Scripting CompuCell3D extension modules in Python

Tutorial

V3.2.1

Authors:
Maciej Swat, Christopher Mueller, James Glazier, Julio Belmonte
Last modified:05/01/08

The focus of this manual is to teach you how to use Python scripting language to develop CompuCell3D extension modules. First thing first, we will assume that you have a working knowledge of Python. We do not expect you to be a Python guru, rather we assume that you know how to write simple Python scripts that use functions, classes, dictionaries, lists (or sequences). If, however it will be your first attempt to use Python I strongly recommend that you go over basic Python programming literature. My favorite is a book by David Beazley (creator of SWIG) "Python Essential Reference". In his book he teaches you Python fundamentals in less the 20 pages which saves you plenty of time. And the book is inexpensive either (on Amazon it is less than 30 US $).

Python extensions to CompuCell3D, Python modules and these tutorials have been developed by Maciej Swat and Christopher Mueller and James Glazier.

OK, let's get started.

CompuCell3D scripting

Once Python scripting became available in CompuCell3D it turned CompuCell3D into fully fledged simulation environment with level of flexibility known from Matlab or Mathematica. Of course, I am not saying that CompuCell3D is as sophisticated as those two commercial products but rather, that it gives you the ability to create simulations with much greater degree of flexibility than it was possible with xml model description files.

If you decide to use Python scripting for CompuCell3D you will be running Python script in which you will call C++ and Python modules. Very first question that comes right now to your mind is "What about the performance, scripting languages are slow, aren't they"? The answer to this question could be easily turned into Computer Science paper but to make long story short, we say 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. What matters here 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. Most professional programmers will tell you that scripting languages are crucial to increase your productivity and frankly based, on our experience with Python all we can say is that it is really true. Without further ado. let's begin CompuCell3D/Python tutorials.

CompuCell3D in Python

Python written simulations still need to be accompanied by xml file. Remember that Python is used to extend CompuCell3D, so the way it works is you open up a Player and open up an xml file that contains your simulation description. Now, if you want to use Python extensions for your simulation you go to Python menu and click "Run Python Script". Then you go to "Python->Configure Python

Extensions...” and choose a Python script in which you have coded the simulation. In practice you do need to code entire simulation. What you do is you use a template file and modify it in certain places to get access to your Python-written extension modules. If it sounds too complicated now, do not worry, we will go step-by-step through several examples and this will show you that is is not that difficult.

Before we go any further one more word of explanation: when you use “pure C++” CompuCell3D the flow of the simulation is hard coded in C++, you do not have any access to it. all you can do is to write plugins or steppables. In Python the flow of the simulation is coded in Python and this gives you the ability to do whatever you want with it. So why it might be tempting to mess around with different settings there, I would recommend that unless you fully understand what you are doing you are better off using Python script with simulation description as a template and modify it inly in certain places as it will be shown. Otherwise you might get cryptic error messages and your simulation will crash. Well, just a word of caution.

To run CompuCell3D simulations written in Python you first need to prepare Python script, or I should say copy and modify existing one.

Our first example will be a very simple one. We will run cell sorting and then we will add a module that will print every 10 MCS a list of cell id cell type and cell volume. It is not very complicated module and not a very useful in this particular context, but it will show you how to access from Python level all the cells, how to iterate over the list of cells and extract certain information.

OK, let's take a look at the template used to describe in Python flow of CompuCell3D simulation.

