steering_steppables import ChemotaxisSteering
```

(continues on next page)

(continued from previous page)

```
chemotaxisSteering=ChemotaxisSteering(_simulator=sim,_frequency=100)
steppableRegistry.registerSteppable(chemotaxisSteering)

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

we write something much simpler

```
from cc3d import CompuCellSetup
from .bacterium_macaophage_2D_steering_steppables import ChemotaxisSteering

CompuCellSetup.register_steppable(steppable=ChemotaxisSteering(frequency=100))

CompuCellSetup.run()
```

We import CompuCellSetup module from cc3d package as well as the steppable we want to instantiate. After that we register the steppable by calling

```
CompuCellSetup.register_steppable(steppable=ChemotaxisSteering(frequency=100))
```

and then start the simulation by calling

```
CompuCellSetup.run()
```

This is much simpler than before and main change is that we no longer store references to key CC3D objects sim - simulator object and simthread - object representing Player in the main script. Those objects are now handled behind-the-scenes by the new code-base . You can still easily access them though.

## 34.2 Steppable Class

The new Steppable class is quite similar to the old one but as before we no longer need to pass simulator in the constructor of the class. For example.

```
from cc3d.core.PySteppables import *

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

The rest of the steppable structure is very similar as in the CC3D 3.x.

Note that we import steppable class using

```
from cc3d.core.PySteppables import *
```

As we mentioned before, most of the CC3D-related Python modules are now submodules of the cc3d python package

## 34.3 Deprecation Warnings for Old API

Most of the old API still works in the new CC3D. If you notice absence of certain functions please let us know and we will fix it. In the process of reworking CC3D API we removed deprecated functions or functions that were eliminated because they were not needed anymore. Old API was preserved but we added depreciation warning. It is quite likely, therefore, that when you run CC3D Simulation you may see a lot of depreciation warnings. Most of them will look as follows

```
SBMLSolverLegacy/Simulation/SBMLSolverLegacySteppables.py:47: DeprecationWarning: Call to deprecated method addSBMLToCell. (You should use : add_sbml_to_cell) -- Deprecated since version 4.0.0.
```

You may ignore those warnings for now but we highly encourage you to replace old API calls with eh new ones. Most importantly, Twedit++ uses new API so if you need assistance you may always refer to CC3D Python of Twedit++

## 34.4 Simplified Programmatic Steering of CC3DML Parameters

Previous version of CC3D allowed to programmatically change values of CC3DML parameters. For example, you could run simulation and adjust chemotaxis Lambda from a Python script. The code that was required to make those adjustments was , at best, quite confusing and therefore this feature was a source a frustration among users. The new CC3D fixes this issue. The solution comes from the world of JavaScript and HTML. All that is required is tagging of the CC3DML element using `id` attribute and referring to it from Python script. we present a simple example below and a separate section on programmatic steering can be found in later chapters of this manual

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

Here in the CC3DML code we added `id="macro_chem"` tag to element that we want to modify from Python steppable script. One important thing to keep in mind is that the tags for different elements need to be distinct

In python script we modify Lambda attribute as follows:

```
def step(self, mcs):
    if mcs > 100 and not mcs % 100:
        vol_cond_elem = self.get_xml_element('macro_chem')
        vol_cond_elem.Lambda = float(vol_cond_elem.Lambda) - 3
```

where first statement `vol_cond_elem = self.get_xml_element('macro_chem')` fetches a reference to the CC3DML element and the second modifies `vol_cond_elem.Lambda = float(vol_cond_elem.Lambda) - 3` assigns new value of Lambda

As a reminder we present equivalent code in the old version of CC3D

```
def step(self,mcs):
    if mcs>100 and not mcs%100:

        attrVal=float(self.getAttributeValue('Lambda',[ 'Plugin','Name','Chemotaxis'],
        ['ChemicalField','Name','ATTR'],[ 'ChemotaxisByType','Type','Macrophage']))
        self.setAttributeValue('Lambda',attrVal-3,[ 'Plugin','Name','Chemotaxis'],
        ['ChemicalField','Name','ATTR'],[ 'ChemotaxisByType','Type','Macrophage'])
    self.updateXML()
```

As you can see the new code is easy to understand while the old one is quite a mouthful... For this reason we completely removed the old way of programmatic CC3DML steering from the new CC3D.

## 34.5 Accessing Fields

Starting with CC3D 4.0.0 all fields declared in the simulation can accessed using quite natural syntax:

```
self.field.FIELD_NAME
```

where FIELD\_NAME is replaced with actual field name:

For example to access field called fgf8 you type:

```
self.field.fgf_8
```

Like in previous releases if you are dealing with scalar fields (or a cell field) you may use slicing operators familiar from `numpy` package. For example to assign a patch of concentration of you would type:

```
self.field.fgf_8[10:20, 20:30, 0] = 12.3
```

## 34.6 SBML Solver

We also changed the way you use SBML solver. While the old syntax still works we feel that the new way of interacting with SBMLSolve submodule is more natural. Take a look at the example

