CompuCell3D Python Scripting manual

Version 3.7.5

Maciej H. Swat, Julio Belmonte

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

1. Contents

The focus of this manual is to teach you how to use Python 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. http://hetland.org/writing/instant-hacking.html, http://hetland.org/writing) or simply purchase a book on introductory Python programming.

2. 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 to 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 clumsym but over-optimized code. If you have any doubts about this philosophy ask any programmer or professor of SoftwareEngineering 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 youstill need to develop high performance C++ modules, CompuCell3D andTwedit++have excellent tools which make even C++ programing quite pleasurable (Hint: look at CC3D C++ menu in the Twedit++)

3. 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 disctinc 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`
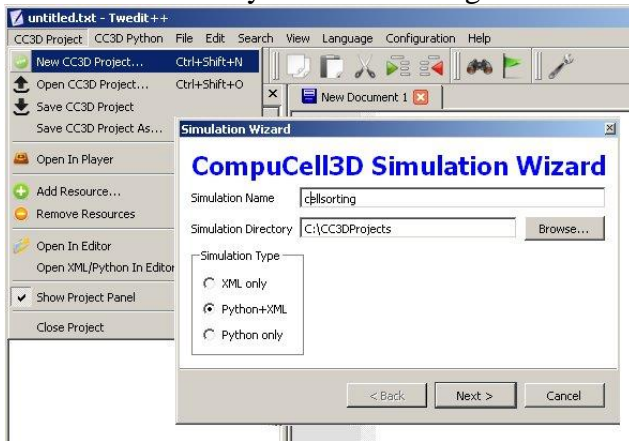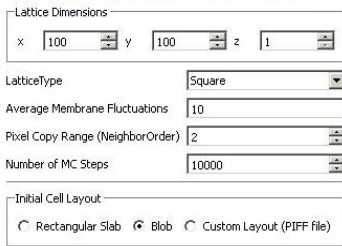
Figure 1 Invoking the CompuCell3D Simulation Wizard from Twedit++.

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.
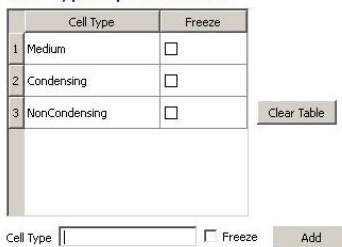
Figure 2 Specification of basic cell-sorting properties in Simulation Wizard.

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

Figure 3 Specification of cell-sorting cell types in Simulation Wizard.

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

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

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):

```
<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>
```

---

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

Cellsorting.cc3d stores names of the files 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 modelrs can refer to file resources using partial paths i.e. 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`

```
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()

CompuCellSetup.initializeSimulationObjects(sim,simthread)

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

from cellsortingSteppables import cellsortingSteppable
steppableInstance=cellsortingSteppable(sim,_frequency=1)
steppableRegistry.registerSteppable(steppableInstance)

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

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

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

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

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

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

```
sim,simthread = CompuCellSetup.getCoreSimulationObjects()
CompuCellSetup.initializeSimulationObjects(sim,simthread)
```
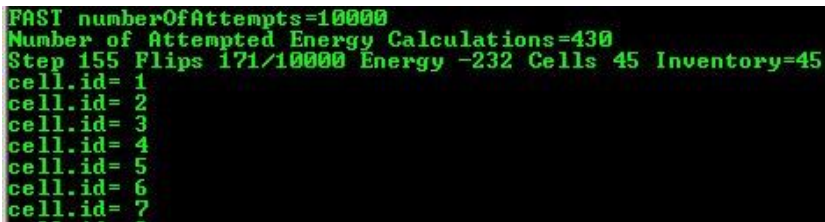
We then create a steppable **registry** (a Python **container** that stores steppables, *i.e.*, a list of all steppables that the Python code can access) and pass it to the function that runs the simulation. We also create and register cellsortingSteppable:

```
steppableRegistry=CompuCellSetup.getSteppableRegistry()

from cellsortingSteppables import cellsortingSteppable
steppableInstance=cellsortingSteppable(sim,_frequency=1)
steppableRegistry.registerSteppable(steppableInstance)

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

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 PySteppables import *
import CompuCell
import sys
class cellsortingSteppable(SteppableBasePy):

    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator,_frequency)
    def start(self):
        # any code in the start function runs before MCS=0
        pass
    def step(self,mcs):
        #type here the code that will run every _frequency MCS
        for cell in self.cellList:
            print "cell.id=",cell.id
    def finish(self):
        # Finish Function gets called after the last MCS
        pass
```

Inside step function we have the following code snippet:

```
        for cell in self.cellList:
            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 Pyton 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 (which in turn imports SteppablePy). The nice feature of inheritance is that oncve 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) .

4. SteppableBasePy class

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

```
for cell in self.cellList:
```

to iterate over a list of all cells in the simulation. Where does `self.cellList` come from and how it acceses/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.cellList` is a member of a SteppableBasePy class. All CC3D Python steppable inherit this class and consequently `self.cellList` 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.cellList` 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.cellList:
```

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:

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`. 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 programming 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.

5. 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++ let's you do it with pretty much single click. In the CC3D Project Panel right-click on Steppable Python file and choose Add Steppable option:
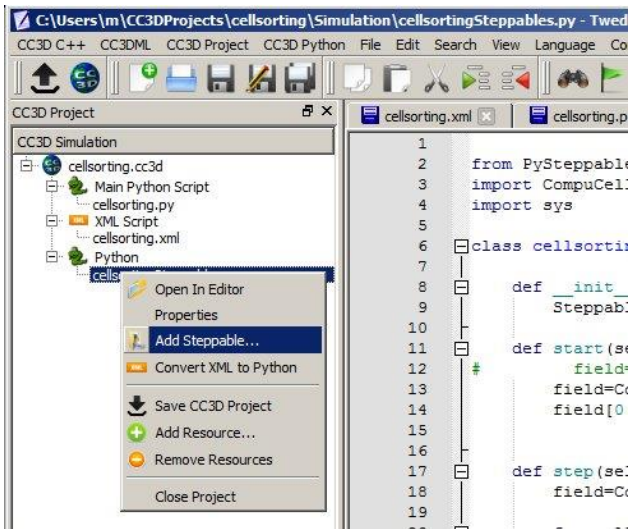
Figure 8 Adding seppable using Twedit++

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.
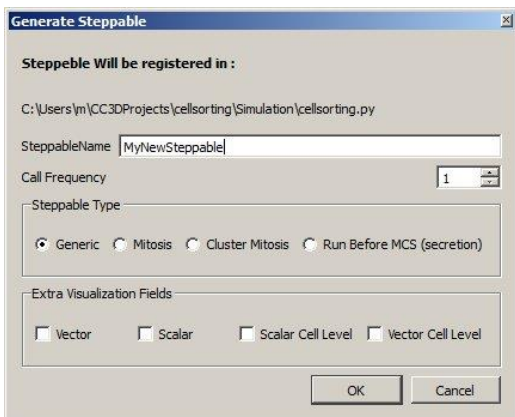
Figure 9 Configuring basic steppable properties in Twedit++.

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

```
from cellsortingSteppables import MyNewSteppable
instanceOfMyNewSteppable=MyNewSteppable(_simulator=sim,_frequency=1)
steppableRegistry.registerSteppable(instanceOfMyNewSteppable)
```

6. Passing information between steppables

When you work with more than one steppable (and it is a good idea to work with several steppables each of which has well defined purpose) you may sometimes need to acces/change member variable of one steppable inside the code of another steppable. If you are seasoned Python programmer, you can easily find workaround. However, we have added a convenience function to `SteppableBasePy` class that makes accessing content of one steppable from another module very easy and, let's say, elegant. Here is an example (see also `Demos\CompuCellPythonTutorial\SteppableCommunication`):

```
class SteppableCommunicationSteppable(SteppableBasePy):
```

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

    def step(self,mcs):
        extraSteppable=self.getSteppableByClassName('ExtraSteppable')
        print 'extraSteppable.sharedParameter=',extraSteppable.sharedParameter

class ExtraSteppable(SteppableBasePy):
    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator,_frequency)
        self.sharedParameter=25

    def step(self,mcs):
        print "ExtraSteppable: This function is called every 1 MCS"
```

In the `SteppableCommunicationSteppable` class, inside step function we fetch (using class name) another `ExtraSteppble` steppable. Once we have access to object of type `ExtraSteppble` we can access/change parameters of this steppable.

Remark: This approach will work fine if you create only one steppable object for each steppable class. In case you create two objects of the same steppable class, the presented method fails. However, most, if not all, CC3D simulations rely on an unwritten rule - one steppable object for each steppable class.

7. 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`

```
<CompuCell3D version="3.6.2">
   <Potts>

      <!-- Basic properties of CPM (GGH) algorithm -->
      <Dimensions x="100" y="100" z="1"/>
      <Steps>10000</Steps>
      <Temperature>10.0</Temperature>
      <NeighborOrder>2</NeighborOrder>
   </Potts>

   <Plugin Name="CellType">

      <!-- Listing all cell types in the simulation -->
      <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">
```
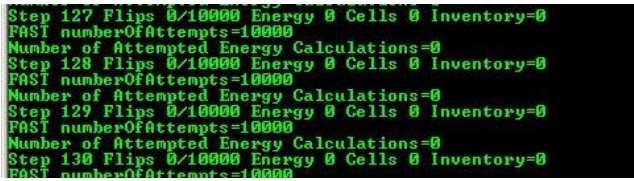
```
        <!-- Module tracking center of mass of each cell -->
    </Plugin>

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

<!--     <Steppable Type="BlobInitializer">     -->
<!--         <Region> -->
<!--             <Center x="50" y="50" z="0"/> -->
<!--             <Radius>20</Radius> -->
<!--             <Gap>0</Gap> -->
<!--             <Width>5</Width> -->
<!--             <Types>Condensing,NonCondensing</Types> -->
<!--         </Region> -->
<!--     </Steppable> -->
</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:

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

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

-13-
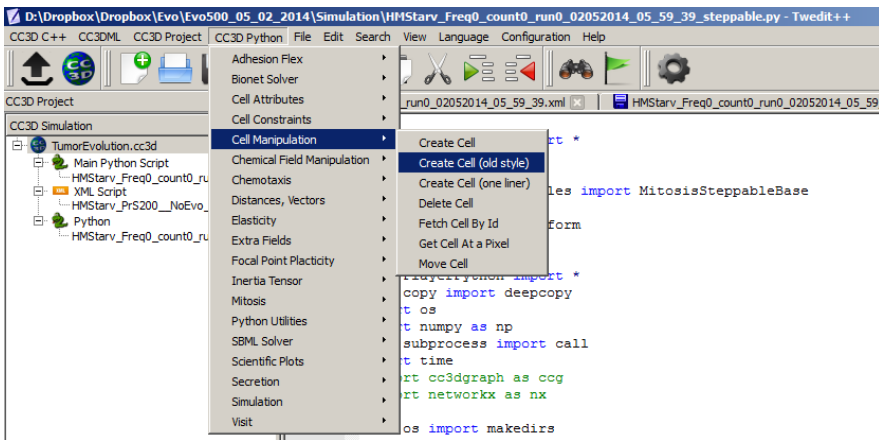
Figure 11 Inserting code snippet in Twedit++ to create cells. Notice that this is a generic code that usualy needs minor customizations.

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:

```
self.newCell(self.CONDENSING)
```

We initialize its type using `self.CONDENSING` class variable that corresponds to an integer assigned to type Condensing. Cell type is an integer value from 1 to 255 and CompuCell3D automatically creates class variables corresponding to each type. By looking at the defitnition 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.

IMPORTANT: 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.cellField[10:14,10:14,0]=cell
```

is based on Numpy syntax. `self.cellField` is a pointer to a C++ lattice which stores pointers to cell objects. In this example our cell is a 5x5 square collection of pxels. 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 os `SteppableBasePy`. To access cell object occupying lattice site, `x,y,z`, we type:

```
cell=self.cellField[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 messge 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.cellList:
            if cell.type==self.CONDENSING:
                self.deleteCell(cell)
```

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

8. 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 codemore 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` demonstrates the use of the functions calculating distance between cells or between any 3D points:

```
class CellDistanceSteppable(SteppableBasePy):

    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator,_frequency)
        self.cellA=None
        self.cellB=None
    def start(self):
        self.cellA=self.potts.createCell()
        self.cellA.type=self.A
```

```
        self.cellField[10:12,10:12,0]=self.cellA

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

    def step(self,mcs):

        distVec=self.invariantDistanceVectorInteger(_from=[10,10,0] ,_to=[92,12,0])
        print 'distVec=',distVec, ' norm=',self.vectorNorm(distVec)

        distVec=self.invariantDistanceVector(_from=[10,10,0] ,_to=[92.3,12.1,0])
        print 'distVec=',distVec, ' norm=',self.vectorNorm(distVec)

        print 'distance invariant='\
        ,self.invariantDistance(_from=[10,10,0] ,_to=[92.3,12.1,0])

        print 'distance =',self.distance(_from=[10,10,0] ,_to=[92.3,12.1,0])

        print 'distance vector between cells ='\
        ,self.distanceVectorBetweenCells(self.cellA,self.cellB)
        print 'inv. vec between cells ='\
        ,self.invariantDistanceVectorBetweenCells(self.cellA,self.cellB)
        print 'distanceBetweenCells = ',self.distanceBetweenCells(self.cellA,self.cellB)
        print 'invariantDistanceBetweenCells = ',\
        self.invariantDistanceBetweenCells(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.invariantDistanceVectorInteger` 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.vectorNorm. 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 pointwith coordinates x,y,z=10,12,-5 you use the following syntax:

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

If we want to calculate invariant vector but with components being floating point numbers we use `self.invariantDistanceVector` 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 thisis amatter 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.
Funtion `self.distance` calculates distance between two points in a naïve way. Sometimes this is allyou need.
Finally the set of last four calls `self.distanceVectorBetweenCells`,
`self.invariantDistanceVectorBetweenCells`, `self.distanceBetweenCells`,
`self.invariantDistanceBetweenCells` calculates distances and vectors between center of masses of cells.
You could replace

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

with

```
self.invariantDistanceVectorBetweenCells(_from=[ self.cellA.xCOM, self.cellA.yCOM, \
self.cellA.yCOM], _to=[=[ self.cellB.xCOM, self.cellB.yCOM, self.cellB.yCOM])
```

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

9. Looping over select cell types. Finding cell in the inventory.

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

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

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

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

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

```
self.cellListByType(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 Maniuplation->Fetch Cell By Id`):

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

The output of this code will look as shown below:

Figure 12 Fetching cell with specified id

Funciton self. attemptFetchingCellById 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. Actually 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, issufficient. We will come back to cell ids and cluster ids later in this manual.

10. 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. 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 snipet shows you how to

accomplish this (the code to open file in the simulation output directory can be inserted from Twedit++ - simply go to `CC3D Python->Python Utilities`):

```
    def step(self,mcs):
        fileName='myOutput+'+str(mcs)+'.txt'
        try:
            fileHandle,fullFileName\
            =self.openFileInSimulationOutputDirectory(fileName,"w")
        except IOError:
            print "Could not open file ", fileName," for writing. "
            return

        for cell in self.cellListByType(self.NONCONDENSING):
            print >>fileHandle, 'cell.id=',cell.id,'volume=',cell.volume

        fileHandle.close()
```

In the step function we create fileName by concatenating `'myOutput'`, current MCS - `str(mcs)`, and extension `'.txt'`. Inside `try/except` statement (refresh you knowledge about Python exceptions) we call `self.openFileInSimulationOutputDirectory` function where first argument is file name and second argument is file open mode. Since we are opening file for writing we use `"w"` . To open file in the read mode we would use `"r"`. Please consult appropriate chapter from Python programing manual for more information about file modes. If CC3D fails to open file in the simuatoin directory we pring error message and retur from step function. If the file open operation is successful we iterate over all cells of type `NonCondensing` and print cell id and cell current volume. Notice that when writing to a file in Python we have to use

```
print >>fileHandle
```

syntax. The reminder of this `print` statemnt looks exactly as a regular `print` statement. Alternatively we can use the following syntax to write to a file:

```
fileHandle.write("formatting string" %(values for formating string))
```

The formatting string contains regular text and formatting characters such as `'\n'` denoting end of line, `%d` denoting integer number, `%f` denoting floating point number and `%s` denoting strings. For more information on this topic please see any Python manual or see online Python documentation.
After we are done with writing we close the file which ensures that file buffers are transferred to a disk. Do not forget to close the file after you are done writing.
Notice that with `self.openFileInSimulationOutputDirectory` function we do not need to know the actual nameof the output directory. This makes things much easier than if we had to construct full file path. If you would prefer to store yourfiles in a separate subfolder of the simulation directory all you have to do is to prepend filename with thename of the subfolder followed by `'/'`. For example the following statement:

```
self.openFileInSimulationOutputDirectory('OUTPUT_SUBFOLDER/myoutput.txt',"w")
```

creates subfolder called `OUTPUT_SUBFOLDER` inside simulation outpur directory and inside this subfolder it opens file `myoutput.txt` for writing. You can replace `OUTPUT_SUBFOLDER` with any partial path e.g. `OUTPUT/TXT_FILES` and CC3D will make sure that all directories specified in the partial paths get created. This greatly simplifies file output operations in the CC3D.

11. Adding plots to the simulation

Some modelers like to monitor simulation progress bydisplaying "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:

```
class cellsortingSteppable(SteppableBasePy):

    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator,_frequency)
    def start(self):
        self.pW=self.addNewPlotWindow(_title='Average Volume And Volume of Cell 1',\
        _xAxisTitle='MonteCarlo Step (MCS)',_yAxisTitle='Variables',
        _xScaleType='linear',_yScaleType='log' )
        self.pW.addPlot('AverageVol',_style='Dots',_color='red',_size=5)
        self.pW.addPlot('Cell1Vol',_style='Steps',_color='black',_size=5)


    def step(self,mcs):
        averVol=0.0
        numberOfCells=0
        for cell in self.cellList:
            averVol+=cell.volume
            numberOfCells+=1

        averVol/=float(numberOfCells)

        cell1=self.attemptFetchingCellById(1)
        print cell

        self.pW.addDataPoint("AverageVol",mcs,averVol) #  name of the data series, x, y
        self.pW.addDataPoint("Cell1Vol",mcs,cell1.volume) #name of the data series, x, y
        # self.pW.showAllPlots() # no longer necessary
```

In the `start` function we create plot window (`self.pW`) – the arguments of this function are self explanatory. After we have plot windows object (self.pW) 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 instantenous volume of cell with id 1:

```
self.pW.addPlot('AverageVol',_style='Dots',_color='red',_size=5)
self.pW.addPlot('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 `_yScaleType='log'` as in the example above.
Alternatively we could Y-axis scale to be logarithmic using the following code:

```
self.pW.setYAxisLogScale()
```

In the `step` function we are calculating average volume of all cells and extract instantenuous volume of cell with id 1. After we are done with calculations we are adding our results to the time series:

```
self.pW.addDataPoint("AverageVol",mcs,averVol) # name of the data series, x, y
self.pW.addDataPoint("Cell1Vol",mcs,cell1.volume) # name of the data series, x, y
```

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

Important: Previous versions of CC3D required users to explicitly update plots by calling `self.pW.showAllPlots()` . **This is no longer necessary**. However if users decide to create plot windows using lower-level api this call may be necessary. However this way of creating plot windows is deprecated.

