Python Scripting Manual for CC3D

Release 4.2.4

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

Jan 31, 2021

Contents

The focus of this manual is to teach you how to use Python 2 scripting language to develop complex CompuCell3D simulations.

> We will assume that you have a working knowledge of Python. You do not have to be a Python guru 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 Instant Python Hacking) or simply purchase a book on introductory Python programming.

Authors

- Maciej H. Swat
- Julio Belmonte
- T.J. Sego
- James Glazier

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)

CHAPTER 3

Introduction

If you have been already using CompuCell3D you probably have realized the limitations of CC3DML (CompuCell3D XML model specification format). Simulations written CC3DML are "static". That means you specify initial cellular behaviors, and throughout the simulation those behaviors descriptions remain unchanged. If your goal is to run simple cell-sorting or grain coarsening simulations CC3DML is all you need. However if you are seriously thinking about building complex biological models you have to look beyond markup-languages. Fortunately, CompuCell3D provides easy to use and learn Python scripting interface which allows users to build complex simulations without writing low-level code which requires compilation. If you have used Matlab or Mathematica you are familiar with such approach – somebody writes all number-crunching functions and provides you with scripting language which you use to "glue" those functions together to build mathematical models. This approach is very successful because it allows non-programmers to enter the arena of mathematical modeling. Python scripting available in CompuCell3D offers modelers significant flexibility to construct models where behaviors of individual cells change (according to user specification) as simulation progresses. In case you wonder if using Python degrades performance of the simulation we want to asure you that unless you use Python "unwisely" you will not hit any performance barrier for CompuCell3D simulations. Yes, there will be things that should be done in C++ because Python will be way too slow to handle certain tasks, however, throughout our two 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++ programing quite pleasurable (Hint: look at CC3D C++ menu in the Twedit++)

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

4.1 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

```python
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
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_macrophage_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.

4.2 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 exampmple.

```
from cc3d.core.PySteppables import *

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

The rest of of the steppable structure is very similar as in the CC3D 3.x.

Note that we import steppable class using

As we mentioned before, most of the CC3D-related Python modules are now submodules of the `cc3d` python package

---

4.3 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. MOsf 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++

4.4 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.getXMLAttributeValue('Lambda',['Plugin','Name','Chemotaxis
→'],['ChemicalField','Name','ATTR'],['ChemotaxisByType','Type','Macrophage']))
        self.setXMLAttributeValue('Lambda',attrVal-3,['Plugin','Name','Chemotaxis'],[
→'ChemicalField','Name','ATTR'],['ChemotaxisByType','Type','Macrophage'])
        self.updateXML()
```

As you can see the new code is easy to inderstand while the old one is quite a mouthful... For this reason we completely removed the old way of programatic CC3DML steering from the new CC3D.

4.5 Accessing Fields

Staring 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
```

4.6 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.

How to use Python in CompuCell3D

The most convenient way to start Python scripting in CC3D is by learning Twedit++. With just few clicks you will be able to create a template of working CC3D simulation which then you can customize to fit your needs. Additionally, each CC3D installation includes examples of simple simulations that demonstrate usage of most important CC3D features and studying these will give you a lot of insight into how to build Python scripts in CC3D.

Hint: Twedit++ has CC3D Python Menu which greatly simplifies Python coding in CC3D. Make sure to familiarize yourself with this conveninent tool.

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 few distinct places, leaving the rest of the template untouched. The goal of the main Python script is to setup a CC3D simulation and make sure that all modules are initialized in the correct order. Typically, the only place where you, as a user, will modify this script is towards the end of the script where you register your extension modules (steppables and plugins).

Another task of main Python script is to load CC3DML file which contains initial description of cellular behaviors. You may ask, why we need CC3DML file when we are using Python. Wasn't the goal of Python to replace CC3DML? There are two answers to this question short and long. The short answer is that CC3DML provides the description of INITIAL cell behaviors and we will modify those behaviors as simulation runs using Python. But we still need a starting point for our simulation and this is precisely what CC3DML file provides. If you, however, dislike XML, and would rather not use separate file you can easily convert CC3DML into equivalent Python function – all you have to do is to use Twedit++ context menu. We will come back to this topic later. For now, let's assume that we will still load CC3DML along with main Python script.

Let us start with simple example. We assume that you have already read "Introduction to CompuCell3D" manual and know how to use Twedit++ Simulation Wizard to create simple CC3D simulation. For completeness, however, we include here basic steps that you need to follow to generate simulation code using Twedit++.

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

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`
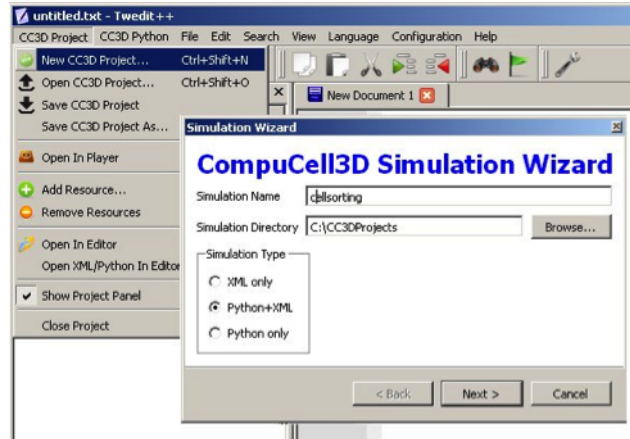
Fig. 1: Figure 1 Invoking the CompuCell3D Simulation Wizard from Twedit++.

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

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

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 Volume Constraint module.

We skip the next page related to Python scripting, after which Twedit++-CC3D generates the draft simulation code. Double clicking on `cellsorting.cc3d` opens both the CC3DML (`cellsorting.xml`) and Python scripts for the model.

The structure of generated CC3D simulation code is stored in .cc3d file (`C:\\CC3DProjects\\cellsorting`):

```xml
<Simulation version="3.6.2">

    <XMLScript Type="XMLScript">Simulation/cellsorting.xml</XMLScript>

    <PythonScript Type="PythonScript">Simulation/cellsorting.py</PythonScript>

    <Resource Type="Python">Simulation/cellsortingSteppables.py</Resource>

</Simulation>
```
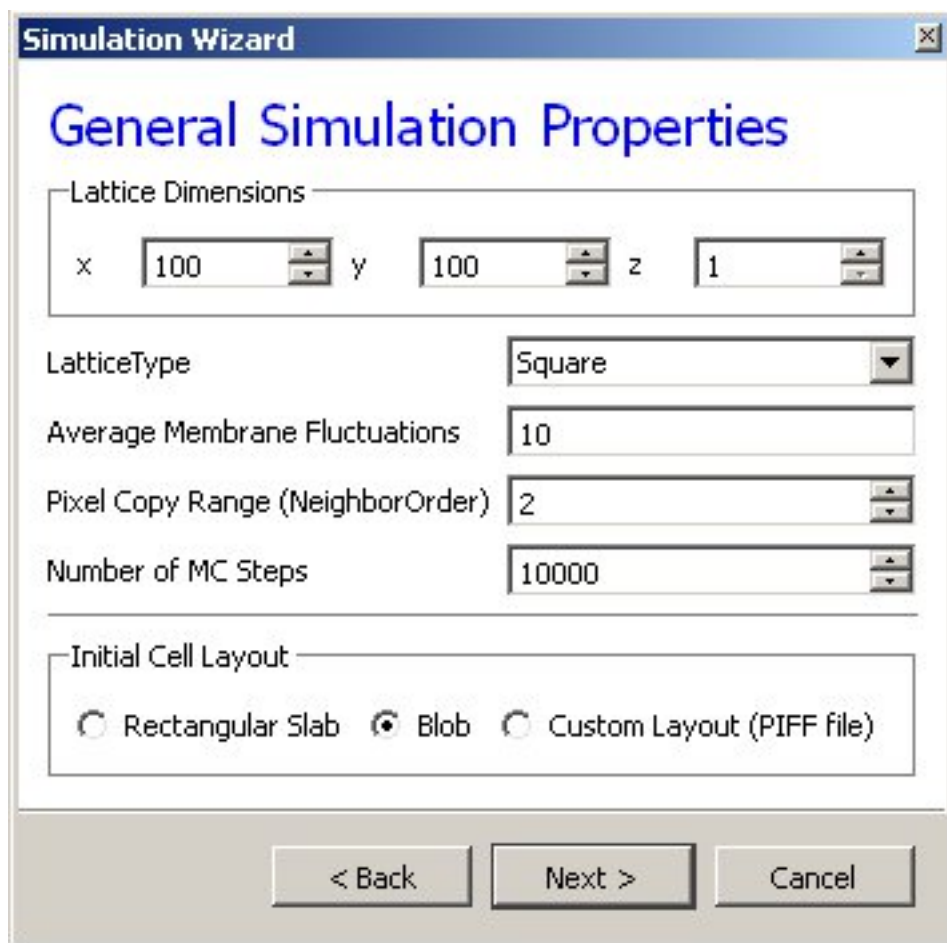
`Cellsorting.cc3d` stores names of the files that actually implement the simulation, and most importantly it tells you that both `cellsorting.xml`, `cellsorting.py` and `cellsortingSteppables.py` are part of the same simulation. CompuCell3D analyzes .cc3d file and when it sees `<PythonScript>` tag it knows that users will be using Python scripting. In such situation CompuCell3D opens Python script specified in .cc3d file (here `cellsorting.py`) and if user specified CC3DML script using `<XMLScript>` tag it loads this CC3DML file as well. In other words, .cc3d file is used to link Python simulation files together in an unbigous way. It also creates "root directory" for simulation so that in the Python or XML code modelers can refer to file resources using partial paths *i.e.*

---

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

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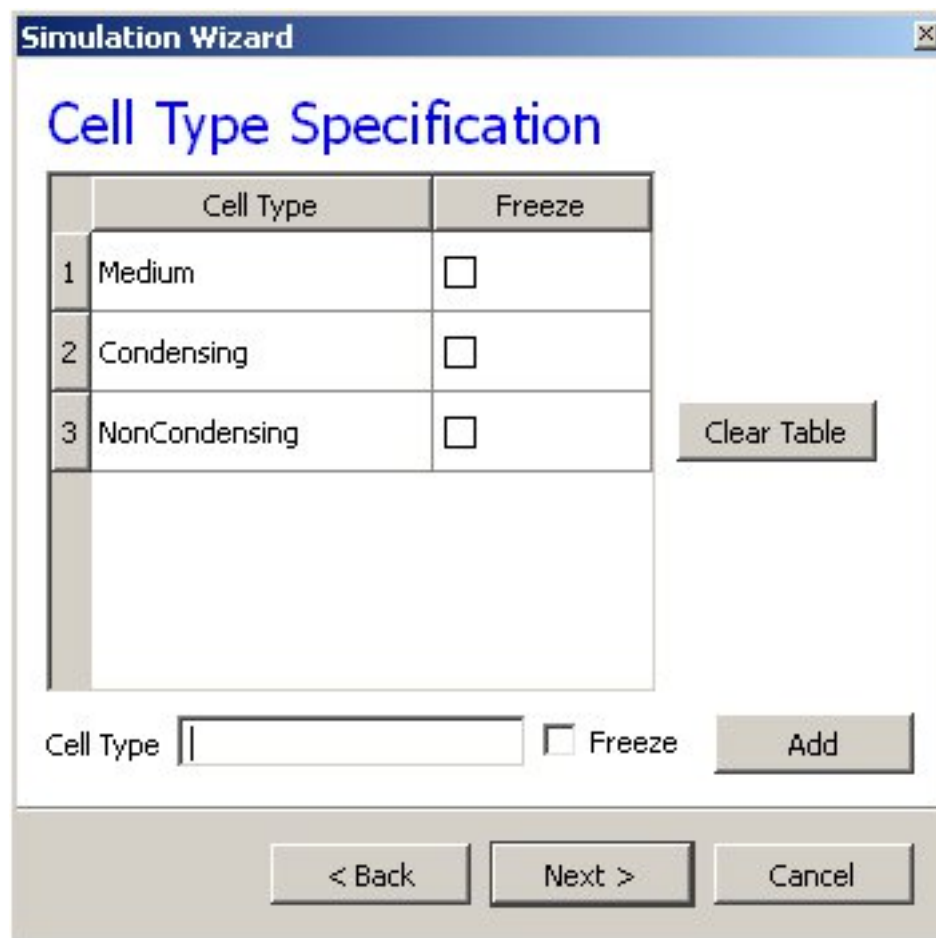
Fig. 3: Figure 3 Specification of cell-sorting cell types in Simulation Wizard.

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

if you store additional files in the Simulation directory you can refer to them via `Simulation\your_file_name` instead of typing full path *e.g.* `C:\CC3DProjects\cellsorting\Simulation\your\_file\_name`. For more discussion on this topic please see CompuCell Manual.

Let's first look at a generated Python code:

File: `C:\CC3DProjects\cellsorting\Simulation\cellsorting.py`

```
1  from cc3d import CompuCellSetup
2  from cellsortingSteppables import cellsortingSteppable
3
4  CompuCellSetup.register_steppable(steppable=cellsortingSteppable(frequency=1))
5
6  CompuCellSetup.run()
```

At the top of simulation main Python script we import `CompuCellSetup` module from `cc3d` package. The `CompuCellSetup` module has 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 cellsortingSteppables import cellsortingSteppable
```
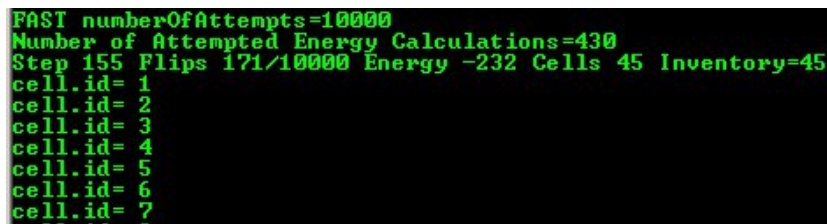
Note: If the the module from which we import steppable (here `cellsortingSteppables`) or the steppable class (here `cellsortingSteppable`) itself contains word `steppable` (capitalization is not important) we can put `.` in front of the module: `from .cellsortingSteppables import cellsortingSteppable`. This is not necessary but some development environments (e.g. PyCharm) will autocomplete syntax. This is quite helpful and speeds up development process.

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

Finally we start simulation using

```
CompuCellSetup.run()
```

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

Figure 5 Printing cell ids using Python script

You may wonder where strings cell.id=1 come from but when you look at `C:\CC3DProjects\cellsorting\Simulation\cellsortingSteppables.py` file, it becomes obvious:

```
from cc3d.core.PySteppables import *

class cellsortingSteppable(SteppableBasePy):
```

(continues on next page)

```python
    def __init__(self,frequency=1):
        SteppableBasePy.__init__(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 step function we have the following code snippet:

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

which prints to the screen id of every cell in the simulation. The step function is called every Monte Carlo Step (MCS) and therefore after completion of each MCS you see a list of all cell ids. In addition to step function you can see start and finish functions which have empty bodies. Start function is called after simulation have been initialized but before first MCS. Finish function is called immediately after last MCS. When writing Python extension modules you have flexibility to implement any combination of these 3 functions (`start`, `step`, `finish`).You can, of course, leave them unimplemented in which case they will have no effect on the simulation.

Let's rephrase it again because this is the essence of Python scripting inside CC3D - each steppable will contain by default 3 functions:

1. `start(self)`

2. `step(self,mcs)`

3. `finish(self)`

Those 3 functions are imported , via inheritance, from `SteppableBasePy`. The nice feature of inheritance is that once you import functions from base class you are free to redefine their content in the child class. We can redefine any combination of these functions. Had we not redefined *e.g.* finish functions then at the end simulation the implementation from `SteppableBasePy` of finish function would get called (which as you can see is an empty function).

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 develope 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.

