CompuCell3D Python Scripting manual

Version 3.6.0

Maciej H. Swat, Julio Belmonte, Benjamin L. Zaitlen

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

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.

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.

I case you wonder if using Python degrades perfoemance 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 80 % of times Python will make your life way easier and will not impose ANY noticeable degradation in the performance. Whatis more important is that 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. No time is wasted for dealing with compilation/installation of C/C++ modules and Python script you will write will run on any operating system (Mac, Windows, Linux).

How to use Python in CompuCell3D

Probably the best way to learn Python scripting in CC3D is (besides reading this manual) to study examples in cluded in CC3D package. Those are very simple simulations that demonstrate usage of most important CC3D features.
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, leaveing 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 there is good news too. You may write Python function which encodes the content of the CC3DML file using straightforward Python syntax. We will explain how to do this later. For now, let's assume that we will still load CC3DML along with main Python script.

Let's look at a simple example first:
File: examples_PythonTutorial\cellsort_2D\Simulation\cellsort_2D.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()

#Create extra player fields here or add attributes

CompuCellSetup.initializeSimulationObjects(sim,simthread)

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

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

File: examples_PythonTutorial\cellsort_2D\Simulation\cellsort_2D.xml

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

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

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

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

 <Plugin Name="CenterOfMass"/>

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

</CompuCell3D>
```

File: examples_PythonTutorial\cellsort_2D\cellsort_2D.cc3d

```
<Simulation version="3.5.1">
    <XMLScript>Simulation/cellsort_2D.xml</XMLScript>
    <PythonScript>Simulation/cellsort_2D.py</PythonScript>
</Simulation>
```

What is going on here? Two xml files and one Python script - what goes where?
When you want to run this simulation you open up .cc3d file - examples_PythonTutorial\cellsort_2D\cellsort_2D.cc3d – which, as you can tell, stores names of the files files that actually implement the simulation, and most importantly the .cc3d file tells you that both `cellsort_2D.xml` and `cellsort_2D.xml` 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 cellsort_2D.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. For more discussion on this topic please see CompuCell Manual.

Let's analyze main Python script now.
The `import sys` line provides access to standard functions and variables needed to manipulate the Python runtime environment. The next two lines,

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

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

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

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

Next, we create and initialize the core CompuCell3D modules:

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

We then create a steppable *registry* (a Python *container* that stores steppables, *i.e.*, a list of all steppables that the Python code can access) and pass it to the function that runs the simulation:

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

Once we open .cc3d file in CompuCell3D the simulation begins to run. As you can see looks exactly as a basic cell-sorting simulation that you typically would run using just cellsort_2D.xml configuration file. We hope however that at this point we don't have to remind you why we went to extra trouble of bringing Python to the picture.

Our next task is to extend above simulation and give you a preview of what you can actually accomplish in Python. Now that you know how CompuCell3D simulations are structured we will only show files which are relevant to the discussion.

Looping over cell list

The next example is based on cellsort_2D_info_printer example. We will show you how to extract soma basic information from CC3D while simulation is running.

The module we will develop here is called in CompuCell3D terminology a steppable. Those are modules which are called by CompuCell3D engine after every MCS or with user specified frequency.

A little bit of theory first. A Python steppable typically inherits SteppableBasePy class which provides a lot of functionality and hides a lot of boiler-plate code so that scripting is easier.

If you don't know what inheritance is here is a simple example: assume you have a class, call it BaseClass with 3 functions functionA, function and functionC:

```
class BaseClass:
    def __init__(self):
        print "Constructor"

    def functionA(self):
        print "Hello I am function A"

    def functionB(self):
        print "Hello I am function B"

    def functionC(self):
        print "Hello I am function C"
```

You also want to have another class called UsefulSmartClass which has exactly the same 3 functions and one more function called (you guessed it right) functionD. You have two choices

1) Copy and paste in which case you end up with the following class

```
class UsefulButNotSmartClass:
    def __init__(self):
        print "Constructor"

    def functionA(self):
        print "Hello I am function A"

    def functionB(self):
        print "Hello I am function B"

    def functionC(self):
        print "Hello I am function C"

    def functionD(self):
        print "Hello I am function D"
```

2) Use inheritance:

```
class UsefulSmartClass(BaseClass):
    def __init__(self):
        BaseClass.__init__(self)


    def functionD(self):
        print "Hello I am function D"
```

When you use the two classes (UsefulSmart and UsefulButNotSmartClass) the results will be the same. They both have 4 functions. Without going into too much details inheritance is a way of importing content of one class into another. Yes, it saves you typing and hides code. There is way more than that to inheritance but for our purposes let's focus on those two aspects of inheritance – hiding code in the base class importing it to the child class (in our case UsefulSmartClass).

As we have already mentioined our CompuCell3D steppables will inherit SteppableBasePy class. The code for this class is stored in PythonSetupScripts\PySteppables.py in your CC3D installation directory if you are interested what's inside it (as you start developing more complex simulations you will almost certainly look into this code anyway). Detailed information on functions/members of SteppableBasePy is presented in the Appendix A. Let's take a look at our steppable:

File: examples_PythonTutorial\cellsort_2D_info_printer\Simulation\ cellsort_2D_steppables_info_printer.py

```
from PySteppables import *
import CompuCell
import sys


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


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


    def step(self,mcs):
        print "This function is called every 10 MCS"
        for cell in self.cellList:
            print "CELL ID=",cell.id, " CELL TYPE=",cell.type
        if not ( mcs % 20 ):
            counter=0
            for cell in self.cellListByType(1):
```

```
                print "BY TYPE CELL ID=",cell.id,\
                " CELL TYPE=",cell.type,

                counter+=1

            for cell in self.cellListByType(2):

                print "BY TYPE CELL ID=",cell.id, " CELL \
                TYPE=",cell.type

                counter+=1

    def finish(self):

        print "Called at the end of simulation"
```

As you can see 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. So in our example we redefine all 3 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) .

As you can already tell all interesting things happen in the `step` function. We are visiting all cells in the simulation and print their cell id and cell type:

```
        for cell in self.cellList:

            print "CELL ID=",cell.id, " CELL TYPE=",cell.type
```

It looks like a bit of magic here. Out of the blue we get `self.cellList` which behaves like Python container with CC3D cell objects. The way it happens is that in the constructor of SteppableBasePy base class we do a lot of initialization andamong other things we initialize class member called `self.cellList` to point to Python object which know how to extract cell inventory infoemation from underlying C++ code. Thus by using inheritance we have gained access to internal C++ structures of CompuCell3D. It is cool, powerful, simple, and dangerous.

Thew only thing that we do inside the `for` loop is to print basic information about each cell – its id and type.

In case you wonder what other cell attributes you can print from Python, use dir(cell) statement to see what attributes are available in each cell:

```
for cell in self.cellList:

    print dir(cell)
```

Now, once we have written a steppable we need to place a reference to it in the main Python script (we are simply extending our original main script template):

```
import sys

from os import environ

from os import getcwd

import string

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



import CompuCellSetup

sim,simthread = CompuCellSetup.getCoreSimulationObjects()



#Create extra player fields here or add attributes

CompuCellSetup.initializeSimulationObjects(sim,simthread)



#Add Python steppables here

steppableRegistry=CompuCellSetup.getSteppableRegistry()



from cellsort_2D_steppables_info_printer import InfoPrinterSteppable

infoPrinterSteppable=InfoPrinterSteppable(_simulator=sim,_frequency=10)

steppableRegistry.registerSteppable(infoPrinterSteppable)



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

We first import `InfoPrinterSteppable` class from `cellsort_2D_steppables_info_printer.py`.

Next we create steppable object (`infoPrinterSteppable`) of class `InfoPrinterSteppable` (in other words we instantiate class `InfoPrinterSteppable`). Finally, and this is essential step we register the newly created **object** with steppableRegistry. The registration is a way of telling CompuCell that the new object we created should be used as a Python steppable in the simulation and its 3 functions - `start`, `step` and `finish` - will be called at appropriate times. Unregistered steppables are simply ignored.

When creating steppable we have passed as first argument of the steppable constructor a reference to simulator object - `_simulator=sim` - (this is main CC3D C++ object) and as a secod argument we `_frequency=10`. _frequency tells CC3D to call `step` function every `_frequency` MCS.

Notice that `infoPrinterSteppable` is a name of the variable whereas `InfoPrinterSteppable` is a name of the class. The difference is small - one starts with capital letter (name of the class) and one with lower case letter (name of the variable). This distinction is important because Python is case-sensitive.

Quite often in your simulation you will want to check on cells of certain types. One way to accomplish could look like this:

```
        for cell in self.cellList:
            if cell.type==2:
               print "CELL ID=",cell.id, " CELL TYPE=",cell.type
```

However you still are looping over all cells and select only those which interest you. A better way is to use cellListByType container which preselects cells of a given type. In the example below only cells of type 2 are selected:

```
            for cell in self.cellListByType(2):
                print "BY TYPE CELL ID=",cell.id, " CELL \
                TYPE=",cell.type
```

Again, all the code that preselects cells is hidden in SteppableBasePy . What you should know though that type selection is done in C++ which makes is much faster than doing it in pure Python – another reason to use `self.cellListByType`.