```
self.pW.showAllPlots() # DEPRECATED
```

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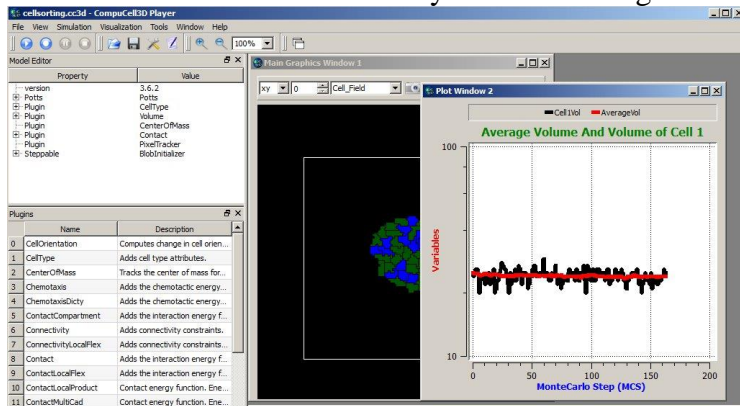
Figure 13 Displaying plot window in the CC3D Player with 2 time-series data.

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.

11.1. 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):

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

    def start(self):

        #initialize setting for Histogram
        self.pW=self.addNewPlotWindow\
        (_title='HIstogram',_xAxisTitle='Cell #',_yAxisTitle='Volume')
        self.pW.addHistogramPlot(_plotName='Hist 1',_color='green',_alpha=100)# _alpha is
        #transparency 0 is transparent, 255 is opaque
        self.pW.addHistogramPlot(_plotName='Hist 2',_color='red')
        self.pW.addHistogramPlot(_plotName='Hist 3',_color='blue')
```

-20-

```
    def step(self,mcs):
        volList = []
        for cell  in  self.cellList:
            volList.append(cell.volume)

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

        self.pW.addHistogram(plot_name='Hist 1' , value_array = gauss ,\
        number_of_bins=10)
        self.pW.addHistogram(plot_name='Hist 2' , value_array = volList ,\
        number_of_bins=10)
        self.pW.addHistogram(plot_name='Hist 3' , value_array = volList ,\
        number_of_bins=50)

        fileName="HistPlots_"+str(mcs)+".png"
        self.pW.savePlotAsPNG(fileName,1000,1000) # here we specify size of the image
        fileName="HistPlots_"+str(mcs)+".txt"
        self.pW.savePlotAsData(fileName)
```

In the `start` function we call `self.addNewPlotWindow` to add new plot window (`self.pW`) 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.pW.addHistogram(plot_name='Hist 2' , value_array = volList ,number_of_bins=10)
```

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

Alternatively we may use slightly more complex way od adding histogram which in some situations may actually give you a bit more control. First we bin array of values using numpy functionality:

```
(n, bins) = numpy.histogram(volList, bins=10)
```

The return values are two numpy arrays: n which specifies center of the bin (we plot it on x axis) and bins which determines stores counts for a given bin.

Important: Make sure you import `random` and `numpy` modules in the steppable file. Place the following code:

```
import random,numpy
```

at the top of the file.

Next you add histogram data output from numpy to the plot using the following call:

```
self.pW.addHistPlotData('Hist 2',n,bins)
```

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 gausian distribution centered at 0.0 and having standard deviation equal to 1.0. We histogram those values using the following code:

```
self.pW.addHistogram(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.

Imnportant: Calling `showAllHistPlots` is no longer necessary

```
self.pW.showAllHistPlots()  #DEPRECATED
```

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

```
self.pW.savePlotAsPNG(fileName,1000,1000)
```

two last arguments of this function represent x and y sizes of the image. We construct `fileName` 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 `savePlotAsData` from the plot windows object:

```
fileName="HistPlots_"+str(mcs)+".txt"
self.pW.savePlotAsData(fileName)
```

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

12. Custom Cell Attributes in Python

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

```
int clock
```

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

```
    def step(self,mcs):

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

        for cell in self.cellList:
            print 'cell.id=',cell.id,' dict=',cell.dict
```
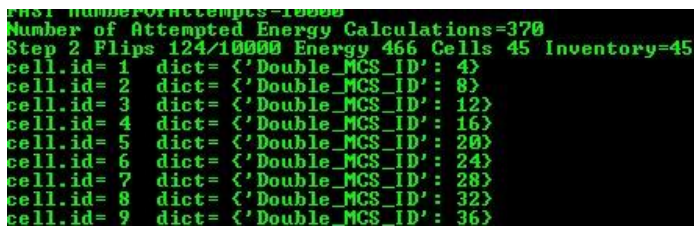
We have two loops that iterate over list of all cells. In the first loop we access dictionary that is attached to each cell:

```
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 acces the dictionary and print its content to the screen. The result will look something like:

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

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

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

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

13. 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\Extra Fields` 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.

13.1. Scalar Field – pixel based

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

```
from math import *
class ExtraFieldVisualizationSteppable(SteppableBasePy):
    def __init__(self,_simulator,_frequency=10):
        SteppableBasePy.__init__(self,_simulator,_frequency)
        self.scalarField=CompuCellSetup.createScalarFieldPy(self.dim,"ExtraField")

    def step(self,mcs):

        self.scalarField[:, :, :]=0.0 # clear field

        for x in xrange(self.dim.x):
            for y in xrange(self.dim.y):
                for z in xrange(self.dim.z):

                    if (not mcs%20):
                        self.scalarField[x,y,z]=x*y

                    else:
                        self.scalarField[x,y,z]=sin(x*y)
```

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

```
self.createScalarFieldPy(self.dim,"ExtraField").
```

Important: Make sure that all calls to functions creating fields are in the __init__ functions so that the Player can display them correctly.
In the step function we initialize self.scalarField using slicing operation:

```
self.scalarField[:, :, :]
```

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.
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` has convenience function called `self.everyPixel` (`CC3D Python->Visit->All Lattice Pixels`) which allows us to compact triple loop to just one line:

```
        for x,y,z in self.everyPixel():
            if (not mcs%20):
                self.scalarField[x,y,z]=x*y
            else:
                self.scalarField[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.everyPixel(5,10,4):
```

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

13.2. Vector Field – pixel based

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

```
class VectorFieldVisualizationSteppable(SteppableBasePy):
    def __init__(self,_simulator,_frequency=10):
        SteppableBasePy.__init__(self,_simulator,_frequency)
        self.vectorField=self.createVectorFieldPy("VectorField")

    def step(self,mcs):

        self.vectorField[:, :, :,:]=0.0 # clear vector field

        for x,y,z in self.everyPixel(10,10,5):
            self.vectorField[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.vectorField` is implemented as Numpy array:

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


13.3. 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:

```
class IdFieldVisualizationSteppable(SteppableBasePy):
    def __init__(self,_simulator,_frequency=10):
        SteppableBasePy.__init__(self,_simulator,_frequency)
        self.scalarCLField=self.createScalarFieldCellLevelPy("IdField")

    def step(self,mcs):
        self.scalarCLField.clear()
        for cell in self.cellList:
            self.scalarCLField[cell]=cell.id*random()
```

As it was the case with other fields we create cell level scalar field in the `__init__` function using `self.createScalarFieldCellLevelPy`. 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 loopover all cells we assign random value to each cell. When we plot `IdField` 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.

13.4. Vector Field – cell level

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

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

        self.vectorCLField=self.createVectorFieldCellLevelPy("VectorFieldCellLevel")

    def step(self,mcs):
        self.vectorCLField.clear()
        for cell in self.cellList:

            if cell.type==1:

                self.vectorCLField[cell]=[cell.id*random(),cell.id*random(),0]
```

Inside `__init__` function we create cell-level vector field using `self.createVectorFieldCellLevelPy` 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.

14.   Automatic Tracking of Cells' Attributes

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

```
 class DemoVisSteppable(SteppableBasePy):

    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator,_frequency)
        self.track_cell_level_scalar_attribute (field_name='COM_RATIO',\
        attribute_name='ratio')

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

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

In the `start` and `step` functions we iterate over all cells and attach a cell attribute `'ratio'` that is equal to the ration of x and y center-of mass coordinates for each cell. In the init function we setup automatic tracking of

this attribute i.e. we create a cell-level scalar field (called `COM_RATIO`) where cells are colored according to the value of their `'ratio'` attribute:

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

The syntax of this function can be found in Twedit Python helper menu: `CC3D Python->Extra Fields Automatic Tracking -> Track Scalar Cell Attribute (__init__)` :
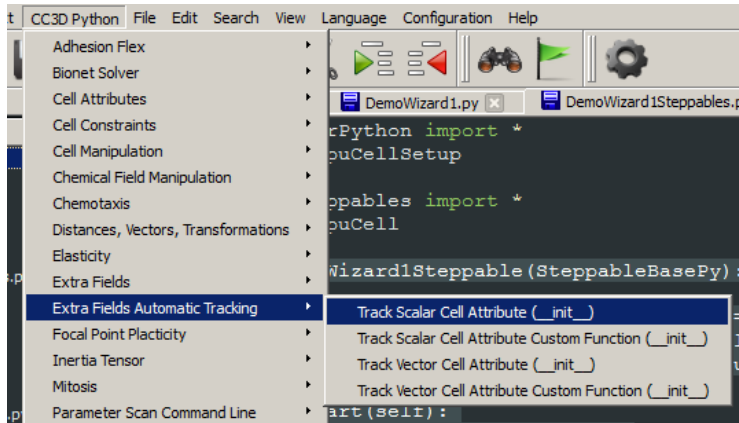
Figure 15 Setting up automatic tracking of cells' scalar attribute using Twedit++

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

```
class DemoVisSteppable(SteppableBasePy):

    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator,_frequency)
        self.track_cell_level_scalar_attribute (field_name='COM_RATIO',\
        attribute_name='ratio')

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

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

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

All we did in the snippet above was to add new field `SIN_COM_RATIO` using the `track_cell_level_scalar_attribute` function. The call to this function almost identical as before except now we also used `function` argument:

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

The meaning of this is the following: for each attribute `'ratio'` attached to a cell a function math.sin(attr_val) will be evaluated where `attr_val` will assume same value as `'ratio'` cell attribute for a given cell. If you are

puzzled about `lambda` Python key word don't be. Python `lambda`'s are a convenient way to define inline functions For example:

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

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

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

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

When we run the simulation above the output may look like in the figure below:
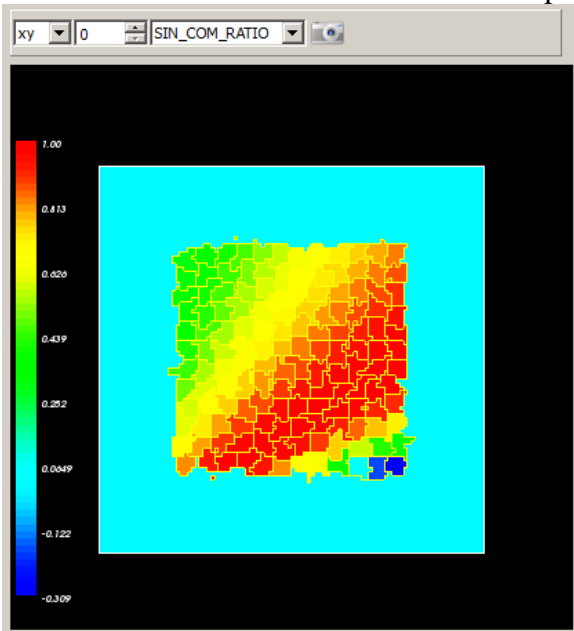
Figure 16. Automatic labeling of cells according to scala cell's attribute

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

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

```
class DemoVisSteppable(SteppableBasePy):

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

    def start(self):
```

```
        for cell in self.cellList:
            cell.dict['com_vector'] = [cell.xCOM, cell.yCOM, 0.0]

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

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

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

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

15. 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`):

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

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

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:

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

```
        self.runBeforeMCS=1
```

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

```
self.field=self.getConcentrationField('ATTR')
lmfLength=1.0;
xScale=1.0
yScale=1.0
zScale=1.0
        # FOR HEX LATTICE IN 2D
#         lmfLength=sqrt(2.0/(3.0*sqrt(3.0)))*sqrt(3.0)
#         xScale=1.0
#         yScale=sqrt(3.0)/2.0
#         zScale=sqrt(6.0)/3.0

for cell in self.cellList:
    #converting from real coordinates to pixels
    xCM=int(cell.xCOM/(lmfLength*xScale))
    yCM=int(cell.yCOM/(lmfLength*yScale))

    if cell.type==3:
        self.field [xCM,yCM,0]=self.field [xCM,yCM,0]+10.0
```

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

15.1. Lattice Conversion Factors

In thecode where we manually implement secretion at the cell'sCOM we use strange looking variables `lmfLength`, `xScale` and `yScale`. 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 Cartiesian 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 `PySteppableBase` built-in functions (`self.cartesian2Hex`, and `self.hex2Cartesian`) – see also Twedit++ CC3D Python menu `Distances`, `Vectors`, `Transformations`:

```
hex_coords=self.cartesian2Hex(_in=[10,20,11])
pt=self.hex2Cartesian(_in=[11.2,13.1,21.123])
```

16.   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:

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

    def start(self):

        for cell in self.cellList:
            if cell.type==self.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.cellList:
                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 Source="FlexibleDiffusionSolverFE" 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.cellList:
            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 code snippet:

```
The Python code would look like:
        for cell in self.cellList:
            if cell.type==self.MACROPHAGE:
                cd=self.chemotaxisPlugin.addChemotaxisData(cell,"ATTR")
                cd.setLambda(30.0)
                cd. setSaturationLinearCoef(100)
                cd.assignChemotactTowardsVectorTypes([self.MEDIUM,self.BACTERIUM])
```

17.   Steering – changing CC3DML parameters on-the-fly.

(You may skip this paragraph in the first reading)

In the CC3D terminology, steering means changing simulation parameters on the fly as simulation is running. In fact the whole point of merging Python scripting with CC3D core code is to enable steering. What about parameters defined in CC3DML? Can they be modified as the simulation runs? The short answer is 'yes' but not all of them. There are two ways of doing it – one way is to use player CC3DML panel and change parameters in the GUI. The second way is to use Python. Python code that alters CC3DML parameters during simulation runtime is quite clumsy-looking. It is not difficult to understand but is quite verbose and, to be honest it is not too much fun to write. Probably the best way to learn how to modify CC3DML from Python is to look at several examples. Let us start with the simplest one (Demos\SimulationSettings\Steering):

Here we will modify two CC3DML parameters one will be Temperature from the CC3DML Potts Section and the other one will be Contact energy between Condensing and NonCondensing.

Let's look at CC3DMLcode first:

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

Potts XML element has 4 child elements – Dimensions, Steps, Temperature and NeighborOrder.

To modify Temperature parameter we use the following code:

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

    def step(self, _mcs):
        # get Potts section of XML file
        pottsXMLData=self.simulator.getCC3DModuleData("Potts")
        # check if we were able to successfully get the section from simulator
        if pottsXMLData:
            # get Temperature XML element
```

```
                temperatureElement=pottsXMLData.getFirstElement("Temperature")
                # check if the attempt was succesful
                if temperatureElement:
                    # get value of the temperature and convert it to a floating point number
                     temperature=float(temperatureElement.getText())
                    # increase temperature by 1.0
                    temperature+=1.0
                    # update XML Temperature element by converting floating point
                    temperatureElement.updateElementValue(str(temperature))
                # finally call simulator.updateCC3DModule(pottsXMLData) to update model
                # parameters - this is actual steering
                self.simulator.updateCC3DModule(pottsXMLData)
```

Each XML element that we fetch using Python (e.g. `pottsXMLData`, `temperatureElement`) has set of member functions that allow us to get to elements nested one level deeper (child elements) or functions which allow us to read and modify element values and element attributes. For the temperatureElement we are modifying its value.

Important: XML stores text. All numbers or other data types stored in the XML are converted to their text representations. Consequently, depending on your needs and particular applications, you may need to convert text to numbers and vice versa when interacrting with XML through Python interface.

In the code presented above we first fetch `Potts` element (`pottsXMLData`) using special function from the simulator object. If the fetching opration was successful we try to fetch first `Temperature` child element of the `Potts` element. If this operation was successful we convert value of the temperature element to floating point number:

```
temperature=float(temperatureElement.getText())
```

we increase value of this number by one and then update value of the `temperatureElement`:

```
temperatureElement.updateElementValue(str(temperature))
```

Notice that we had to convert floating point number temperature back to string using

```
str(temperature)
```

call.

The steppable ends with call to simulator member function that informs CC3D that CC3DML was changed and simulation needs to get new parameters:

```
self.simulator.updateCC3DModule(pottsXMLData)
```

As you can see the code was not hard to write but is quite long and clumsy. We could simplify it a bit but getting rid of comments and if statements that check if fetching operation was successful. In such a case the code would look like that:

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

    def step(self, _mcs):

        pottsXMLData=self.simulator.getCC3DModuleData("Potts")
        temperatureElement=pottsXMLData.getFirstElement("Temperature")
```

```
        temperature=float(temperatureElement.getText())
        temperature+=1.0
        temperatureElement.updateElementValue(str(temperature))
        self.simulator.updateCC3DModule(pottsXMLData)
```

This is not so bad but still it is a lot of work to change one number. But do we have a choice here? In fact we do. All we have to do is to change cell membrane fluctuation amplitude using the following code:

```
newTemperature=51
for cell in self.cellList:
    cell.fluctAmpl= newTemperature
```

In practice you don't use need to modify CC3DML from Python level too often. CC3D has modules e.g. AdhesionFlex, FocalPointPlasticity, VolumeLocalFlex that require initialization of their parameters in Python but also offer much simpler programing interfaces making coding much less cumbersome. Please make sure that before writing complicated CC3DML steering code you familiarize yourself with modules that are designed to be flexible and do not rely on CC3DML-type of steering.
Now let us take a look at the code that alters contact energy, but first quick glance at the CC3DML that we will be modifying:

```
<Plugin Name="Contact">
   <Energy Type1="NonCondensing" Type2="Condensing">11</Energy>
   <Energy Type1="Condensing"    Type2="Condensing">2</Energy>
…
   <NeighborOrder>2</NeighborOrder>
 </Plugin>
```

Our task here is to first fetch Plugin XML Element and then fetch Energy Element for type pair Condensing and NonCondensing. Here is the code that does it:

```
class ContactSteering(SteppablePy):
    def __init__(self,_simulator,_frequency=10):
        SteppablePy.__init__(self,_frequency)
        self.simulator=_simulator

    def step(self,mcs):
        # get <Plugin Name="Contact"> section of XML file
        contactXMLData=self.simulator.getCC3DModuleData("Plugin","Contact")
        # check if we were able to successfully get the section from simulator
        if contactXMLData:
            # get <Energy Type1="NonCondensing" Type2="Condensing"> element
            energyNonCondensingCondensingElement=contactXMLData.getFirstElement\
            ("Energy",d2mss({"Type1":"NonCondensing","Type2":"Condensing"}))
            # check if the attempt was succesful
            if energyNonCondensingCondensingElement:
                # get value of <Energy Type1="NonCondensing" Type2="Condensing"> element
                #and convert it into float
                val=float(energyNonCondensingCondensingElement.getText())
                # increase the value by 1.0
                val+=1.0
                # update <Energy Type1="NonCondensing" Type2="Condensing"> element
                #remembering abuot converting the value bask to string
                energyNonCondensingCondensingElement.updateElementValue(str(val))
            # finally call simulator.updateCC3DModule(contactXMLData) to tell simulator
            #to update model parameters - this is actual steering
            self.simulator.updateCC3DModule(contactXMLData)
```

We first fetch `Plugin` element using simulator object member function:

```
contactXMLData=self.simulator.getCC3DModuleData("Plugin","Contact")
```

When this operation succeeds we attempt to fetch Python object representation for the
`<Energy Type1="NonCondensing" Type2="Condensing">11</Energy>` element:

```
energyNonCondensingCondensingElement=contactXMLData.getFirstElement\
("Energy",d2mss({"Type1":"NonCondensing","Type2":"Condensing"}))
```

Notice that when we call `getFirstElement` member function of the `contactXMLData` we pass the name of the element but also a partil list of element attributes. Here we use `d2mss` function what converts Python dictionary `{"Type1":"NonCondensing","Type2":"Condensing"}` to C++ `map<string,string>`. With so much information passed to `getFirstElement` function only one element fits the description and this is the one that we are looking for. The reminder of the steppable looks almost identical as in the example where we changed temperature.