6.1 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:
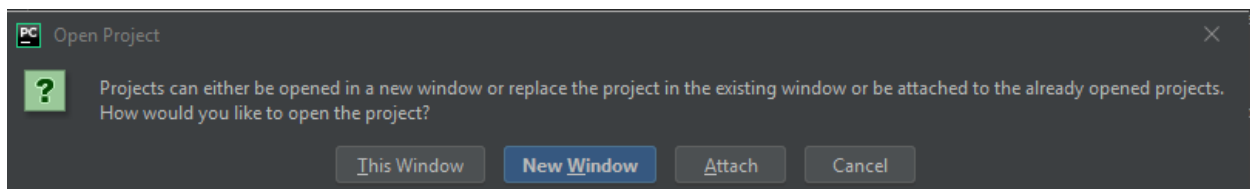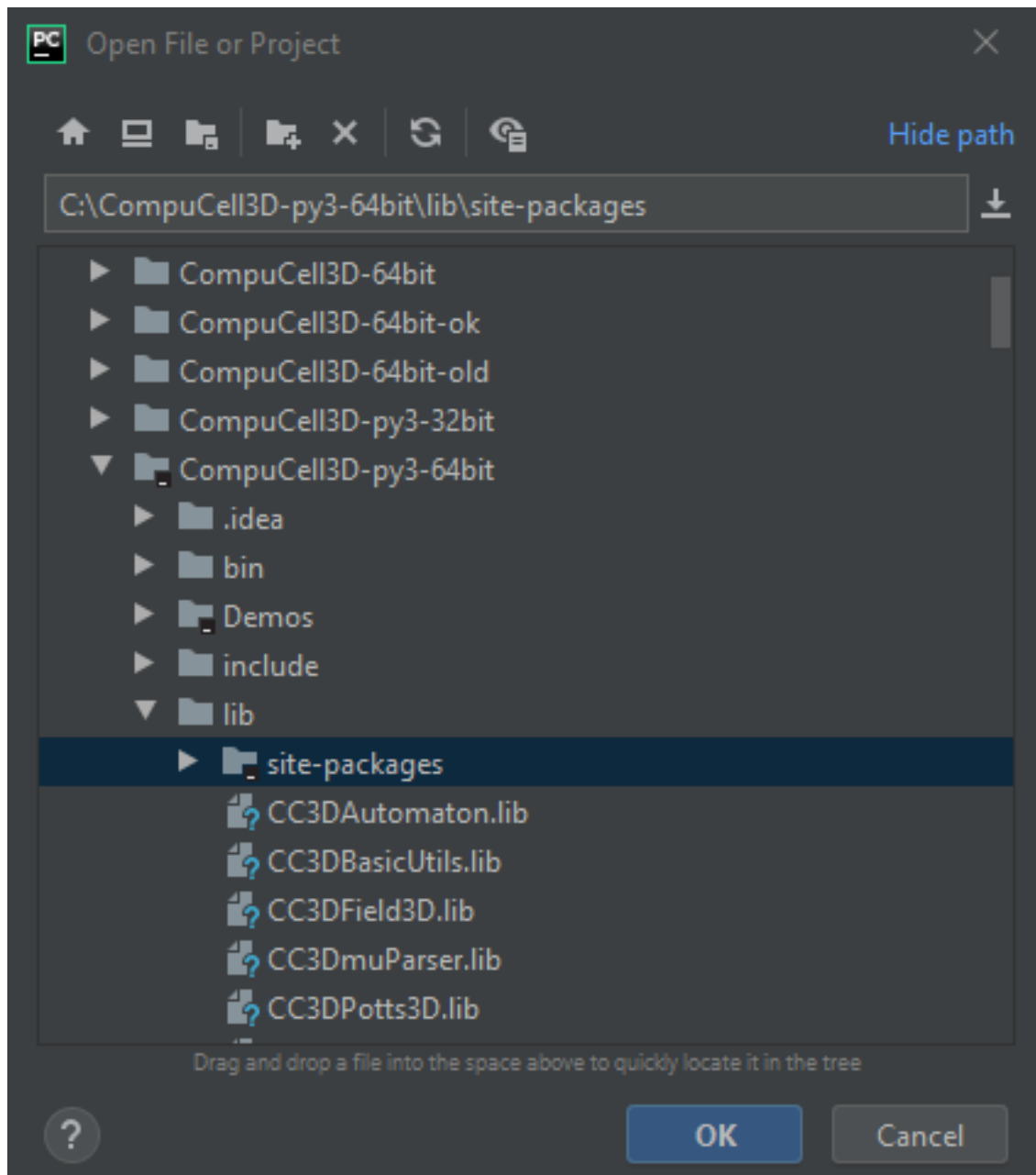
Next, you should see the following window
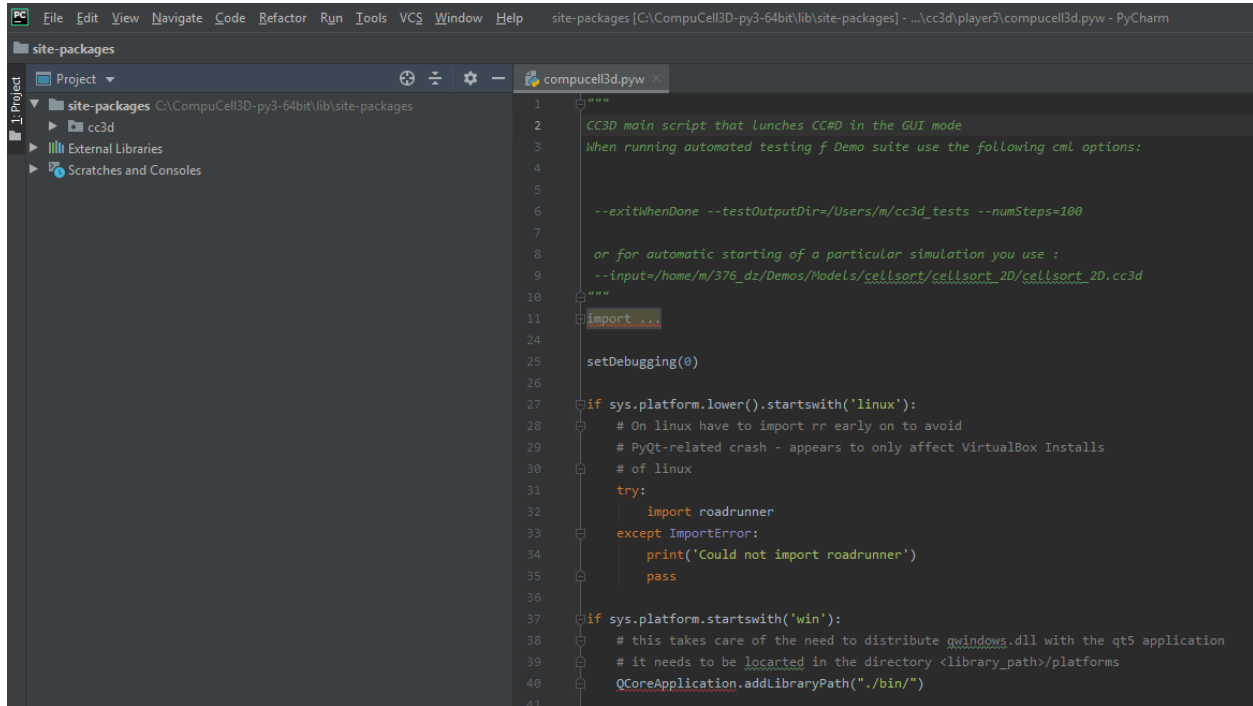
In order to be able to debug CC3D simulations it is best if the `Demos` folder (or any folder where you keep your simulations) also resides under `site-packages`. Simply copy `Demos` folder to `site-packages` folder so that your you PyCharm Project Explorer looks as follows (left panel in PyCharm) - see `Demos` directory listed under `cc3d`:

6.2 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->compucell3d.pyw` and first we double-click to open `compucell3d.pyw` script in the editor and then `right-click` to open up a context menu and from there we choose `Run "compucell3d.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:

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:
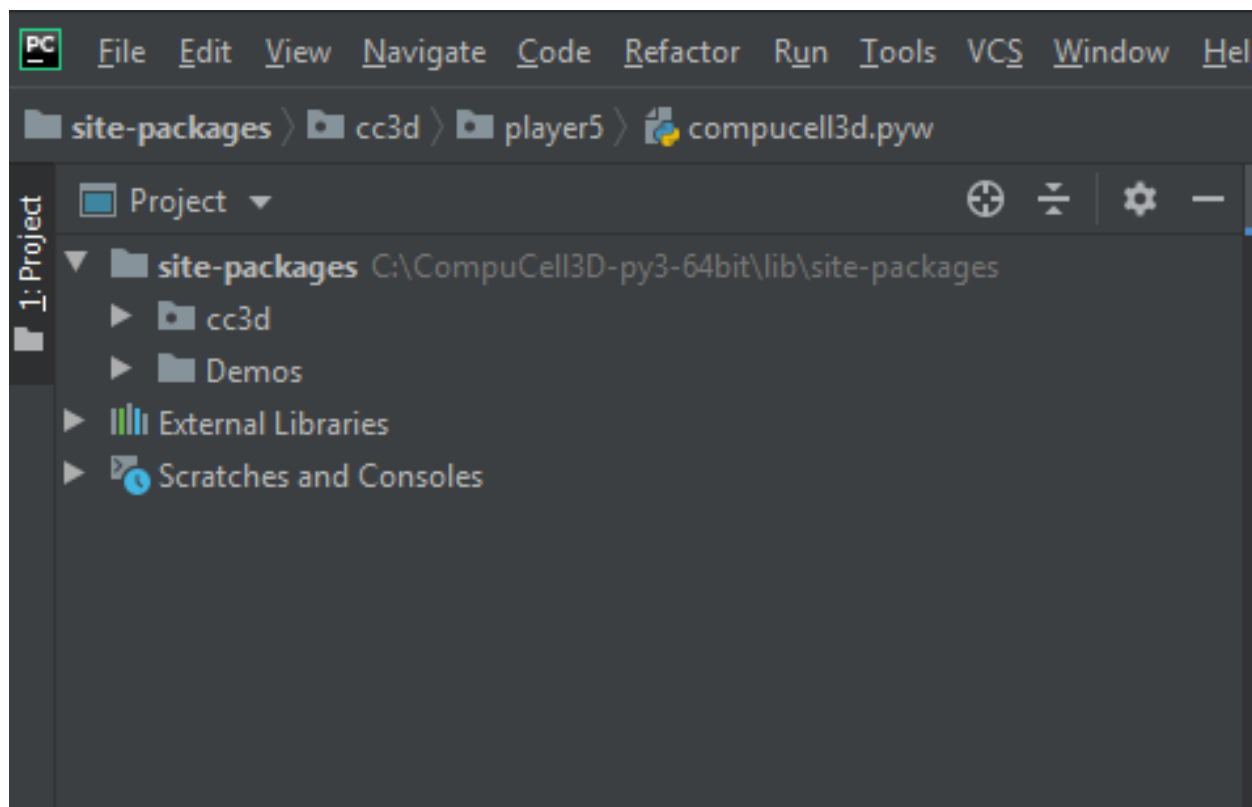
In this dialog we make sure to select options `Virtualenv Enviropnment` and check `Existing Environment` radio-box and then select correct Python interpreter executable. In my case it is located in `c:\CompuCell3D-py3-64bit\python36\python.exe` and if you are on e.g. osx or linux you will need to navigate 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 intepreter 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`

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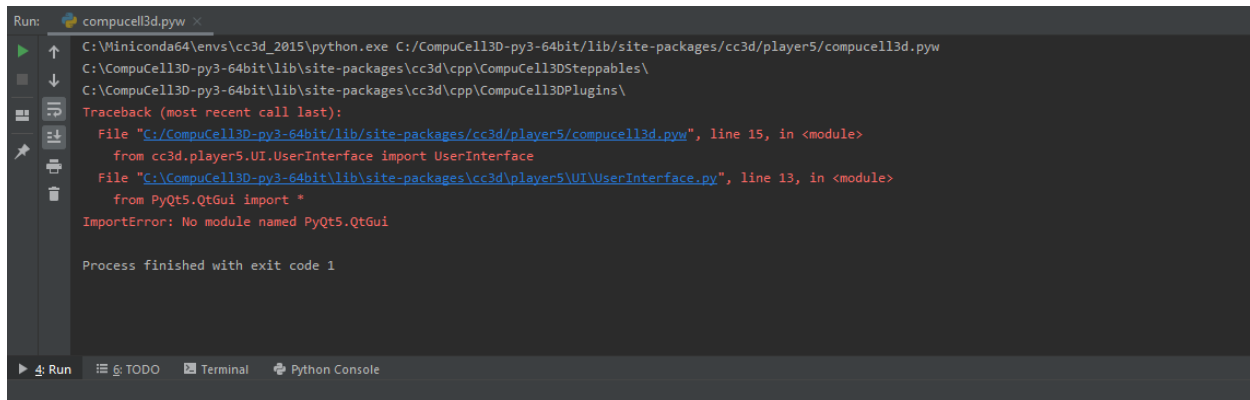
We select Environment Variables pull-down menu by clicking the icon in the right-end of the `Environment Variables` line and the following dialog will open up:

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
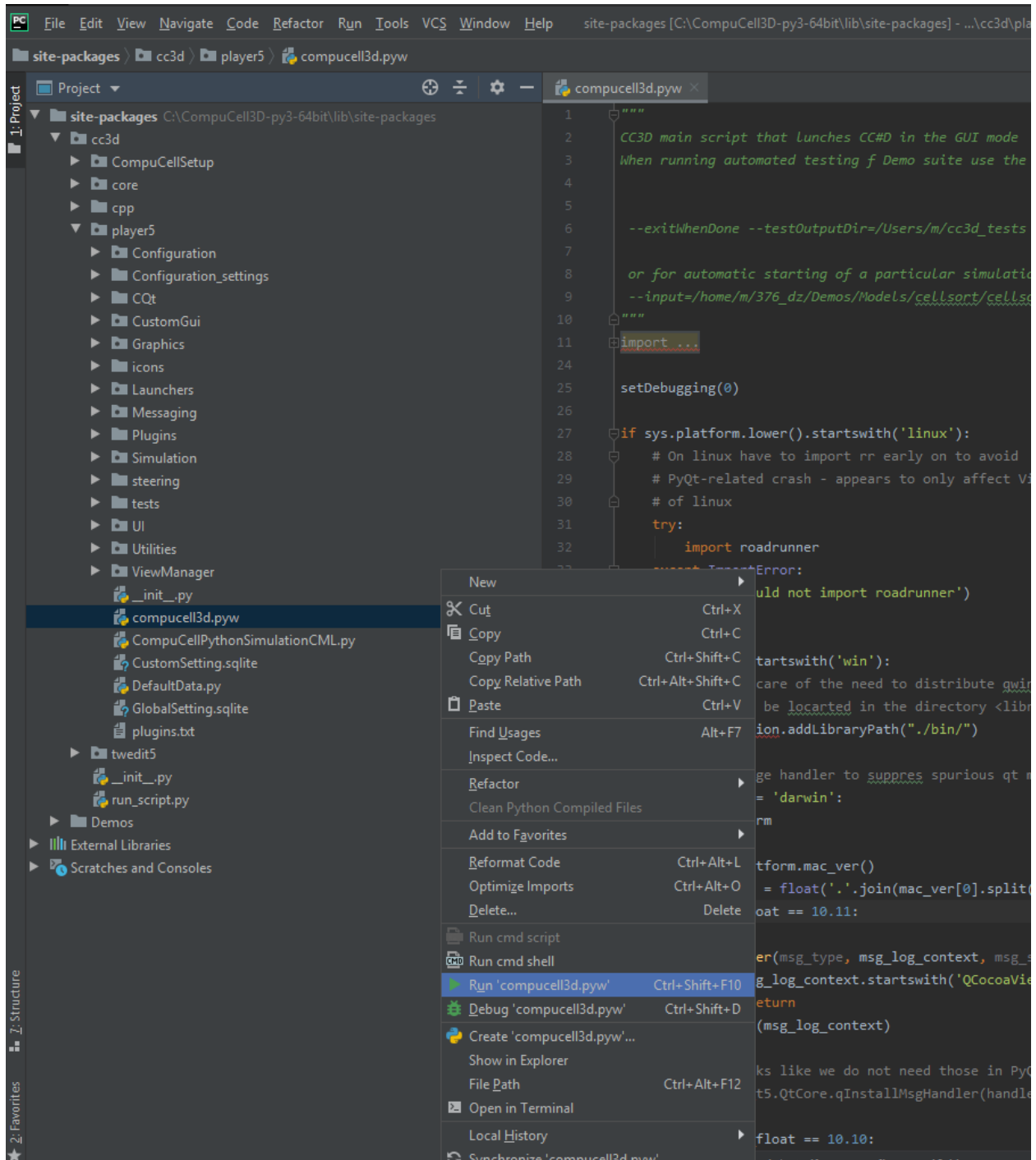
Chapter 6. Running and Debugging CC3D Simulations Using PyCharm

6.2.1 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:

```bash
#!/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
export COMPUCELL3D_BUILD_VERSION=0


echo " "
echo "---- ---- ---- ---- ---- ---- ---- ---- "
```

(continues on next page)

```
echo "  CompuCell3D version $COMPUCELL3D_MAJOR_VERSION.$COMPUCELL3D_MINOR_VERSION.
↪$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/compucell3d.app/Contents/MacOS/python

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

...
```

All the lines that begin with `export` ar used to set local environment variables that are necessary to to get CC3D to run. 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:
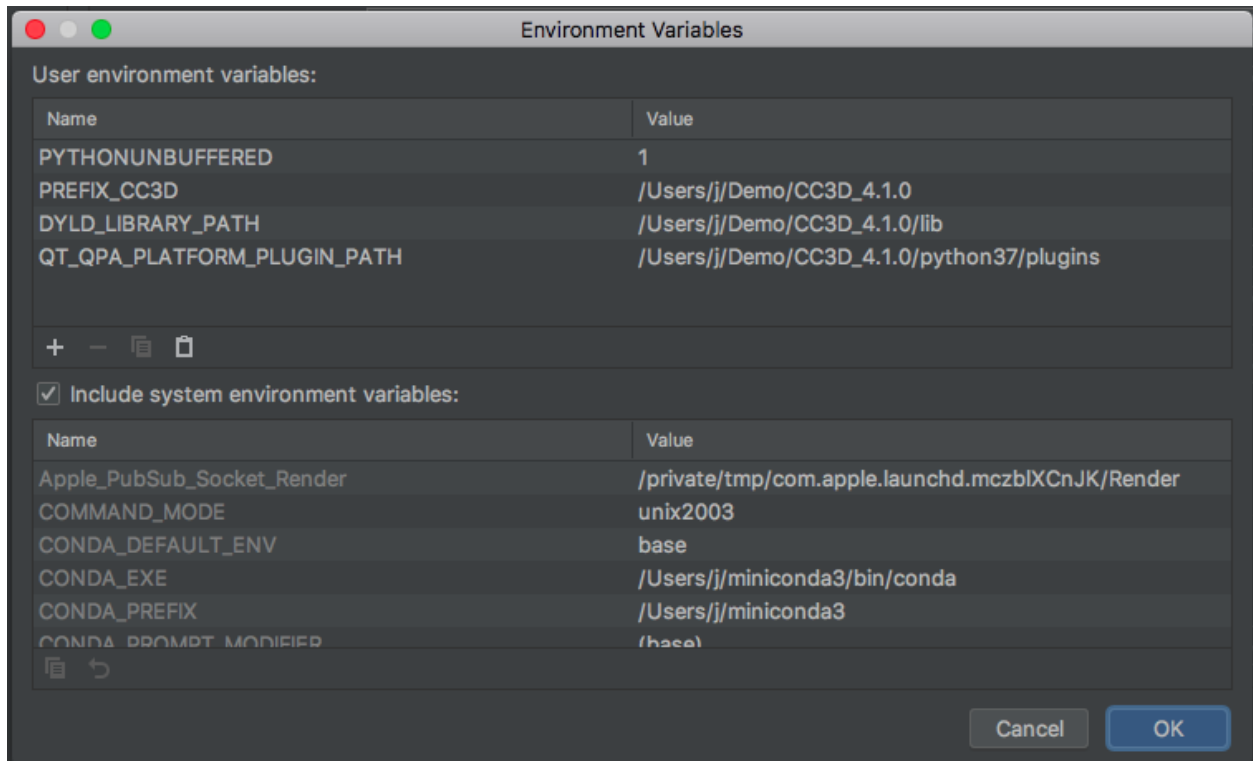
where I set :

```
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
```

6.2. Step 2 - running CC3D simulation from PyCharm. Configuring Python Environment and PREFIX_CC3D

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 `compucell3d.sh` script what are environment variables that are set there and make sure to set them in the PyCharm environment variable panel.