To conclude this example we will show how you can output information from the simulaton to a file:

```
class InfoPrinterSteppable(SteppableBasePy):
    def __init__(self,_simulator,_frequency=10):
        SteppableBasePy.__init__(self,_simulator,_frequency)
        self.file=open("Output.txt","w")
```

```
    def step(self,mcs):


        for cell in self.cellList:
            self.file.write("Cell id=%d type=%d volume=%d\n"\
            %(cell.id,cell.type,cell.volume))
```

The only thing we have changed here was to first open the file (“Output.txt”) for writing :

```
self.file=open("Output.txt","w")
```

and writing content of the file using:

```
self.file.write("Cell id=%d type=%d volume=%d\n"\
%(cell.id,cell.type,cell.volume))
```

The syntax to to write to a file is as follows:

```
file.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.

The reason we have chosen the example with cell inventory simulation is that this is one of the most frequent tasks with CPM modeling. You usually want to run simulation and then iterate over all the cells and do various tasks. This example gives you a template that you may reuse for your simulations.

Important: In the above example 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

```
CompuCell.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
```

Attaching cell attributes

In this example we will teach you how to attach extra attribute to the cell from the Python level. For example, you may want every cell to have a countdown clock that will be recharged once its value reaches zero.
To accomplish this task we will need a steppable that will manage the clock but we also need a way to attach additional attribute that will serve as a clock. One way to do that would be to add line

```
int clock
```

to Cell.h file and recompile entire package. Cell.h is a C++ header file that defines basic properties of the CompuCell3D cells. This, however, is extremely inefficient solution because in this case you will need to recompile the whole package. And if you want to add another attribute, you will be recompiling again. Most important your simulation will not be portable because it will be only runnable using a particular version of CompuCell3D that has been modified.

A much better approach is to do it from Python level. To be fair, in certain cases adding attribute at C++ level makes sense (that's why we have volume, target volume etc. defined in the C++ code for Cell), however, if you need an attribute for one particular simulation, you better do it in Python.

Let's see how it is done in cellsort_2D_extra_attrib example. In the cellsort_2D_extra_attrib.py file insert the following after "#Create extra player fields here or add attributes" line:

```
pyAttributeAdder,LISTAdder=CompuCellSetup.attachListToCells(sim)
```

IMPORTANT: Most of the Python templates can be easily generated using Twedit++ - CC3D edition. Simul;y open new project Wizard and there you will have an option to autogenerate templates of Python-based simulations.

It is important that you keep `pyAttributeAdder,listAdder` on the left hand side of the equation as it prevents those two objects created inside `attachListToCells(sim)` from being garbage collected.

At this point every time a new cell is created it is going to have a additional list attribute attached to it. This list can be modified from Python level as we will show you in a second.

Here is the body of the new steppable that we have just developed:

```
class ExtraAttributeCellsort(SteppableBasePy):

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

        SteppablePy.__init__(self,_simulator,_frequency)



    def step(self,mcs):

        for cell in self.cellList:

            list_attrib=CompuCell.getPyAttrib(cell)

            print "length=",len(list_attrib)

            list_attrib[0:2]=[cell.id*mcs,cell.id*(mcs-1)]

            print "CELL ID modified=",list_attrib[0],"    ",\
            list_attrib[1]
```

As you can see we declared only step function which is OK as we pointed it out earlier.

Inside `step` function we iterate over inventory of cells, however this time instead of just printing cell information we modify additional cell attribute. Moreover we have also done a nice trick. Namely, when we registered `listAdder` we asked CompuCell3D to attach to very cell as additional attribute a list and this list has one empty element (integer number initialized to zero). By doing assignment:

```
list_attrib[0:2]=[cell.id*mcs,cell.id*(mcs-1)]
```

we replace first two entries of the list with `cell.id*mcs , cell.id*(mcs-1)`.

One interesting thing here is the fact that initially we had just one element in the list but using `list_attrib[0:2]` construct we tell Python to replace first two entries of the list with `cell.id*mcs , cell.id*(mcs-1)` and if the list does not have enough entries they will be added automatically.

We can also write the following:

```
list_attrib[0:3]=[cell.id*mcs,cell.id*(mcs-1),"Another text attribute"]
```

In this case we added third entry to the list and in this case it is a simple text constant. This demonstrates that lists in Python can group various objects and they do not need to be of the same type.

As you can see, this is quite powerful technique, which allows you to store all sorts of Python objects inside a list attrached to each CC3D cell.

You need to be however careful when you work with a simulation in which new cells are created because once a cell has been created during the simulation, the content of the additional attribute is initialized to one-element list (with zero being the only list entry). In such a case you need to to account for that fact and if you use additional attribute you first need to make sure that it has been properly initialized (i.e. usually right after create a cell in e.g. mitosis steppable you should modify the content of the list attached to a cell).

One way to do it would be to check either a size of the list or the content of the first element and make sure that they are not the defaults.

Besides attaching Python lists to each CC3D cell, CompuCell3D allows users to use Python dictionaries as cell attributes. Dictionaries are often more convenient to use than lists especially when you will be trying to store many objects. With lists you will have to memorize index of the object, while dictionaries will allow you to reference object using text labels. Here is a example of the main script using a dictionary instead of a list:

```
import sys

from os import environ

from os import getcwd

import string


```

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



import CompuCellSetup

sim,simthread = CompuCellSetup.getCoreSimulationObjects()

#Create extra player fields here or add attributes

pyAttributeAdder,dictAdder=CompuCellSetup.attachDictionaryToCells(sim)

CompuCellSetup.initializeSimulationObjects(sim,simthread)

#Add Python steppables here

steppableRegistry=CompuCellSetup.getSteppableRegistry()

#here we will add ExtraAttributeCellsort steppable

from cellsort_2D_steppables_extra_attrib import ExtraAttributeCellsort

extraAttributeCellsort=ExtraAttributeCellsort(_simulator=sim,\
_frequency=10)

steppableRegistry.registerSteppable(extraAttributeCellsort)



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

And the steppable showing how to use dictionaries is shown below:

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

    def step(self,mcs):
        for cell in self.cellList:
            dict_attrib=CompuCell.getPyAttrib(cell)
            dict_attrib["AttribID"]=cell.id*mcs
            dict_attrib["AttribID_Previous MCS"]=cell.id*(mcs-1)
            print "dict_attrib=",dict_attrib
```

As you can see, the overall structures of the simulations with list or dictionaries are quite similar. The only difference is in the way you access and store the attributes. Many users find working with dictionaries easier than working with lists because they can refer to items in the dictionary using text label. Additionally if your simulation calls for a list instead of dictionary you can always store a list as an element of the dictionary:

```
            dict_attrib=CompuCell.getPyAttrib(cell)
            dict_attrib["ListObject"]=[cell.id*mcs]
```

In Python, you construct a list by enclosing coma-separated objects in square brackets. For example:

```
list=[a,b,c]
```

If you want to construct empty list you simply type:

```
list=[]
```

Adding Python Objects as attributes

The above examples demonstrated how to attach simple attributes to a cell but what if you want to add a more complex attribute , say a class (strictly speaking an object of a given class). In this case the solution is simple. First define a class then create object of this class and attach it to the cell. However there is on **IMPORTANT** detail here - your class **HAS TO** inherit from object which meant it has to be “new style” python class. Let’s see an example:

```
class CustomClass(object): # notice that we inherit from object

    def __init__(self, _x,y):

        self.x=_x

        self.y=_y

    def calculate(self):

        print “this is a result of calculations: ”,x*y

        return x*y



class ModifyDictAttribute(SteppableBasePy):

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

        SteppableBasePy.__init__(self, _simulator ,_frequency)

    def start(self):

        for cell in self.cellList:

        print " MODIFY ATTRIB CELL ID=",cell.id

        dictionary=CompuCell.getPyAttrib(cell)

        dictionary["customClass"]=CustomClass(cell.id,cell.id+1)
```

```
    def step(self,mcs):

        for cell in self.cellList:

            if not mcs % 20:

                dictionary=CompuCell.getPyAttrib(cell)

                print "CustomClass \
calculations=",dictionary["customClass"].calculate()
```

If you do not inherit from object and try to use user defined class as an attribute you will get segfault whenever cell is destroyed. The take home message here is that whenever using classes as cell attributes always inherit from object to avoid run time errors

Important: CurrentCC3D implementation allows users to attach list or dictionary to a cell but not both in the way presented above. However, as we already mentioned, you may always insert a list into a dictionary or insert dictionary to a list so in practice this easy work-around allows you to actually have both list and dictionary remembering that you need to one extra step to make things work.

Modifying attributes of CellG object

So far, the only attributes of a cell we have been modifying were those that we attached during runtime in the form of list or dictionary. However, CC3D allows users to modify core cell attributes i.e. those which are visible to the C++ portion of the CC3Dcode. 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: while you can freely modify any attribute of the CellG object we have to warn you that modifying some of the attributes may corrupt the simulation or even crash CC3D. We are working on fixing this issue. For now, before trying to modify any CellG object properties, please see manual to check if you are allowed to do so.