The next example demonstrates how to update attribute of the XML element. You can find full code in `Demos\Models\bacterium_macrophage_2D_steering`. Again let us look at the CC3DML that we will be modifying:

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

We would like to periodically decrease `lambda` chemotaxis by `3` units. This is how we do it in Python:

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

    def step(self,mcs):
        if mcs>100 and not mcs%100:
            # get <Plugin Name="Chemotaxis"> section of XML file
            chemotaxisXMLData=self.simulator.getCC3DModuleData("Plugin","Chemotaxis")
            # check if we were able to successfully get the section from simulator
            if chemotaxisXMLData:
                # get <ChemicalField Source="FlexibleDiffusionSolverFE" Name="ATTR" >
                chemicalField=chemotaxisXMLData.getFirstElement("ChemicalField",\
                d2mss({"Source":"FlexibleDiffusionSolverFE", "Name":"ATTR"}))
                # check if the attempt was succesful
                if chemicalField:
                    # get <ChemotaxisByType Type="Macrophage" Lambda="xxx"/>
                    chemotaxisByTypeMacrophageElement=chemicalField.\
                    getFirstElement("ChemotaxisByType",d2mss({"Type":"Macrophage"}))
                    if chemotaxisByTypeMacrophageElement:
                        # get value of Lambda from <ChemotaxisByType> element
                        # notice that no conversion fro strin to float is necessary as
                        #getAttributeAsDouble returns floating point value

                        lambdaVal=chemotaxisByTypeMacrophageElement.\
                        getAttributeAsDouble("Lambda")
                        print "lambdaVal=",lambdaVal
                        # decrease lambda by 0.2
```

```
                        lambdaVal-=3
                        # update attribute value of Lambda converting float to string
                        chemotaxisByTypeMacrophageElement.\
                        updateElementAttributes(d2mss({"Lambda":str(lambdaVal)}))
            self.simulator.updateCC3DModule(chemotaxisXMLData)
```

As you can see the structure of the code is very similar to the previous 2 examples. First we fetch `Plugin` element describing Chemotaxis properties:

```
chemotaxisXMLData=self.simulator.getCC3DModuleData("Plugin","Chemotaxis")
```

Next, we fetch ChemicalField element:

```
chemicalField=chemotaxisXMLData.getFirstElement\
("ChemicalField",d2mss({"Source":"FlexibleDiffusionSolverFE", "Name":"ATTR"}))
```

and using `ChemicalField` element we get `ChemotaxisByType` element:

```
chemotaxisByTypeMacrophageElement=chemicalField.getFirstElement\
("ChemotaxisByType",d2mss({"Type":"Macrophage"}))
```

Using `chemotaxisByTypeMacrophageElement` we fetch its attribute lambda convert it to floating point number decrease by `3` units and assing new value of `lambda`:

```
lambdaVal=chemotaxisByTypeMacrophageElement.getAttributeAsDouble("Lambda")
lambdaVal-=3
chemotaxisByTypeMacrophageElement.updateElementAttributes\
(d2mss({"Lambda":str(lambdaVal)}))
```

The rest of the code is analogous to the previous examples. This completes the discussion of CC3DML steering.

17.1.Simplifying steering - XML access path

The above examples demonstrate how to steer CC3DML-based part of the simulation in a fairly verbose way i.e. the amount of code is quite significant. In 3.7.1 we have introduced more compact way to access and modify CC3DML elements: Let us look at the implementation of the `ContactSteeringAndTemperature` steppable using new style coding:

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

    def step(self,mcs):

        temp=float(self.getXMLElementValue(['Potts'],['Temperature']))
        self.setXMLElementValue(temp+1,['Potts'],['Temperature'])

        val=float(\
        self.getXMLElementValue(\
        ['Plugin','Name','Contact'],['Energy','Type1','NonCondensing','Type2','Condensing']))

        self.setXMLElementValue\
        (val+1,['Plugin','Name','Contact'],['Energy','Type1','NonCondensing','Type2','Condensing'])

        self.updateXML()
```

Instead of using verbose code to access CC3DML elements we now specify access path to particular element . Access path is a sequence of Python lists. First element of each list is the name of the CC3DML element followed by a sequence of pairs (attribute,value) which fully specify the XML element:

```
[ElementName,AttrName1,AttrValue1,Attr2,AttrValue2,…, AttrNameN,AttrValueN]
```

In the CC3DML code below:

```
 <CompuCell3D>
<Potts>
   <Dimensions x="100" y="100" z="1"/>
   <Anneal>10</Anneal>
   <Steps>10000</Steps>
   <Temperature>10</Temperature>
   <Flip2DimRatio>1</Flip2DimRatio>
   <NeighborOrder>2</NeighborOrder>
 </Potts>


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

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

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

to access Temperature element from the Potts section we construct our access path following one simple rule:

1. Recursively identify child element of the current element that leads you to the desired place in the CC3DML code. **Notice:** we skip root element `<CompuCell3D>` element.

In our example to access `<Temperature>` element we first locate `<Potts>` as a child of `<CompuCell3D>` element (remember in the access path we do not include `<CompuCell3D>` element) and
then `<Temperature>` appears to be child of the `<Potts>`. Hence our access path is a simple sequence of two Python lists, each list with one element:
```
['Potts'],['Temperature']
```

A bit more complex, but much simpler than our original coding is the example where we locate one of the Contact plugin elements. <Plugin Name="Contact"> is a child of the <CompuCell3D> hence:

```
['Plugin','Name','Contact']
```

will be first Python list of ther access path. Notice that the first element of this list is the same as name of the child element (Plugin) and the two next elements constitute an XML attribute-value pair. In other words, XML's `Name="Contact"` gets translated into `'Name','Contact'` of the Python list.

Now we locate correct <Energy> element. Since there are many of these the correnct indentifiacation of the one which is of interest for us will require specification of all its attributes: Type1="NonCondensing" Type2="Condensing". Consequently our access path from `<Plugin>` to `<Enenrgy>` will look as follows:

```
['Energy','Type1','NonCondensing','Type2','Condensing']
```

And the full path is simply

```
['Plugin','Name','Contact'], ['Energy','Type1','NonCondensing','Type2','Condensing']
```

18. 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 youcan 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:

```
<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"/>
```
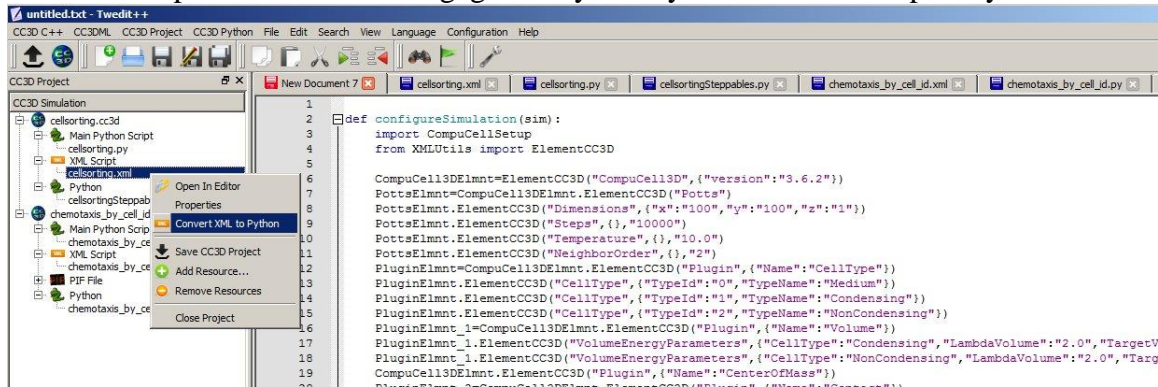
```
    </Plugin>
```

And then at equivalent Python code:

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

    CompuCell3DElmnt=ElementCC3D("CompuCell3D",{"version":"3.6.2"})
    PottsElmnt=CompuCell3DElmnt.ElementCC3D("Potts")
    PottsElmnt.ElementCC3D("Dimensions",{"x":"100","y":"100","z":"1"})
    PottsElmnt.ElementCC3D("Steps",{},"10000")
    PottsElmnt.ElementCC3D("Temperature",{},"10.0")
    PottsElmnt.ElementCC3D("NeighborOrder",{},"2")
    PluginElmnt=CompuCell3DElmnt.ElementCC3D("Plugin",{"Name":"CellType"})
    PluginElmnt.ElementCC3D("CellType",{"TypeId":"0","TypeName":"Medium"})
    PluginElmnt.ElementCC3D("CellType",{"TypeId":"1","TypeName":"Condensing"})
    PluginElmnt.ElementCC3D("CellType",{"TypeId":"2","TypeName":"NonCondensing"})
    PluginElmnt_1=CompuCell3DElmnt.ElementCC3D("Plugin",{"Name":"Volume"})
    PluginElmnt_1.ElementCC3D("VolumeEnergyParameters",\
   {"CellType":"Condensing","LambdaVolume":"2.0","TargetVolume":"25"})
    PluginElmnt_1.ElementCC3D("VolumeEnergyParameters",\
   {"CellType":"NonCondensing","LambdaVolume":"2.0","TargetVolume":"25"})
```

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

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

We attach a child element (Potts) to ComuCell3D element and a return value of this call is object representing Potts element:

```
PottsElmnt=CompuCell3DElmnt.ElementCC3D("Potts")
```

We looks at the XML and notice that Potts element has several child elements – e.g. `Dimensions`, `Temperature` etc… We attach all of these child elements to `Potts` element:

```
PottsElmnt.ElementCC3D("Dimensions",{"x":"100","y":"100","z":"1"})
PottsElmnt.ElementCC3D("Temperature",{},"10.0")
```

We hope you see the pattern. The general rule is this. To create root element you use function `ElementCC3D` from XML Utils – see how we created `CompuCell3D` element. When you want to attach child element we call `ElementCC3D` member function of the parent element e.g.:

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

This syntax can 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 objectrepresenting 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:

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

    cc3d=ElementCC3D("CompuCell3D")
    potts=cc3d.ElementCC3D("Potts")
    potts.ElementCC3D("Dimensions",{"x":100,"y":100,"z":1})
…
    CompuCellSetup.setSimulationXMLDescription(cc3d)

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

import CompuCellSetup

sim,simthread = CompuCellSetup.getCoreSimulationObjects()

configureSimulation(sim)

CompuCellSetup.initializeSimulationObjects(sim,simthread)


from PySteppables import SteppableRegistry
steppableRegistry=SteppableRegistry()


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

The actual placement of `configureSimulation` function in the main script matters. It has to be called right before

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

Finally, one important remark: Twedit++ has CC3DML helpermenu which pastesready to use CC3DML code for all availale modules. This means that when you work with XML and you wantto add cell types, insertsyntax 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.

19. 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 reals system thet chemotaxis is the reason for directed motion) and sometimes we simply apply energy term which simulates a force. In the CC3D manual we show how to apply constant force to

all cells or on a type-by-type basis. Here let us concentrate on a situation where we apply force to individual cells and how change its value and the direction. You can check simulation code in `Demos\CompuCellPythonTutorial\CellMotility`. To be able to use force in our simulation (we need to include `ExternalPotential` Plugin in the CC3DML:

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

Let us look at the steppable code:

```
from random import uniform

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

    def start(self):
        print "This function is called once before simulation"

        # iterating over all cells in simulation
        for cell in self.cellList:
            break
            # Make sure ExternalPotential plugin is loaded
            # negative lambdaVecX makes force point in the positive direction
            cell.lambdaVecX=10.1*uniform(-0.5,0.5) # force component along X axis
            cell.lambdaVecY=10.1*uniform(-0.5,0.5) # force component along Y axis
#           cell.lambdaVecZ=0.0 # force component along Z axis


    def step(self,mcs):

        for cell in self.cellList:

            cell.lambdaVecX=10.1*uniform(-0.5,0.5) # force component along X axis
            cell.lambdaVecY=10.1*uniform(-0.5,0.5) # force component along Y axis
```

Once `ExternalPotential` plugin has been loaded we assign a constant force in a given direction by initializing `lambdaVecX`, `lambdaVecY`, `lambdaVecZ` cell attributes.

Remark: 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 manual.

20. 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.

21. Checking if two cell objects point to different cells

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.areCellsDifferent(cell1,cell2)
```

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

```
def areCellsDifferent(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
```

22. 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,_simulator,_frequency=100):
        SteppableBasePy.__init__(self,_simulator,_frequency)

    def start(self):
        for cell in self.cellList:
            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.cellList:
            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 glogal settings.

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

23. Controling steppable call frequency. Stoppping 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
steppableInstance=cellsortingSteppable(sim,_frequency=1)
steppableRegistry.registerSteppable(steppableInstance)
```

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.stopSimulation()
```

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.setMaxMCS(100000)
```

to extend simulation to 100000 MCS.

24. Building a wall (destroying sounds too cliché).

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 betwwe lattice boundary and cell contributes 0 to the contact energy. Thus, whenall 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 asfFrozen. 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.buildWall(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.

25. 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:

```
class BuildWall3DSteppable(SteppableBasePy):

    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator,_frequency)
    def start(self):

        self.buildWall(self.WALL)


    def step(self,mcs):

        if mcs==4:
            self.destroyWall()
            self.resizeAndShiftLattice(_newSize=(80,80,80), _shiftVec=(10,10,10))
            self.buildWall(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 runtimes 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.

26. 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 runtimes. But what does a small cell mean? If we are on a 100x100 lattice and cells have approx. 5-7 pixels in "diameter" then if we use 4 cores then each core will be responsible for 50x50 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 sinlge 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:

```
class DynamicNumberOfProcessorsSteppable(SteppableBasePy):

    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator,_frequency)

    def step(self,mcs):
        if mcs==10:
            self.resizeAndShiftLattice(_newSize=(400,400,1), _shiftVec=(100,100,0))

        if mcs==100:
            self.changeNumberOfWorkNodes(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 * 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 ,simpley go to `CC3D Python->Simulation` menu and choose appropriate option.

27. 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:

```
def step(self,mcs):
    for cell in self.cellList:
        print "*********NEIGHBORS OF CELL WITH ID ",cell.id,
```

```
                for neighbor , commonSurfaceArea in self.getCellNeighborDataList(cell):
                    if neighbor:
                        print "neighbor.id",neighbor.id," commonSurfaceArea=",commonSurfaceArea
                    else: print "Medium commonSurfaceArea=", commonSurfaceArea
```

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

```
for neighbor , commonSurfaceArea in self.getCellNeighborDataList(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++.

28.   Accesing concentration fields managed of PDE solvers

Many of CC3D simulations have at least one diffusing field which represents some sort of chemical it can be growth factor, toxin or nutrient. The concentration fields are created by CC3D PDE solvers. One of the most common tasks that modelsrs implement is modifying cellular behaviours based on the chemical concentration at the center of mass of a cell (or for that matter any other point belonging to a given cell).

In this example we will show you how to extract and modify values of the concentration fields.

You can take a look at `CompuCellPythonTutorial\diffusion` example if you want a quick preview of code that deals with diffusion fields. The task here is quite simple. We first have to get a handle to the field and then using Numpy-like syntax either read or modify field values.

The example that we will implement here is the following. We will start with "regular" cellsorting cell layout where condensing and non condensing cells are mixed together. In the corner of the lattice we will place pulse of the chemical and will let the chemical diffuse. We will monitor the values of the concentration at the center of mass of each cell and if this value will be greater than `0.1` we will change cell type to `Condensing`. Assuming that concentration pulse is big enough all cells will become `Condensing` after some time. Let's tak a a look at the code:

```
class cellsortingSteppable(SteppableBasePy):

    def __init__(self,_simulator,_frequency=1)
        SteppableBasePy.__init__(self,_simulator,_frequency)

    def start(self):
        field=self.getConcentrationField("FGF")
        field[0,0,0]=2000

    def step(self,mcs):
        field=self.getConcentrationField("FGF")

        for cell in self.cellList:
            if field[cell.xCOM,cell.yCOM,cell.zCOM]>0.1:
                cell.type=self.CONDENSING
```

In the `start` function we get a handle to a diffusion field `FGF`. This field is defined in the XML using the following code:

```
    <Steppable Type="FlexibleDiffusionSolverFE">
```

```
    <DiffusionField>
        <DiffusionData>
            <FieldName>FGF</FieldName>
            <DiffusionConstant>0.1</DiffusionConstant>
            <DecayConstant>1e-05</DecayConstant>
        </DiffusionData>
    </DiffusionField>

</Steppable>
```

XML code above defines diffusion and decay constants but says nothing about initial conditions. We could have defined initial conditions in the XML but we chose to do this in the `start` function of the Python-based steppable.

The content of the start function is almost self-explanatory. In the first line of the function we get a handle to concentration field. Notice that we pass `"FGF"` to `self.getConcentrationField` function. It is exactly the same name as we declared in XML. If you use field name that is undefined inth XML you will get `None` object in return. In the second line we initialize field to have concentration 2000 units at location `x=0`, `y=0`, `z=0`. If we wanted to extend the area of initial concentration of 200 units we could have used the following Numpy slicing operation

```
field[0:5:1,0:5:1,0]
```

and this would put 2000 units of concentration in the 5x5 square at the origin of the lattice. Slicing works as follows first number specifies lower bound, second specifies upper bound (the maximum index is upper bound minus one) and third specifies step.In our example `0:5:1` will select indices `0,1,2,3,4`.
When we type `0:6:2` then we will select indices `0,2,4` – again `6` being upper bound is not selected.For more information about Numpy slicing please see numpy tutorial online:

http://wiki.scipy.org/Tentative_NumPy_Tutorial

In the `start` function we first get a handle to the `FGF` field, and then we iterate over each cell in the simulation. We check if `FGF` concentration at the center of mass of each cell is greater than `0.1`:

```
if field[cell.xCOM,cell.yCOM,cell.zCOM]>0.1:
```

and if so we change cell type to `Condensing`. Notice that to access center of mass of a cell we we need to include `CentterOfMass` plugin in the XML using the following code:

```
<Plugin Name="CenterOfMass">
```

All Twedit++ -generated templates put this plugin by default but if you type XML manually you need to remember about this module. CentrOFMass plugin tracksand keeps up-to-date centerofmass of each cell. To access COM value from Python we use the following syntax:

```
cell.xCOM
cell.yCOM
cell.zCOM
```

When you run the simulation you will notice that gradually all of the cells will turn into Condensing.

28.1. Min/Max field values

To access min or max of concentration fiels (i.e. defined in the PDE solver) you simply type

```
minVal=field.min()
```

or

```
maxVal=field.max()
```

29. Mitosis

In developmental simulations we often need to simulate cells which grow and divide. In earlier versions of CompuCell3D we had to write quite complicated plugin to do that which was quite cumbersome and unintuitive. 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
import CompuCell
import sys
from random import uniform
import math


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

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

                cells_to_divide.append(cell)

        for cell in cells_to_divide:
            # to change mitosis mode uncomment proper line below
            self.divideCellRandomOrientation(cell)
            # these are valid option

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

    def updateAttributes(self):
        self.parentCell.targetVolume = 25.0 # reducing parent target volume
        self.cloneParent2Child()

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

The `step` function is quite simple – we iterate over all cells in the simulation and check if the volum 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.

Important: the name of the function where we update attributes after mitosis has to be exactly `updateAtttributes`. If it is called differently CC3D will not call it automatically. We can obviouslycall such function by hand, immediately we do the mitosis but this is not very elegant solution.

The `updateAttributes` 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`:

```
parentCell.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 `updateAttributes` function is `self.cloneParent2Child()`. 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.cloneAttributes(sourceCell=self.parentCell, targetCell=self.childCell,\
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 updateAttribute function where we manually set parent and child properties could look like that:

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

        childCell.targetVolume=parentCell.targetVolume
        childCell.lambdaVolume=parentCell.lambdaVolume
        if parentCell.type==self.CONDENSING:
            childCell.type=self.NONCONDENSING
```

```
        else:
            childCell.type=self.CONDENSING
```

Remark: It is important to divide cells outside the loop where we iterate over entire cell inventory. If we keep dividing cells in the 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:

```
class VolumeParamSteppable(SteppablePy):
    def __init__(self,_simulator,_frequency=1):
        SteppablePy.__init__(self,_frequency)
        self.simulator=_simulator
        self.inventory=self.simulator.getPotts().getCellInventory()
        self.cellList=CellList(self.inventory)
    def start(self):
        for cell in self.cellList:
            cell.targetVolume=25
            cell.lambdaVolume=2.0
    def step(self,mcs):
        for cell in self.cellList:
            cell.targetVolume+=1
```

Again, this is quite simple module whre 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:

```
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`.

29.1. 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.setParentChildPositionFlag` 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.