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

All the hard work you have done so far will pay up 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

6.3.1 Debugging Simulation

To Debug a simulation we open CompuCell3D in the debug mode by clicking `Debug` located to the right of the `Run` button:
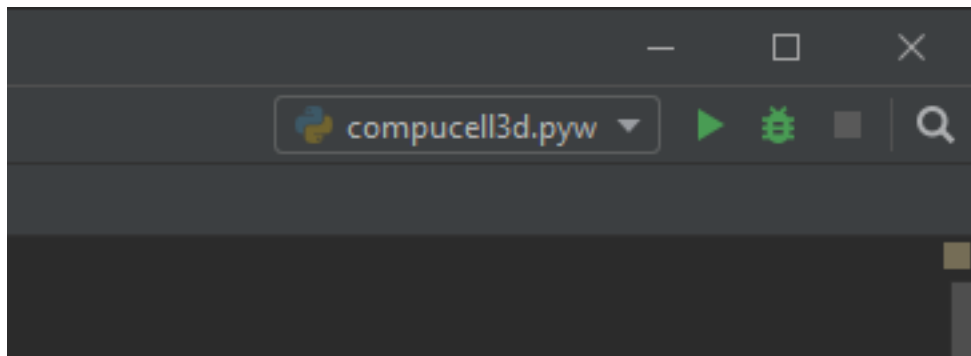
The player will open up. You may start the simulation by pressing `Step` button on the player. While the simulation is running we would like to inspect actual variable inside Python steppable. To to so we open up a simulation script we want to debug. In my case I will open simulation in `c:\CompuCell3D-py3-64bit\lib\site-packages\Demos\Models\cellsort\cellsort_2D_growing_cells_` 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\Demos\Models\cellsort\cellsort_2D_growing_cells_` `py` in PyCharm editor.

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 variables. 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:
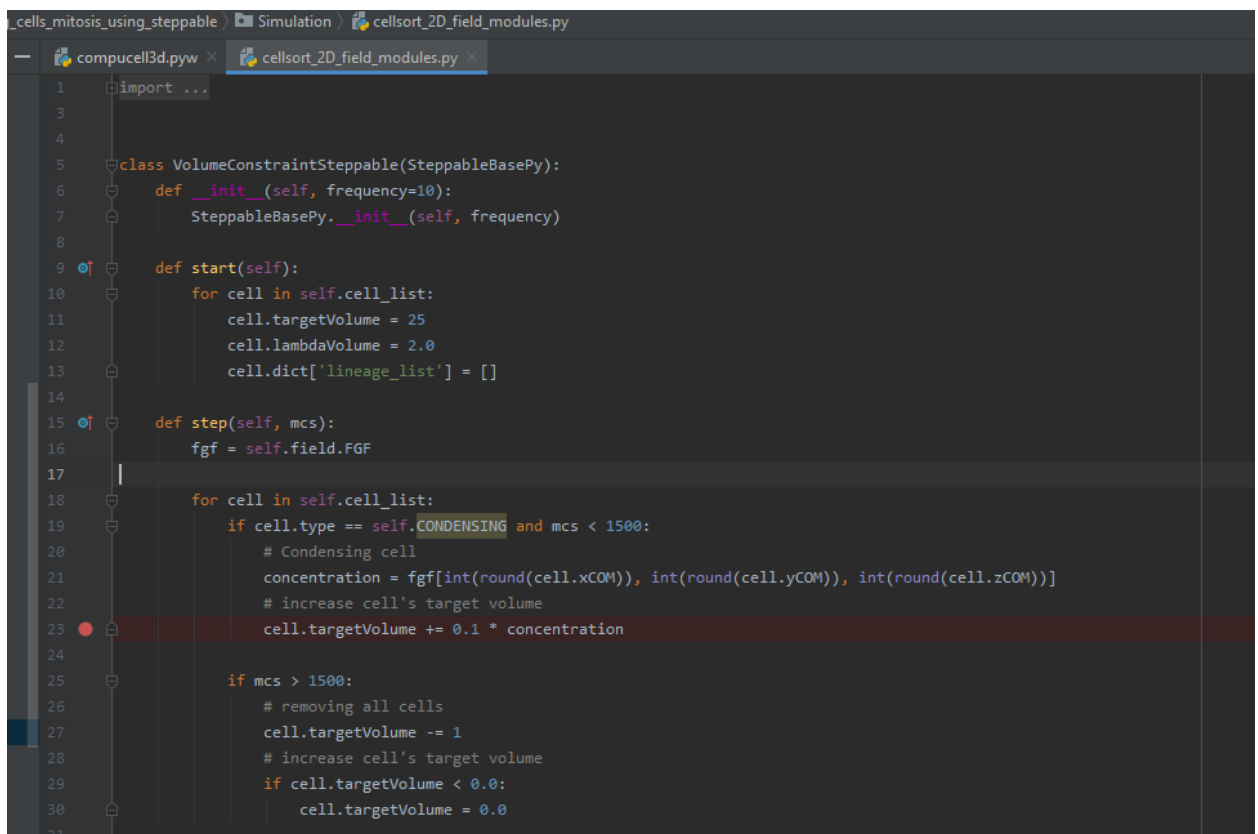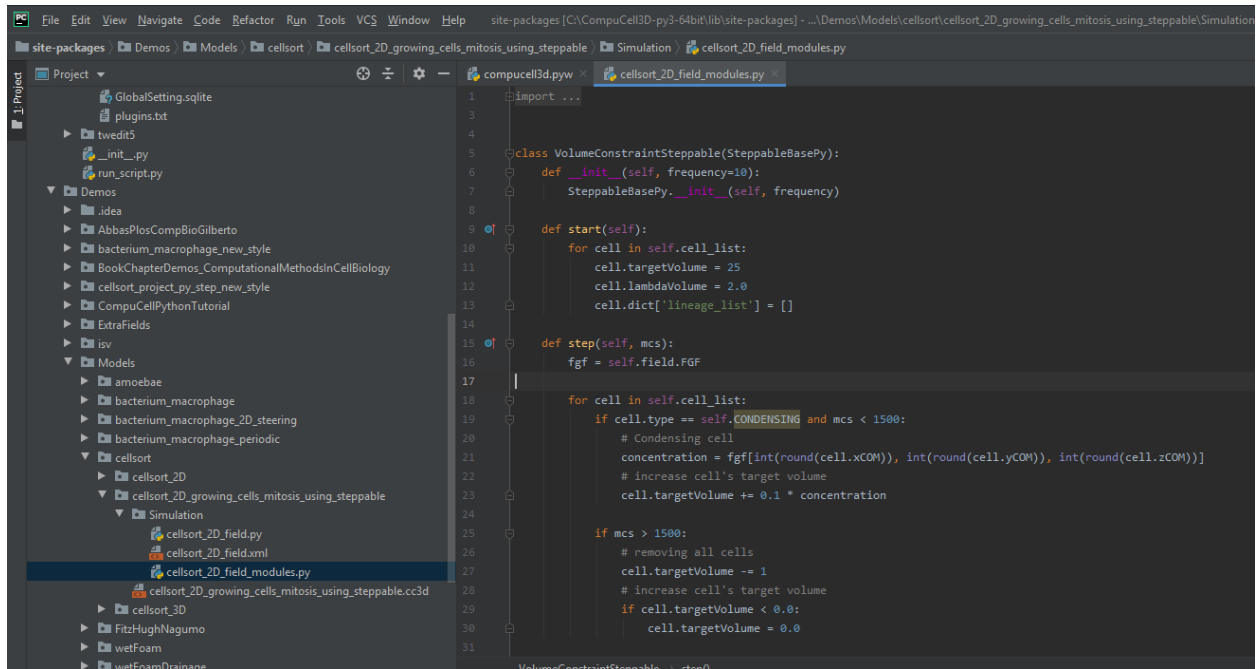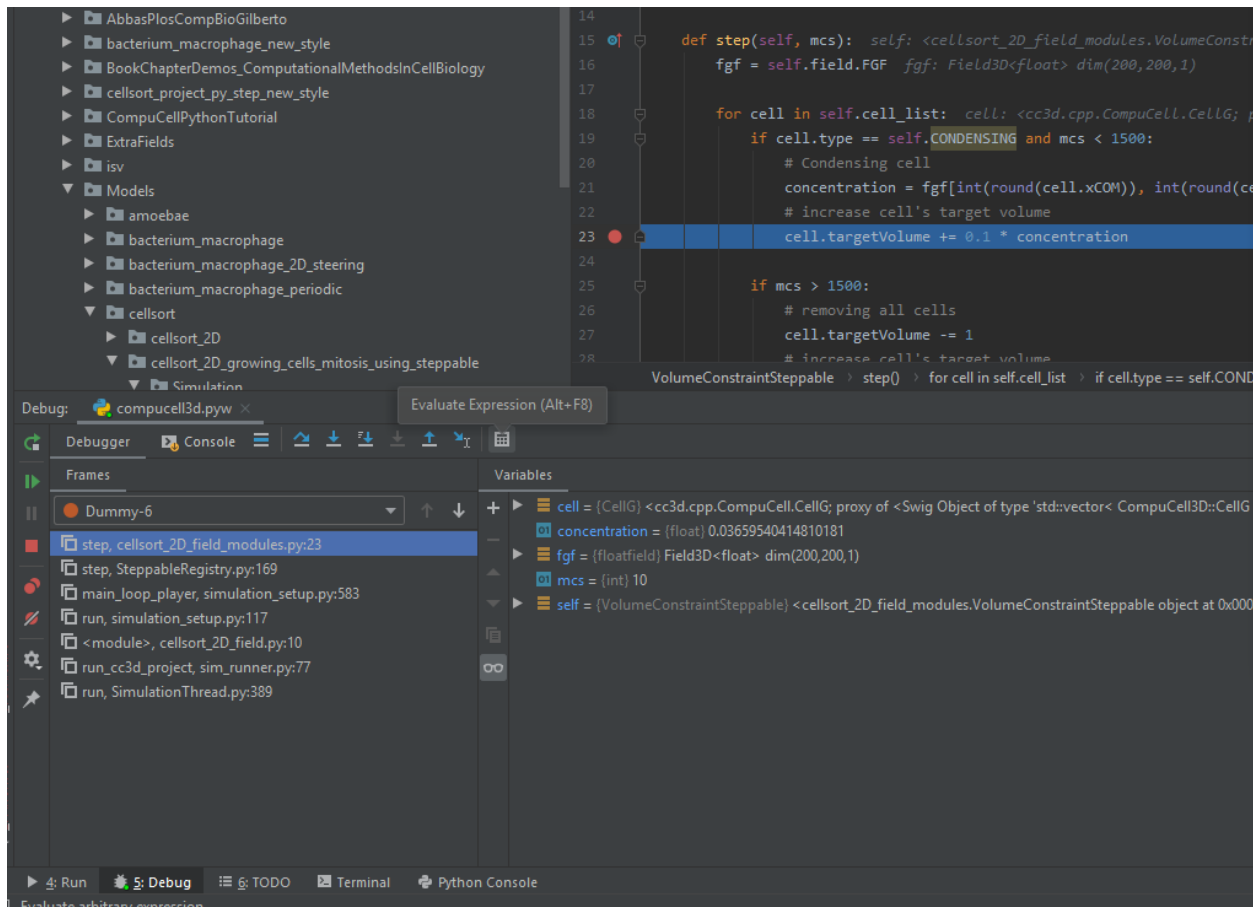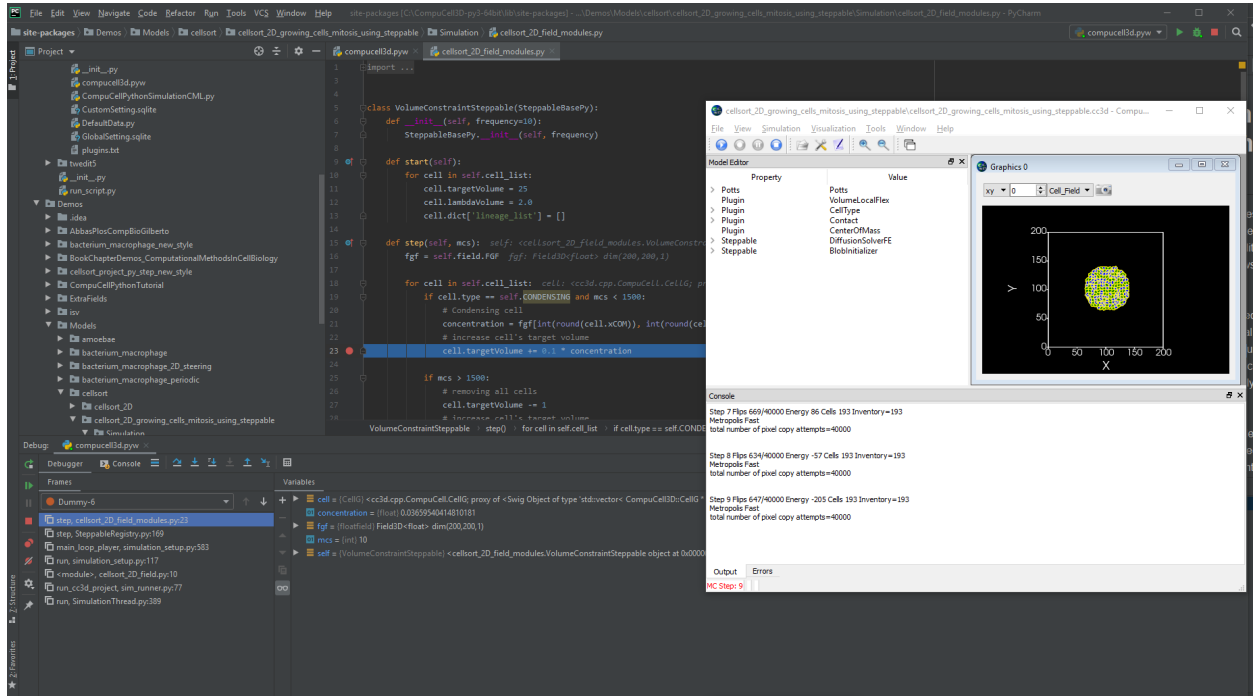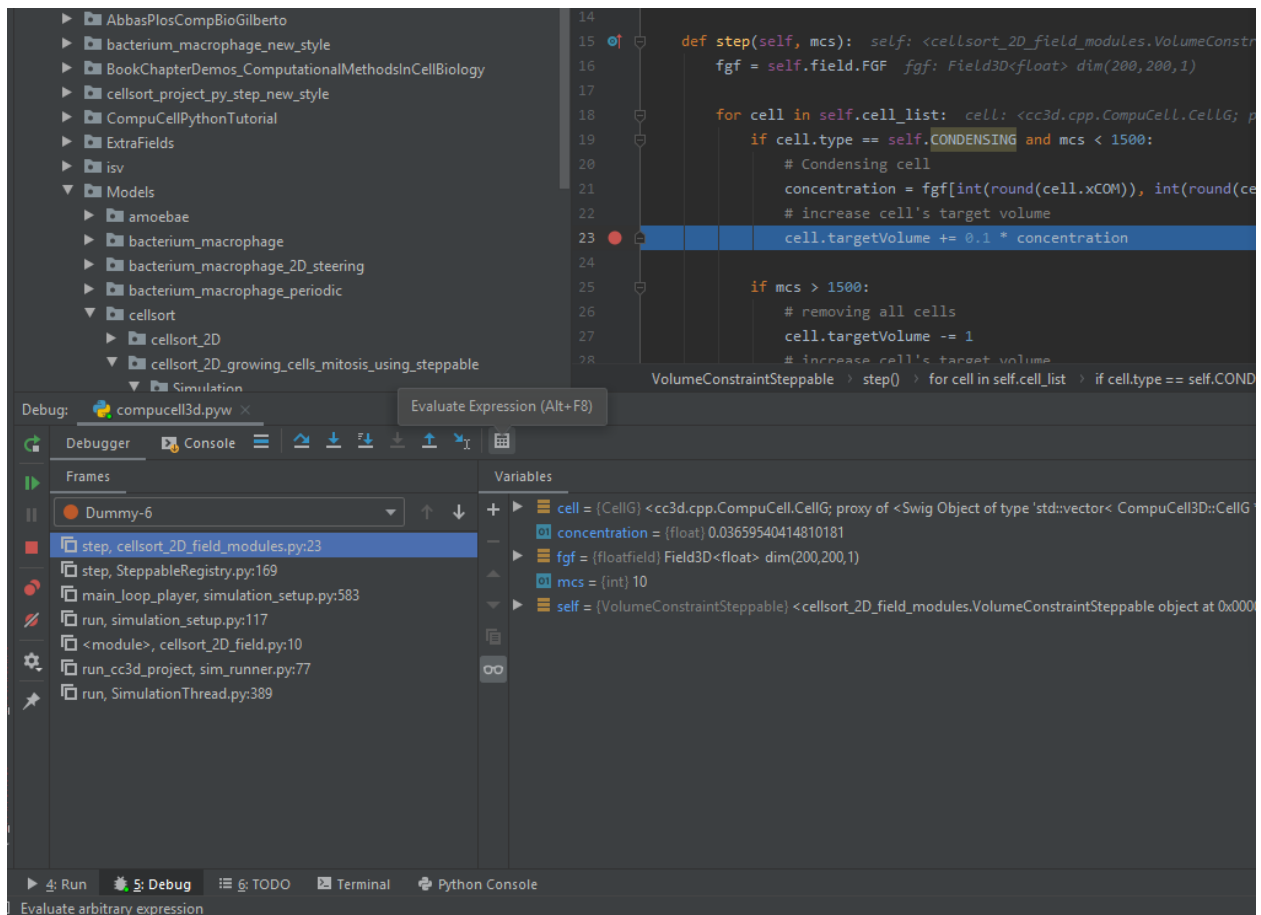
We can advance code execution by one line by hitting F8 or clicking `Step Over` from the debug menu. This will advance us to the next line of steppable. At this point we may open second `Evaluate Expression` window and this time type `concentration` to check the value of concentration variable and in 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:
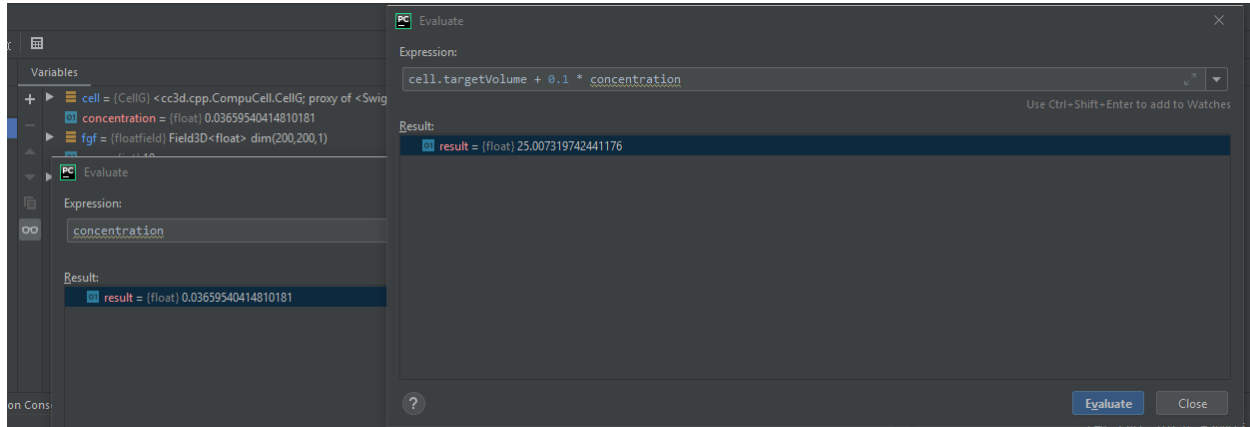
A very important feature of a breakpoint is the ability to enable them if certain condition is met. For example we want to break when concentration is greater than 0.5. To do so we right-click on the breakpoint red-circle
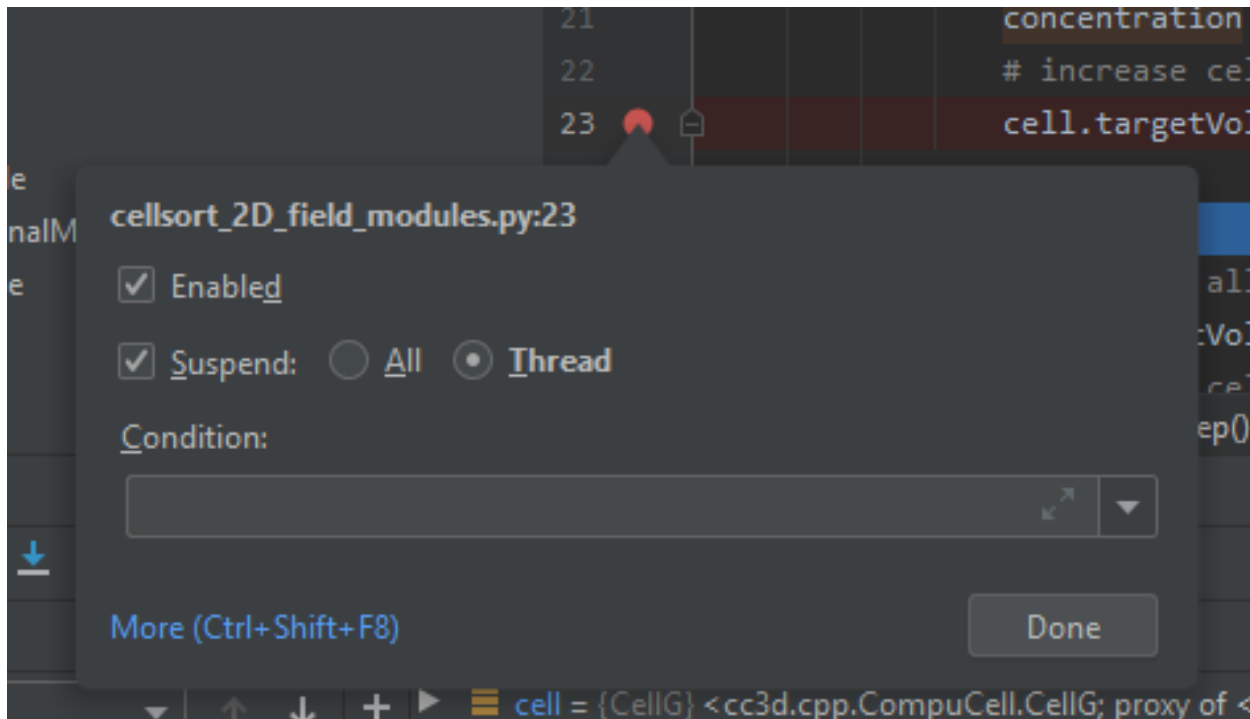
and in the line below we enter `concentration > 0.5` and click `Done` :