```
model_file = 'Simulation/test_1.xml'

self.add_free_floating_sbml(model_file=model_file, model_name='Medium_dp2')

Medium_dp2 = self.sbml.Medium_dp2
Medium_dp2['S1'] = 10
Medium_dp2['S2'] = 0.5
```

Similarly, as in the case of regular fields, we access free floating sbml models using the followng syntax

```
self.sbml.Medium_dp2
```

where Medium\_dp2 is a label that we assigned to particular free-floating SBML model (i.e. the one not associated with a particular CC3D cell).

To add and access SBML model to a particular cell we use the following syntax:

```
model_file = 'Simulation/test_1.xml'

cell_20 = self.fetch_cell_by_id(20)

self.add_sbml_to_cell(model_file=model_file, model_name='dp', cell=cell_20)

cell_20.sbml.dp['S1'] = 1.3
```

In the code snippet above we first access a cell with `id=20` using `self.fetch_cell_by_id` function - we assume that cel with `id=20` exists. Next we add SBML model to a cell with `id=20` and then use

```
cell.sbml.SBML_MODEL_NAME['SPECIES_NAME'] = VALUE
```

to modify concentration in the SBML model

In our example the above template looks as follows:

```
cell_20.sbml.dp['S1'] = 1.3
```

We will cover SBML solver in details in later chapters

This completes transition guide.



## SECRETION PLUGIN (LEGACY VERSION FOR PRE-V3.5.0)

 **Warning**

While we still support Secretion plugin as described in this section, we observed performance degradation when declaring `<Field>` elements inside the plugin. To resolve this issue we encourage users to implement secretion “by cell type” in the PDE solver and keep using secretion plugin to implement secretion on a per-cell basis using Python scripting.

 **Note**

In version 3.6.2 Secretion plugin should not be used with DiffusionSolverFE or any of the GPU-based solvers.

In earlier versions of CC3D, secretion was part of PDE solvers. We still support this mode of model description however, starting in 3.5.0 we developed separate plugin which handles secretion only. Via secretion plugin we can simulate cellular secretion of various chemicals. The secretion plugin allows users to specify various secretion modes in the CC3DML file – CC3DML syntax is practically identical to the SecretionData syntax of PDE solvers. In addition to this Secretion plugin allows users to manipulate secretion properties of individual cells from Python level. To account for possibility of PDE solver being called multiple times during each MCS, the Secretion plugin can be called multiple times in each MCS as well. We leave it up to user the rescaling of secretion constants when using multiple secretion calls in each MCS.

 **Note**

Secretion for individual cells invoked via Python will be called only once per MCS.

Typical CC3DML syntax for Secretion plugin is presented below:

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

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

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

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

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

---

CHAPTER  
**THIRTYSIX**

---

## **APPENDIX A: LIST OF BASE STEPPABLE FUNCTIONS**

In this appendix, we present an alphabetical list of member functions and objects of the SteppableBasePy class from which all steppables should inherit:

`add_antimony_to_cell` - translates Antimony model to SBML and forwards it to `add_sbml_to_cell`  
`add_antimony_to_cell_ids` - translates Antimony model to SBML and forwards it to `add_sbml_to_cell_ids`  
`add_antimony_to_cell_types` - translates Antimony model to SBML and forwards it to `add_sbml_to_cell_types`  
`add_cellml_to_cell` - translates CellML model to SBML and forwards it to `add_sbml_to_cell`  
`add_cellml_to_cell_ids` - translates CellML model to SBML and forwards it to `add_sbml_to_cell_ids`  
`add_cellml_to_cell_types` - translates CellML model to SBML and forwards it to `add_sbml_to_cell_types`  
`add_free_floating_antimony` - translates Antimony model to SBML and forwards it to `add_free_floating_sbml`  
`add_free_floating_cellml` - translates CellML model to SBML and forwards it to `add_free_floating_sbml`  
`add_free_floating_sbml` - adds free floating SBML solver object to the simulation  
`add_new_plot_window` - adds new plot windows to the Player display  
`add_sbml_to_cell` - attaches SBML solver object to individual cell  
`add_sbml_to_cell_ids` - attaches SBML solver object to individual cells with specified ids  
`add_sbml_to_cell_types` - attaches SBML solver object to cells with specified types  
`adhesionFlexPlugin` - a reference to C++ AdhesionFlexPlugin object. None if plugin not used.  
`are_cells_different` - function determining if two cell objects are indeed different objects  
`boundaryMonitorPlugin` - a reference to C++ BoundaryMonitorPlugin object. None if plugin not used  
`boundaryPixelTrackerPlugin` - a reference to C++ BoundaryPixelTrackerPlugin object. None if plugin not used  
`build_wall` - builds wall of cells (They have to be of cell type which has Freeze attribute set in the Cell Type Plugin) around the lattice  
`cell_field` - reference to cell field.  
`cell_list` - cell list. Allows iteration over all cells in the simulation  
`cell_list_by_type` - function that creates on the fly a list of cells of given cell types.  
`cellOrientationPlugin` - a reference to C++ CellOrientationPlugin object. None if plugin not used  
`cellTypeMonitorPlugin` - a reference to C++ CellTypeMonitorPlugin object. None if plugin not used  
`centerOfMassPlugin` - a reference to C++ CenterOfMassPlugin object. None if plugin not used

`change_number_of_work_nodes` - function that allows changing number of worknodes use dby the simulation