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="VolumeLocalFlex"/>
```

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.

To register `TypeSwitcherAndVolumeParamSteppable` in the main Python stript we use the following code:

```
from cellsort_2D_steppables_extra_attrib import
TypeSwitcherAndVolumeParamSteppable
ts=TypeSwitcherAndVolumeParamSteppable(sim,100)
steppableRegistry.registerSteppable(ts)
```

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. By neighbor we mean adjacent cell which has common surface area with the cell in mind. As you will see iteratin g over cell neighbors is as easy as visiting all the cells in the simulation. To enable neighbor tracking you have to include NejghborTracker plugin in the XML or in Python code which replaces XML. For details see examples_PythonTutorial\cellsort_2D_neighbor_tracker example.The actual class where we implement algorithm to visig cell neighbors is shown below:

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

    def step(self,mcs):
        self.cellList=CellList(self.inventory)
        for cell in self.cellList:
            cellNeighborList=self.getCellNeighbors(cell)
            print "*********NEIGHBORS OF CELL WITH ID ",cell.id,
            for neighborSurfaceData in cellNeighborList:
                if neighborSurfaceData.neighborAddress:
                    print \
            "neighbor.id",neighborSurfaceData.neighborAddress.id,\
            " commonSurfaceArea=",neighborSurfaceData.commonSurfaceArea
                else:
                    print \
      "Medium commonSurfaceArea=",neighborSurfaceData.commonSurfaceArea
```

In the outer `for` loop we iterate over all cells. For each cell in the simulation we construct a list of its neighbors using

```
cellNeighborList=self.getCellNeighbors(cell)
```

As you should know by now, the `getCellNeighbors` funcrtion is defined inside `SteppableBasePy` class and is available inside our steppable by means of inheritance.

cellNeighborList is a collection of `neighborSurfaceData` structures which store information about memory address of CellG object (`neighborSurfaceData.neighborAddress`) which is a neighbor of a given cell and common surface area between cell and the neighbor (`neighborSurfaceData.commonSurfaceArea`). Notice that we use mysteriously looking statement:

```
if neighborSurfaceData.neighborAddress:
      …
else:
      …
```

We do this to check if neighbor address has a non-zero value. If neighborAddress is zero it means that our neighbor is a medium cell which in CC3D is implemented as a NULL pointer and as such medium does not have any properties (id, type etc). Thus trying to refer to e.g. id pf a medium would crash the simulation. Therefore a solution is to check if we have non medium cell and in this case we are free to access its id, type etc.. and in the case of Medium cell we only print a common surface area between cell and its neighbor (Medium).

Accesing concentration fields managed of PDE solvers

In this example we will show you how you may extract and modify values of the concentration field which come from CC3D PDE solvers. We will use diffusion_2D.xml which is a simple simulation where you only solve diffusion equation. To make thing simple we will not use any cells and we will wsolve diffusion equation for one field - FGF. The actual code for this simulation is show in examples_PythonTutorial\diffusion.

The code of the steppable is shown below:

```
class ConcentrationFieldDumperSteppable(SteppableBasePy):

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

        SteppableBasePy.__init__(self,_simulator,_frequency)

    def step(self,mcs):

        fileName="FGF_"+str(mcs)+".dat"

        field=CompuCell.getConcentrationField(self.simulator,"FGF")

        pt=CompuCell.Point3D()

        if field:

            try:

                import CompuCellSetup
```

```
                fileHandle,fullFileName=CompuCellSetup.\
                openFileInSimulationOutputDirectory(fileName,"w")

            except IOError:

                print "Could not open file ", fileName

            for i in xrange(self.dim.x):

                for j in xrange(self.dim.y):

                    for k in xrange(self.dim.z):

                        pt.x=i

                        pt.y=j

                        pt.z=k

                        fileHandle.\
                        write("%d\t%d\t%d\t%f\n"%\
                        (pt.x,pt.y,pt.z,field.get(pt)))

            fileHandle.close()
```

.

The highlights of the steppable are:

1) `self.dim` variable is a part of `SteppableBasePy` and it contains information about lattice dimensions.
2) When opening a file for writing it is best to use `CompuCellSetup.openFileInSimulationOutputDirectory` convenience function. It will create new file in the output directory or, if you are not saving screenshots (see Configuration options in the Player) it will use user specified path (relative to CC3D installation directory) to create new file. You may also specify absolute path in thich case it will create new file according to your absolute path.
3) We access the field from CC3D PDe solvers using `CompuCell.getConcentrationField(self.simulator,"FGF")` Here we accessed field called FGF which matches tha name defined in the XML.
4) To access value of the field at the specified location we use:

   As you can see we specify the location using `pt=CompuCell.Point3D().pt` has 3 components `x,y,z` and they denote lattice coordinates. We use `CompuCell.Point3D()` to access or modify most of CC3D fields (cell field, Concentration fields from PDE solvers etc…).
5) We use familiar syntax to write to the file. We close the file after we are done with writing `fileHandle.close()`.
6) To modify value of the PDE solver field we use the following examplesyntax: `field.set(pt,1200)`.