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

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

23 cell.targetV

cellsort_2D_field_modules.py:23

☑ Enabled

☑ Suspend: ○ All   ⦿ Thread

Condition:

concentration > 0.5

More (Ctrl+Shift+F8)   Done

Debug: compucell3d.pyw ✕

Resume Program   Console ≡

Frames   Variab

≡ Dummy-6   ↑ ↓ + ▶

step, cellsort_2D_field_modules.py:25
step, SteppableRegistry.py:169
main_loop_player, simulation_setup.py:583
run, simulation_setup.py:117

---

cellsort_2D_growing_cells_mitosis_using_steppable\cellsort_2D_growing_cells_mitosis_using_steppable.cc3d - Compu...   —   ☐   ✕

File   View   Simulation   Visualization   Tools   Window   Help

Model Editor   ⊟ ✕

Property	Value
> Potts	Potts
Plugin	VolumeLocalFlex
> Plugin	CellType
> Plugin	Contact
Plugin	CenterOfMass
> Steppable	DiffusionSolverFE
> Steppable	BlobInitializer

Graphics 0

xy ▼ 0   Cell_Field ▼

200

150

6.3. Step 3 - Debugging (stepping through) CC3D simulation and exploring other PyCharm features 39

the `concentration` variable we will see that indeed its value is greater than 0.5:
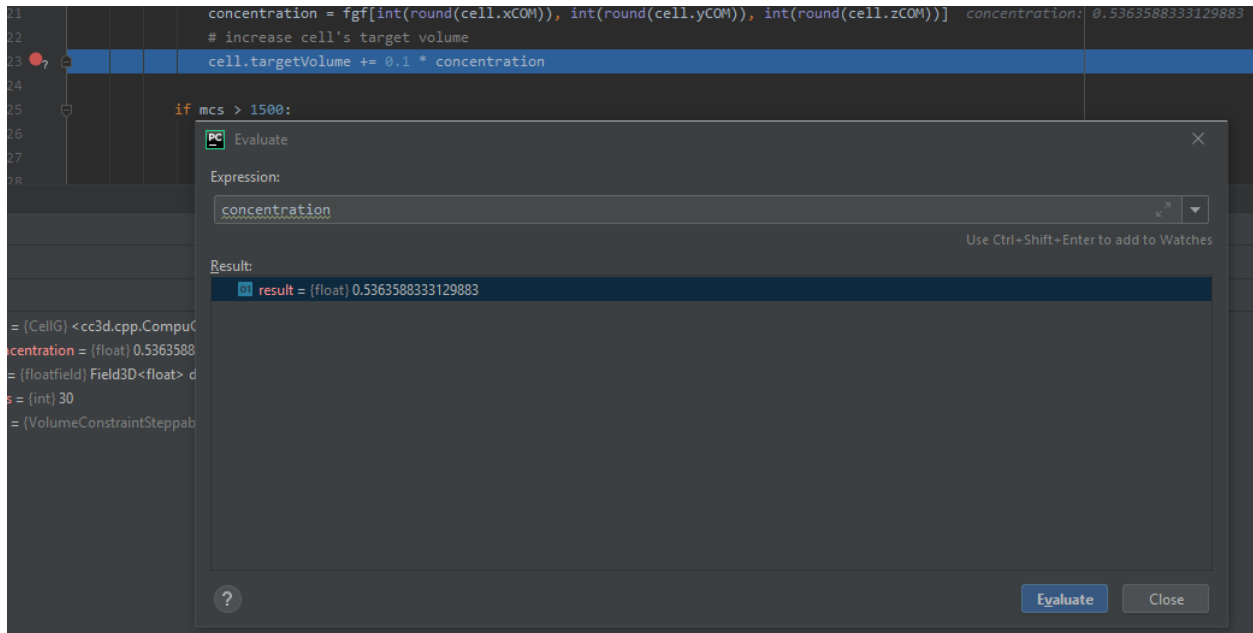
This technique of adding conditional breakpoints is quite useful when debugging simulations. If you have a lot of cells you do not want to step every single line of the loop by hitting F8. you want to press `Play` on the player and then have debugger inspect stop condition and stop once the condition has been satisfied.

This isa 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

6.4 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 candidadates, a function awe are looking for is `self.new_cell` and is listed somewhere in the middle:

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
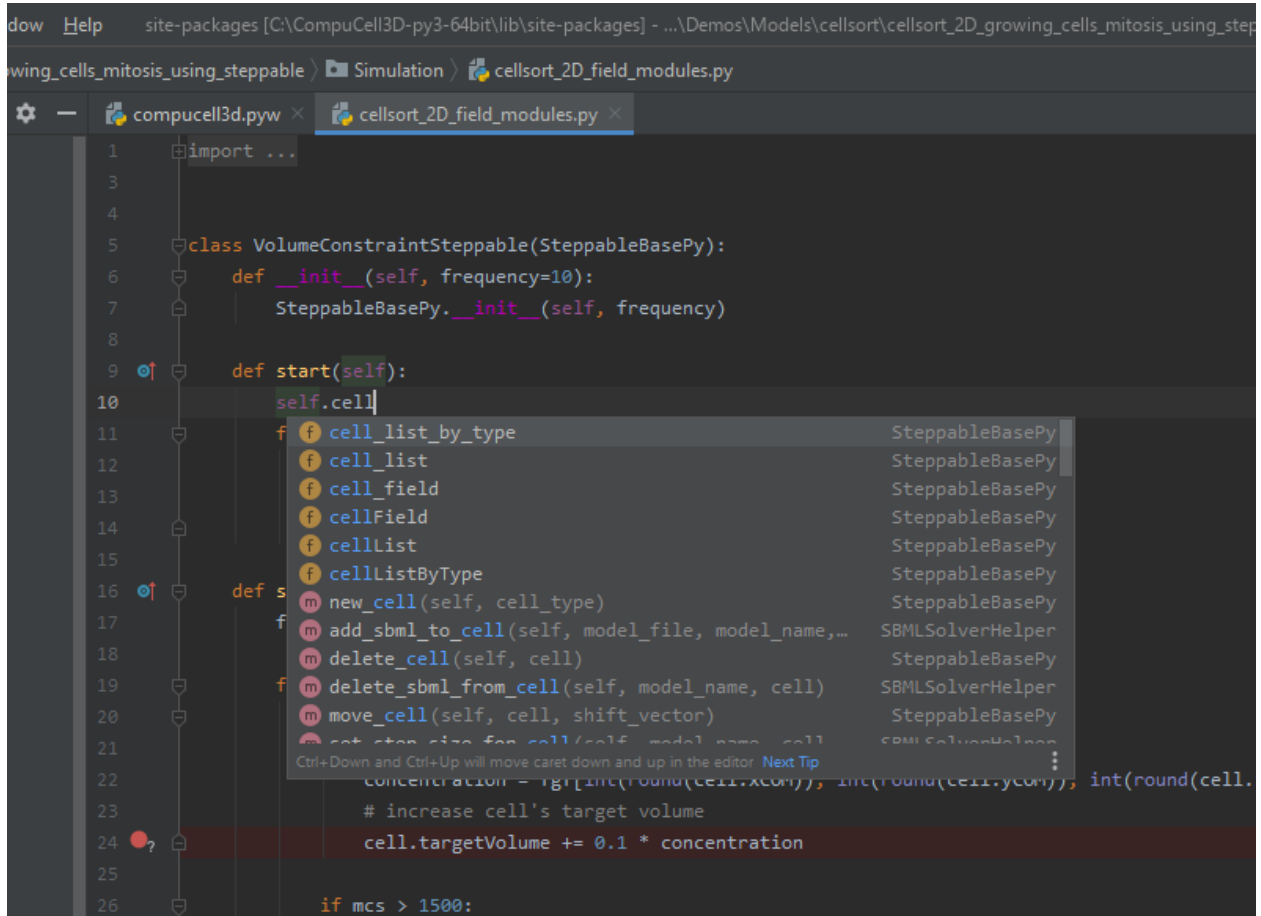
available and if you see something interesting you can always lookup documentation to see if indeed this function matches your needs. MOst importantly you can always suggest additional functions to be added to the steppables The best way to do it is to open up a ticket at https://github.com/CompuCell3D/CompuCell3D/issues. All you need is github account (those are free) nad you are ready to be part of CC3D development team.

6.5 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 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.

SteppableBasePy class

In the example above you may wonder how it is possible that it is sufficient to type:

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

to iterate over a list of all cells in the simulation. Where does `self.cell_list` come from and how it accesses/stores information about all cells? The full answer to this question is beyond the scope of this manual so we will give you only a hint what happens here. The `self.cell_list` is a member of a `SteppableBasePy` class. All CC3D Python steppable inherit this class and consequently `self.cell_list` is a member of all steppables (please see a chapter on class inheritance from any Python manual if this looks unfamiliar). Under the hood the `self.cell_list` is a handle, or a "pointer", if you prefer this terminology, 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. We use self.cellList to access C++ cell objects usually iterating over entire list of cells. The cell in the

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

Note: old syntax `self.cellList` is still supported

---

is a pointer to C++ cell object. You can easily see what members C++ cell object has by modifying the step function as follows:

```
def step(self, mcs):
    for cell in self.cellList:
        print dir(cell)
        break
```

The result looks as follows:

```
Step 0 Flips 94/10000 Energy 806 Cells 45 Inventory=45
['__class__', '__del__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattr__', '__getattribute__', '__hash_
_', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str_
_', '__subclasshook__', '__swig_destroy__', '__swig_getmethods__', '__swig_setmethods__', '__weakref__', 'angle', 'avera
geConcentration', 'clusterId', 'clusterSurface', 'ecc', 'extraAttribPtr', 'flag', 'fluctAmpl', 'iXX', 'iXY', 'iXZ', 'iYY
', 'iYZ', 'iZZ', 'id', 'lX', 'lY', 'lZ', 'lambdaClusterSurface', 'lambdaSurface', 'lambdaVecX', 'lambdaVecY', 'lambdaVec
Z', 'lambdaVolume', 'pyAttrib', 'setSurface', 'setVolume', 'setclusterId', 'setecc', 'setextraAttribPtr', 'setiXX', 'set
iXY', 'setiXZ', 'setiYY', 'setiYZ', 'setiZZ', 'setid', 'setlX', 'setlY', 'setpyAttrib', 'setxCM', 'setxCOM', 'setxCOMPre
v', 'setyCM', 'setyCOM', 'setyCOMPrev', 'setzCM', 'setzCOM', 'setzCOMPrev', 'subtype', 'surface', 'targetClusterSurface'
, 'targetSurface', 'targetVolume', 'this', 'type', 'volume', 'xCM', 'xCOM', 'xCOMPrev', 'yCM', 'yCOM', 'yCOMPrev', 'zCM'
, 'zCOM', 'zCOMPrev']
```

Figure 6 Checking out properties of a cell C++ object

The dir built-in Python function prints out names of members of any Python object. Here it printed out members of `CellG` class which represents CC3D cells. We will go over these properties later.