`check_if_in_the_lattice` - convenience function that determines if 3D point is within lattice boundaries

`chemotaxisPlugin` - a reference to C++ ChemotaxisPlugin object. None if plugin not used

`cleanDeadCells` - function that calls step function from VolumetrackerPlugin to remove dead cell. Advanced use only. Deprecated in CC3D 4.x

`cleaverMeshDumper` - a reference to C++ CleaverMeshDumper object. None if module not used. Experimental. Deprecated in CC3D 4.0.0

`clone_attributes` - copies all attributes from source cell to target cell. Typically used in mitosis. Allows specification of attributes that should not be copied.

`clone_parent_2_child` - used in mitosis plugin. Copies all parent cell attributes to the child cell.

`clone_cluster_attributes` - typically used in mitosis with compartmentalized cells. Copies attributes from cell in a source cluster to corresponding cell in the target cluster. Allows specification of attributes that should not be copied

`clone_parent_cluster_2_child_cluster` - used in mitosis with compartmentalized cells. Copies all attributes from cell in a parent cluster to corresponding cell in the child cluster

`cluster_list` - Python-iterable list of clusters. Obsolete

`clusterSurfacePlugin` - a reference to C++ ClusterSurfacePlugin object. None if module not used.

`clusterSurfaceTrackerPlugin` - a reference to C++ ClusterSurfaceTrackerPlugin object. None if module not used.

`clusters` - Python-iterable list of clusters.

`connectivityGlobalPlugin` - a reference to C++ ConnectivityGlobalPlugin object. None if module not used.

`connectivityLocalFlexPlugin` - a reference to C++ ConnectivityLocalFlexPlugin object. None if module not used.

`contactLocalFlexPlugin` - a reference to C++ ContactLocalFlexPlugin object. None if module not used.

`contactLocalProductPlugin` - a reference to C++ ContactLocalProductPlugin object. None if module not used.

`contactMultiCadPlugin` - a reference to C++ ContactMultiCadPlugin object. None if module not used.

`contactOrientationPlugin` - a reference to C++ ContactOrientationPlugin object. None if module not used.

`copy_sbml_simulators` - function that copies SBML Solver objects from one cell to another

`create_scalar_field_cell_level_py` - function creating cell-level scalar field for Player visualization.

`create_scalar_field_py` - function creating pixel-based scalar field for Player visualization.

`create_vector_field_cell_level_py` - function creating cell-level vector field for Player visualization.

`create_vector_field_py` - function creating pixel-based vector field for Player visualization.

`delete_cell` - function deleting cell.

`delete_free_floating_sbml` - function deleting free floating SBML Solver object with a given name.

`delete_sbml_from_cell` - function deleting SBML Solver object with a given name from individual cell.

`delete_sbml_from_cell_ids` - function deleting SBML Solver object with a given name from individual cells with specified ids.

`delete_sbml_from_cell_types` - function deleting SBML Solver object with a given name from individual cells of specified types.

`destroy_wall` - function destroying wall of frozen cells around the lattice (if the wall exists)

`dim` - dimension of the lattice

`distance` - convenience function calculating distance between two 3D points

`distance_between_cells` - convenience function calculating distance between COMs of two cells.

`distance_vector` - convenience function calculating distance vector between two 3D points

`distance_vector_between_cells` - convenience function calculating distance vector between COMs of two cells

`elasticityTrackerPlugin` - a reference to C++ ElasticityTrackerPlugin object. None if module not used.

`every_pixel` - Python-iterable object returning tuples (x,y,z) for every pixel in the simulation. Allows iteration with user-defined steps between pixels.

`every_pixel_with_steps` - internal function used by `everyPixel`.

`fetch_cell_by_id` - fetches cell from cell inventory with specified id. Returns None if cell cannot be found.

`finish` - core function of each CC3D steppable. Called at the end of the simulation. User provide implementation of this function.

`focalPointPlasticityPlugin` - a reference to C++ FocalPointPlasticityPlugin object. None if module not used.

`frequency` - steppable call frequency.

`get_anchor_focal_point_plasticity_data_list` - returns a list anchored links

`get_box_coordinates` - returns the two points defining the smallest box containing all cells in simulation.

`get_cell_boundary_pixel_list` - function returning list of boundary pixels

`get_cell_neighbor_data_list` - function returning Python-iterable list of tuples (neighbor, common surface area) that allows iteration over cell neighbors

`get_cell_pixel_list` - function returning Python-iterable list of pixels belonging to a given cell

`get_cluster_cells` - function returning Python iterable list of cells in a cluster with a given cluster id.

`get_copy_of_cell_boundary_pixels` - function creating and returning new Python-iterable list of cell pixels of all pixels belonging to a boundary of a given cell.

`get_copy_of_cell_pixels` - function creating and returning new Python-iterable list of cell pixels of all pixels belonging to a given cell.

`get_elasticity_data_list` - function returning Python-iterable list of C++ ElasticityData objects. Used in conjunction with `ElasticityPlugin`

`get_energy_calculations` - function returning iterator of flip result and dictionary of effective energies by energy function for all flip attempts of the most recent Monte Carlo step. Requires `EnergyFunctionCalculator` with Type “Statistics”.