Here we have stored concentration of 1200 inside at the location specified by pt.

Adding and managing extra fields for visualization purposes

Quite often in your simulation you will want to label cells using scalar field or vector fields. In other words you will want to create 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

It is best to learn how to use those four kinds of fields by studying simple example in examples_PythonTutorial\ExtraFields. In the main script we tell CC3D to create Python-accessible additional fields:

```
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()


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




CompuCellSetup.initializeSimulationObjects(sim,simthread)

#Create extra player fields
dim=sim.getPotts().getCellFieldG().getDim()
extraPlayerField=simthread.createFloatFieldPy(dim,"ExtraField")
idField=simthread.createScalarFieldCellLevelPy("IdField")

vectorField=simthread.createVectorFieldPy(dim,"VectorField")

vectorCellLevelField=simthread.createVectorFieldCellLevelPy("VectorFiel
dCellLevel")
… # steppables registration and main loop call
```

It is quite important to put field registration in the appropriate place in the code – right after the line.

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

If you forget where to put the registration of the fields Twedit++ will autogenerate template script for you with extra field registration in the correct places.

The reminder of the main scrip looks as below:

```
#Add Python steppables here
from PySteppablesExamples import SteppableRegistry
steppableRegistry=SteppableRegistry()

from ExtraFields_steppables import ExtraFieldVisualizationSteppable
extraFieldVisualizationSteppable=ExtraFieldVisualizationSteppable(_simu
lator=sim,_frequency=10)
extraFieldVisualizationSteppable.setScalarField(extraPlayerField)
steppableRegistry.registerSteppable(extraFieldVisualizationSteppable)

from ExtraFields_steppables import IdFieldVisualizationSteppable
idFieldVisualizationSteppable=IdFieldVisualizationSteppable(_simulator=
sim,_frequency=10)
idFieldVisualizationSteppable.setScalarField(idField)
steppableRegistry.registerSteppable(idFieldVisualizationSteppable)

from ExtraFields_steppables import VectorFieldVisualizationSteppable
vectorFieldVisualizationSteppable=VectorFieldVisualizationSteppable(_si
mulator=sim,_frequency=10)
vectorFieldVisualizationSteppable.setVectorField(vectorField)
steppableRegistry.registerSteppable(vectorFieldVisualizationSteppable)

from ExtraFields_steppables import
VectorFieldCellLevelVisualizationSteppable
vectorFieldCellLevelVisualizationSteppable=VectorFieldCellLevelVisualiz
ationSteppable(_simulator=sim,_frequency=10)
vectorFieldCellLevelVisualizationSteppable.setVectorField(vectorCellLev
elField)
steppableRegistry.registerSteppable(vectorFieldCellLevelVisualizationSt
eppable)

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

We simply create 4 simple steppables each of which will manipulate different kind of field.
To see how to manipulate extra fields let's let's see how steppables look like. First we import all functions from PLayerPython module.this module contains the functions necessary to manipulate extra fields.

```
from PlayerPython import *
from math import *
```

Our first steppable will manipulate pixel based scalar field:

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

```
        SteppableBasePy.__init__(self,_simulator,_frequency)

    def setScalarField(self,_field):
        self.scalarField=_field

    def start(self):pass

    def step(self,mcs):
        clearScalarField(self.dim,self.scalarField)
        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):
                        value=x*y
                        fillScalarValue(self.scalarField,x,y,z,value)
                    else:
                        value=sin(x*y)
                        fillScalarValue(self.scalarField,x,y,z,value)
```

As you can see here to assign a value to a pixel we use `fillScalarValue` function with the following arguments (listed in the correct order):

1) reference to the field (`self.scalarField`) – as refurned by `simthread.createFloatFieldPy` function in the main script. This function is passed to our steppable using `setScalarField` function.
2) `x,y,z` coordinates of the pixel at which we want to change the value of the field
3) `value` of the field at pixel `x,y,z` .

Notice also that before entering triple loop we have called function `clearScalarField` . This is technically not necessary but is recommended if you want to avoid leftovers from previous field assignments.

Our next steppable shows how to deal with a cell level based scalar field:

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

    def setScalarField(self,_field):
        self.scalarField=_field

    def step(self,mcs):
        clearScalarValueCellLevel(self.scalarField)
        from random import random
        for cell in self.cellList:

            fillScalarValueCellLevel\
            (self.scalarField,cell,cell.id*random())
```

This is quite similar to previous steppable – except this time you are not manipulating pixel-based scalar field but rather assign scalar to the field and associate this scalar with particular cell. To do that we use `fillScalarValueCellLevel` function this takes the following arguments:

1) Reference to the field (`self.scalarField`) - as refurned by `simthread.createScalarFieldCellLevelPy` function
2) Reference to cell (`cell`)
3) A value you want to associate with cell (`cell.id*random()`) .