The simplicity of the above code snippets is mainly due to underlying implementation of `SteppableBasePy` class. You can find this class in `<CC3D\_installation\_dir>/pythonSetupScripts/PySteppables.py`. The definition of this class goes on for several hundreds lines of code (clearly a bit too much to present it here). If you are interested in checking out what members this class has use the dir Python function again:

```
def step(self,mcs):
    print ('Members of SteppableBasePy class')
    print dir(self)
```

You should know from Python programing manual that `self` refers to the class object. Therefore by printing `dir(self)` we are actually printing Python list of all members of `cellsortingSteppable` class. Because `cellsortingSteppable` class contains all the functions of `SteppableBasePy` class we can inspect this way base class `SteppableBasePy`. The output of the above simulation should look as follows:

Figure 7 Printing all members of SteppableBasePy class

If you look carefully, you can see that cellList is a member of `SteppabeBasePy` class. Alternatively you can study source code of `SteppablBasePy`.

One of the goals of this manual is to teach you how to effectively use features of `SteppableBasePy` class to create complex biological simulations. This class is very powerful and has many constructs which make coding simple.

Adding Steppable to Simulation using Twedit++

In the above example Python steppable was created by a simulation wizard. But what if you want to add additional steppable? Twedit++ lets you do it with pretty much single click. In the CC3D Project Panel right-click on Steppable Python file and choose Add Steppable option:
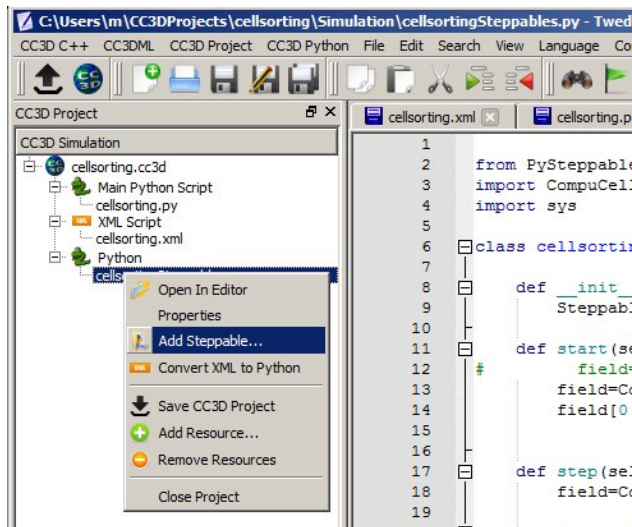
Figure 8 Adding seppable using Twedit++

The dialog will pop up where you specify name and type of the new steppable, call frequency. Click OK and new steppable gets added to your code.
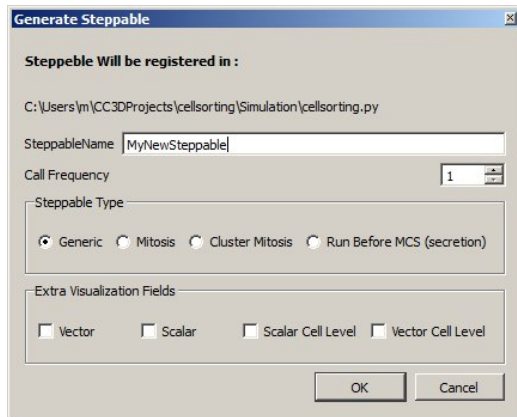
Figure 9 Configuring basic steppable properties in Twedit++.

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

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

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:

```python
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:

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

    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!).

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\\cellsorting\\Simulation\\cellsorting.xml`

```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>
    <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:

Figure 10 Output from simulation that has no cells

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

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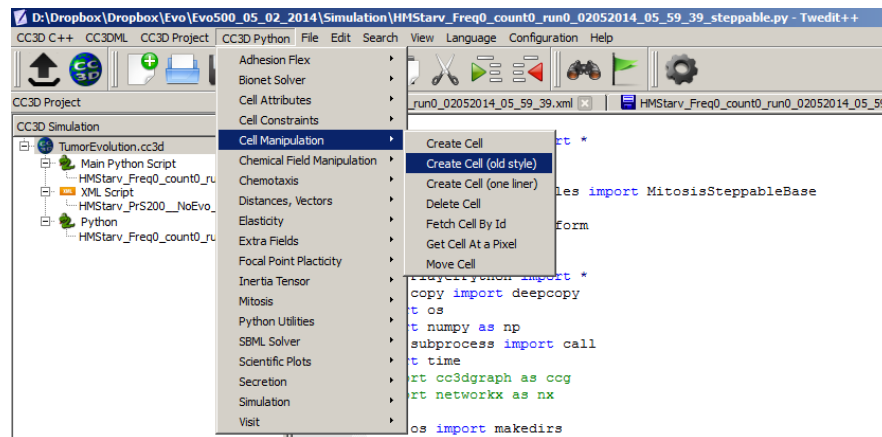
Figure 11 Inserting code snippet in Twedit++ to create cells. Notice that this is a generic code that usualy needs minor customizations.

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:

```python
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 corre-

sponding 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.

CHAPTER 11

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 righ 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:

```python
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
```

(continues on next page)

53

```python
        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:

```python
[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

```python
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.yCOM],
    p2=[ self.cellB.xCOM, self.cellB.yCOM, self.cellB.yCOM]
)
```

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

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`):

```python
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

```python
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:

```python
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`):

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

The output of this code will look as shown below:

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.

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 subfolders 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`):

```python
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:

```python
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:

```python
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]))
```

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

```python
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.

Adding plots to the simulation

Some modelers like to monitor simulation progress by displaying "live" plots that characterize 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 example code to get a flavor of what is involved when you want to work with plots in CC3D:

```python
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
        )

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

    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)
```

(continues on next page)

```
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 plot window (`self.plot_win`) – the arguments of this function are self-explanatory. After we have plot windows object (`self.plot_win`) we are adding actual plots to it. Here, we will plot two time-series data, one showing average volume of all cells and one showing instantaneous volume of 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)
```

We are specifying here plot symbol types (`Dots`, `Steps`), their sizes and colors. The first argument is then 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 logarithmic axis by using `y_scale_type='log'` as in the example above.

In the `step` function we are calculating average volume of all cells and extract instantaneous volume of cell with id 1. After we are done with calculations we are adding our results 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 current Monte Carlo Step (mcs) whereas the third is actual quantity that we want to plot on Y axis. We are done at this point

The results of the above code may look something like:
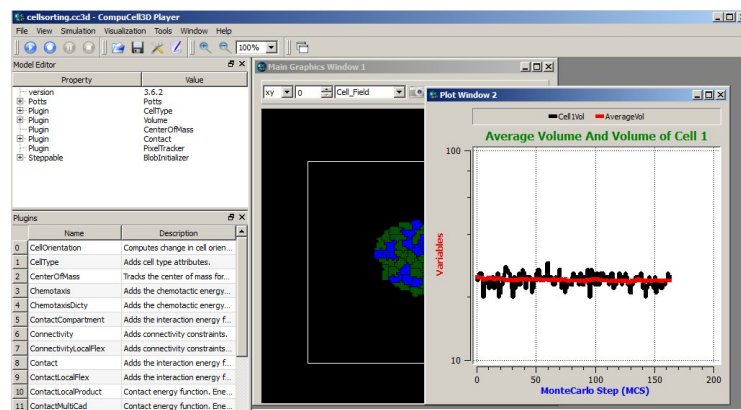
Figure 13 Displaying plot window in the CC3D Player with 2 time-series data.

Notice that the code is fairly simple and, for the most parts, self-explanatory. However, the plots are not particularly pretty and they all have same style. This is because this simple code creates plots based on same template. The plots are usable but if you need high quality plots you should save your data in the text data-file and use stand-alone plotting programs. Plots provided in CC3D are used mainly as a convenience feature and used to monitor current state of the simulation.

14.1 Histograms

Adding histograms to CC3D player is a bit more complex than adding simple plots. This is because you need to first process data to produce histogram data. Fortunately Numpy has the tools to make this task relatively simple. An example `scientificHistBarPlots` in CompuCellPythonTutorial demonstrates the use of histogram. Let us look at the example steppable (you can also find relevant code snippets in `CC3D Python-> Scientific Plots` menu):

```python
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)

        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)␣
→+ ".png")

            # 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 new plot window -`self.plot_win`- to the

Player. Subsequently we specify display properties of different data series (histograms). Notice that we can specify opacity using `alpha` parameter.

In the step function we first iterate over each cell and append their volumes to Python list. Later plot 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)
```

that takes an array of values and the number of bins and adds histogram to the plot window.

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.

At the end of the steppable we output histogram plot as a png image file using:

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

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

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

---

We construct `file_name` in such a way that it contains MCS in it. The image file will be written in the simulation outpt directory. 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.

CHAPTER 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:

```python
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.:

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

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

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.

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.

```python
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):
```

(continues on next page)

```
        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 exrta 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.

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:

```python
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()]
```

Th 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*random(), y*random(), z*random()] vector. Internally, self.field.VectorField is implemented as numpy array:

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

16.3 Scalar Field – cell level

Pixel based fields are appropriate for situations where we want to assign scalar of 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 ver 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:

```python
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.

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.

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:

```python
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:

```python
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__`):
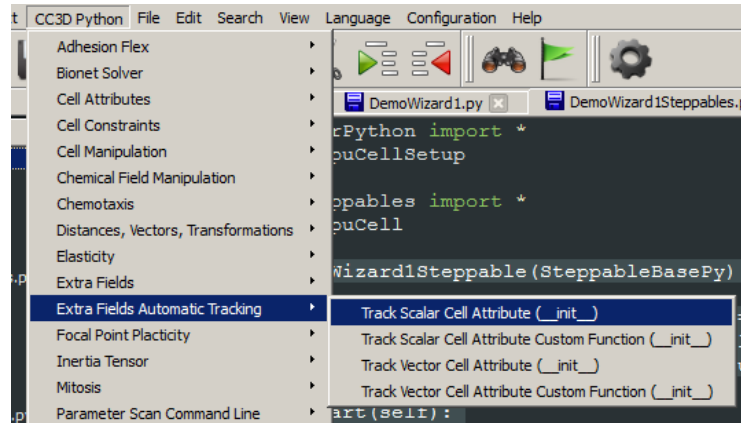
Figure 15 Setting up automatic tracking of cells' scalar attribute using Twedit++

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:

```python
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')

        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:

```python
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:

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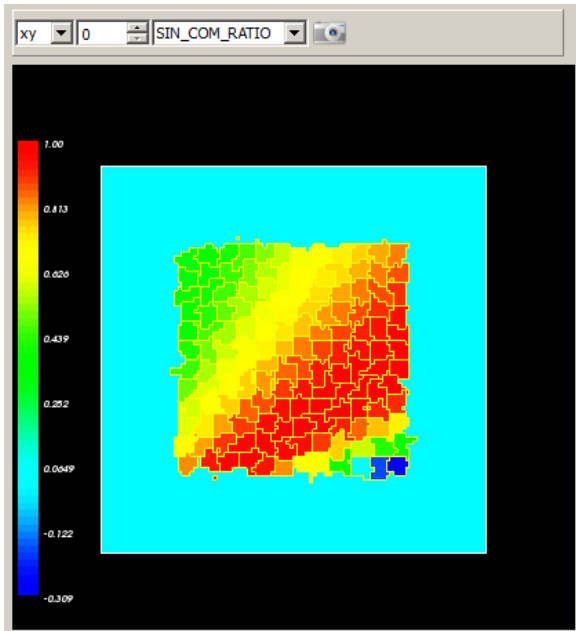
Figure 16. Automatic labeling of cells according to scala cell's attribute

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:

```python
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.

Field Secretion

PDE solvers in the CC3D allow users 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 `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`):

```python
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:
                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: As we mentioned in the introductory section we switched Python functions capitalization conventions. 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 should help users with identification of 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 which are of type 3 (notice we use 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. `secreteInsideCell` increases concentration by a given amount (here `300`) in every pixel occupied by a cell. `secreteInsideCellAtBoundary` and `secreteOutsideCellAtBoundary` increase concentration but only in pixels which at the boundary but are inside cell or outside pixels touching cell boundary. Finally, `secreteInsideCellAtCOM` increases concentration in a single pixel that is closest to cell center of mass of a cell.

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` and in the __init__ function it sets `self.runBeforeMCS` flag to `1`:

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

18.1 Direct (but somewhat naive) Implementation

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

```python
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.

18.2 Lattice Conversion Factors

In the code where we manually implement secretion at the cell'sCOM we use strange looking variables `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 lattice. 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.compucell3d.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) – see also Twedit++ CC3D Python menu Distances, Vectors, Transformations:

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

18.3 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 he secretor object.

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

```python
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 == 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 to 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 e.g. secreteInsideCellTotalCount returns object res that is an instance of FieldSecretorResult class that contains the summary of the secreion/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. :

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

For completeness we present a complete list of C++ signatures of all the functions that can be used to fine-control how uptake/secretion happens in CC3D. All those functions are members of the secretor object and are accessible from Python

```cpp
bool _secreteInsideCellConstantConcentration(CellG * _cell, float _amount);

FieldSecretorResult _secreteInsideCellConstantConcentrationTotalCount(CellG * _cell,
    float _amount);

bool _secreteInsideCell(CellG * _cell, float _amount);

FieldSecretorResult _secreteInsideCellTotalCount(CellG * _cell, float _amount);
```

```
bool _secreteInsideCellAtBoundary(CellG * _cell, float _amount);

FieldSecretorResult _secreteInsideCellAtBoundaryTotalCount(CellG * _cell, float _
→amount);

bool _secreteInsideCellAtBoundaryOnContactWith(CellG * _cell, float _amount,
const std::vector<unsigned char> & _onContactVec);

FieldSecretorResult _secreteInsideCellAtBoundaryOnContactWithTotalCount(CellG * _cell,
float _amount, const std::vector<unsigned char> & _onContactVec);

bool _secreteOutsideCellAtBoundary(CellG * _cell, float _amount);

FieldSecretorResult _secreteOutsideCellAtBoundaryTotalCount(CellG * _cell, float _
→amount);

bool _secreteOutsideCellAtBoundaryOnContactWith(CellG * _cell, float _amount,
const std::vector<unsigned char> & _onContactVec);

FieldSecretorResult  _secreteOutsideCellAtBoundaryOnContactWithTotalCount(CellG * _
→cell,
float _amount, const std::vector<unsigned char> & _onContactVec);

bool secreteInsideCellAtCOM(CellG * _cell, float _amount);

FieldSecretorResult secreteInsideCellAtCOMTotalCount(CellG * _cell, float _amount);

bool _uptakeInsideCell(CellG * _cell, float _maxUptake, float _relativeUptake);

FieldSecretorResult _uptakeInsideCellTotalCount(CellG * _cell, float _maxUptake,
→float _relativeUptake);

bool _uptakeInsideCellAtBoundary(CellG * _cell, float _maxUptake, float _
→relativeUptake);

FieldSecretorResult _uptakeInsideCellAtBoundaryTotalCount(CellG * _cell, float _
→maxUptake, float _relativeUptake);

bool _uptakeInsideCellAtBoundaryOnContactWith(CellG * _cell, float _maxUptake,
float _relativeUptake,const std::vector<unsigned char> & _onContactVec);

FieldSecretorResult _uptakeInsideCellAtBoundaryOnContactWithTotalCount(CellG * _cell,
float _maxUptake, float _relativeUptake, const std::vector<unsigned char> & _
→onContactVec);

bool _uptakeOutsideCellAtBoundary(CellG * _cell, float _maxUptake, float _
→relativeUptake);

FieldSecretorResult _uptakeOutsideCellAtBoundaryTotalCount(CellG * _cell, float _
→maxUptake, float _relativeUptake);

bool _uptakeOutsideCellAtBoundaryOnContactWith(CellG * _cell, float _maxUptake,
 float _relativeUptake,const std::vector<unsigned char> & _onContactVec);

FieldSecretorResult _uptakeOutsideCellAtBoundaryOnContactWithTotalCount(CellG * _cell,
float _maxUptake, float _relativeUptake, const std::vector<unsigned char> & _
→onContactVec);
```

```
bool uptakeInsideCellAtCOM(CellG * _cell, float _maxUptake, float _relativeUptake);

FieldSecretorResult   uptakeInsideCellAtCOMTotalCount(CellG * _cell, float _maxUptake,
→float _relativeUptake);
```

For example if we want to use `uptakeInsideCellAtCOMTotalCount(CellG * _cell, float _maxUptake, float _relativeUptake);` from python we would use the following code:

In this case `_cell` is a `cell` object that we normally deal with in Python, `_maxUptake` has value of `3` and `_relativeUptake` is set to `0.1`

In similar fashion we could use remaining functions listed above

Chemotaxis on a cell-by-cell basis

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

```python
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.MACROPHAGE:
                cd = self.chemotaxisPlugin.addChemotaxisData(cell, "ATTR")
                cd.setLambda(30.0)
                cd.assignChemotactTowardsVectorTypes([self.MEDIUM, self.BACTERIUM])
                break

    def step(self, mcs):
        if mcs > 100 and not mcs % 100:
            for cell in self.cell_list:
                if cell.type == self.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"/>              -->
   </ChemicalField>
 </Plugin>
```

As you can see we have commented out `ChemotaxisByType` but leaving information about fields so that chemotaxis plugin can fetch pointers to the fields. Clearly, leaving such definition of chemotaxis in the CC3DML would have no effect on 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 cell of type Macrophage we assign to it object called `ChemotaxisData` (we use `self.chemotaxisPlugin.addChemotaxisData` function to do that). ChemotaxisData object allows definition of all chemotaxis properties available via CC3DML but here we apply them to single cells. In our example code we set lambda describing chemotaxis strength and cells types that don't inhibit chemotaxis by touching our cell (in other words, 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 first cell of type `Macrophage`.