`get_field_secretor` - function returning Secretor object that allows implementation of secretion in a cell-by-cell fashion.

`get_focal_point_plasticity_data_list` - function returning Python-iterable list of C++ FocalPointPlasticity-Data objects. Used in conjunction with `FocalPointPlasticityPlugin`.

`get_focal_point_plasticity_neighbor_list` - function returning a Python-iterable list of all cell objects linked to a cell. Used in conjunction with `FocalPointPlasticityPlugin`.

`get_focal_point_plasticity_num_neighbors` - function returning the number of links attached to a cell. Used in conjunction with `FocalPointPlasticityPlugin`.

`get_focal_point_plasticity_is_linked` - function returning a Boolean signifying whether two cells are linked. Used in conjunction with `FocalPointPlasticityPlugin`.

`get_focal_point_plasticity_initiator` - function returning which of two linked cells initiated the link, or None if two cells are not linked. Used in conjunction with `FocalPointPlasticityPlugin`.

`get_internal_focal_point_plasticity_data_list` - function returning Python-iterable list of C++ InternalFocalPointPlasticityData objects. Used in conjunction with FocalPointPlasticityPlugin.

`get_pixel_neighbors_based_on_neighbor_order` - function returning Python-iterable list of pixels which are within given neighbor order of the specified pixel

`get_plasticity_data_list` - function returning Python-iterable list of C++ tPlasticityData objects. Used in conjunction with PlasticityPlugin. Deprecated

`get_sbml_simulator` - gets RoadRunner object for a given cell

`get_sbml_state` - gets Python-dictionary describing state of the SBML model.

`get_sbml_value` - gets numerical value of the SBML model parameter

`get_type_name_by_cell` - gets string name of cell type

`init` - internal use only

`invariant_distance` - calculates invariant distance between two 3D points

`invariant_distance_between_cells` - calculates invariant distance between COMs of two cells.

`invariant_distance_vector` - calculates invariant distance vector between two 3D points

`invariant_distance_vector_between_cells` - calculates invariant distance vector between COMs of two cells.

`invariant_distance_vector_integer` - calculates invariant distance vector between two 3D points. Keeps vector components as integer numbers

`inventory` - inventory of cells. C++ object

`lengthConstraintPlugin` - a reference to C++ LengthConstraintPlugin object. None if module not used.

`momentOfInertiaPlugin` - a reference to C++ MomentOfInertiaPlugin object. None if module not used.

`move_cell` - moves cell by a specified shift vector

`neighborTrackerPlugin` - a reference to C++ NeighborTrackerPlugin object. None if module not used.

`new_cell` - creates new cell of the user specified type

`normalize_path` - ensures that file path obeys rules of current operating system

`numpy_to_point_3d` - converts numpy vector to Point3D object

`open_file_in_simulation_output_folder` - function returning the file handle and output path of a file in the simulation output folder. Returns None, None if the file cannot be opened.

`output_dir` - simulation output directory

`pixelTrackerPlugin` - a reference to C++ PixelTrackerPlugin object. None if module not used.

`plasticityTrackerPlugin` - a reference to C++ PlasticityTrackerPlugin object. None if module not used.

`point_3d_to_numpy` - converts Point3D to numpy vector

`polarization23Plugin` - a reference to C++ Polarization23Plugin object. None if module not used.

`polarizationVectorPlugin` - a reference to C++ PolarizationVectorPlugin object. None if module not used.

`potts` - reference to C++ Potts object

`reassign_cluster_id` - reassigned cluster id. **Notice:** you cannot type `cell.clusterId=20`. This will corrupt cell inventory. Use `reassignClusterId` instead

`remove_attribute` - internal use

`resize_and_shift_lattice` - resizes lattice and shifts its content by a specified vector. Throws an exception if operation cannot be safely performed.

`runBeforeMCS` - flag determining if steppable gets called before (runBeforeMCS=1) Monte Carlo Step or after (runBeforeMCS=0). Default value is 0.

`secretionPlugin` - a reference to C++ SecretionPlugin object. None if module not used.

`set_focal_point_plasticity_parameters` - convenience function for setting various focal point plasticity parameters for a cell. Used in conjunction with FocalPointPlasticityPlugin.

`set_max_mcs` - sets maximum MCS. Used to increase or decrease number of MCS that simulation shuold complete.

`set_sbml_state` - used to pass dictionary of values of SBML variables

`set_sbml_value` - sets single SBML variable with a given name

`set_step_size_for_cell` - sets integration step for a given SBML Solver object in a specified cell

`set_step_size_for_cell_ids` - sets integration step for a given SBML Solver object in cells of specified ids

`set_step_size_for_cell_types` - sets integration step for a given SBML Solver object in cells of specified types

`set_step_size_for_free_floating_sbml` - sets integration step for a given free floating SBML Solver object

`shared_steppable_vars` - reference to a global dictionary shared by all steppables.

`simulator` - a reference to C++ Simulator object

`start` - core function of the steppable. Users provide implementation of this function

`step` - core function of the steppable. Users provide implementation of this function

`stop_simulation` - function used to stop simulation immediately

`timestep_cell_sbml` - function carrying out integration of all SBML models in the SBML Solver objects belonging to cells.

`timestep_free_floating_sbml` - function carrying out integration of all SBML models in the free floating SBML Solver objects