Notice that we also called `clearScalarValueCellLevel` function to previous values from the field.

Once we know how to manipulate scalar fields we pretty much can figure out how to deal with vector fields. Let's see how we manipulate pixel based vector field:

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

    def setVectorField(self,_field):
        self.vectorField=_field

    def step(self,mcs):
        maxLength=0
        clearVectorField(self.dim,self.vectorField)
        import math
        for x in xrange(0,self.dim.x,5):
            for y in xrange(0,self.dim.y,5):
                for z in xrange(self.dim.z):

                    pt=CompuCell.Point3D(x,y,z)

                    insertVectorIntoVectorField(\
                    self.vectorField,pt.x, pt.y,pt.z, x, y, z)
```

To insert a vector into the field we use `insertVectorIntoVectorField` function with the following arguments:

1) Reference to the field (`self.vectorField`) – as returned by `simthread.createVectorFieldPy` call in the main script
2) Coordidates of the pixel at which the vesctor is to be inserted
3) `x,y,z` components of the vector

Next steppable shows how to manipulate cell-level vector field

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

    def setVectorField(self,_field):
        self.vectorField=_field

    def step(self,mcs):
        clearVectorCellLevelField(self.vectorField)
        for cell in self.cellList:
            if cell.type==1:
```

```
                insertVectorIntoVectorCellLevelField(\
                self.vectorField,cell, cell.id, cell.id, 0.0)
```

We use `insertVectorIntoVectorCellLevelField` function with the following arguments:

1) reference to the field (`self.vectorField`) – as returned by a call to `simthread.createVectorFieldPy` function.
2) Reference to a cell
3) `x,y,z` componenets of the vector

You should remember that all those4 kinds of field discussed here are for display p[urposes 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.

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 (see example 9). 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 examples_PythonTutorial/steppableBasedMitosis

Let's see how we implement mitosis steppable:

```
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):
        parentCell=self.mitosisSteppable.parentCell
        childCell=self.mitosisSteppable.childCell

        parentCell.targetVolume=25
        childCell.targetVolume=parentCell.targetVolume
        childCell.lambdaVolume=parentCell.lambdaVolume
        if parentCell.type==1:
            childCell.type=2
        else:
            childCell.type=1
```

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 of course call 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 ini 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
```

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 fast if we keep increasing targetvolume and not resetting it the target volume of parent cell would be higher for each consecutive mitosis event. Therefore you sohu8ld always make sure that attributes of parent and child cells are adjusted properly in the `updateAttribute` function.

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 VolumeLocalFlex plugin instead of “regular” Volume plugin with global parameters.

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 concentratoiin 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)
```

Analogously proceed for remaining components of the centroid. Additionally , make sure you include CenterOfMass plugin in the XML or the above calls will return 0’s.

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 dividing line. What you may observe then that if you reassign cell type of 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, in the `updateAttributes` function, you may randomly “swap” child cell with parent cell. To be precise you should not simply swap references to cells but rather swap cell types as well as swap attributes:

```
    def updateAttributes(self):
```

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

        tempType=parentCell.type
        parentCell.type=childCell.type
        childCell.type=tempType
        # reassign parameters for child and parent
```

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):
```

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

        compartmentListChild\
        =self.inventory.getClusterCells(childCell.clusterId)
        compartmentListParent\
        =self.inventory.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
```

The steppable is quite simulat 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 due to functionality built into CellInventory (self.inventory):

```
        compartmentListChild\
        =self.inventory.getClusterCells(childCell.clusterId)
```

The call above returns a list of cells in a cluster with `clusterID` specified by `childCell.clusterId`. In the subsequent `for` loop we iterate over list of cells in the parent and child clusters and assign appropriate values of volume constrain parameters. Notice that because `compartmentListChild` is indexable (ie. we can access directly any element of the list provided our index is not out of bounds) we can in a single loop visit compartments of paraent and child clusters:

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

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

Notice that no where 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.

Implementing Energy Functions in Python

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 modules which 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:

```
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.