In the step function iterate through all cells and search for 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 is that lambda chemotaxis will go from positive number to negative number and cell that initially chemotaxed up the concentration gradient, now will 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 type of chemotaxis that corresponds in the CC3DML to the following call:

```
<ChemotaxisByType ChemotactTowards="CELL_TYPES" Lambda="1.0" SaturationCoef="100.0"␣
↪Type="CHEMOTAXING_TYPE"/>
```

The Python code would look like:

```
for cell in self.cell_list:
    if cell.type == self.MACROPHAGE:
        cd = self.chemotaxisPlugin.addChemotaxisData(cell, "ATTR")
        cd.setLambda(30.0)
        cd.setSaturationCoef(100)
        cd.assignChemotactTowardsVectorTypes([self.MEDIUM, self.BACTERIUM])
```

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

```
<ChemotaxisByType ChemotactTowards="CELL_TYPES" Lambda="1.0" SaturationLinearCoef="10.
↪1" Type="CHEMOTAXING_TYPE"/>
```

we would use the following Python snippet:

```
for cell in self.cell_list:
    if cell.type == self.MACROPHAGE:
        cd = self.chemotaxisPlugin.addChemotaxisData(cell, "ATTR")
        cd.setLambda(30.0)
        cd.setSaturationLinearCoef(100)
        cd.assignChemotactTowardsVectorTypes([self.MEDIUM, self.BACTERIUM])
```

CHAPTER 20

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)
```

(continues on next page)

83

```
    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.

20.1 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.

20.2 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_macrophage_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:
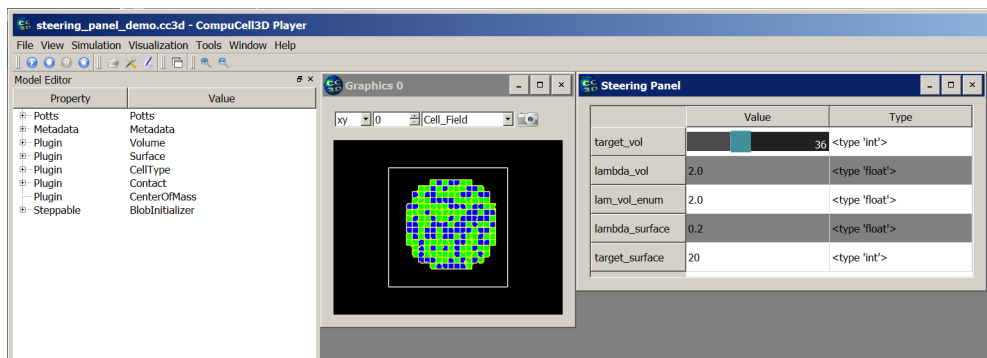
All the code that is necessary to get steering panel for Python parameters working will reside in Python steppable file.

To specify steering panel we have to define `add_steering_panel` function to our steppable (the function has to be called exactly that):

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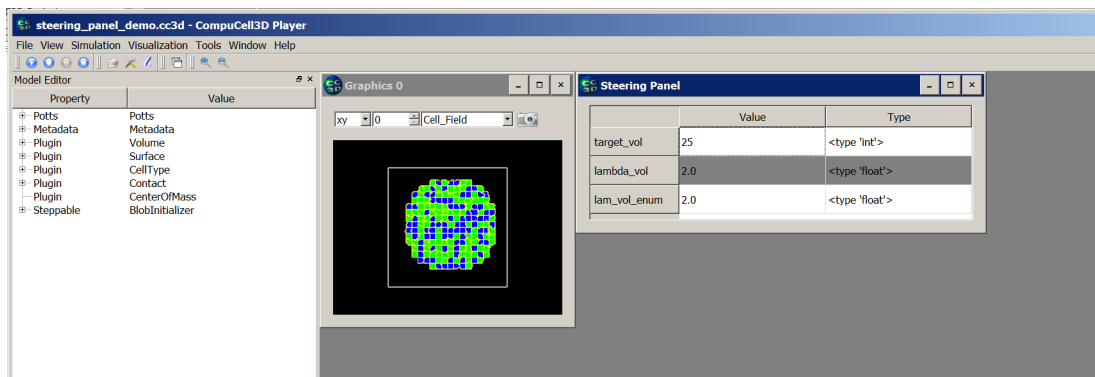
Note: You can add steering panel from multiple steppables. In such a case CC3D will gather all parameters you defined in the `add_steering_panel` function and display them in a single panel. For example, if our code has two steppables and we add steering parameters in both of them:

```
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')
```

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

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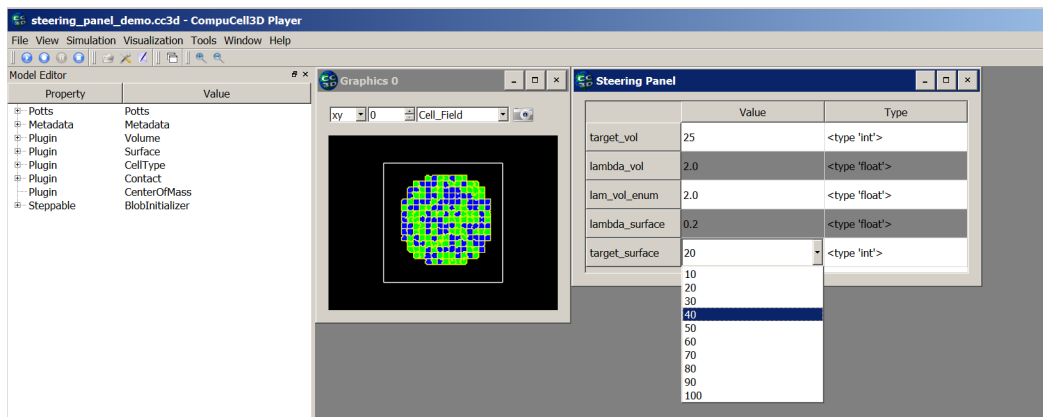
Notice that the last parameter `target_surface` uses `combobox` and appropriately the widget displayed is a pull-down-list

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:

```python
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:

```python
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:
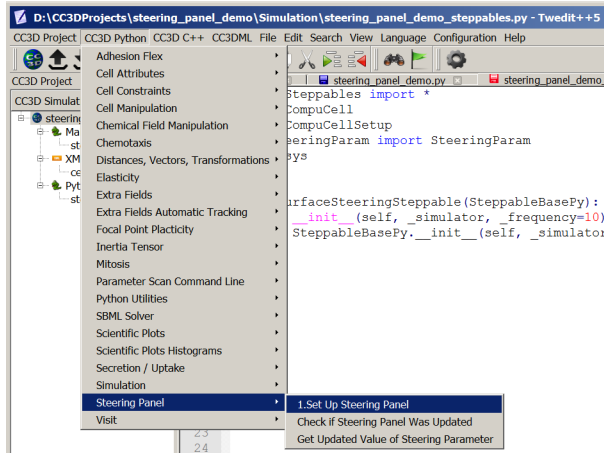
Replacing CC3DML with equivalent Python syntax

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 is is to right click XML file in the project panel and you will see option Convert XML To Python. When you choose this option Twedit++ will gegerate Python syntax which can replace your XML:
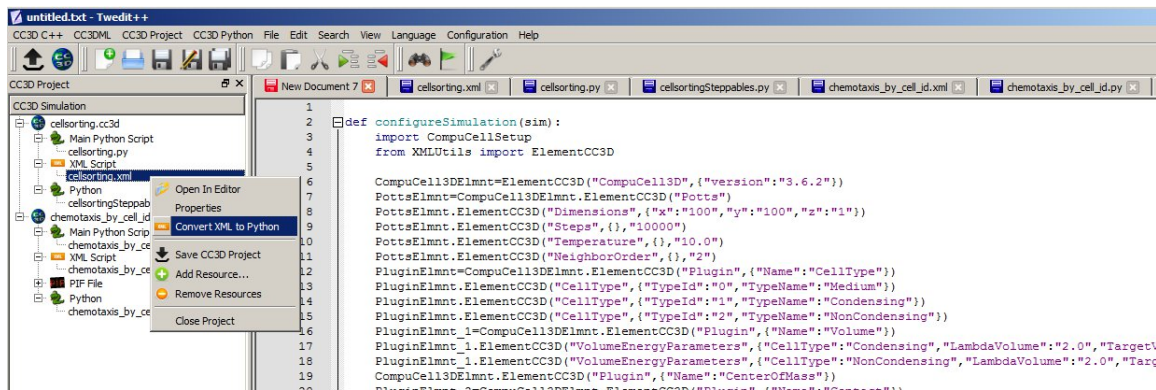
Figure 17 Generating Python code that replaces XML in Twedit++.

If we look at the XML code:

```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">
```

(continues on next page)

```xml
    <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>
```

And then at equivalent Python code:

```python
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:

```python
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:

```python
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.:

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

This syntax can bepresented 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:

```python
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.

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:

```python
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:
```

(continues on next page)

```
        # 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, 05)` 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 cels. This is however beyond the scope of this introductory

Setting cell membrane fluctuation ona 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.

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:

```python
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
```

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 simuation 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.

Controling steppable call frequency. Stopping simulation on demand or increasing maximum Monte Carlo Step.

When you create steppable using Twedit++, the editor will plunk template steppable code and will register this steppable in the main Python script. By default such steppable will be called after each completed MCS – as code snippet below shows:

```
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.

Sometimes in the simulation it may happen that initially you want to call steppable, say, every 50 MCS but later as the simulation goes on you may want to call it every 500 MCS or not call it at all. In such a case all you need to to 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.

On few occasions instead of waiting for a simulation to go through all MCS's you may have a metric determining if it is sensible to continue simulation or not. In case you want to stop simulation on demand. CC3D has useful function call that does exactly that. Place the following code (`CC3D Python->Simulation->Stop Simulation`)

```
self.stop_simulation()
```

anywhere in the steppable and after this call simulation will get stopped.

Inverse situation may also occur – you want to run 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.

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 froze 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"/>
   <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 int 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.

Resizing the lattice

When you have mitosis in your simulation the numbers of cells usualy 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:

```python
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))
```

(continues on next page)

```
        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 cell,s 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`.

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 x 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 x 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:

```python
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.

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:

```python
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:

```python
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 0x0000000007388EA0> >, 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 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
```

30.1 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.

30.2 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

30.3 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 ad 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_ist.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`

30.4 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 dicitonary where the key is a type io 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})
...
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`

Mitosis

In developmental simulations we often need to simulate cells which grow and divide. In earlier versions of Compu-Cell3D we had to write quite complicated plugin to do that which was quite cumbersome and unintuitive. The only advantage of the plugin was that exactly after the pixel copy which had triggered mitosis condition CompuCell3D called cell division function immediately. This guaranteed that any cell which was supposed divide at any instance in the simulation, actually did. However, because state of the simulation is normally observed after completion of full a Monte Carlo Step, and not in the middle of MCS it makes actually more sense to implement Mitosis as a steppable. Let us examine the simplest simulation which involves mitosis. We start with a single cell and grow it. When cell reaches critical (doubling) volume it undergoes Mitosis. We check if the cell has reached doubling volume at the end of each MCS. The folder containing this simulation is `CompuCellPythonTutorial/steppableBasedMitosis`

Let's see how we implement mitosis steppable:

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


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

    def step(self, mcs):

        cells_to_divide = []
        for cell in self.cell_list:
            if cell.volume > 50:
                cells_to_divide.append(cell)

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

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

        self.clone_parent_2_child()

        # implementing
        if self.parent_cell.type == self.CONDENSING:
            self.child_cell.type = self.NONCONDENSING
        else:
            self.child_cell.type = self.CONDENSING
```

The `step` function is quite simple – we iterate over all cells in the simulation and check if the volume of the cell is greater than 50. If it is we append this cell to the list of cells that will undergo mitosis. The actual mitosis happens in the second loop of the step function.

We have a choice there to divide cells along randomly oriented plane (line in 2D), along major, minor or user specified axis. When using user specified axis you specify 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 updateAttributes 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 called differently CC3D will not call it automatically. We can obviously call such function by hand, immediately we do the mitosis but this is not very elegant solution.

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 parent to 25:

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

Had we forgotten to do that parent cell would keep 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 attributes of parent and child cells are adjusted properly in the updateAttribute 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 child cell. That includes python dictionary attached to a cell. It is completely up to you to call this function or do manula copy of select attributes from parent to child cell.

If you would like to use automatic copy of parent attributes but skip certain dictionary elements (i.e. elements of the `cell.dict`) you would use the following call:

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

where 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_attribute` function where we manually set `parent` and `child` properties could look like that:

```python
def updateAttributes(self):

    self.child_cell.targetVolume = self.parent_ell.targetVolume
    self.child_cell.lambdaVolume = self.parent_ell.lambdaVolume
    if self.parent_cell.type == self.CONDENSING:
        self.child_cell.type = self.NONCONDENSING
    else:
        self.child_cell.type = self.CONDENSING
```

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

If you study the full example you will notice second steppable that we use to tom implement cell growth. Here is this steppable:

```python
class VolumeParamSteppable(SteppablePy):
    def __init__(self, frequency=1):
        SteppablePy.__init__(self, frequency)
        self.cellList = CellList(self.inventory)

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

    def step(self, mcs):
        for cell in self.cell_list:
            cell.targetVolume += 1
```

Again, this is quite simple module where in start function we assign `targetVolume` and `lambdaVolume` to every cell. In the step function we iterate over all cells in the simulation and increase target volume by 1 unit. As you may suspect to get it to work we have to make sure that we use Volume without any parameters in the CC3DML plugin instead of `Volume` plugin with parameters specified in the CC3DML.

At this point you have enough tools in your arsenal to start building complex simulations using CC3D. For example, combining steppable developed so far you can write a steppable where cell growth is dependent on the value of e.g. FGF concentration at the centroid of the cell. To get x coordinate of a centroid of a cell use the following syntax: .. code-block:: python

> cell.xCOM

or in earlier versions of CC3D

```
cell.xCM/float(cell.volume)
```

Analogous code applies to remaining components of the centroid. Additionally , make sure you include CenterOfMass plugin in the XML or the above calls will return 0's.

Python helper for mitosis is available from Twedit++ `CC3D Python->Mitosis.`

31.1 Directionality of mitosis - a source of possible simulation bias

When mitosis module divides cells (and, for simplicity, let's assume that division happens along vertical line) then the parent cell will always remain on the same side of the line i.e. if you run have a "stem" cell that keeps dividing all of it's offsprings will be created on the same side of the dividing line. What you may observe then that if you reassign cell type of a child cell after mitosis than in certain simulations cell will appear to be biased to move in one direction of the lattice. To avoid this bias you need to set call `self.set_parent_child_position_flag` function from `Base` class of the `Mitosis` steppable. When you call this function with argument 0 then relative position of parent and child cell after mitosis will be randomized (this is default behavior). When the argument is negative integer the child cell will always appear on the right of the parent cell and when the argument is positive integer the child cell will appear always on the left hand side of the parent cell.

Dividing Clusters (aka compartmental cells)

So far we have shown examples of how to deal with cells which consisted of only simple compartments. CC3D allows to use compartmental models where a single cell is actually a cluster of compartments. A cluster is a collection of cells with same clusterId . If you use "simple" (non-compartmentalized) cells then you can check that each such cell has distinct id and clusterId. An example of compartmental simulation can be found in `CompuCellPythonTutorial/clusterMitosis`. The actual algorithm used to divide clusters of cells is described in the appendix of the CompuCell3D manual.