`timestep_sbml` - function carrying out integration of all SBML models in all SBML Solver objects

`translate_to_sbml_string` - function returning a string of SBML model specification translated from Antimony or CellML model specification file or string

`typeIdTypeNameDict` - internal use only - translates type id to type name

`vector_norm` - function calculating norm of a vector

`volumeTrackerPlugin` - a reference to C++ VolumeTrackerPlugin object. None if module not used.

Additionally MitosisPlugin base has these functions:

`child_Cell` - a reference to a cell object that has jus been created as a result of mitosis

`parent_cell` - a reference to a cell object that underwent mitisos. After mitosis this cell object will have smalle volume

`set_parent_child_position_flag` - function which sets flag determining relative positions of child and parent cells after mitosis. Value 0 means that parent child position will be randomized between mitosis event. Negative integer value means parent appears on the 'left' of the child and positive integer values mean that parent appears on the 'right' of the child.

`get_parent_child_position_flag` - returns current value of parentChildPositionFlag.

`divide_cell_random_orientation` - divides parent cell using randomly chosen cleavage plane.

`divide_cell_orientation_vector_based` - divides parent cell using cleavage plane perpendicular to a given vector.

`divide_cell_along_major_axis` - divides parent cell using cleavage plane along major axis

`divide_cell_along_minor_axis` - divides parent cell using cleavage plane along minor axis

`update_attributes` - function called immediately after each mitosis event. Users provide implementation of this function.

---

CHAPTER  
**THIRTYSEVEN**

---

## APPENDIX B: LIST OF CELL ATTRIBUTES

In this appendix, we present an alphabetical list of CellG attributes:

`clusterId`` - cluster id

`clusterSurface` - total surface of a cluster that a given cell belongs to. Needs ClusterSurface Plugin

`dict` - stores any custom attributes you wish to assign to the cell

`ecc` - eccentricity of cell . Needs MomentOfInertia plugin

`extraAttribPtr` - a C++ pointer to Python dictionary attached to each cell

`flag` - integer variable - unused. Can be used from Python

`fluctAmpl` - fluctuation amplitude. Default value is -1

`iXX` - xx component of inertia tensor. Needs MomentOfInertia Plugin

`iXY` - xy component of inertia tensor. Needs MomentOfInertia Plugin

`iXZ` - xz component of inertia tensor. Needs MomentOfInertia Plugin

`iYY` - yy component of inertia tensor. Needs MomentOfInertia Plugin

`iYZ` - yz component of inertia tensor. Needs MomentOfInertia Plugin

`iZZ` - zz component of inertia tensor. Needs MomentOfInertia Plugin

`id` - cell id

`lX` - x component of orientation vector. Set by MomentOfInertia

`lY` - y component of orientation vector. Set by MomentOfInertia

`lZ` - z component of orientation vector. Set by MomentOfInertia

`lambdaClusterSurface` - lambda (constraint strength) of cluster surface constraint.Needs ClusterSurface Plugin

`lambdaSurface` - lambda (constraint strength) of surface constraint. Needs Surface Plugin

`lambdaVecX` - x component of force applied to cell. Needs ExternalPotential Plugin

`lambdaVecY` - y component of force applied to cell. Needs ExternalPotential Plugin

`lambdaVecZ` - z component of force applied to cell. Needs ExternalPotential Plugin

`lambdaVolume` - lambda (constraint strength) of volume constraint. Needs Volume Plugin

`subtype` - currently unused

`surface` - instantaneous cell surface. Needs Surface plugin

`targetClusterSurface` - target value of cluster surface constraint.Needs ClusterSurface Plugin

**targetSurface** - target value of surface constraint. Needs Surface Plugin

**targetVolume** - target value of volume constraint. Needs Volume Plugin

**type** - cell type

**volume** - instantaneous cell volume. Needs VolumeTracker plugin which is loaded by default by every CC3D simulation.

**xCM** - numerator of x-component expression for cell centroid

**xCOM** - x component of cell centroid

**xCOMPrev** - x component of cell centroid from previous MCS

**yCM** - numerator of y-component expression for cell centroid

**yCOM** - y component of cell centroid

**yCOMPrev** - y component of cell centroid from previous MCS

**zCM** - numerator of z-component expression for cell centroid

**zCOM** - z component of cell centroid

**zCOMPrev** - z component of cell centroid from previous MCS

## DIRECT CALL TO COMPUCELL3D FROM PYTHON

In a typical situation, you will most often run CC3D using GUI. You create simulation, run it, look at the results, modify parameters, run again and the process continues. But what if you would like to be more “methodical” in finding optimal parameters? CC3D comes with a convenience module that allows you to directly call CC3D as a Python function, pass information to your simulations, and get return value(s) from them. As you can tell already, functionality like this allows you to use CC3D as a black box that takes inputs and returns outputs in advanced, integrated workflows (*e.g.*, integration with various optimization packages to find desired set of parameters for your simulation).

### 38.1 How does it work?

 Note

All examples from this section can be found in CompuCell3D/core/Demos/CallableCC3D

#### 38.1.1 Step 1

If you want to call CC3D engine from a Python function, pass information to it and/or get information from it, then you need to modify your existing simulation to process incoming information and/or return outgoing information. This is a very easy step - take a look at the code below:

```
from cc3d.core.PySteppables import *
from random import random

class CellsortSteppable(SteppableBasePy):
    def __init__(self, frequency=10):
        SteppableBasePy.__init__(self, frequency)

    def start(self):
        input_val = self.external_input

    def step(self, mcs):
        if mcs == 100:
            self.external_output = 200.0 + random()
```

This is a “regular” steppable. There are two new parts of interest. The first new part is where we retrieve a value from `external_input`. The value of this property is set in Python *before* CC3D is called and is passed to CC3D during instantiation of a simulation through a Python function. The second new part is the line where we modify `external_output`. This property that will persist even after simulation is finished is the way we set what is returned by our simulation. Here, our return value is set to be a sum of number `200.0` and a random number between 0 and 1.

This return value can be set at any point in the simulation. In particular it often makes sense to set it in the `finish` function, but for illustration purposes we set it in `step` function.

### 38.1.2 Step 2

Once we have a simulation we need to write a Python script from which we will call our value-returning simulations. We will first show a minimal example where we call CC3D a few times from a Python script, pass a value to each simulation and store the return value of each simulation in a list. Here is the code:

```

1  from os.path import dirname, join, expanduser
2  from cc3d.CompuCellSetup.CC3DCaller import CC3DCaller
3
4
5 def main():
6
7     number_of_runs = 4
8
9     # You may put a direct path to a simulation of your choice here and comment out
10    # simulation_fname line below
11    # simulation_fname = <direct path to your simulation>
12    simulation_fname = join(dirname(dirname(__file__)), 'cellsort_2D', 'cellsort_2D.cc3d')
13
14    # this creates a list of simulation file names where simulation_fname is repeated
15    # number_of_runs times
16    # you can create a list of different simulations if you want.
17    sim_fnames = [simulation_fname] * number_of_runs
18
19    ret_values = []
20    for i, sim_fname in enumerate(sim_fnames):
21        cc3d_caller = CC3DCaller(
22            cc3d_sim_fname=sim_fname,
23            screenshot_output_frequency=10,
24            output_dir=join(root_output_folder, f'cellsort_{i}'),
25            sim_input=i,
26            result_identifier_tag=i
27        )
28
29        ret_value = cc3d_caller.run()
30        ret_values.append(ret_value)
31
32
33
34 if __name__ == '__main__':
35     main()
```

In line 1 we import functions from `os.path` package that will be used to create paths to files. In line 2 we import `CC3DCaller` class. `CC3DCaller` object runs single simulation and returns simulation return value.

 Note

Simulation return value is a dictionary. This allows for quite a lot of flexibility. In particular, you are not limited to a single return value but can use multiple return values.

In line 11 we construct a path to a simulation that takes inputs and returns a value. This is a simulation that is bundled with CC3D. If you want to run different simulation you would replace code in line 11 with a direct path to your simulation. Line 12 defines location where we will write simulation output files (think of it as custom version of CC3DWorkspace folder that CC3D normally uses for simulation output).

### Note

When you rerun your multiple simulations using script above you may want to make sure that simulation output folders are empty to avoid overwriting output from previous runs

In line 16 we construct a list of simulations we want to run. Notice that we use Pythonic syntax to create a list with multiple copies of the same element. `[simulation_fname] * number_of_runs` constructs a list where `simulation_fname` is repeated `number_of_runs` times.

In line 18 we create a list that will hold results. Line 19 starts a loop where we iterate over the simulation paths we stored in the list `sim_fnames`.

In line 20 we create `CC3DCaller` object where we pass simulation name, screenshot output frequency, output directory for this specific simulation, input to the simulation and a tag (identifier) that is used to identify return results. In our case we use integer number `i` as an input and identifier but you can be more creative. In general, whatever is passed to the keyword argument `sim_input` is available to our steppables as the property `external_input`, and whatever we set the steppable property `external_output` to in our steppables will be returned. Finally in line 28 we execute simulation and get return value of the simulation and in line 29 is appended to `ret_values`. Line 31 prints return values.

If we run this script the output of print statement in line 31 will look something like (because we use `random()` function we do not know exact outputs):