30.   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" cells then you can check that each such cell has distinct `id` and `clusterId`. An example of compartmental simulation can be found in `examples_PythonTutorial\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:

```
class MitosisSteppableClusters(MitosisSteppableClustersBase):
    def __init__(self,_simulator,_frequency=1):
        MitosisSteppableClustersBase.__init__(self,_simulator, _frequency)

    def step(self,mcs):

        for cell in self.cellList:
            clusterCellList=self.getClusterCells(cell.clusterId)
            for cellLocal in clusterCellList:

        mitosisClusterIdList=[]
        for compartmentList in self.clusterList:
            clusterId=0
            clusterVolume=0
            for cell in CompartmentList(compartmentList):
                clusterVolume+=cell.volume
                clusterId=cell.clusterId

            if clusterVolume>250:
                mitosisClusterIdList.append(clusterId)
        for clusterId in mitosisClusterIdList:
            # to change mitosis mode uncomment one of the lines below
            self.divideClusterRandomOrientation(clusterId)
            # self.divideClusterOrientationVectorBased(clusterId,1,0,0)
            # self.divideClusterAlongMajorAxis(clusterId)
            # self.divideClusterAlongMinorAxis(clusterId)

    def updateAttributes(self):
        compartmentListParent=self.getClusterCells(self.parentCell.clusterId)

        for i in xrange(compartmentListParent.size()):
            compartmentListParent[i].targetVolume/=2.0
        self.cloneParentCluster2ChildCluster()
```

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 getClusterCells which returns a list of cells belonging to a cluster with a given cluster id:

```
compartmentListParent = self.getClusterCells(self.parentCell.clusterId)
```

The call above returns a list of cells in a cluster with `clusterID` specified by self.`parentCell.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 `compartmentListParent` is indexable (ie. we can access directly any element of the list provided our index is not out of bounds).

```
        for i in xrange(compartmentListParent.size()):
            compartmentListParent[i].targetVolume/=2.0
```

Notice that nowhere in the update attribute finction 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 `updateAttributes` function is `self.cloneParentCluster2ChildCluster()`. 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:

```
compartmentListParent = self.getClusterCells(self.parentCell.clusterId)

compartmentListChild=self.getClusterCells(self.childCell.clusterId)

self.cloneClusterAttributes(self,sourceCellCluster = compartmentListParent,\
targetCellCluster= compartmentListChild ,\
no_clone_key_dict_list = ['ATTR_NAME_1' , 'ATTR_NAME_2'] )
```

where `cloneClusterAttributes` 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:

```
class MitosisSteppableClusters(MitosisSteppableClustersBase):
    def __init__(self,_simulator,_frequency=1):
        MitosisSteppableClustersBase.__init__(self,_simulator, _frequency)

    def step(self,mcs):

        for cell in self.cellList:
            clusterCellList=self.getClusterCells(cell.clusterId)
            for cellLocal in clusterCellList:

        mitosisClusterIdList=[]
        for compartmentList in self.clusterList:
            clusterId=0
            clusterVolume=0
            for cell in CompartmentList(compartmentList):
                clusterVolume+=cell.volume
                clusterId=cell.clusterId

            if clusterVolume>250:
                mitosisClusterIdList.append(clusterId)
        for clusterId in mitosisClusterIdList:
            # to change mitosis mode uncomment one of the lines below
            self.divideClusterRandomOrientation(clusterId)
            # self.divideClusterOrientationVectorBased(clusterId,1,0,0)
            # self.divideClusterAlongMajorAxis(clusterId)
            # self.divideClusterAlongMinorAxis(clusterId)

    def updateAttributes(self):
```

```
            parentCell=self.mitosisSteppable.parentCell
            childCell=self.mitosisSteppable.childCell

            compartmentListChild\
            =self.getClusterCells(childCell.clusterId)
            compartmentListParent\
            =self.getClusterCells(parentCell.clusterId)

            for i in xrange(compartmentListChild.size()):
                compartmentListParent[i].targetVolume/=2.0

                compartmentListChild[i].targetVolume\
                =compartmentListParent[i].targetVolume
                compartmentListChild[i].lambdaVolume\
                =compartmentListParent[i].lambdaVolume
```

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

31. 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 with 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 cluyster 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.inventory.reassignClusterId(cell,550)
```

When you attempt to use former syntax CC3D will throw an exception and inform you that you need to chang the syntax.

32. 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 refered 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:

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

    def start(self):

        # adding options that setup SBML solver integrator - these are optional
        options={'relative':1e-10,'absolute':1e-12,'steps':10,'stiff':False}
        self.setSBMLGlobalOptions(options)

        modelFile='Simulation/test_1.xml'

        initialConditions={}
        initialConditions['S1']=0.00020
        initialConditions['S2']=0.000002

        self.addSBMLToCellIds(_modelFile=modelFile,_modelName='dp',\
        _ids=range(1,11),_stepSize=0.5,_initialConditions=initialConditions)

        self.addFreeFloatingSBML(_modelFile=modelFile,_modelName='Medium_dp',\
        _stepSize=0.5,_initialConditions=initialConditions)
        self.addFreeFloatingSBML(_modelFile=modelFile,_modelName='Medium_dp1',\
        _stepSize=0.5,_initialConditions=initialConditions)
        self.addFreeFloatingSBML(_modelFile=modelFile,_modelName='Medium_dp2')
        self.addFreeFloatingSBML(_modelFile=modelFile,_modelName='Medium_dp3')
        self.addFreeFloatingSBML(_modelFile=modelFile,_modelName='Medium_dp4')

        cell20=self.attemptFetchingCellById(20)

        self.addSBMLToCell(_modelFile=modelFile,_modelName='dp',_cell=cell20)

    def step(self,mcs):

        self.timestepSBML()

        cell10=self.inventory.attemptFetchingCellById(10)
        print 'cell=',cell10

        speciesDict=self.getSBMLState(_modelName='Medium_dp2')
        print 'speciesDict=',speciesDict.values()
```

```
        state={}
        state['S1']=10
        state['S2']=0.5
        if mcs==3:
            self.setSBMLState('Medium_dp2',_state=state)

#        if mcs==5:
#            self.deleteSBMLFromCellIds(_modelName='dp',_ids=range(1,11))


        if mcs==7:
            cell25=self.inventory.attemptFetchingCellById(25)
            self.copySBMLs(_fromCell=cell10,_toCell=cell25)
```

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 rmodel 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:

```
        modelFile='Simulation/test_1.xml'

        initialConditions={}
        initialConditions['S1']=0.00020
        initialConditions['S2']=0.000002
```

Remark: 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 `addSBMLToCellIds` function let us focus on statements that open the start function:

```
        options={'relative':1e-10,'absolute':1e-12,'steps':10,'stiff':False}
        self.setSBMLGlobalOptions(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`.
After we define 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 `setSBMLGlobalOptions`. 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.getSBMLGlobalOptions()
```

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:

```
self.addSBMLToCellIds(_modelFile=modelFile,_modelName='dp',\
_ids=range(1,11),_stepSize=0.5,_initialConditions=initialConditions)
```

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 arbitrary model identifier that you use to retrieve model values. The name of the function is `addSBMLToCellIds` and the third argument is a Python list that contains cell ids to which CC3D wil attach aninstance of the SBML Solver.

Remark: 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 SBML Solver function that associates models with a cell or adds free floating model call RoadRunnerLib functions that parse SBML, translate it to C, compile generated C code to dynamically loaded library, load the library and make it ready for use. 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.addFreeFloatingSBML` 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.addFreeFloatingSBML` 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 fincitons demonstrate how to add SBML Solver object to a single cell:

```
cell20=self.attemptFetchingCellById(20)
self.addSBMLToCell(_modelFile=modelFile,_modelName='dp',_cell=cell20)
```

Instead of passing listof cell ids we pass cell object (`cell20`).
We can also associate SBML model with certain cell types using the following syntax:

```
self.addSBMLToCellTypes(_modelFile=modelFile,_modelName='dp',\
_types=[self.A,self.B],_stepSize=0.5,_initialConditions=initialConditions)
```

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.timestepSBML`. 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 soecies after integration step we can call e.g.

```
state=self.getSBMLState(_modelName='Medium_dp2')
print state=',state.values()
```

These functions check and print model variables for free floating model called `Medium_dp2`.
The next set of function calls:

```
        state={}
        state['S1']=10
        state['S2']=0.5
        if mcs==3:
            self.setSBMLState('Medium_dp2',_state=state)
```

set new state for for free floating model called `Medium_dp2`. If we wanted to retrieve state of the model `dp` belonging to cell object called `cell20` we would use the following syntax:

```
state=self.getSBMLState(_modelName='dp',_cell=cell20)
```

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

```
state={}
state['S1']=10
state['S2']=0.5
self.setSBMLState(_modelName='dp',_cell=cell20,_state=state)
```

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

```
self.deleteSBMLFromCellIds(_modelName='dp',_ids=range(1,11))
```

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

```
self.deleteSBMLFromCellTypes(_modelName='dp',_types=range[self.A,self.B])
```

from single cell:

```
self.deleteSBMLFromCell(_modelName='dp',_cell=cell20)
```

or delete free floating SBML Solver object:

```
self.deleteFreeFloatingSBML(_modelName='Medium_dp2')
```

Remark: When cells get deleted all SBML Solver models are deleted automatically. You do not need to call deleteSBML functions in such a case.

Sometimes you may encounter a need to clone all SBML models from one cell to another (e.g. in the mitosis updateAttributes function where you clone SBML Solver objects from parent cell to a child cell). SBML Solver lets you do that very easily:

```
cell10=self.inventory.attemptFetchingCellById(10)
cell25=self.inventory.attemptFetchingCellById(25)
self.copySBMLs(_fromCell=cell10,_toCell=cell25)
```

What happens here is that source cell (_fromCell) provides SBML Solver object templates and based on these templates new SBML Solver objects are gets created and CC3D assigns them to target cell (_toCell). All the state variables in the targer 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:

```
cell10=self.inventory.attemptFetchingCellById(10)
cell25=self.inventory.attemptFetchingCellById(25)
self.copySBMLs(_fromCell=cell10,_toCell=cell25, _sbmlNames=['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 multicellular implementation of Delta-Notch mutual inhibitory coupling. In this juxtacrine signaling process, a cell's level of membrane-bound Delta depends on its intracellular level of activated Notch, which in turn depends on the average level of membrane-bound Delta of its neighbors. In such a situation, the Delta-Notch dynamics of the cells in a tissue sheet will depend on the rate of cell rearrangement and the fluctuations it induces. While the example does not explore the richness due to the coupling of subcellular networks with intercellular networks and cell behaviors, it already shows how different such behaviors can be from those of their non-spatial simplifications. We begin with the Ordinary Differential Equation (*ODE*) Delta-Notch patterning model of Collier in which juxtacrine signaling controls the internal levels of the cells' Delta and Notch proteins. The base model neglects the complexity of the interaction due to changing spatial relationships in a real tissue:

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

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

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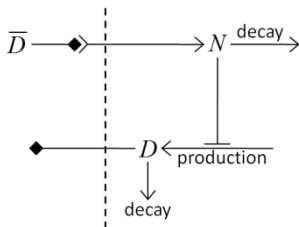

Figure 18 Diagram of Delta-Notch feedback regulation between and within cells.

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:

```
class DeltaNotchClass(SteppableBasePy):
    def __init__(self,_simulator,_frequency):
        SteppableBasePy.__init__(self,_simulator,_frequency)
    def start(self):

        modelFile='Simulation/DN_Collier.sbml'
        self.addSBMLToCellTypes(_modelFile=modelFile,_modelName='DN',\
        _types=[self.TYPEA],_stepSize=0.2)

        #Initial conditions
        import random

        state={} #dictionary to store state veriables of the SBML model

        for cell in self.cellList:
```

```
                state['D'] = random.uniform(0.9,1.0)
                state['N'] = random.uniform(0.9,1.0)
                self.setSBMLState(_modelName='DN',_cell=cell,_state=state)

                cellDict=self.getDictionaryAttribute(cell)
                cellDict['D']=state['D']
                cellDict['N']=state['N']

    def step(self,mcs):
        for cell in self.cellList:

            D=0.0; nn=0
            for neighbor , commonSurfaceArea in self.getCellNeighborDataList(cell)
                if neighbor:
                    nn+=1
                    state=self.getSBMLState(_modelName='DN',_cell=neighbor)

                    D+=state['D']
            if (nn>0):
                D=D/nn

            state={}
            state['Davg']=D
            self.setSBMLState(_modelName='DN',_cell=cell,_state=state)

            state=self.getSBMLState(_modelName='DN',_cell=cell)
            cellDict= self.getDictionaryAttribute(cell)
            cellDict['D']=D
            cellDict['N']=state['N']
        self.timestepSBML()
```

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 - `Davg`). We pass `Davg` to the SBML Solver for each cell and then carry out integration for the new time step. Before calling `self.timestepSBML` 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 subcellular scales.

33.   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 packkeges 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:
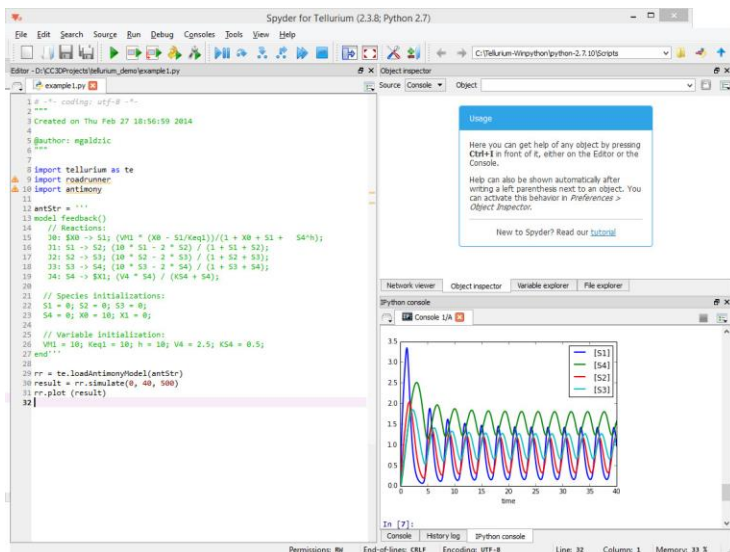

Hopefully this was easy. Now let's build our own model. We start creating our new reaction-kinetics model by pressing "New File" button - the first button on the left in the main toolbar:
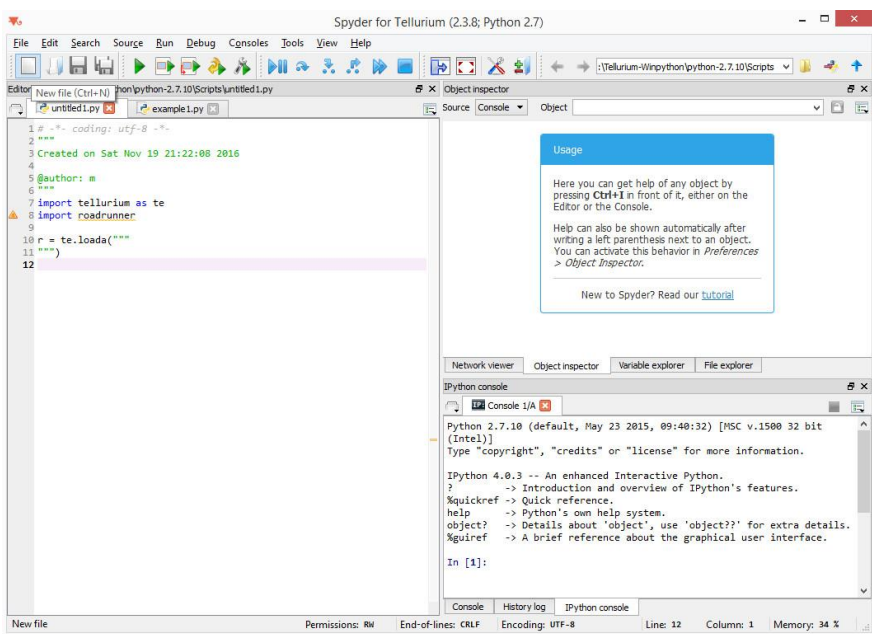
Pressing "New File" buttin opens up a Tellurium template for reaction-kinetics model. All we need to do is to define the model and add few statements that will export it in the SBML format.

Let's start by defining the model. The model definition uses language called Antimony. To learn more about Antimony please visit its tutorial page http://tellurium.analogmachine.org/documentation/antimony-tutorial/ . Antimony is quite intuitive and allows to specify system of chemical reaction in a way that is very natural to anybody who has basic understanding of chemical reaction notation. In our case we will define a simple relaxation oscillator that consists of two floating species (s_1 and s_2) and two boundary species (x_0, x_3). Boundary species serve as sources/sinks of the reaction and their concentrations remain constant.Concentration of floating species change with time. The presented example has been developed by Herbert Sauro (University of Washington) - one of the three "founding fathers" of SBML, author of many books on reaction kinetics and lead

contributor to the Tellurium and Systems Biology Workbench packages. Let's look at the reaction schematics we are about to implement:
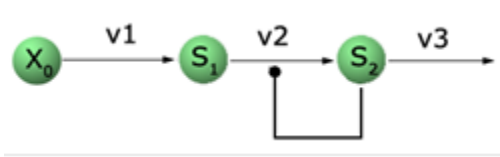
Qualitatively, x0 is being kept at a constant concentration of 1.0 (we assume arbitrary units here) and s1 accumulates at the constant rate v1. s1 also gets "transformed" to s2 at the rate v2 which depends on concentration of s1 and s2 in such a way that when s1 and s2 are at relatively high values the reaction "spepds 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 oscilaltions. 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:

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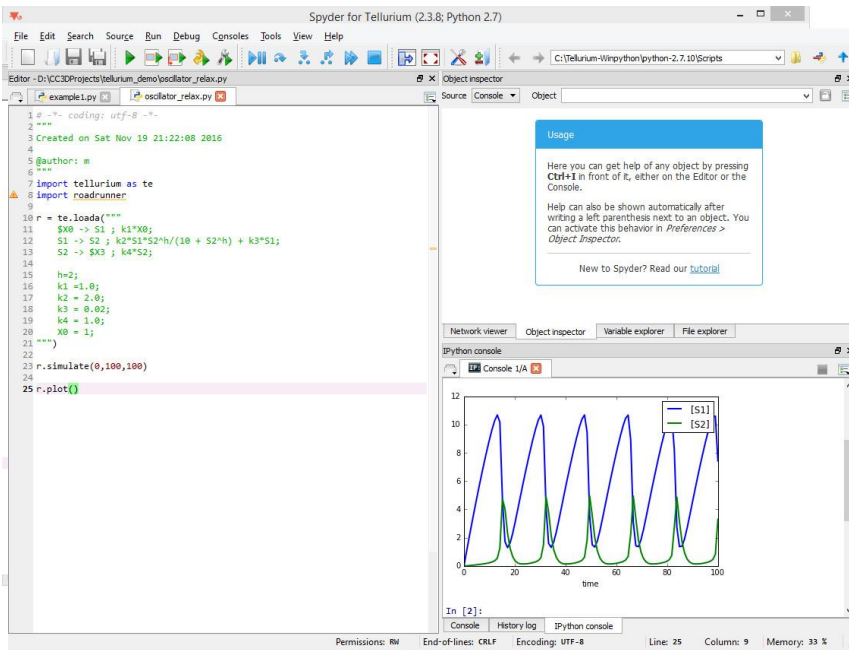
Notice that we need to add two lines of code - one to actually solve the model

```
r.simulate(0,100,100)
```

And another one to plot the results

```
r.plot()
```

The general philosophy of Tellurium is that you desine 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

34. Parameter Scans

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.

To describe parameter scan users use Twedit++ to generate an XML file which describes the parameter scan. This XML file is referenced from `.cc3d` file (see example below) and this is how CC3D figures out that it has to scan parameter space:

```
<Simulation version="3.5.1">
   <XMLScript Type="XMLScript">Simulation/CellSorting.xml</XMLScript>
   <PythonScript Type="PythonScript">Simulation/CellSorting.py</PythonScript>
   <Resource Type="Python">Simulation/CellSortingSteppables.py</Resource>
   <ParameterScan Type="ParameterScan">Simulation/ParameterScanSpecs.xml</ParameterScan>
</Simulation>
```

As compared to "regular" `.cc3d` file the one we show above has extra line with `ParameterScan` XML element. Clearly if we comment this line we can run the simulation in a single-run mode. This is great advantage of introducing additional XML file for parameter scans – the simulation code (CC3DML, Python scripts) is unmodified and can be run as-is in case modelers decide to tunr off scanning of the parameter space.

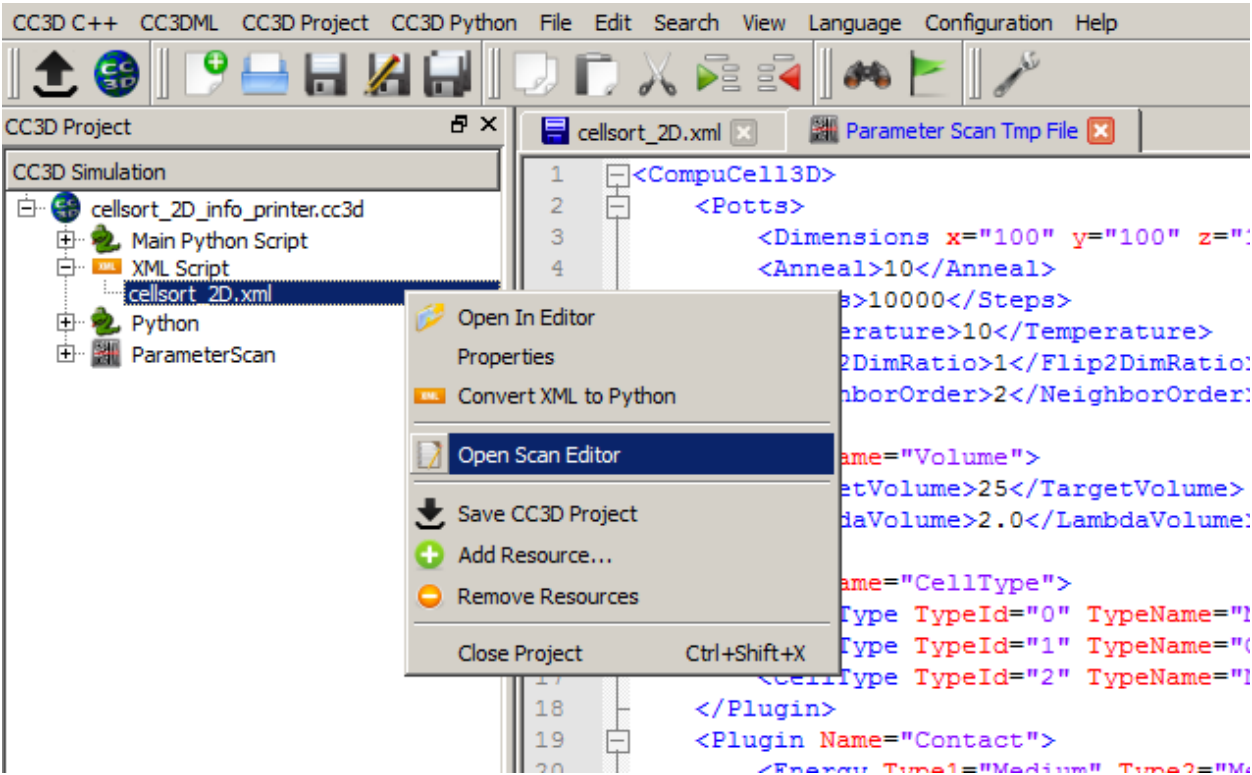### 34.1.Setting up Parameter Scan Using Twedit++

To setup up parameter scan we open any valid `.cc3d` project file in Twedit++:
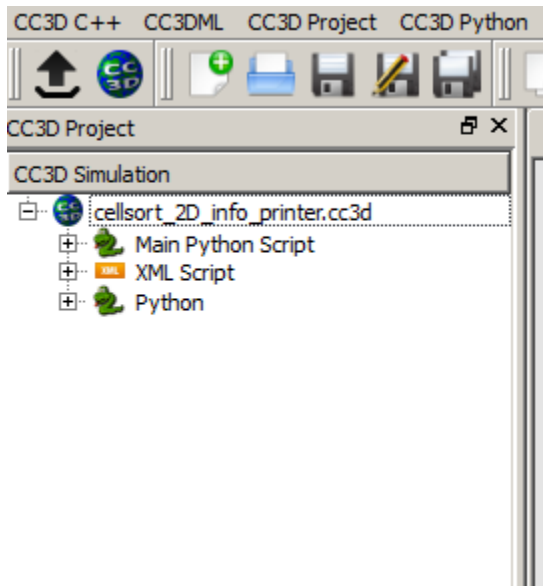
We right-click on the project and select `Add Parameter Scan` option:
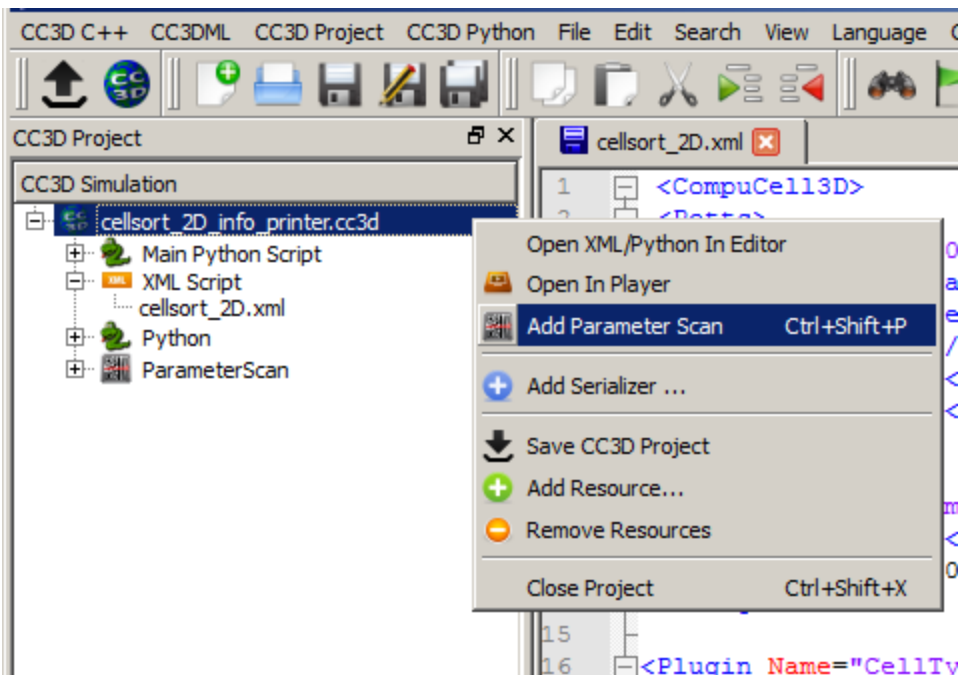
Notice that Parameter Scan appears as a part of the `.cc3d` project. To add a parameter from the CC3DML to the parameter scan we right-click on the CC3DML file and select `Open Scan Editor`:


Notice that Twedit++ opens up a new tab called `Parameter Scan Tmp File`. This is read-only file that you use to select parameters for scanning purposes. To do that, you click in the desired place of this file. For example if you want to to run simulation with different `Temperature` parameters you click in the line with `Temperature` parameter and then right-click to get access to `Add To Scan…` option:
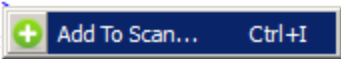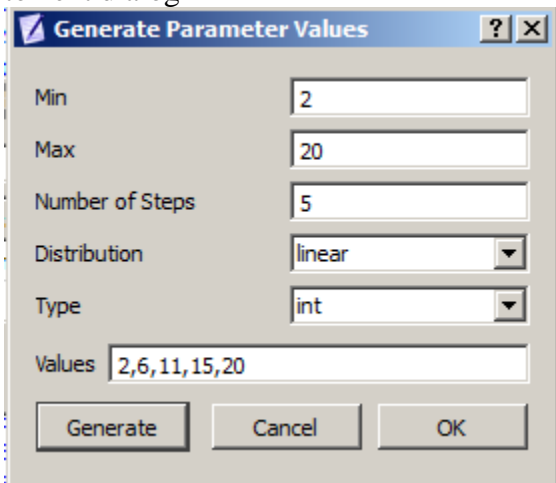
After you choose this option Twedit++ displays parameter scan configuration dialog:
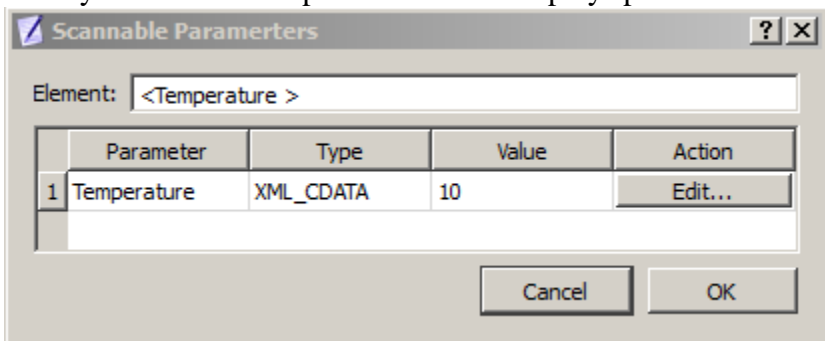

This dialog displays all parameters in the given XML element which can change. In our case of a simple XML element we have only oneway to change the element and it is through its value (`10`). When we click edit we get to next dialog


Where we fill minimum, maximum value, select value type (we picked integer but we could have picked float as well) and generate values (given by the `Number  Of Steps`) assording to given distribution (here we picked linear which means we generate 5 equally spaced values between 2 and 20 inclusive). When we open parameter scan XML file we see that our choices made in the GUI got translated into XML format:
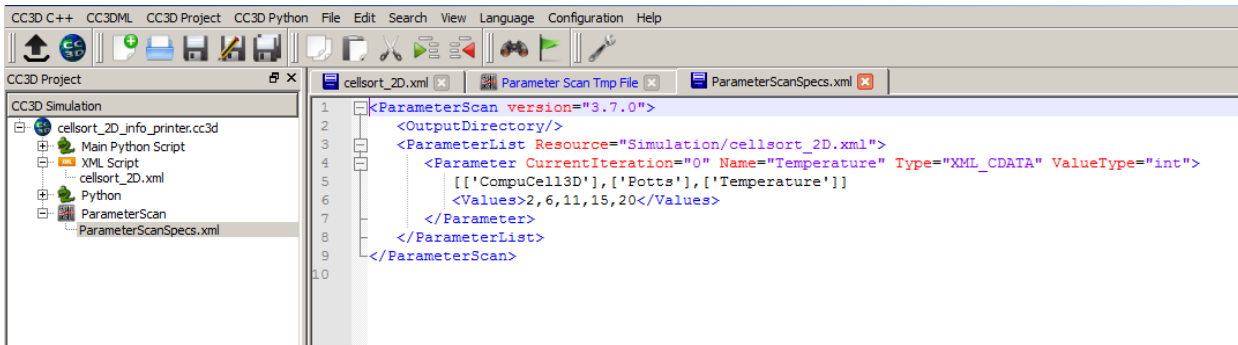
To add more parameters we position the cursor in the desired location in the `Parameter Scan Tmp File` tab, right click to select `Add To Scan…` option and followsteps outlined above. When we select more complicated element e.g. `<Dimensions>` we will get the following parameter configuration dialog:
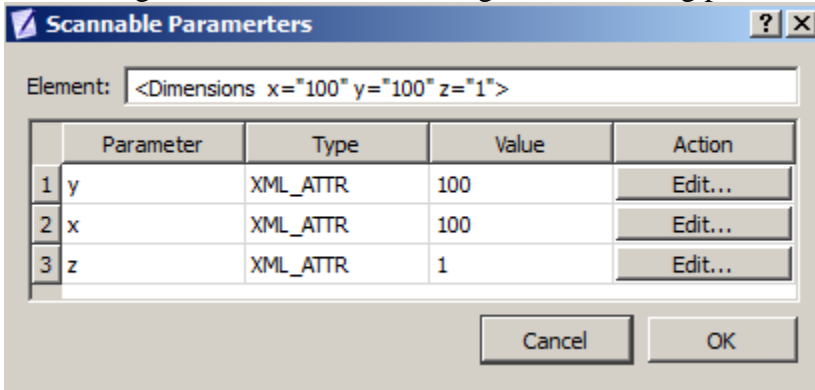
Notice that for this element there are 3 values which can change when we pick , say `y`, and generate values in the next pop-up widget we will end up with the following parameter scan XML file:
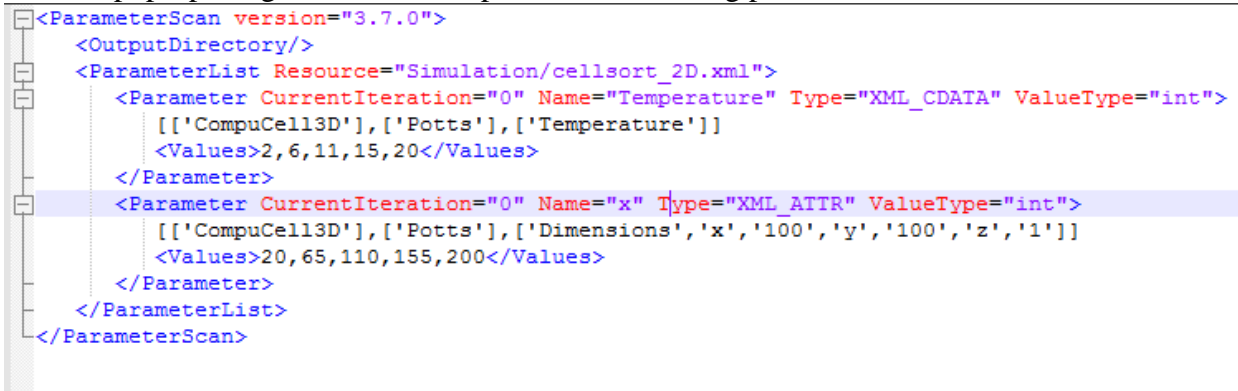

As you can tell this XML file gets updated automatically so users do not need to type XML file.
Adding Python parameters to parameter scan is even easier. We open up Python file using `Open Scan Editor` option:
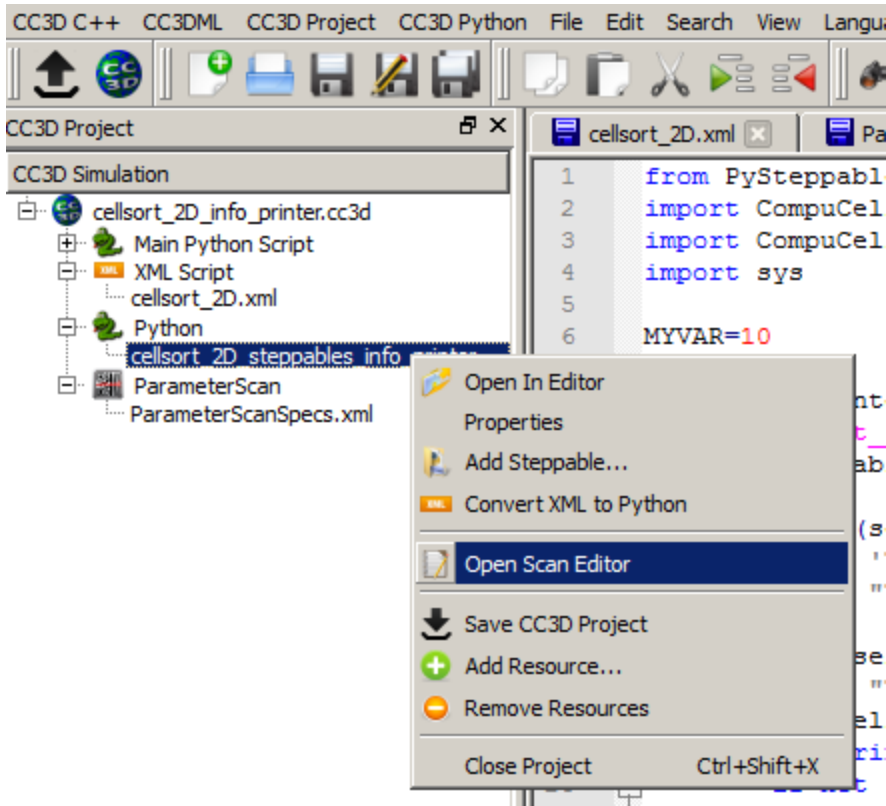
Later we go to the `Parameter Scan Tmp File Tab` which now has Python code open in the read-only mode and right click on any global variable to add it to the parameter scan. We have only one global variable defined in our demo script (`MYVAR`) so in the entire script only one line which defines `MYVAR` can be used to ope up Parameter scan dialog:
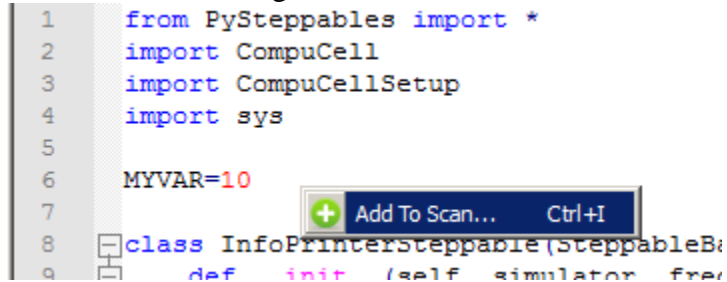
As a result `Generate Parameter Values` dialog will pop up and we use it in exactly the same way we used it for the XML parameters:
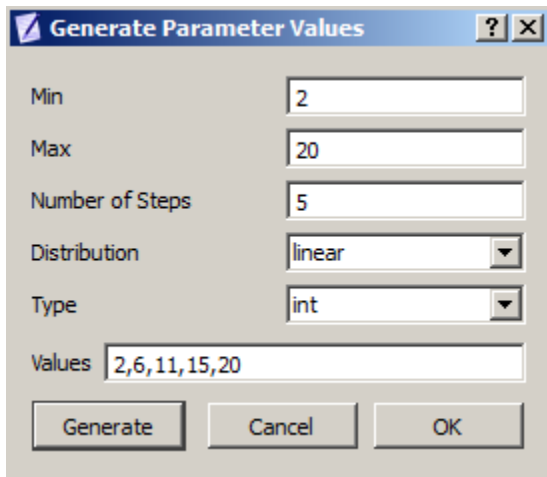
After we are doen our parameter scan XML file looks as follows:

```xml
<ParameterScan version="3.7.0">
    <OutputDirectory/>
    <ParameterList Resource="Simulation/cellsort_2D.xml">
        <Parameter CurrentIteration="0" Name="Temperature" Type="XML_CDATA" ValueType="int">
            [['CompuCell3D'],['Potts'],['Temperature']]
            <Values>2,6,11,15,20</Values>
        </Parameter>
        <Parameter CurrentIteration="0" Name="x" Type="XML_ATTR" ValueType="int">
            [['CompuCell3D'],['Potts'],['Dimensions','x','100','y','100','z','1']]
            <Values>20,65,110,155,200</Values>
        </Parameter>
    </ParameterList>
    <ParameterList Resource="Simulation/cellsort_2D_steppables_info_printer.py">
        <Parameter CurrentIteration="0" Name="MYVAR" Type="PYTHON_GLOBAL" ValueType="int">
            <Values>2,6,11,15,20</Values>
        </Parameter>
    </ParameterList>
</ParameterScan>
```

Finally we can also edit the output directory of the parameter scan results by manually editing the parameter scan XML. Here we set it to