Let's look at how we can divide "compact" clusters and by compact, we mean "blob shaped" clusters:

```python
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
```

(continues on next page)

```
            # 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 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 cell. This time however, after mitosis happens you have to reassign properties of `children` compartments and of `parent` compartments which usually means iterating over list of compartments. Conveniently this iteration is quite simple and `SteppableBasePy` class has a convenience function `get_cluster_cells` which returns a list of cells belonging to a cluster with a given cluster id:

```
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 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, cluster mitosis module assigns cell types to all the cells of child cluster and it does it in such a way so that child cell looks like a quasi-clone of 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_lis_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 manual setting of the parent and child cells you would use the flowing code:

```python
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_ell.clusterId)
        compartment_list_parent = self.get_cluster_cells(parent_cell.clusterId)
```

```python
    for i in range(len(compartment_list_child)):
        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
```

Python helper for mitosis is available from Twedit++ CC3D Python->Mitosis.

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 nd 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 go 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, **Physiologicaly 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 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 a C++ RoadRunnerLib developed by Herbert Sauro team. RoadRunnerLib in turn is based on the C# code written by Frank Bergmann. CC3D uses RoadRunnerLib 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 tapping into SBML Solver functionality. See `CC3DPython->SBML Solver` menu for options available.

Let us look at the example steppable that uses SBML Solver:

```python
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',
        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)

    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 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 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 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 optin setting should be first thing you do in your Python steppable file. If you want ot 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 path to SBML models file. The second one is model alias - it is a name you choose for model. It is an arbitrary model identifier that you use to retrieve model values. The third argument is a Python list that contains cell ids to which CC3D wil attach an instance of the SBML Solver.

Note: Each cell will get separate SBML solver object. SBML Solver objects associated with cells or free floating SBML Solvers are independent.

The fourth argument specifies the size of the integration step – here we use value of `0.5` time unit. The fifth argument passes initial conditions dictionary. Integration step size and initial conditions arguments are optional.

Each SBMLSolver function that associates models with a cell or adds free floating model calls `libroadrunner` functions that parse SBML, translate it to 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.

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. This is required because when we want to refer to such solver to extract model values we will do it using model name. The reason all models attached to cells have same name was that when we refer to such model we pass cell object and a name and this uniquely identifies the model. Free floating models need to have distinct names to be uniquely identified. Notice that last 3 calls to `self.add_free_floating_sbml` do not specify step size (we use default step size 1.0 time unit) nor initial conditions (we use whatever defaults are in the SBML code).

Finally, last two lines of start functions demonstrate how to add SBML Solver object to a single cell:

```
cell_20 = self.fetch_cell_by_id(20)
self.add_sbml_to_cell(model_file=model_file, model_name='dp', cell=cell_20)
```

Instead of passing list of cell ids we pass cell object (`cell_20`).

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 list of cell ids we pass list of cell types.

Let us move on to step function. 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 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 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 new state for for free floating model called `Medium_dp2`. If we wanted to print state of the model dp belonging to cell object called `cell20` we would use the following syntax:

```
print('cell_20, dp=', cell_20.sbml.dp.values())
```

To assign new values to dp model variables for cell20 we use the following syntax:

```
cell_20.sbml.dp['S1'] = 10
cell_20.sbml.dp['S2'] = 0.5
```

Note: We access free-floating SBML solved via `self.sbml.MODEL_ALIAS` syntax whereas SBML solvers associated with a particular cell are accessed using reference to cell objects e.g. `cell_20.sbml.MODEL_ALIAS`

Another useful operation within SBML Solver capabilities is deletion of models. This comes handy when at certain point in your simulation you no longer need to solve ODE's described in the SBML model. This is the syntax that deletes SBML from cell ids:

```
self.delete_sbml_from_cell_ids(model_name='dp', cell_ids=list(range(1, 11)))
```

As you probably suspect, we can delete SBML Solver instance from cell types:

```
self.delete_sbml_from_cell_types(model_name='dp' ,cell_types=range[self.A,self.B])
```

from single cell:

```
self.delete_sbml_from_cell(model_name='dp',cell=cell20)
```

or delete free floating SBML Solver object:

```
self.delete_free_floating_sbml(model_name='Medium_dp2'))
```

Note: When cells get deleted all SBML Solver models 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 up-dateAttributes function where you clone SBML Solver objects from 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 third argument - a Python list that specifies which models to copy. Here we are copying only dp models. All other models associated with parent cells will not be copied.

This example demonstrates most important capabilities of SBML Solver. The next example shows slightly more complex simulation where we reset initial condition of the SBML model before each integration step (`Demos/SBMLSolverExamples/DeltaNotch`).

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 juxtacrine signaling process, a cell's level of membrane-bound Delta depends on its intracellular level of activated Notch, which in turn depends on the average level of membrane-bound Delta of its neighbors. In such a situation, the Delta-Notch dynamics of the cells in a tissue sheet will depend on the rate of cell rearrangement and the fluctuations it induces. While the example does not explore the richness due to the coupling of 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 Ordinary Differential Equation (*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:

$$\frac{dD}{dt} \;=\; \left(\nu \times \frac{1}{1 + bN^h} - D\right) \tag{34.1}$$

$$\frac{dN}{dt} \;=\; \frac{\bar{D}^k}{a + \bar{D}^k} - N \tag{34.2}$$

where and are the concentrations of activated Delta and Notch proteins inside a cell, is the average concentration of activated Delta protein at the surface of the cell's neighbors, and are saturation constants, and are Hill coefficients, and is a constant that gives the relative lifetimes of Delta and Notch proteins.
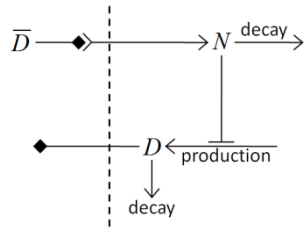
Figure 18 Diagram of Delta-Notch feedback regulation between and within cells.

For the sake of simplicity let us assume that we downloaded SBML model implementing Delta-Notch ODE's. How do we use such SBML model in CC3D? Here is the code:

```python
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 encounteting 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):

        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']
```

(continues on next page)

```
        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 using random values so that each cell has different SBML starting state. We also store the initial SBML in cell dictionary for visualization purposes – see full code in the `Demos/SBMLSolverExamples/DeltaNotch`. In the step function for each cell we visit its neighbors and sum value of Delta in the neighboring cells. We divide this value by the number of neighbors (this gives average Delta concentration in the neighboring cells - `D_avg`). We pass Davg to the SBML Solver for each cell and then carry out integration for the new time step. Before calling `self.timestep_sbm` function we store values of Delta and Notch concentration in the cell dictionary, but we do it for the visualization purposes only. As you can see from this example SBML Solver programing interface is convenient to use, not to mention SBML Solver itself which is very powerful tool which allows coupling cell-level and sub-cellular scales.

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:
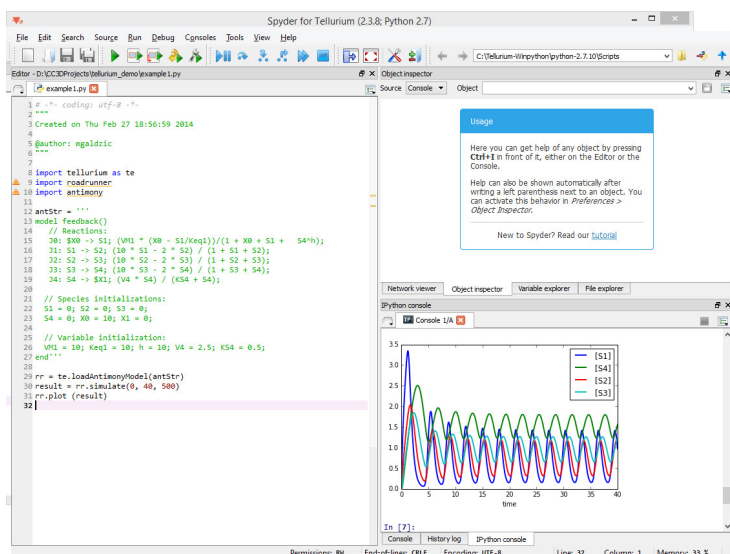

Hopefully this was easy. Now let's build our own model. We start creating our new reaction-kinetics model by pressing "New File" button - the first button on the left in the main toolbar:
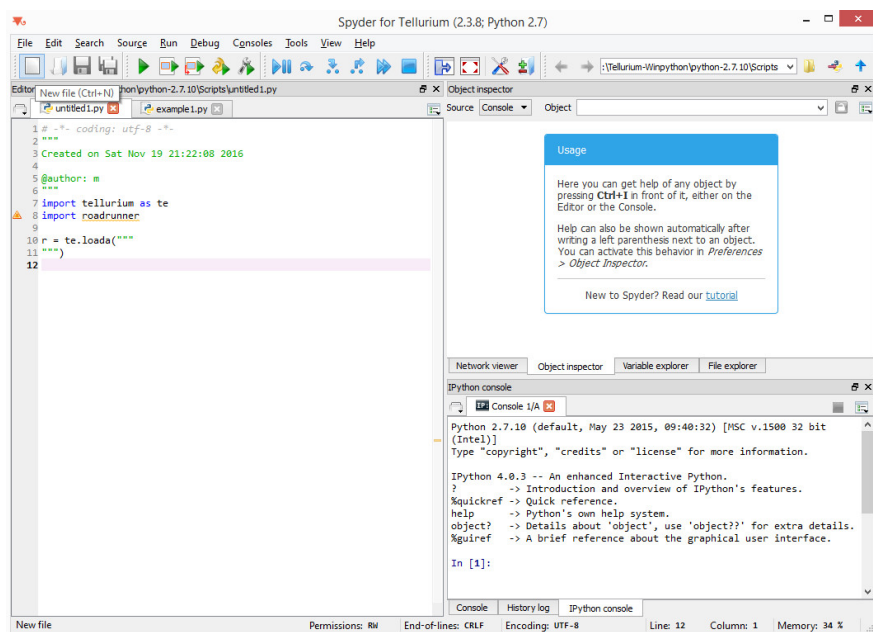
Pressing `New File` button opens up a Tellurium template for reaction-kinetics model. All we need to do is to define the model and add few statements that will export it in the SBML format.

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 (`S1` and `S2`) and two boundary species (`X0`, `X3`). 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:
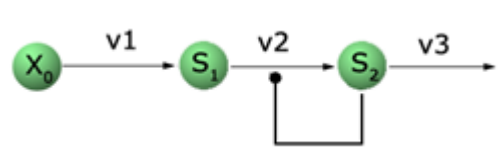
Qualitatively, `X0` is being kept at a constant concentration of 1.0 (we assume arbitrary units here) and `S1` accumulates at the constant rate `v1`. `S1` also gets "transformed" to `S2` at the rate `v2` which depends on concentration of S1 and S2 in such a way that when S1 and S2 are at relatively high values the reaction "speeds up" depleting quickly concentration of `S1`. Once `S1` concentration is low reaction `S1 -> S2` slows down allowing `S1` 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 `$X0 -> S1` represents reaction and `k1*X0` is a rate law that describes the speed at which species `X0` transform to `S1`.

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 `S2` into `X3`. 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:

```python
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:
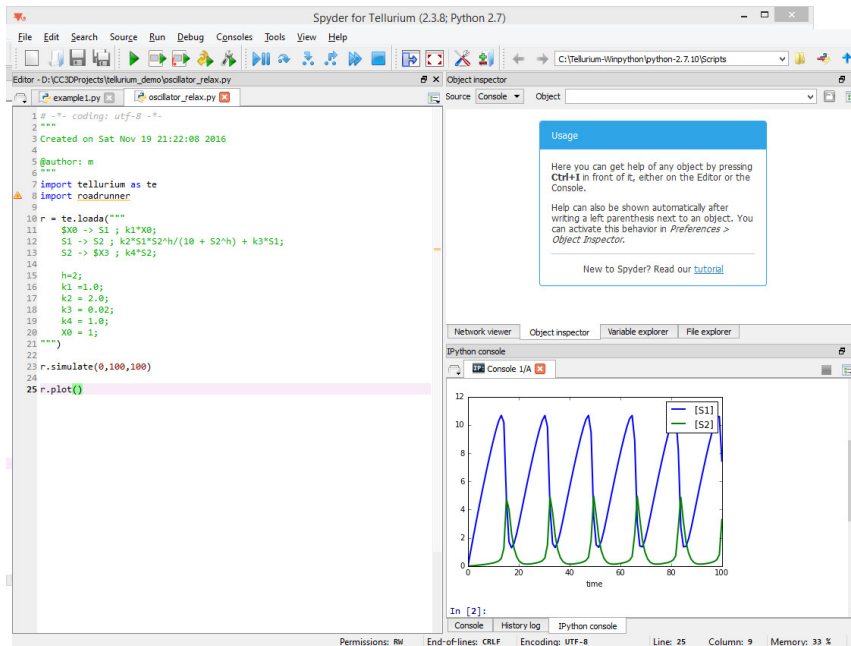
Notice that we need to add two lines of code - one to actually solve the model

```
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\oscilator_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

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[1] model definition languages. Antimony and CellML models can be 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 function `add_sbml_to_cell` there are corresponding functions `add_antimony_to_cell` and `add_cellml_to_cell`). 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)
```

---

[1] 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.

```python
    def start(self):

        antimony_model_string = """model rkModel()
        $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;
        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:

```python
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)

    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.

CHAPTER 37

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 sia 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 som 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:
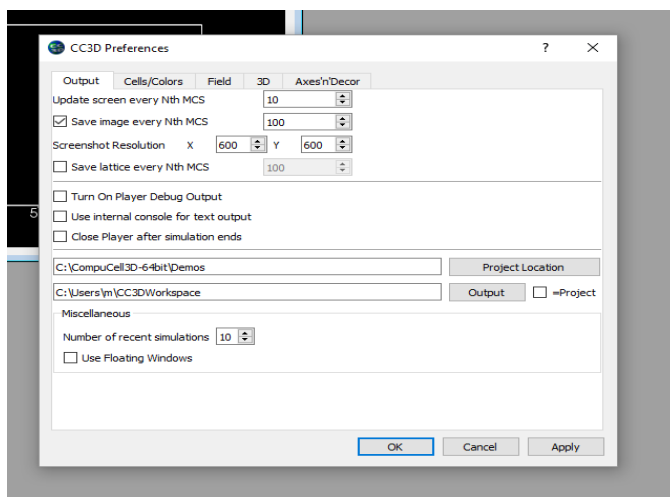

Fig 1. Enable screenshot output - check box next to `Save image every Nth MCS` and choose screenshot output frequency
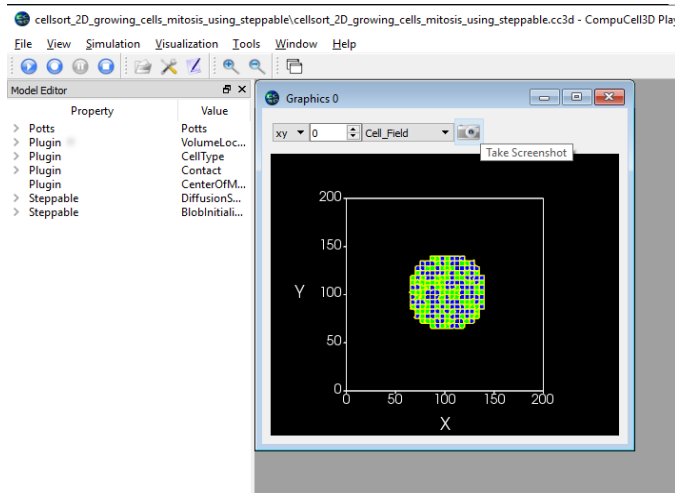
Fig 2. Open up simulation and start running it. Press `Pause` and click `camera button` (the button next to `Take screenshot` tool-tip) on the graphics configuration you would like to save.
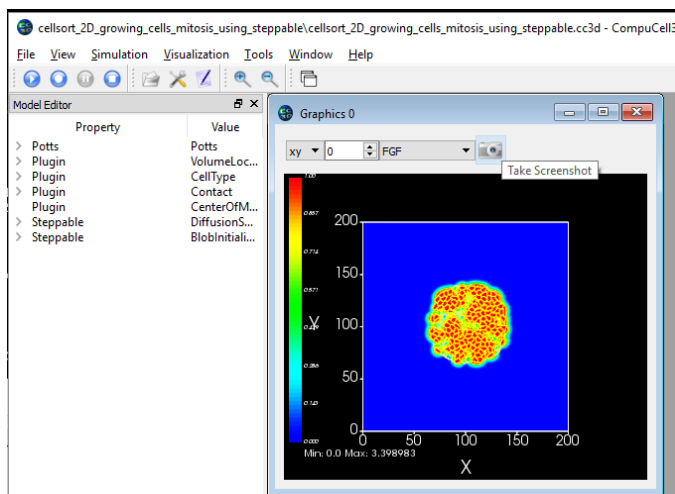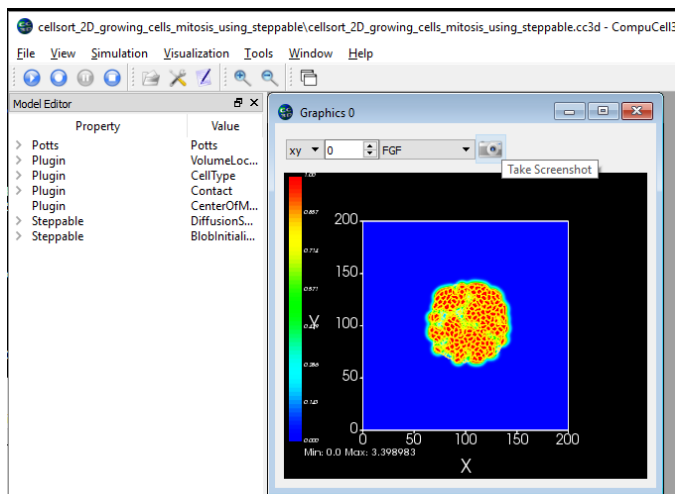

Fig 3. Repeat the sam process on other graphics configurations you would like to output as screenshots. Here we are adding screenshots for FGF field and for the cell field in 3D. See pictures above

The screenshot configuration data folder is stored along the simulation code in the original `.cc3d` project location:
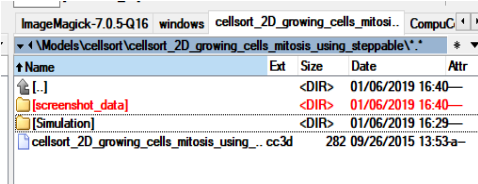
Fig 4. When you click camera button , CC3D will store screenshot configuration data in the `screenshot data` folder and it will become integral part od `.cc3d` project. Every time you run a simulation screenshots described there will be output to the CC3DWorkspace folder - unless you disable taking of 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:
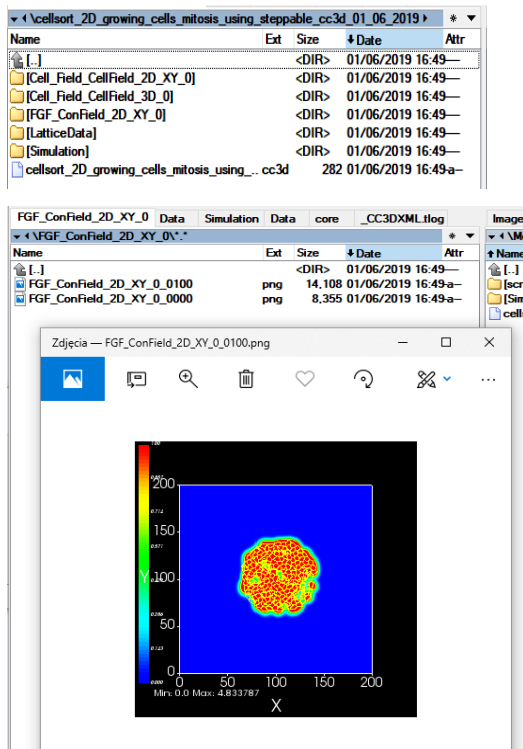

Fig 5. Screenshots are written to simulation output folder (*i.e.* subfolder of `CC3DWorkspace`)

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 paramter 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.

Implementation of parameter scans requires users to write simple JSON file with parameter scan specification and replacing actual values in the CC3DML or Python scripts with template markers. 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":{
```