```
return values [{'tag': 0, 'result': 200.8033875687598}, {'tag': 1, 'result': 200.  
˓→6628249954859}, {'tag': 2, 'result': 200.6617630355885}, {'tag': 3, 'result': 200.  
˓→30450775355195}]
```

Notice that a single simulation returns a dictionary as a result. For example simulation with tag 1 returned `{'tag': 1, 'result': 200.6628249954859}`. By “consuming” this dictionary in Python we can extract identifier using `ret_values[1]['tag']` syntax and if we want to get the result we would use `ret_values[1]['result']`.

Notice that in this example `ret_values[1]['result']` is a floating point number but you can write your simulation in such a way that the result can be another dictionary where you could return multiple values.

## 38.1.3 Applications

We mentioned it at the beginning, but the examples we are showing here are only to illustrate a technique of how to call CC3D engine from Python script. Executing several simulations inside a Python loop is not that exciting but coupling it to an optimization algorithm or sensitivity analysis script is actually more practical. We include a simple example of integrating CC3D with the SciPy optimization module to do model calibration in `CompuCell3D/core/Demos/CallableCC3D/ChemotaxisCalibrateDemo`.

### 38.1.4 Step 3

In order to run above script you need to set up the environment that is shipped with the CC3D binary distribution or the one that you used to compile CC3D. Let's get started. We will walk you through the steps necessary to run the above scripts on various platforms.

#### Note

Users who installed CC3D directly from conda only need to activate the conda environment in which CC3D is installed to configure the environment from a terminal.

Let's start with windows.

### 38.1.5 Windows

Open a terminal in the root directory of the CC3D installation and issue `call` on the script `conda-shell.bat`,

```
call conda-shell
```

The environment is now configured to execute CC3D in Python.

Next I navigate to location where my script from Step 2 is installed

```
cd c:\CompuCell3D-py3-64bit\Datas\CC3DCaller\cc3d_call_single_cpu
```

I replace the line 11 of the script from Step 2

```
simulation_fname = join(dirname(dirname(__file__)), 'cellsort_2D', 'cellsort_2D.cc3d')
```

with

```
simulation_fname = r'c:\CompuCell3D-py3-64bit\Datas\CallableCC3D\cellsort_2D\cellsort_2D.  
cc3d'
```

The reason I am doing it because in real application you probably have to do it anyway. You specify directly what simulation you want to run

Finally, in the console I execute the following:

```
python cc3d_call_single_cpu.py
```

Make sure that you are in the correct directory when you run the last command.

### 38.1.6 Linux

Running simulation on Linux is very similar to running on Windows. We start by opening a terminal in the root directory of the CC3D installation and calling `source` on `conda-shell.sh`,

```
source conda-shell.sh
```

Next, I go to `/home/m/411_auto/Datos/CC3DCaller\cc3d_call_single_cpu` and execute the script from Step 2. It is also useful to change line 11 of the script from

```
simulation_fname = join(dirname(dirname(__file__)), 'cellsort_2D', 'cellsort_2D.cc3d')
```

to

```
simulation_fname = '/home/m/411_auto/Demos/CC3DCaller/cellsort_2D/cellsort_2D.cc3d'
```

```
cd /home/m/411_auto/Demos/CC3DCaller/cc3d_call_single_cpu
python cc3d_call_single_cpu.py
```

In your output you should see the following lines