```xml
<OutputDirectory>InfoPrinter_ParameterScan</OutputDirectory>
```

```xml
<ParameterScan version="3.7.0">
    <OutputDirectory>InfoPrinter_ParameterScan</OutputDirectory>
    <ParameterList Resource="Simulation/cellsort_2D.xml">
        <Parameter CurrentIteration="0" Name="Temperature" Type="XML_CDATA" ValueType="int">
            [['CompuCell3D'],['Potts'],['Temperature']]
            <Values>2,6,11,15,20</Values>
        </Parameter>
        <Parameter CurrentIteration="0" Name="x" Type="XML_ATTR" ValueType="int">
            [['CompuCell3D'],['Potts'],['Dimensions','x','100','y','100','z','1']]
            <Values>20,65,110,155,200</Values>
        </Parameter>
    </ParameterList>
    <ParameterList Resource="Simulation/cellsort_2D_steppables_info_printer.py">
        <Parameter CurrentIteration="0" Name="MYVAR" Type="PYTHON_GLOBAL" ValueType="int">
            <Values>2,6,11,15,20</Values>
        </Parameter>
    </ParameterList>
</ParameterScan>
```

34.2. Running Parameter Scans

As you can see defining parameters scans is quite easy. Running simulation is easy as well and you have two options. The most straightforward (but probably not the best – keep reading to find the recommended way of running) is to open up the project in the Player and run it:
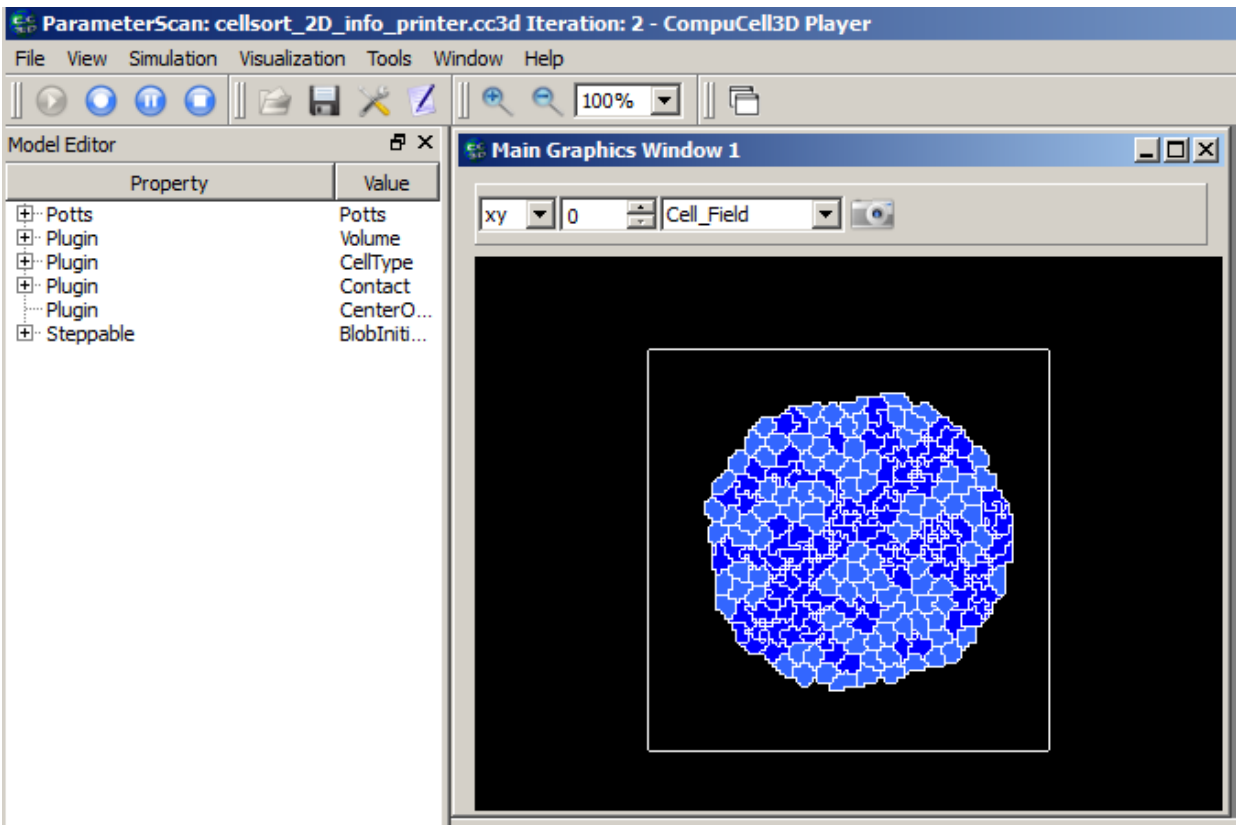
Notice that the title bar of the player informs you that you are running parameter scan . It also displays number of the parameter simulation that is currently being executed. (Iteration: 2). The results of the parameter scans are written to the Simulation Output directory :
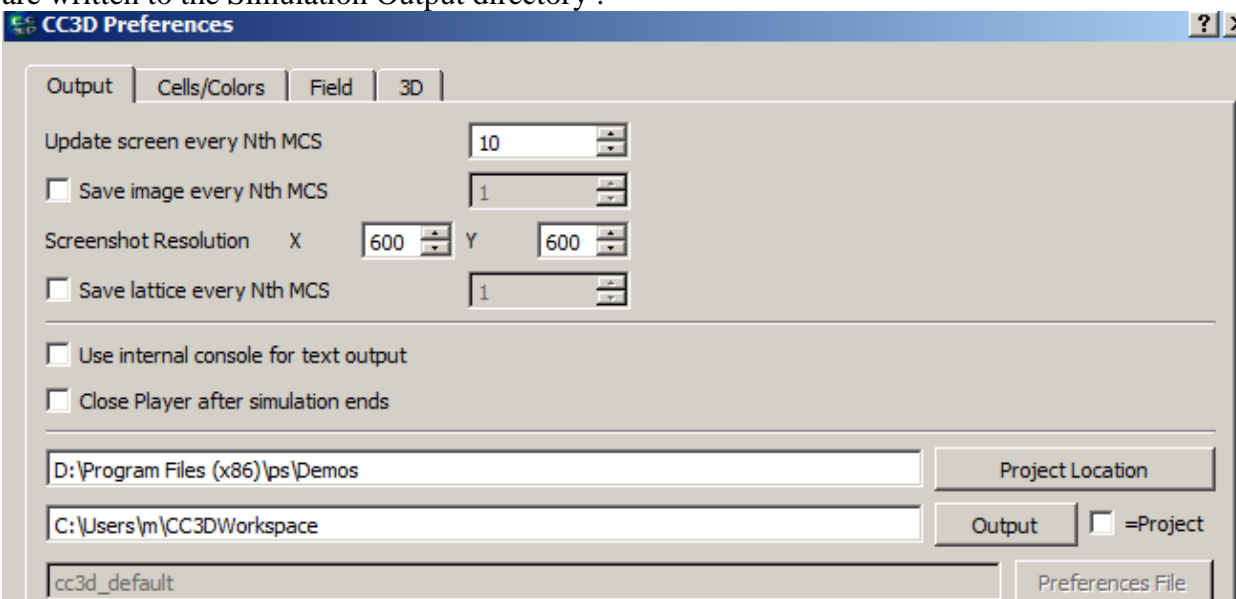

In my case it is `c:\Users\m\CC3DWorkspace`, the default setting for simulation outoput directory is is `<your home directory>/CC3DWorkspace`.

When we look into this directory we will see that it contains subdurectories corresponding to the consecutive simulation runs – each such simulation corresponds to a different set of parameters:

Since at the moment CC3D is performin exhaustive parameter scan you have to be careful witht the number of parameters you scan and number of values each parameter can assume. Simply put the total number of simulation to run when doing exhaustive parameter scan is equal to a producs of number of values of each parameter. Scanning 10 parameters with 10 values each will require 10 billion simulations. This is a large number and it will take quite a lot of time before all results are ready. On the other hand this might be your gateway to apply for one of the Big Data grants…

Important: When parameter scan runs it modifies ParameterScan.xml file. In particular it record current iteration for each parameter. To rerun parameter scan again from the beginning you need to set `CurrentIteration` values for each parameter to 0. You can do it manually or use Twedit++ option `Reset Parameter Scan`:
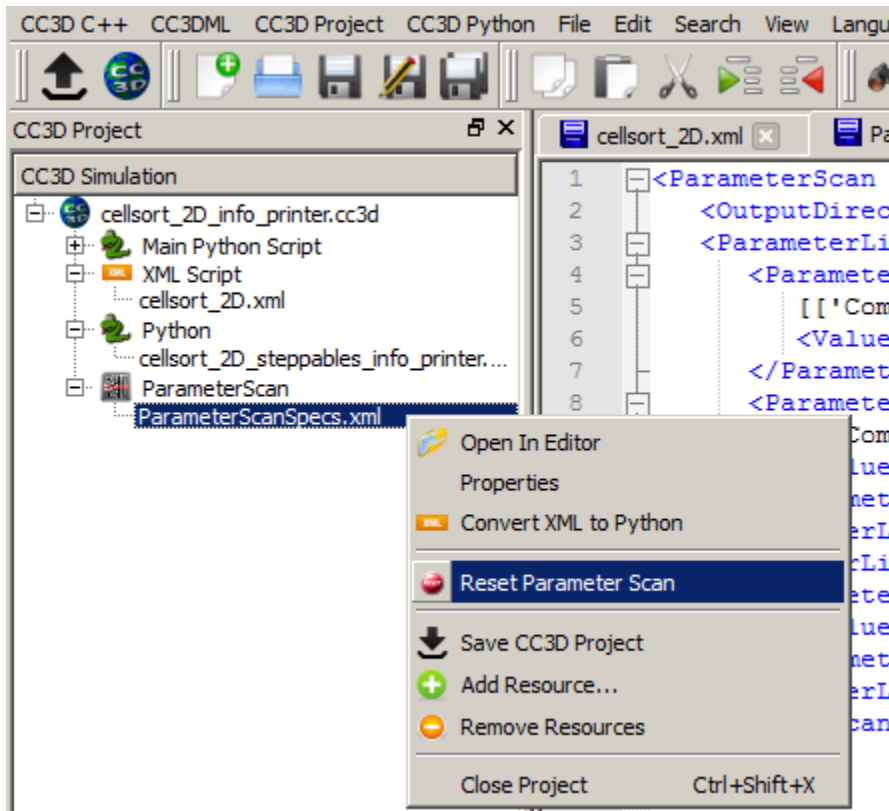
In a nutshell this is all it takes to run parameter scan in CC3D 3.7.1. We will add more options to this feature but at least now you can probe your parameter scans without writing clumsy-looking Python wrappers.

The recommended way of running parameter scan is via script called `paramScan.sh` . `paramScan.sh` takes same command line arguments as `compucell3d.sh` or `runScript.sh` (notice on windows we use `.bat` extension and on OSX `.command`). Here is the syntax:

```
paramScan.sh -i <cc3d project file> --guiScan -maxNumberOfRuns=20 [remaining command line
options used in compucell3d.sh or runScript.sh]
```

`--guiScan` – will ensure that `paramScan.sh` will be run using Player. If you do not use this option the parameter scan will internally use `runScrip.sh`, hence no gui, which is often preferred way of running multiple jobs on clusters. With `--guiScan` enabled `paramScan.sh` calls `compucell3d.sh` internally so you can pass any options you would normally use with `compucell3d.sh`. When `–guiScan` is disabled `paramScan.sh` calls `runScript.sh` so you any options you would normally use with `runScript.sh`.

`--maxNumberOfRuns` - using this option you can stop parameter scan after given number of simulations. By default the parameter scan will run until all the simulation have been finished. This option is most helpful during debugging stage

`[remaining command line options used in compucell3d.sh or runScript.sh]` – here you simply pass additional options you would use with - see explanation of the `--guiScan` switch for explanation

Example commands:
```
paramScan.sh –i vascular.cc3d  --noOutput
paramScan.sh –i vascular.cc3d --guiScan
```

The benefit of using `paramScan.sh` is that it is fault tolerant. Say, your simulation crashes in the middle because e.g. somewhere in the Python script you divide by 0. `paramScan.sh` will handle this situation and start new subsequent simulation. If you used Player or runScript.sh directly to run parameter scan the parameter scan would simply stop in that situation. When you do large runs on clusters you might want to keep running scan even if some of the simulations run into trouble. This is why using paramScan.sh is preferred way of running parameter scans, starting with 3.7.3 version.

You may also find example command to run parameter scan in Twedit++: `CC3D Python->Parameter Scan Command Line`.

34.3.Parameter Scan Configuration Details

Let us now discuss how we describe parameter scan for parameters defined in the CC3DML file. In our example we will be scanning parameters defined in the Potts section:
```
<CompuCell3D version="3.6.2">
   <Potts>
      <Dimensions x="100" y="100" z="1"/>
      <Steps>10</Steps>
      <Temperature>10.0</Temperature>
      <NeighborOrder>2</NeighborOrder>
   </Potts>
```

Let us look into XML file describing parameter scan itself (note that you do not need to type this code manually – Twedit++ will generate it for you with just few clicks):
```
<ParameterScan version="3.7.0">
   <OutputDirectory>CellSorting_ParameterScan</OutputDirectory>
   <ParameterList Resource="Simulation/CellSorting.xml">
      <Parameter CurrentIteration="0" Name="y" Type="XML_ATTR" ValueType="float">
         [['CompuCell3D','version','3.6.2'],['Potts'],['Dimensions','x','100','y','100','z','1']]
         <Values>65.0,110.0,120.0</Values>
      </Parameter>
      <Parameter CurrentIteration="0" Name="Steps" Type="XML_CDATA" ValueType="int">
         [['CompuCell3D','version','3.6.2'],['Potts'],['Steps']]
         <Values>1,2,3,4,5,6</Values>
      </Parameter>
   </ParameterList>
   <ParameterList Resource="Simulation/CellSortingSteppables.py">
      <Parameter CurrentIteration="0" Name="MYVAR" Type="PYTHON_GLOBAL" ValueType="int">
         <Values>0,1,2</Values>
      </Parameter>
      <Parameter CurrentIteration="0" Name="MYVAR1" Type="PYTHON_GLOBAL" ValueType="string">
         <Values>"abc1,abc2","abc"</Values>
      </Parameter>
```

```
      </ParameterList>
</ParameterScan>
```

`<OutputDirectory>CellSorting_ParameterScan</OutputDirectory>` is the first child of the root XML element (`<ParameterScan version="3.7.0">`). It specifies the name of the directory where CC3D will store the results of the scan. As it is always the convention in CC3D this directory name is relative to the simulation output path (default setting `<your_home_directory>/CC3DWorkspace`).

The next two elements are "container elements" (`<ParameterList>`) where we store description for each parameter we want to scan. We have separate container element for each file in the .cc3d simulation. Here, we scan paramaters defined in two files `Simulation/CellSorting.xml` and `Simulation/CellSortingSteppables.py`. Consequantly we have two container XML elements:

```
<ParameterList Resource="Simulation/CellSorting.xml">
```

and

```
<ParameterList Resource="Simulation/CellSortingSteppables.py">
```

Each subelement of `<ParameterList>` is an element how one particular parameter will be scanned. Let us concentrate first on the XML parameters.

```
      <Parameter CurrentIteration="1" Name="y" Type="XML_ATTR" ValueType="float">
        [['CompuCell3D','version','3.6.2'],['Potts'],['Dimensions','x','100','y','100','z','1']]
        <Values>65.0,110.0,120.0</Values>
      </Parameter>
```

Each `<Parameter>` element has CurrentIteration attribute which determines an index of the current value of the paramters. In this example this index has value `1`, hence it points to second element of the list `<Values>`. In this case the value of the parameter will be `110.0` . The name of the parameter in the XML file which will assume value of `110.0` is `y` as indicated by `Name="y"` attribute. The parameter `y` is an attribute of the

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

element of the CC3DML file as shown above.For this reason, in the parameter scan XML file we set the type of this parameter to be `XML_ATTR` (`Type="XML_ATTR"`) and we also set its `value-type` to be float (`ValueType="float"`). While dimensions are integer numbers we purposely set it to `float` to demonstrate that it is up to the modeler to assign correct value type in the `<Parameter>` element. In this case no harm will be done as the float will be converted to integer during parsing but in general one has to be aware that specifying incorrect value type may result in malfunctioning simulation.

The value of the `<Parameter>` element:

```
[['CompuCell3D','version','3.6.2'],['Potts'],['Dimensions','x','100','y','100','z','1']]
```

Determines XML access path that allows CC3D to locate correct parameter `y` in the CC3DML file. For details on how to construct access path please check Steering section of theis manual. The only change as comparet to steering is that we include `['CompuCell3D','version','3.6.2']` as a root element and the entire access path is containe in list bracket Python operators:

```
[[RootSpec],[Child1Spec],[Child1_1 spec],…]
```

As compared to XML parameter scan specification, specifying Python parameter scan is much simpler. This is because In Python we only allow global parameters to be scanned. Consequently each of the `<Parameter>` elements will have structure similar to the one below:

```
      <Parameter CurrentIteration="0" Name="MYVAR" Type="PYTHON_GLOBAL" ValueType="int">
        <Values>0,1,2</Values>
      </Parameter>
```

Here the parameter that we will scan has name `MYVAR`, is of value-type integer (`int`) and its type is `PYTHON_GLOBAL`.

For completness we include Python code snippet which lists `MYVAR`:

```
from PySteppables import *
import CompuCell
import sys

MYVAR=10
```

```
MYVAR1='new str'

class CellSortingSteppable(SteppableBasePy):

    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator,_frequency)