File: *cellsort_2D.py*

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

sys.path.append(environ["PYTHON_MODULE_PATH"])
sys.path.append(getcwd()+"/examples_PythonTutorial")

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)
```

This is a basic template that we will be modifying to customize our simulations. To get the examples working we had to add module search path so that Python knows where to find the modules. Lines:

```
from os import getcwd
```

and

```
sys.path.append(getcwd()+"/examples_PythonTutorial")
```

are used to set additional search path to be the one of directory called “examples_PythonTutorial” (this is where all the examples from this tutorial are stored) of the current directory.

For now try running cellsort_2D.py accompanied by cellsort_2D.xml file to make sure that everything works fine. As you can see the only thing cellsort_2D.py does is to run your simulation described in cellsort_2D.xml file. However it runs the simulation accessing all the CompuCell objects through Python. and this is exactly what we want.

Example 1 – *Loop on Cell's List*

Now let's develop a simple Python steppable that will print every 10 MCS a list of cell id cell type and cell volume.
Important: All the steppables from this tutorial are stored in cellsort_2D_steppables.py file.

To do this open up editor and type the following script:

File: *cellsort_2D_steppables.py*

```
from PySteppables import *
import CompuCell
import sys

class InfoPrinterSteppable(SteppablePy):
   def __init__(self,_simulator,_frequency=10):
      SteppablePy.__init__(self,_frequency)
      self.simulator=_simulator
      self.inventory=self.simulator.getPotts().getCellInventory()
      # this creates an iterable list of cells
      self.cellList=CellList(self.inventory)

   def start(self):
      print "This function is called once before 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," volume=",cell.volume
```

First three lines are boiler plate for creating new steppable – they include necessary Python files. Next you define a class `InfoPrinterSteppable` which inherits from `SteppablePy` class. What `SteppablePy` inheritance does it assures that your steppable has three mandatory functions that steppables should have:

```
start(self)
step(self,_mcs)
finish(self)
```

Those functions are called by a `steppableRegistry` (see cellsort_2D.py) with predefined frequency -

i.e. `start` and `finish` are called once at the beginning and at the end of the simulation and `step` is called every user defined number of MCS.

If you do not define a one of the mandatory functions nothing will happen and the default implementation of those function (empty function) defined in `SteppablePy` will be called.

The most important initializing function of the above class is the constructor. There we set references (or pointers if you prefer) to simulator object from C++ code and we get also a reference to cell inventory also from C++ code.
Now, the `start(self)` function has a trivial body - we print there a simple message.
Notice that we have not defined `finish(self)` function.

Now the most important function that does useful work is `step(self,_mcs)`. We will copy code of the above function and put comments every other line so that you know what is going on.

File: *cellsort_2D_steppables.py*

```
def step(self,mcs):
   print "This function is called every 10 MCS"
   #message printing
   #Looping over all cells
   for cell in self.cellList:
      print "CELL ID=",cell.id, " CELL TYPE=",cell.type," volume=",cell.volume
      #printing cell id, cell type and cell volume
```

We hope you agree that the implementation of this steppable was not too complicated. There were some strange names on the way but you will get used to them and if you forget them there are many examples included with CompuCell3D so you may always look it up. The implementation of the above functionality in C++ would be not much more complicated, however, you would need to compile the module, install it in the appropriate directory and would not be able to make changes as efficiently as when you are using Python.

The `self.inventory` and `self.cellList` objects contain most up-to-date list of cells present in the simulation. The difference between the two is that `self.cellList` wraps the content of `self.inventory` and offers easier way of iterations over all the cells.

I 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:
   dir(cell)
```

Now, once you have written this steppable you need to place a reference to it in the main simulation code. This is how you do it: start with cellsort_2D.py file that we have just studied and add the following lines (put the lines right after these

File: *cellsort_2D.py*

```
#Add Python steppables here
```

```
steppableRegistry=CompuCellSetup.getSteppableRegistry()

from cellsort_2D_steppables import InfoPrinterSteppable
infoPrinterSteppable=InfoPrinterSteppable(_simulator=sim,_frequency=10)
steppableRegistry.registerSteppable(infoPrinterSteppable)
```

Notice here 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.