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)
```

We will introduce Better error checking in the next releases of CC3D to make sure that errors like this one is clearly communicated to the user when it occurs.

Appendix A

In this appendix we present members of the SteppableBasePy class from which all steppables should inherit:

self.simulator - reference to C++ simulator object
self.potts - reference to C++ simulator object
self.dim - x,y,z, dimensions of the lattice
self.inventory - C++ object - inventory of CC3D cells
self.clusterInventory - - C++ object - inventory of clusters

self.cellList - Python-iterable list of cells
self.cellListByType - Python-iterable list of cells of a given type.
self. cellListByType(2) returns list of all the cells of type 2

self.clusterList- - Python-iterable list of clusters
All CC3D plugins exposed to Python AND registered in XML or Python are automatically being passed to SteppableBasePy at the beginning of the simulation.
self.volumeTrackerPlugin - a reference to C++ VolumeTrackerPlugin object. None if plugin not used

self.neighborTrackerPlugin - a reference to C++ NeighborTrackerPlugin object. None if plugin not used

self.focalPointPlasticityPlugin - a reference to C++ FocalPointPlasticityPlugin object. None if plugin not used

self.chemotaxisPlugin - a reference to C++ ChemotaxisPlugin object. None if plugin not used
self.boundaryPixelTrackerPlugin - a reference to C++ BoundaryPixelTrackerPlugin object. None if plugin not used
self.pixelTrackerPlugin - a reference to C++ PixelTrackerPlugin object. None if plugin not used

self.elasticityTrackerPlugin - a reference to C++ ElasticityTrackerPlugin object. None if plugin not used
self.plasticityTrackerPlugin - a reference to C++ PlasticityTrackerPlugin object. None if plugin not used
self.connectivityLocalFlexPlugin - a reference to C++ ConnectivityLocalFlexPlugin object. None if plugin not used
self.lengthConstraintLocalFlexPlugin - a reference to C++ LengthConstraintLocalFlexPlugin object. None if plugin not used
self.contactLocalFlexPlugin - a reference to C++ ContactLocalFlexPlugin object. None if plugin not used

self.contactLocalProductPlugin - a reference to C++ ContactLocalProductPlugin object. None if plugin not used

self.contactMultiCadPlugin - a reference to C++ ContactMultiCadPlugin object. None if plugin not used

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

self.cellOrientationPlugin - a reference to C++ CellOrientationPlugin object. None if plugin not used

self.polarizationVectorPlugin - a reference to C++ PolarizationVectorPlugin object. None if plugin not used

self.momentOfInertiaPlugin - a reference to C++ MomentOfInertiaPlugin object. None if plugin not used

self.secretionPlugin - a reference to C++ SecretionPlugin object. None if plugin not used

All the functions will return None if for any reason object requested cannot be found. Please check if returned object is different than None object before using the object in your simulation to avoid simulation crash.

self.getClusterCells(_clusterId) - function returning Python iterable list of cells in the cluster with cluster id = _clusterId.

self.getCellNeighbors(_cell) - function returning Python-iterable list of neighbors of cell = _cell

self.getFocalPointPlasticityDataList(_cell) - function returning Python-iterable list of FocalPointPlasticityData of cell=_cell. For details of FocalPointPlasticityData see CC3D manual.

self.getInternalFocalPointPlasticityDataList(_cell) - function returning Python-iterable list of InternalFocalPointPlasticityData of cell=_cell. For details of InternalFocalPointPlasticityData see CC3D manual.

self.getCellBoundaryPixelList(_cell) - function returning Python-iterable list of pixels belonging to the boundary of cell = _cell

self.getCellPixelList(_cell) - function returning Python-iterable list of pixels of cell = _cell

self.getElasticityDataList(_cell) - function returning Python-iterable list of ElasticityData of cell=_cell. For details of ElasticityData see CC3D manual.

self.getPlasticityDataList(_cell) - function returning Python-iterable list of PlasticityData of cell=_cell. For details of PlasticityData see CC3D manual.

self.getFieldSecretor(_fieldName) - returns Secretor object used to secrete chemicals into the field with name = _fieldName. For details see CC3D manual

self.cleanDeadCells() - function which has to be called from Python if you decide to remove a cell directly (i.e. by manually overwriting cell's pixel with pixels of ther cells including Medium) from Python.

self.deleteCell(_cell) - function deleting cell = _cell by overwriting cell pixel with Medium pixel. You have to call self.cleanDeadCells() after you delete cell.

self.createNewCell(_type,_pt,_xSize,_ySize,_zSize=1) - a function creating a cuboidal (rectangular in 2D) cell ot type=type at the location specified by lattice point _pt and of sizes specified by _xSize,_ySize, _zSize. _zSize is by default 1.

self.moveCell(_cell, _shiftVector) - function shifting cell = _cell by a vector = _shiftVector.

self.checkIfInTheLattice(_pt) chcecks if a point _pt is in the lattice

In typical application users inherit SteppableBasePy to import all the functionality available in SteppableBasePy.

Notice that MitosisSteppableBase and ClusterMitosisBase all inherit SteppableBasePy.