```
return values [{'tag': 0, 'result': 200.8033875687598}, {'tag': 1, 'result': 200.
˓→6628249954859}, {'tag': 2, 'result': 200.6617630355885}, {'tag': 3, 'result': 200.
˓→30450775355195}]
```

### 38.1.7 Mac

To run script from Step 2 you would follow described in the Linux section above. The only difference is that you will be using `conda-shell.command` environment variable setter script instead of `conda-shell.sh`. As before, all you need to run is `source conda-shell.command` to set up the environment and modify script from Step 2 to include direct path to the simulation.



## CHANGING NUMBER OF WORKNODES

CompuCell3D allows multi-core executions of simulations. We use checker-board algorithm to deal with the CPM part of the simulation. This algorithm restricts minimum partition size. As a rule of thumb, if you have cells that are large or are fragmented and spread out throughout the lattice, you should not use multiple cores. If your cells are relatively small using multiple cores can give you substantial boost in terms of simulation run times. But what does a small cell mean? If we are on a  $100 \times 100$  lattice and cells have approx. 5-7 pixels in “diameter” then if we use 4 cores then each core will be responsible for  $50 \times 50$  piece of the lattice. This is much bigger than our cell. However as we increase number of cores it may happen that lattice area processed by a single core is comparable in size to a single cell. This is a recipe for disaster. In such a case two (or more) CPUs may modify attributes of the same cell at the same time. This is known as race condition and CC3D does not provide any protection against such situation. The reason CC3D leaves it up to the user to ensure that race conditions do not occur is performance – protecting against race conditions would lead to slower code putting in question the whole effort to parallelize CC3D.

PDE solvers used in CC3D don't exhibit any side effects associated with increasing number of cores. As a matter of fact parallelizing PDE solvers provides the biggest boost to the simulation. We estimate that with 3-4 diffusing fields in the simulation, CC3D spends 80-90% of its runtime solving PDEs.

An example, `DynamicNumberOfProcessors` in `Demos/SimulationSettings` demonstrates how to change number of CPUs used by the simulation:

```
from cc3d.core.PySteppables import *

class DynamicNumberOfProcessorsSteppable(SteppableBasePy):

    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)

    def step(self, mcs):
        if mcs == 10:
            self.resize_and_shift_lattice(new_size=(400, 400, 1), shift_vec=(100, 100, 0))

        if mcs == 100:
            self.change_number_of_work_nodes(8)
```

At `MCS = 10` we resize the lattice and shift its content and at `MCS = 100` we change number of CPU's to 8. Actually what we do here is we chane number of computational threads to 8 and it is up to operating system to assign those threads to different processors. When we have 8 processors usually operating system will try to use all 8 CPU's In case our CPU count is lower some CPU's will execute more than one computational CC3D thread and this will give lower performance compared to the case when each CPU handles one CC3D thread.

As usual Twedit++ offers help in pasting template code ,simply go to CC3D Python->Simulation menu and choose appropriate option.



---

**CHAPTER  
FORTY**

---

**AUTHORS**

- Maciej H. Swat
- Julio Belmonte
- T.J. Sego
- James Glazier
- Peter Fyffe



---

**CHAPTER  
FORTYONE**

---

**FUNDING**

From early days CompuCell3D was funded by science grants. The list of funding entities include

- National Institutes of Health (NIH)
- US Environmental Protection Agency (EPA)
- National Science Foundation (NSF)
- Falk Foundation
- Indiana University (IU)
- IBM

The development of CC3D was funded fully or partially by the following awards:

- National Institutes of Health, National Institute of Biomedical Imaging and Bioengineering, U24 EB028887, “Dissemination of libRoadRunner and CompuCell3D”, (09/30/2019 – 06/30/2024)
- National Science Foundation, NSF 1720625, “Network for Computational Nanotechnology - Engineered nanoBIO Node”, (09/1/2017-08/31/2022)
- National Institutes of Health, National Institute of General Medical Sciences, R01 GM122424, “Competitive Renewal of Development and Improvement of the Tissue Simulation Toolkit”, (02/01/2017 - 01/31/2021)
- Falk Medical Research Trust Catalyst Program, Falk 44-38-12, “Integrated in vitro/in silico drug screening for ADPKD”, (11/30/2014-11/29/2017)
- National Institutes of Health, National Institute of General Medical Sciences, National Institute of Environmental Health Sciences and National Institute of Biomedical Imaging and Bioengineering, U01 GM111243 “Development of a Multiscale Mechanistic Simulation of Acetaminophen-Induced Liver Toxicity”, (9/25/14-6/30/19)
- National Institutes of Health, National Institute of General Medical Sciences, R01 GM076692, “Competitive Renewal of MSM: Multiscale Studies of Segmentation in Vertebrates”, (9/1/05-8/31/18)
- U. S. Environmental Protection Agency, R835001, “Ontologies for Data & Models for Liver Toxicology”, (6/1/11-5/30/15)
- National Institutes of Health, National Institute of General Medical Sciences, R01 GM077138, “Competitive Renewal of Development and Improvement of the Tissue Simulation Toolkit”, (9/1/07-3/31/15).
- U. S. Environmental Protection Agency, National Center for Environmental Research, R834289, “The Texas-Indiana Virtual STAR Center: Data-Generating in vitro and in silico Models of Development in Embryonic Stem Cells and Zebrafish”, (11/1/09-10/31/13)
- Pervasive Technologies Laboratories Fellowship (Indiana University Bloomington) (12/1/03-11/30/04).
- IBM Innovation Institute Award (9/25/03-9/24/06)

- National Science Foundation, Division of Integrative Biology, IBN-0083653, “BIOCOMPLEXITY–Multiscale Simulation of Avian Limb Development”, (9/1/00-8/31/07)

---

CHAPTER  
**FORTYTWO**

---

**REFERENCES**

1. Bassingthwaite, 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., Goligorsky, 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.