    def step(self,mcs):
        global MYVAR
        print 'MYVAR=',MYVAR
        for cell in self.cellList:
            if cell.type==self.DARK:
                cell.lambdaVecX=-0.5
```

35. 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. CompuCell3D makes all such tasks very easy as you will shortly see. First let's learn how to set up a simulation that can save itself and later restart:

Open Twedit++ and open the simulation you'd like to serialize and the restart. For example let's choose `SBMLSolverOscillatorDemo.cc3d`:
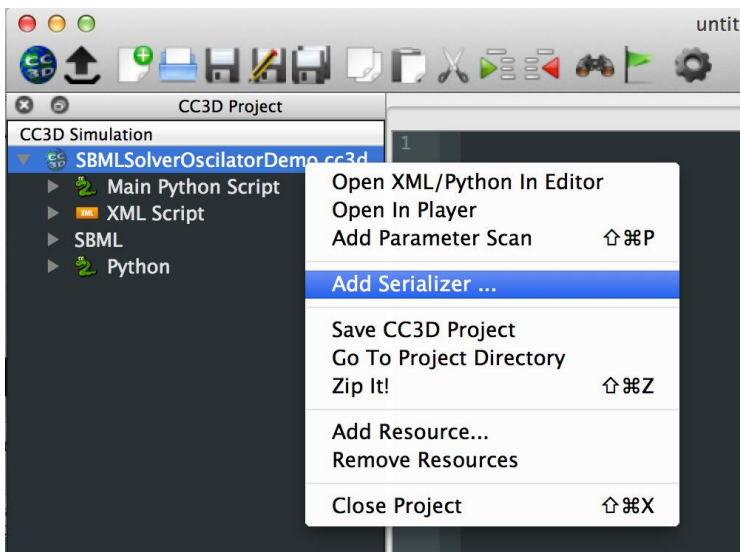

Here we right-click on project tab (`SBMLSolverOscillatorDemo.cc3d`) and choose `Add Serializer…` . In the pop-up window we change snapshot `Output Frequency` to `100` , click `Allow Serialization` and click `OK`:
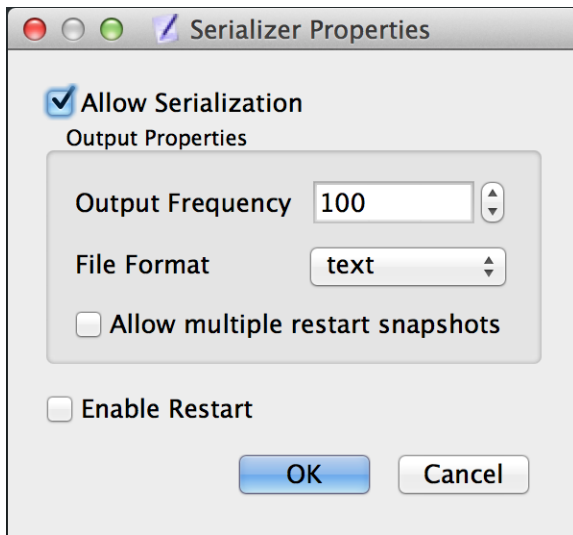
At this point our simulation is set up to take complete snapshots every 100 MCS.

Important: If you want to keep every single snapshot you need to click Allow multiple restart snapshots otherwise only the most recent snapshot will be kept on the hard drive

Before we proceeed we need to save the simulation by right clicking on project tab (`SBMLSolverOscillatorDemo.cc3d`) and choosing `Save CC3D Project`:
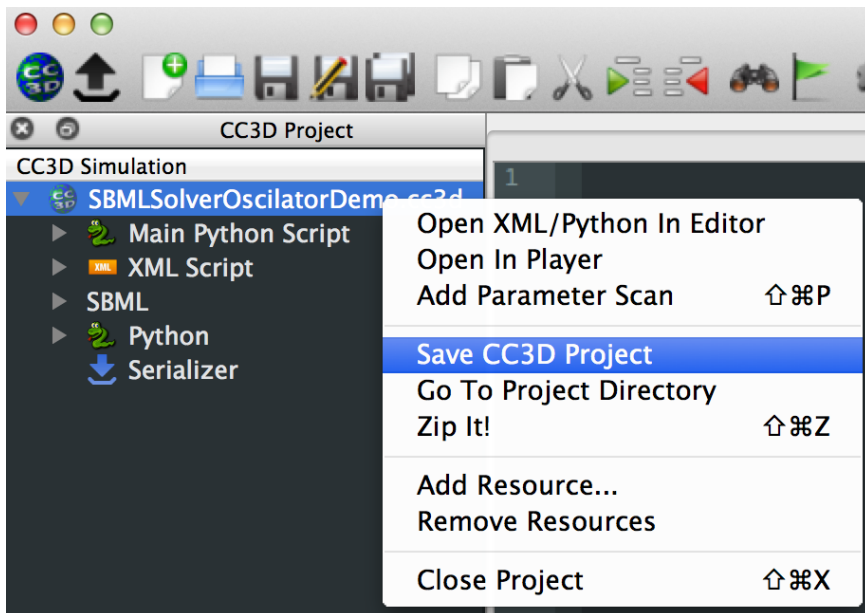
Notice how Serializer was added at the bottom of our CC3D project. If you right-click on it you can make changes to serialization configurations. Just remember to Save CC3D Project when you are done.

If we run our project in Player it will run as usual except that every 100 MCS it will save complete restart snapshot of itself to the hard drive. If you wonder where the snapshots will go open op CC3D Player settings and look at the Output Tab (as you can tell all the output will go to `/Users/m/CC3DWorkspace`):
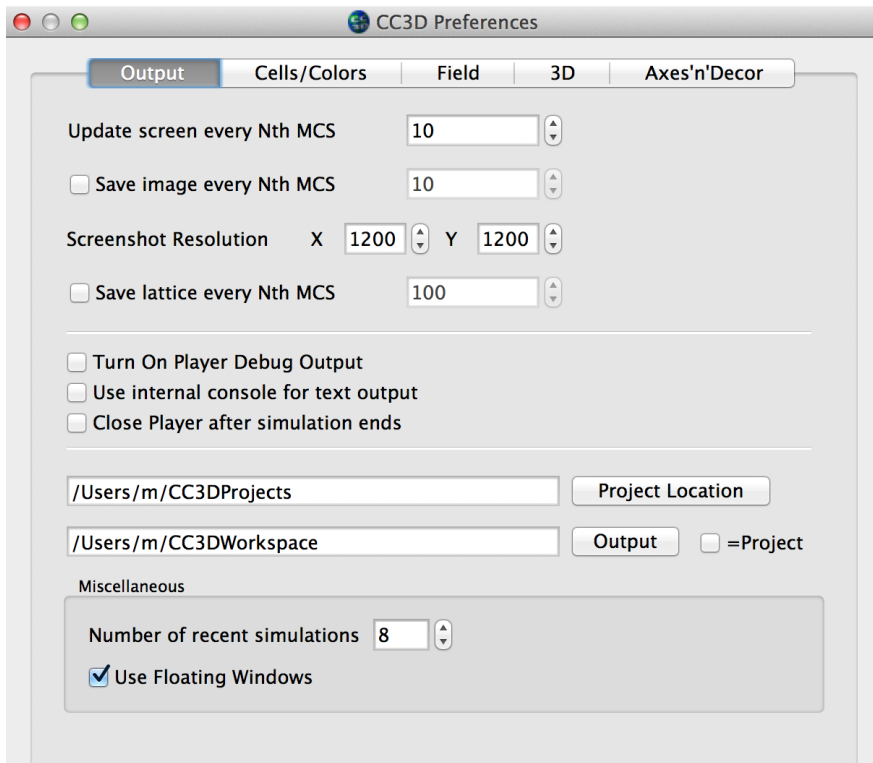
Here is the screenshot of the simulation that we ran up to 300 MCS:
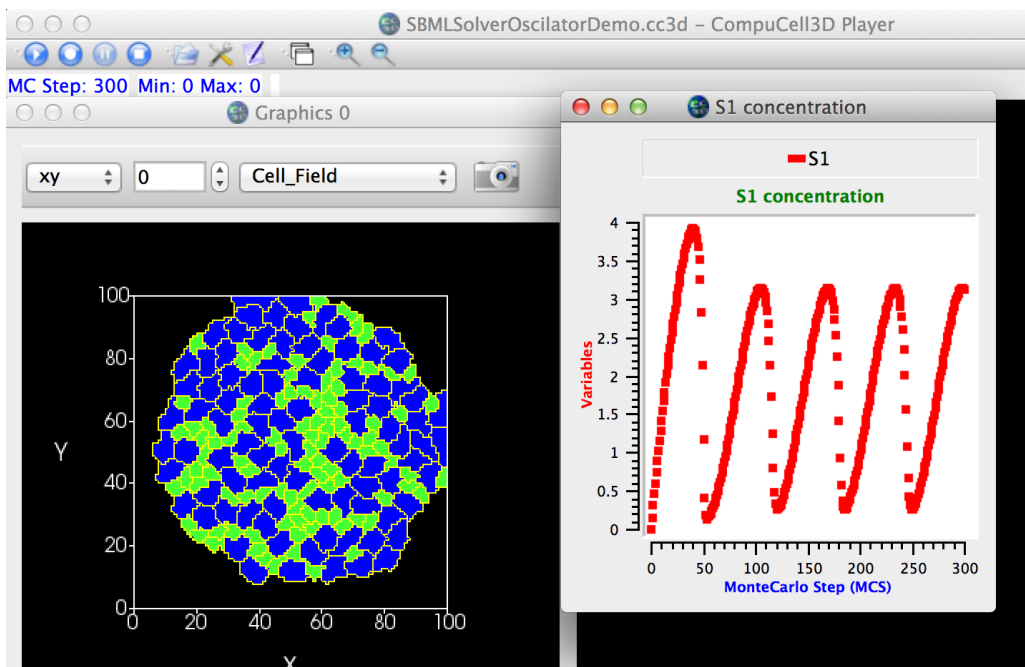
Let's take a look at the content of Simulation output directory (`/Users/m/CC3DWorkspace`):
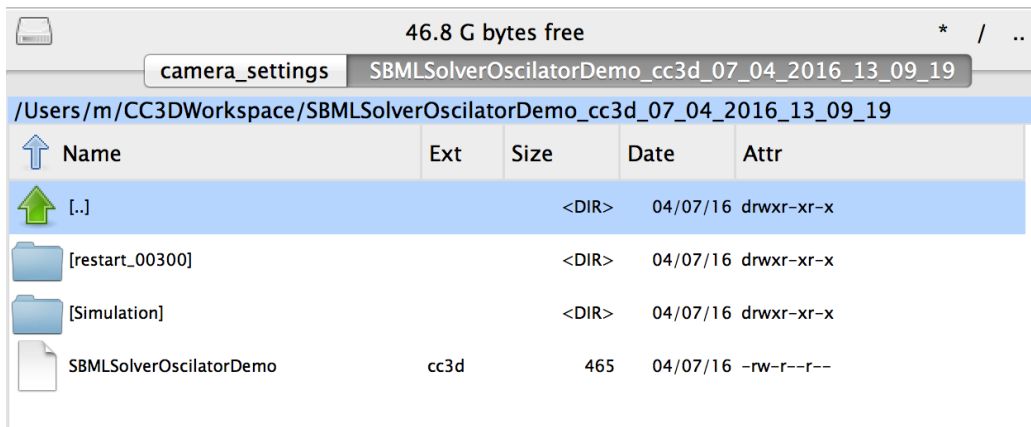
As you can see, CC3D saved time-stamped simulation output directory (`/Users/m/CC3DWorkspace/SBMLSolverOscilatorDemo_cc3d_07_04_2016_13_09_19/`) and inside it you can find `restart_00300` folder that contains all the information needed to restart the simulation at `t=300 MCS`. To restart the simulation we need to do one important step - open in Twedit++ saved simulation (SBMLSovlerOscillatorDemo.cc3d) from restart folder (`/Users/m/CC3DWorkspace/SBMLSolverOscilatorDemo_cc3d_07_04_2016_13_09_19/restart_00300`) , disable serialization and enable restart:
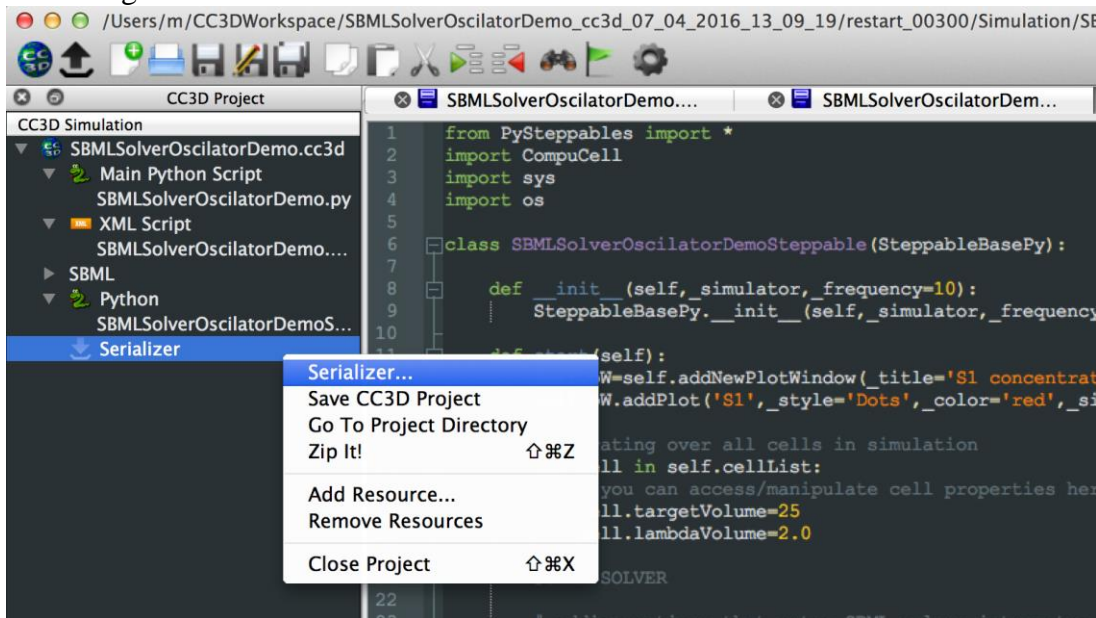
> **Important**: At this step we are modifying simulation from the restart folder NOT the original simulation that we started with and NOT the simulation stored in the top-level output folder - `/Users/m/CC3DWorkspace/SBMLSolverOscilatorDemo_cc3d_07_04_2016_13_09_19/`

Here we go:

Notice, at the top of the Twedit++ window, that we are working with the saved snapshot of the simulation stored in the `restart_00300` subfolder. We will edit Serializer settings of the `.cc3d` project stored in this subfolder to allow restart and disable serialization. As before after we make changes we need to remember to save our project:
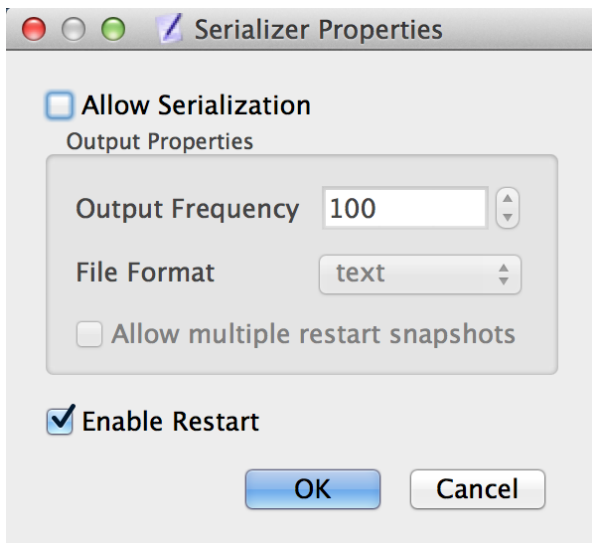
If you are curious what happened here when we change serialization settings you may open `.cc3d` project before and after you make changes and you will see that with serialization enabled our cc3d project looks as follows:

```
<Simulation version="3.5.1">
   <XMLScript Type="XMLScript">Simulation/SBMLSolverOscilatorDemo.xml</XMLScript>
   <PythonScript Type="PythonScript">Simulation/SBMLSolverOscilatorDemo.py</PythonScript>
   <Resource Type="Python">Simulation/SBMLSolverOscilatorDemoSteppables.py</Resource>
   <Resource Type="SBML">Simulation/oscli.sbml</Resource>
   <SerializeSimulation AllowMultipleRestartDirectories="False" FileFormat="text" OutputFrequency="100"/>
</Simulation>
```

And after it will the last line will get replaced to allow restart:

```
<Simulation version="3.5.1">
   <XMLScript Type="XMLScript">Simulation/SBMLSolverOscilatorDemo.xml</XMLScript>
   <PythonScript Type="PythonScript">Simulation/SBMLSolverOscilatorDemo.py</PythonScript>
   <Resource Type="Python">Simulation/SBMLSolverOscilatorDemoSteppables.py</Resource>
   <Resource Type="SBML">Simulation/oscli.sbml</Resource>
   <RestartSimulation RestartDirectory="restart"/>
</Simulation>
```

If you do not like clicking you can easily ope up your favorite editor and modify content of the `.cc3d` project file to allow saving of restart snapshots and to allow restarting from the snapshots.

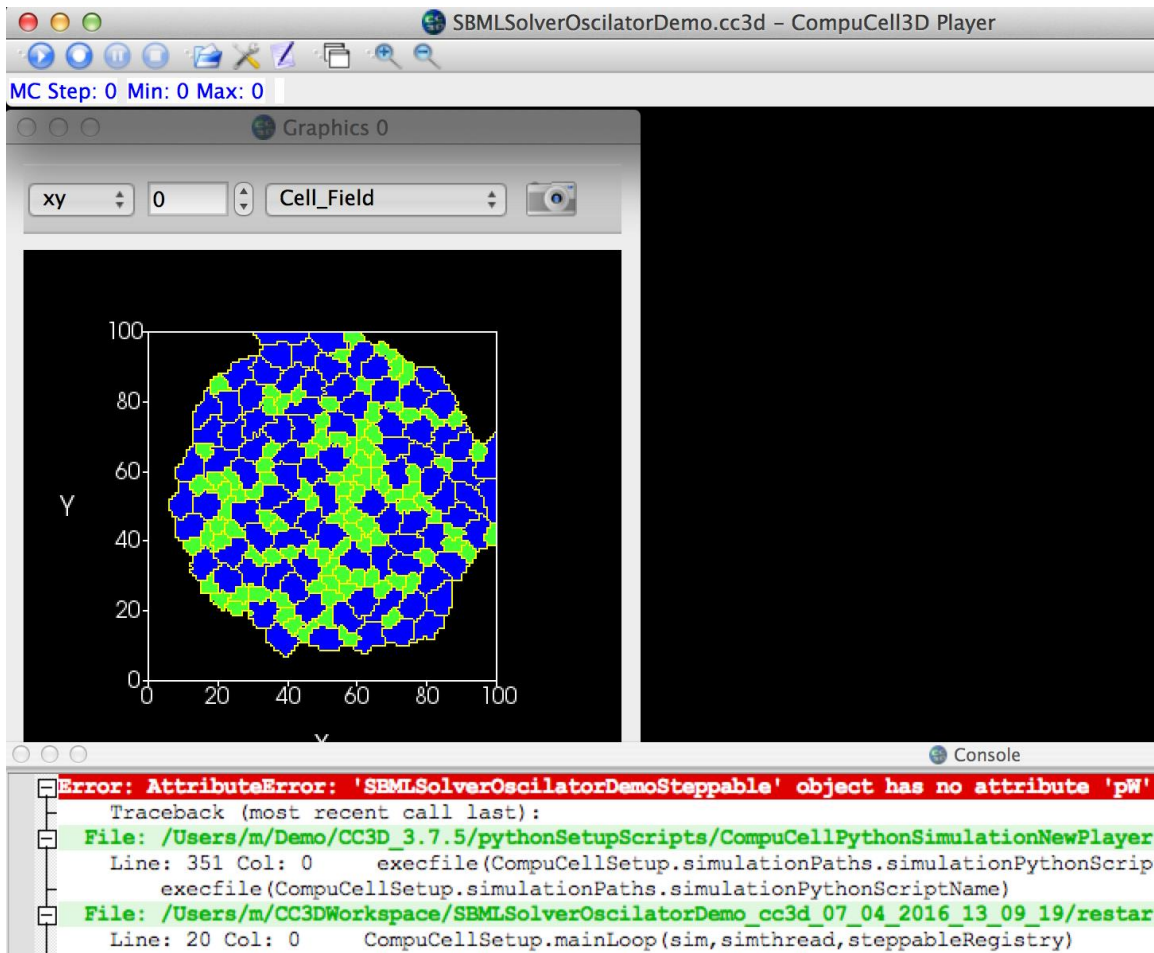Let's see what happens if we run our project form the snapshot saved snapshot. In Player we navigate to `/Users/m/CC3DWorkspace/SBMLSolverOscilatorDemo_cc3d_07_04_2016_13_09_19/restart_00300` and open `SBMLSolverOscillatorDemo.cc3d`:

We got an error. What happens is easy to explain. During restart `start` functions of steppables are not being run. Since in our original simulation we create plots inside start function, clearly the plots are not created during restart and since `step` function refers to plot window object the error arises. Take a look at our original code and see if you can follow what I explained here:

```
class SBMLSolverOscilatorDemoSteppable(SteppableBasePy):

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

    def start(self):
        self.pW=self.addNewPlotWindow(_title='S1 concentration',\
        _xAxisTitle='MonteCarlo Step (MCS)',_yAxisTitle='Variables')
        self.pW.addPlot('S1',_style='Dots',_color='red',_size=5)

        # iterating over all cells in simulation
        for cell in self.cellList:
            # you can access/manipulate cell properties here
            cell.targetVolume=25
            cell.lambdaVolume=2.0

        ...

    def step(self,mcs):

        ...

        self.pW.showAllPlots()
        self.timestepSBML()
```

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 reatart initialization)

Here is the code – changes are highlighted using bold code:

```
class SBMLSolverOscilatorDemoSteppable(SteppableBasePy):

    def __init__(self,_simulator,_frequency=10):
        SteppableBasePy.__init__(self,_simulator,_frequency)
        self.pW = None

    def initialize_plots(self):
        if self.pW:
            return

        self.pW=self.addNewPlotWindow(_title='S1 concentration',
        _xAxisTitle='MonteCarlo Step (MCS)',_yAxisTitle='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.cellList:
            # you can access/manipulate cell properties here
            cell.targetVolume=25
            cell.lambdaVolume=2.0

        ...

    def step(self,mcs):

        self.initialize_plots()

        ...