(continues on next page)

```
                "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:

```xml
<CompuCell3D version="4.0.0">

    <Potts>
        <Dimensions x="100" y="{{y_dim}}" z="1"/>
        <Steps>{{steps}}</Steps>
        <Temperature>10.0</Temperature>
        <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:

```python
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.

38.1 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 thy can be lengthy. For this reason Player has a very convenient tool that let's 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 o 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:
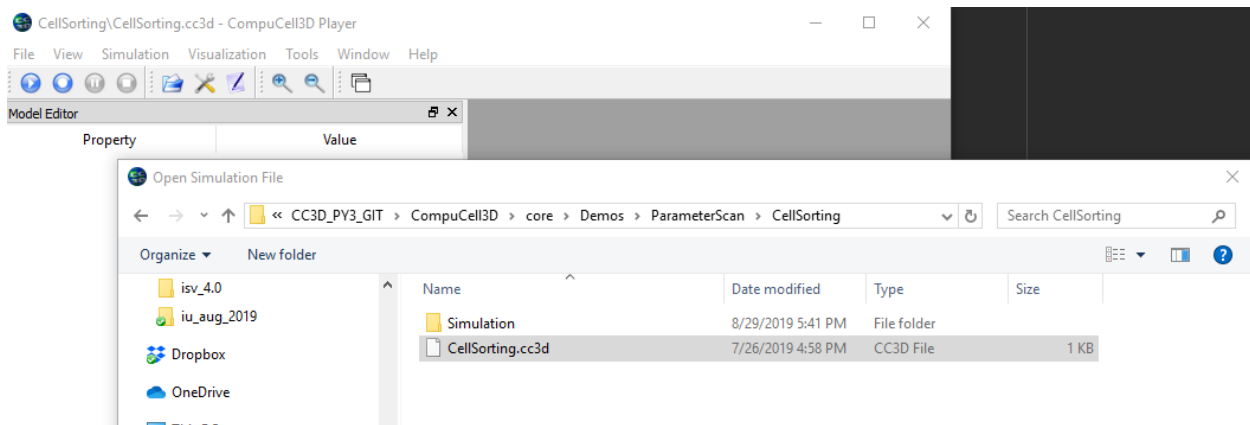
Next, when you click "Play" or "Step" buttons on the PLayer's tool bar you will get the following po-pup dialog:
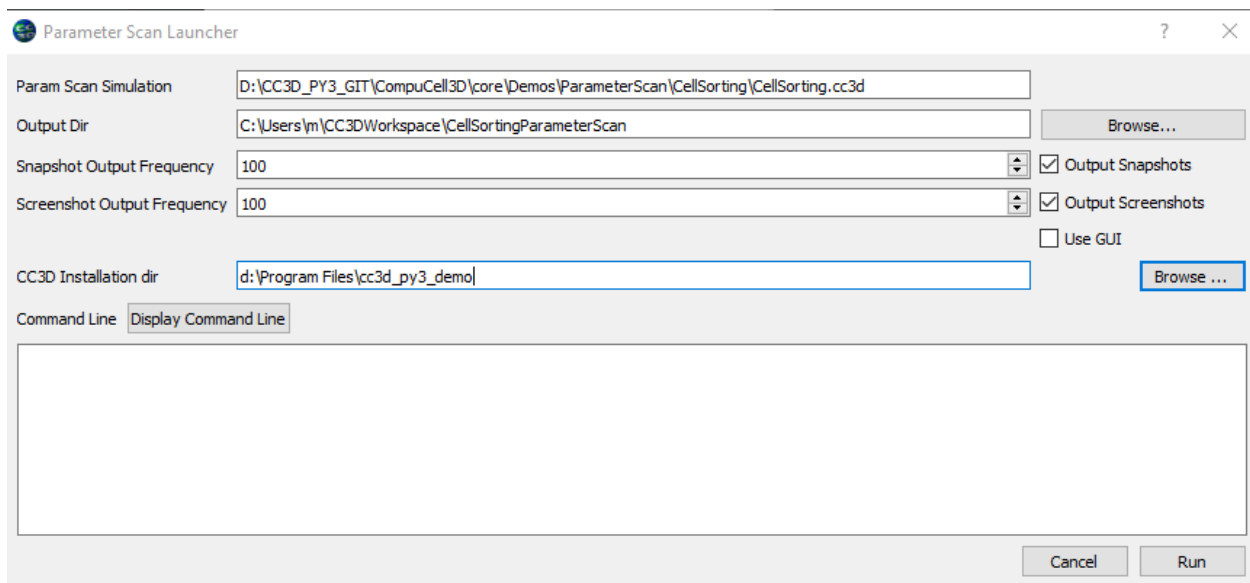
This dialog gives you options to configure how parameter scan gets executed:

`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 ofter 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 ofter 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 characteers. 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 you 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> --screenshot-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

---

## 38.2 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 caes but
   when you are dealing with a more sophisticated cases when you require e.g. points to be distributed logarith-mically then you woudl need to pregenerate such list in external program (e.g. Python console) and copy/paste values into parameter scan file. Fortunately CC3D allows you o 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.

CHAPTER 39

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 `500x500x500` 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. Compu-Cell3D makes all such tasks very easy. The only thing you need to do is to click `Enable output of restart snapshots`
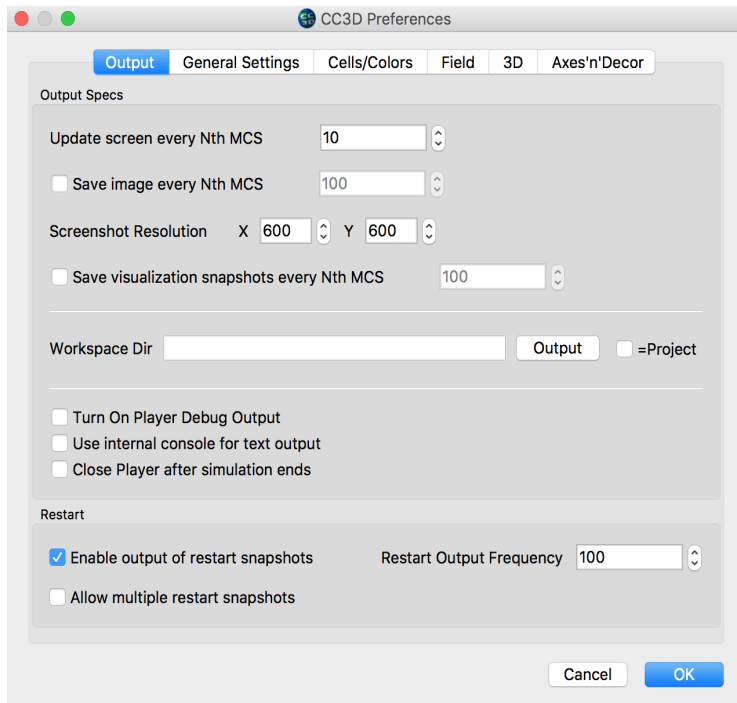
Alternatively if you run simulation using command line script `runScript` you need to pass the following argument to your command line:

```
--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:

```python
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
```

(continues on next page)

```
        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)',
                                            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.

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).

40.1 How does it work?

Note: All examples from this section can be found in `CompuCell3D/core/Demos/CallableCC3D`

40.1.1 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:

```python
from cc3d.core.PySteppables import *
from cc3d import CompuCellSetup
from random import random


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

    def start(self):
```

(continues on next page)

```
        pg = CompuCellSetup.persistent_globals
        input_val = pg.input_object


    def step(self, mcs):
        if mcs == 100:
            pg = CompuCellSetup.persistent_globals
            pg.return_object = 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 `CompuCellSetup.persistent_globals.input_object`. This variable 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 `CompuCellSetup.persistent_globals.return_object`. This variable 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. When accessing persistent_globals object , do not forget to include necessary import: `from cc3d import CompuCellSetup`.

40.1.2 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:

```python
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
   →simulation_fname line below
10     # simulation_fname = <direct path your simulation>
11     simulation_fname = join(dirname(dirname(__file__)), 'cellsort_2D', 'cellsort_2D.
   →cc3d')
12     root_output_folder = join(expanduser('~'), 'CC3DCallerOutput')
13
14     # this creates a list of simulation file names where simulation_fname is repeated
   →number_of_runs times
15     # you can create a list of different simulations if you want.
16     sim_fnames = [simulation_fname] * number_of_runs
17
18     ret_values = []
19     for i, sim_fname in enumerate(sim_fnames):
20         cc3d_caller = CC3DCaller(
21             cc3d_sim_fname=sim_fname,
22             screenshot_output_frequency=10,
23             output_dir=join(root_output_folder,f'cellsort_{i}'),
24             sim_input=i,
25             result_identifier_tag=i
26         )
27
```

```
28          ret_value = cc3d_caller.run()
29          ret_values.append(ret_value)
30
31      print('return values', ret_values)
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 `CompuCellSetup.persistent_globals.input_object`, and whatever we set `CompuCellSetup.persistent_globals.return_object` 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.

40.1.3 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`.

40.1.4 Step 3

In order to run above script you need to set up few environment variables and, in particular, specify location of appropriate Python interpreter. This must be a Python interpreter that is either shipped with CC3D binary distribution or a one that you used to compile CC3D against. Let's get started. We will walk you steps necessary to run above scripts on various platforms. For your convenience we provide simple scripts where you specify two paths (CC3D installation path and Path to Python interpreter) and then the script takes care of setting your environment

Let's start with windows.

40.1.5 Windows

Go to `CompuCell3D/core/Demos/CallableCC3D/environment_var_setters/ cc3d_caller_env_var_set_windows.bat` and open it in your editor and you will see the following content:

```
@ECHO OFF
@SET PREFIX_CC3D=<path to where cc3d is installed>
@SET PYTHON_INSTALL_PATH=<path to where python used for cc3d is installed>
@SET PYTHONPATH=%PREFIX_CC3D%\lib\site-packages
```

Replace it with actual paths to where CC3D is installed and to location where python interpreter used with CC#D resides

For example in my case CC3D is installed to `c:\CompuCell3D-py3-64bit\` so I modify the script as follows:

```
@ECHO OFF
@SET PREFIX_CC3D=`c:\CompuCell3D-py3-64bit
@SET PYTHON_INSTALL_PATH=`c:\CompuCell3D-py3-64bit\python36
@SET PYTHONPATH=%PREFIX_CC3D%\lib\site-packages
```

I save this script as `c:\CompuCell3D-py3-64bit\Demos\CC3DCaller\environment_var_setters\win_set_path. bat`. I open console and execute this script by typing:

```
cd c:\CompuCell3D-py3-64bit\Demos\CC3DCaller\environment_var_setters
win_set_path.bat
```

Next I navigate to location where my script from `Step 2` is installed

```
cd c:\CompuCell3D-py3-64bit\Demos\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\Demos\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.

40.1.6 Linux

Running simulation on Linux is very similar to running on Windows. We start by modifying script `Demos/CallableCC3D/environment_var_setters/cc3d_caller_env_var_set_linux.sh`:

```
#!/bin/sh
current_directory=$(pwd)

# necessary to enforce standard convention for numeric values specification on non-
→English OS
export LC_NUMERIC="C.UTF-8"

# export PREFIX_CC3D=/home/m/411_auto
export PREFIX_CC3D=<path to where cc3d is installed>

# export PYTHON_INSTALL_PATH=/home/m/miniconda3/envs/cc3d_2021/bin
export PYTHON_INSTALL_PATH=<path to where python executable is. Make sure it is same␣
→python as used by cc3d>

...
```

and in the line that says `export PREFIX_CC3D` we specify where CC3D is installed in your linux system and in line with `export PYTHON_INSTALL_PATH` we specify path to where python interpreter used in CC3D is installed

In my case I changed the script as follows and saved it under `/home/m/411_auto/pathset.sh`

```
#!/bin/sh
current_directory=$(pwd)

# necessary to enforce standard convention for numeric values specification on non-
→English OS
export LC_NUMERIC="C.UTF-8"

export PREFIX_CC3D=/home/m/411_auto
export PYTHON_INSTALL_PATH=/home/m/miniconda3/envs/cc3d_2021/bin

...
```

Note: My `PYTHON_INSTALL_PATH` points to Python installation folder that is outside CC3D installation folder. This is because I compiled CC3D on linux. If you do not compile but want to use Python that is bundled with the distribution you would type `export PYTHON_INSTALL_PATH=$PREFIX_CC3D/Python37/bin`. Regardless of what your configuration is make sure that you are specifying here a path to the folder in which Python interpreter resides.

Next, in the open console I execute path setting script :

```
source /home/m/411_auto/pathset.sh
```

Note, it is essential to call `source` to ensure that the paths you set in the `pathset.sh` script will "carry over" to your open console. On windows we did not have to do it.

Next, I go to `/home/m/411_auto/Demos/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}]
```

40.1.7 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 `cc3d_caller_env_var_set_mac.command` environment variable setter script instead of `cc3d_caller_env_var_set_mac.sh`. As before, all you need to set is the Path to Where CC3D is installed and path to the folder where Python interpreter resides. The remaining steps are analogous as those for linux i.e. modify environment setter script, run `source cc3d_caller_env_var_set_mac.command` to set environment variables and modify script from `Step 2` to include direct path to the simulation

Implementing Energy Functions in Python

Warning: Previous versions of CC3D allowed implementing energy functions in Python. Starting with 4.0.0 we are deprecating this feature. We strongly recommend that you write CC3D extensions such as energy functions or lattice monitors in C++ to achieve optimal performance. If you would like to write your own energy function we strongly recommend that you do this in C++. Twedit++ has C++ module assistant that generates template for any type of CompuCell3D C++ module and makes overall C++ CompuCell3D module development much easier. Go to CC3D C++ -> Generate New Module . . .

Appendix A

In this appendix we present 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 corresponging 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 corresponging 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 floarting 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 beween two 3D points

`distance_vector_between_cells` - convenience function calculating distance vector beween 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++ FocalPointPlasticityData 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 withing 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` - reassignes 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 of after (runBeforeMCS=1). 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.

Appendix B

In this appendix we present alphabetical list of CellG attributes:

clusterId`` - cluster id

`clusterSurface` - total surface of a cluster that a given cell belongs to. Needs ClusterSurface Plugin

`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 or SurfaceTracker 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