        self.pW.showAllPlots()
        self.timestepSBML()
```

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.

After we make all those changes in our simulation code and rerun it we will get the following:
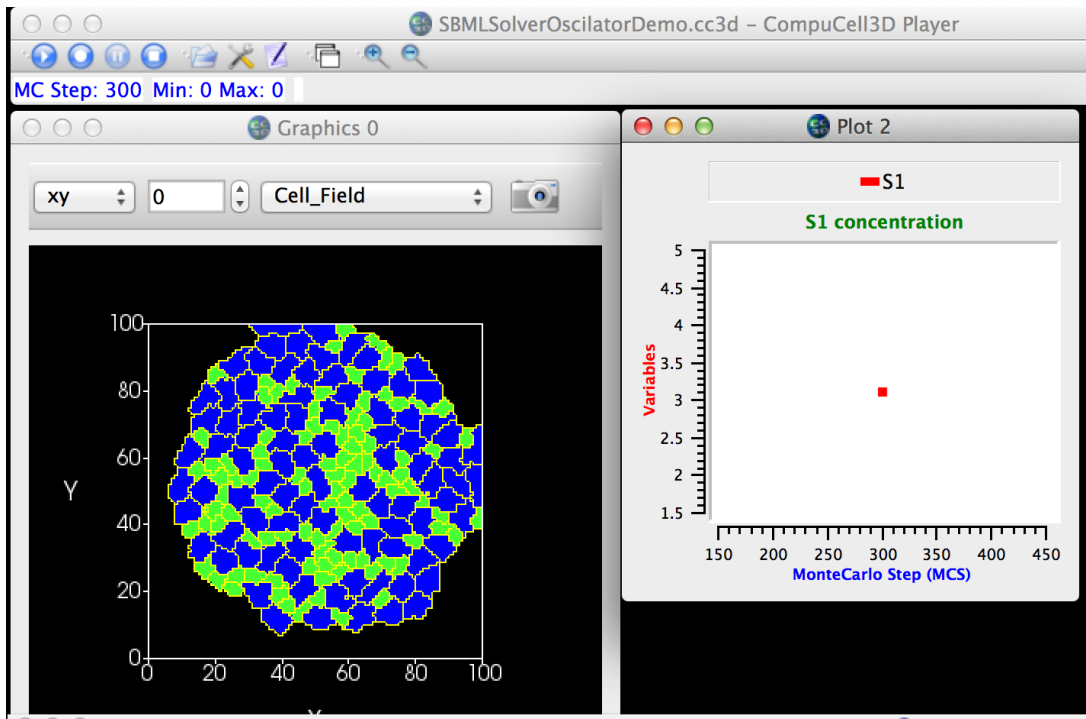
Notice that the cell layout is identical to the one we got at `t=300 MCS` and that out plot restarts at `t=300 MCS`. If we let simulation run for a few more MCS you will see that the `S1` soncentration will start falling as it would if we had continued our original run:
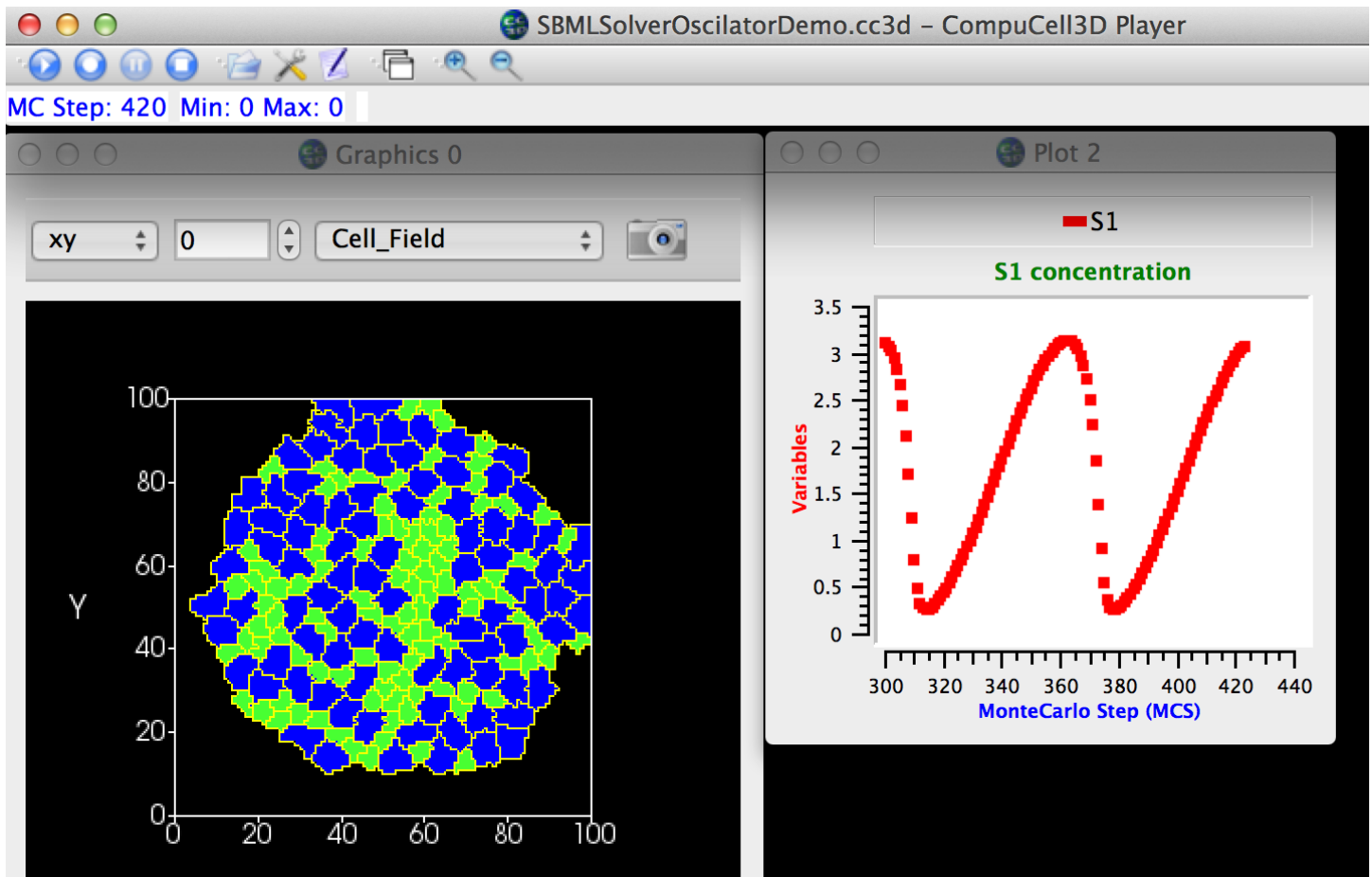
36. Implementing Energy Functions in Python

Important: This feature was implemented as a demo and should not be use in the "production" simulations. Energy functions implemented in Python are much slower than C++ counterparts. 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 …`

CompuCell3D allows users to develop energy functions and lattice monitors in Python. However, we, recommend that if you do need to write such module, you do it in C++. With parallel version of CC3D it makes little sense to build Python moduleswhich are called serially. Even if we could call them in a truly parallel fashion they still would be a big performance bottleneck. For completeness we provide brief description of how to do it. Feel free to skip this section though. In practice modules presented here are almost never used.
First let's take a look how to develop an energy function that calculates a change in volume energy. We will use example from `examples_PythonTutorial\cellsort_2D_with_py_plugin`. In the XML file we make sure that instead of calling `Volume` energy plugin we call:

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

`VolumeTracker` module tracks changes in cells' volume but does not calculate any energy.The implementation of energy function will we done in Python:

```python
from PyPlugins import *
class VolumeEnergyFunctionPlugin(EnergyFunctionPy):

    def __init__(self,_energyWrapper):
        EnergyFunctionPy.__init__(self)
        self.energyWrapper=_energyWrapper
        self.vt=0.0
        self.lambda_v=0.0

    def setParams(self,_lambda,_targetVolume):
        self.lambda_v=_lambda;
        self.vt=_targetVolume

    def changeEnergy(self):
        energy=0
        newCell=self.energyWrapper.getNewCell()
        oldCell=self.energyWrapper.getOldCell()

        if(newCell):
            energy+=self.lambda_v*(1+2*(newCell.volume-self.vt))
        if(oldCell):
            energy+=self.lambda_v*(1-2*(oldCell.volume-self.vt))
        return energy
```

The most important here is `changeEnergy` function. This is where the calculation takes place. Of course when we create the plugin object in the main Python script we will need to make a call to `setParams` function because, that is how we set parameters for this plugin. The `changeEnergy` function calculates the difference in the volume energy for `oldCell` and `newCell`. The volume energy is given by the formula:

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

Consequently the change in the volume energy for `newCell` (the one whose volume will increase due to pixel-copy) is:

$$\Delta E_{newCell} = \lambda(V_{newCell} + 1 - V_{target})^2 - \lambda(V_{newCell} - V_{target})^2 = \lambda(1 + 2(V_{newCell} - V_{target}))$$

for the old cell (the one whose volume will decrease after pixel-copy) the corresponding formula is:

$$\Delta E_{oldCell} = \lambda(V_{oldCell} - 1 - V_{target})^2 - \lambda(V_{oldCell} - V_{target})^2 = \lambda(1 - 2(V_{oldCell} - V_{target}))$$

And overall change of energy is:

$$\Delta E = \Delta E_{oldCell} + \Delta E_{newCell}$$

So as you can see this `changeEnergy` function just implements the formulas that we have just described. notice that sometimes `oldCell` or `newCell` might be a medium cells so that's why we are doing checks for cell being non-null to avoid segmentation faults.:

```
        newCell=self.energyWrapper.getNewCell()
        oldCell=self.energyWrapper.getOldCell()
        if(newCell):
```

Notice also that references to `newCell` and `oldCell` are accessible through `energyWrapper` object. This is a C++ object that stores pointers to `oldCell` and `newCell` every pixel-copy attempt. It also stores Point3D object that contains coordinates of the lattice location at which a given pixel-copy attempt takes place.
Now if you look into `cellsort_2D_with_py_plugin.py` you will see how we use Python plugins in the simulation:

```
import CompuCellSetup

sim,simthread = CompuCellSetup.getCoreSimulationObjects()


import CompuCell #notice importing CompuCell to main script has to be done after call to
getCoreSimulationObjects()

#Create extra player fields here or add attributes or plugins
energyFunctionRegistry=CompuCellSetup.getEnergyFunctionRegistry(sim)

from cellsort_2D_plugins_with_py_plugin import VolumeEnergyFunctionPlugin
volumeEnergy=VolumeEnergyFunctionPlugin(energyFunctionRegistry)
volumeEnergy.setParams(2.0,25.0)

energyFunctionRegistry.registerPyEnergyFunction(volumeEnergy)

CompuCellSetup.initializeSimulationObjects(sim,simthread)

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

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

After a call to `getCoreSimulationObjects()` we create special object called energyFunctionRegistry that is responsible for calling Python plugins that calculate energy every spin flip attempt. Then we create volume

energy plugin that we have just developed and initialize its parameters. Subsequently we register the plugin with `EenergyFunctionRegistry`:

```
energyFunctionRegistry.registerPyEnergyFunction(volumeEnergy)
```

Let's run our simulation now. As you may have noticed the use of this simple plugin slowed down CompuCell3D more than 10 times. So clearly energy functions is not what you should be implementing in Python too often.

37. Appendix A

In this appendix we present alphabetical list of member functions and objects of the SteppableBasePy class from which all steppables should inherit:

`addFreeFloatingSBML` – adds free floating SBML solver object to the simulation

`addNewPlotWindow` - adds new plot windows to the Player display

`addSBMLToCell` – attaches SBML solver object to individual cell

`addSBMLToCellIds` – attaches SBML solver object to individual cells with specified ids

`addSBMLToCellTypes` – attaches SBML solver object to cells with specified types

`adhesionFlexPlugin` - a reference to C++ AdhesionFlexPlugin object. None if plugin not used.

`areCellsDifferent` - function determining if two cell objects are indeed different ojects

`attemptFetchingCellById` – fetches cell from cell inventory with specified id. Returns None if cell cannot be found.

`boundaryMonitorPlugin` - a reference to C++ BoundaryMonitorPlugin object. None if plugin not used

`boundaryPixelTrackerPlugin` - a reference to C++ BoundaryPixelTrackerPlugin object. None if plugin not used

`buildWall` – builds wall of cells (They have to be of cell type which has Freeze attribute set in the Cell Type Plugin) around the lattice

`cellField` – reference to cell field.

`cellList` – cell list. Allows iteration over all cells in the simulation

`cellListByType` – 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

`changeNumberOfWorkNodes` – function that allows changing number of worknodes use dby the simulation

`checkIfInTheLattice` – 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.

`cleaverMeshDumper` - a reference to C++ CleaverMeshDumper object. None if module not used. Experimental

`cloneAttributes` – copies all attributes from source cell to target cell. Typically used in mitosis. Allows specification of attributes that should not be copied.

`cloneParent2Child` – used in mitosis plugin. Copies all parent cell attributes to the child cell.

`cloneClusterAttributes` – 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

`cloneParentCluster2ChildCluster` - used in mitosis with compartmentalized cells. Copies all attributes from cell in a parent cluster to corresponging cell in the child cluster

`clusterInventory` – reference to C++ that serves as inventory of clusters

`clusterList` – 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.

`compilerExeFile` – name of C compiler used by SBML Solver.

`compilerSupportPath` – path to C compiler working directory - used by SBML Solver

`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.

`copySBMLs` – function that copies SBML Solver objects from one cell to another

`createNewCell` – functionfor creating new CC3D cell

`createScalarFieldCellLevelPy` – function creating cell-level scalar field for Player visualization.

`createScalarFieldPy` – function creating pixel-based scalar field for Player visualization.

`createVectorFieldCellLevelPy` – function creating cell-level vector field for Player visualization.

`createVectorFieldPy` -– function creating pixel-based vector field for Player visualization.

`deleteCell` – function deleting cell.

`deleteFreeFloatingSBML` - function deleting free floarting SBML Solver object with a given name.

`deleteSBMLFromCell` - function deleting SBML Solver object with a given name from individual cell.

`deleteSBMLFromCellIds` - function deleting SBML Solver object with a given name from individual cells with specified ids.

`deleteSBMLFromCellTypes` - function deleting SBML Solver object with a given name from individual cells of specified types.

`destroyWall` – 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

`distanceBetweenCells` - convenience function calculating distance between COMs of two cells.

`distanceVector` - convenience function calculating distance vector beween two 3D points

`distanceVectorBetweenCells` - convenience function calculating distance vector beween COMs of two cells

`elasticityTrackerPlugin` - a reference to C++ ElasticityTrackerPlugin object. None if module not used.

`everyPixel` – Python-iterable object returning tuples (x,y,z) for every pixel in the simulation. Allows iteration with user-defined steps between pixels.

`everyPixelWithSteps` – internal function used by everyPixel.

`extraInit` – internal function

`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.

`getAnchorFocalPointPlasticityDataList`

`getCellBoundaryPixelList` – function returning list of boundary pixels

`getCellByIds` – function that attemts fetching cell by cell id and cluste id. See also attemptFetchingCellById

`getCellNeighborDataList` – function returning Python-iterable list of tuples (neighbor, common surface area) that allows iteration over cell neighbors

`getCellNeighbors` – function returning Python-iterable list of NeighborSurfaceData objects. Slightly obsolete

`getCellPixelList` – function returning Python-iterable list of pixels belonging to a given cell

`getClusterCells` – function returning Python iterable list of cells in a cluster with a given cluster id.

`getConcentrationField` - function returning reference to a concentration field with a given name. Returns None if field not found

`getCopyOfCellBoundaryPixels` – function creating and returning new Python-iterable list of cell pixels of all pixels belonging to a boundary of a given cell.

`getCopyOfCellPixels` – function creating and returning new Python-iterable list of cell pixels of all pixels belonging to a given cell.

`getDictionaryAttribute` - function returning Python-dictionary attached to each cell.

`getElasticityDataList` – function returning Python-iterable list of C++ ElasticityData objects. Used in conjunction with ElasticityPlugin

`getFieldSecretor` – function returning Secretor object that allows implementation of secretion in a cell-by-cell fashion.

`getFocalPointPlasticityDataList` – function returning Python-iterable list of C++ FocalPointPlasticityData objects. Used in conjunction with FocalPointPlasticityPlugin.

`getInternalFocalPointPlasticityDataList` – function returning Python-iterable list of C++ InternalFocalPointPlasticityData objects. Used in conjunction with FocalPointPlasticityPlugin.

`getPixelNeighborsBasedOnDistance` – function returning Python-iterable list of pixels which are withing given distance of the specified pixel

`getPixelNeighborsBasedOnNeighborOrder` – function returning Python-iterable list of pixels which are withing given neighbor order of the specified pixel

`getPlasticityDataList` – function returning Python-iterable list of C++ tPlasticityData objects. Used in conjunction with PlasticityPlugin. Deprecated

`getSBMLSimulator` – gets RoadRunner object

`getSBMLState` – gets Python-dictionary describing state of the SBML model.

`getSBMLValue` – gets numerical value of the SBML model parameter

`getSteppableByClassName` – fetches steppable object using class name

`getSteppableListByClassName` – fetches list of steppable objects using class name.

`init` – internal use only

`invariantDistance` – calculates invariant distance between two 3D points

`invariantDistanceBetweenCells` - calculates invariant distance between COMs of two cells.

`invariantDistanceVector` - calculates invariant distance vector between two 3D points

`invariantDistanceVectorBetweenCells` - calculates invariant distance vector between COMs of two cells.

`invariantDistanceVectorInteger` - 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.

`moveCell` – moves cell by a specified shift vector

`neighborTrackerPlugin` - a reference to C++ NeighborTrackerPlugin object. None if module not used.

`newCell` – creates new cell of the user specified type

`normalizePath` - ensures that file path obeys rules of current operating system

`numpyToPoint3D` – converts numpy vector to Point3D object

`openFileInSimulationOutputDirectory` – opens file using use specified file open mode in the 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.

`point3DToNumpy` – 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

`reassignClusterId` – reassignes cluster id. **Notice:** you cannot type cell.clusterId=20. This will corrupt cell inventory. Use reassignClusterId instead

`removeAttribute` – internal use

`resizeAndShiftLattice` - 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.

`setFrequency` - sets steppable call frequency (equivalend to self.frequency=FREQ_VALUE)

`setMaxMCS` – sets maximum MCS. Used to increase or decrease number of MCS that simulation shuold complete.

`setSBMLState` - used to pass dictionary of values of SBML variables

`setSBMLValue` – sets single SBML variable with a given name

`setStepSizeForCell` – sets integration step for a given SBML Solver object in a specified cell

`setStepSizeForCellIds` – sets integration step for a given SBML Solver object in cells of specified ids

`setStepSizeForCellTypes` – sets integration step for a given SBML Solver object in cells of specified types

`setStepSizeForFreeFloatingSBML` – sets integration step for a given free floating SBML Solver object

`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

`stopSimulation` – function used to stop simulation immediately

`tempDirPath` – temporaty directory path used by SBML solver

`timestepCellSBML` - function carrying out integration of all SBML models in the SBML Solver objects belonging to cells.

`timestepFreeFloatingSBML` - function carrying out integration of all SBML models in the free floating SBML Solver objects

`timestepSBML` - function carrying out integration of all SBML models in all SBML Solver objects

`typeIdTypeNameDict` – internal use only

`vectorNorm` – function calculating norm of a vector

`volumeTrackerPlugin` - a reference to C++ VolumeTrackerPlugin object. None if module not used.

Additionally MitosisPlugin base has these functions:

`childCell` – a reference to a cell object that has jus been created as a result of mitosis

`parentCell` – a reference to a cell object that underwent mitisos. After mitosis this cell object will have smalle volume

`setParentChildPositionFlag` - 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.

`getParentChildPositionFlag` – returns current value of parentChildPositionFlag.

`divideCellRandomOrientation` - divides parent cell using randomly chosen cleavage plane.

`divideCellOrientationVectorBased` - divides parent cell using cleavage plane perpendicular to a given vector.

`divideCellAlongMajorAxis` - divides parent cell using cleavage plane along major axis

`divideCellAlongMinorAxis` - divides parent cell using cleavage plane along minor axis

`updateAttributes` – function called immediately after each mitosis event.Users provide implementation of this function.

38. 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 intrtia tensor. Needs MomentOfInertia Plugin

`iXY` – xy component of intrtia tensor. Needs MomentOfInertia Plugin

`iXZ` – xz component of intrtia tensor. Needs MomentOfInertia Plugin

`iYY` – yy component of intrtia tensor. Needs MomentOfInertia Plugin

`iYZ` – yz component of intrtia tensor. Needs MomentOfInertia Plugin

`iZZ` – zz component of intrtia 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` – instantenous 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` - instantenous cell volume. Needs Volumertacker 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