Full script, now called cellsort_2D_info_printer.py appears below (notice we have added few comments):

File: *cellsort_2D_info_printer.py*

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

sys.path.append(environ["PYTHON_MODULE_PATH"])
sys.path.append(getcwd()+"/examples_PythonTutorial")

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()

# we import newly written steppable into main script
from cellsort_2D_steppables import InfoPrinterSteppable

# this line effectively invokes constructor of our new steppable. Notice that
# we can set frequency for the steppable. Here we duplicate the default
# settings but equally wel you may change it to 20, 30, or whatever you want
# notice that here we have created object  infoPrinterSteppable which is of
# type  InfoPrinterSteppable. It is important that you distinguish objects and
# types
infoPrinterSteppable=InfoPrinterSteppable(_simulator=sim,_frequency=10)

# now we register newly created object  infoPrinterSteppable with steppable
# registry. From now on this steppable will be called automatically with
# the frequency that you requested
steppableRegistry.registerSteppable(infoPrinterSteppable)

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

It really is not too difficult, is it?

Sometimes it is useful to write information not to the screen (as we did above) but to a file. Let's

rewrite above steppable so that the output would go to the file called “OutputFile.txt”:

File: *cellsort_2D_steppables.py*

```
class InfoPrinterSteppable(SteppablePy):
   def __init__(self,_simulator,_frequency=10):
      SteppablePy.__init__(self,_frequency)
      self.simulator=_simulator
      self.inventory=self.simulator.getPotts().getCellInventory()
      # this creates an iterable list of cells
      self.cellList=CellList(self.inventory)
      file=open(“OutputFile.txt”,w)

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

   def step(self,mcs):
      print "This function is called every 10 MCS"
       for cell in self.cellList:
          file.write(“CELL ID=%d CELL TYPE=%d volume=%d\n” %
(cell.id,cell.type,cell.volume))
```

Two steps are necessary to write to a file – opening a file and writing to it. We open a file using `open` statement and write using the following syntax:

```
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 flioating 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=cellField.get(pt)
cell2=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 `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
```

Example 2 – *Add Cell Attributes*

In this example we will teach you how to attach extra attribute to the cell from the Python level. For example you 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. Well, I do not have to tell you by now that this is extremely inefficient way because in this case you will need to recompile the whole package. And if you want to add another attribute, yes, 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.

Way 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 a "casual" attribute, you better do it from Python.

Let's see how it is done. In the cellsort_2D.py file insert the following after "`#Create extra player fields here or add attributes`" line:

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

You may also see how it is done in cellsort_2D_extra_attrib.py file. 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. As you can see we declared only `step` function which is OK as we pointed it out earlier.

File: *cellsort_2D_steppables.py*

```
class ExtraAttributeCellsort(SteppablePy):
   def __init__(self,_simulator,_frequency=10):
      SteppablePy.__init__(self,_frequency)
      self.simulator=_simulator
      self.inventory=self.simulator.getPotts().getCellInventory()
      self.cellList=CellList(self.inventory)

   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 the body of the step function is very similar to the previous steppable that we have developed. We simply 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). Now, what we do in line

```
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. So this is quite powerful technique, you effectively extend one entry list into multiple entry list and this happens in a very simple way at the Python level.

You need to be however careful when you have 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 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. it is different than the default attribute with which a new cell was created).

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. This is how our steppable's `step(self,mcs)` function would look

like with this extra check:

File: *cellsort_2D_steppables.py*

```
def step(self,mcs):
   for cell in self.cellList:
      list_attrib=CompuCell.getPyAttrib(cell)
      if len(list_attrib)==1:
         # here we check if the list has size 1 in which case it means
         # it has not been initialized  - as initialized list would
         # have two entries.
         print "Initializing attribute"
         list_attrib[0:2]=[1,2]
         # we initialize pyAttrib list with two entries 1 and 2
      else:
         # if we get here it means that list has been initialized
         list_attrib[0:2]=[cell.id*mcs,cell.id*(mcs-1)]
         print "CELL ID modified=",list_attrib[0],"   ", list_attrib[1]
```

This example demonstrated how you can add additional attributes to cells during run time and how you access them from Python level. With the features that we have just described your simulations can be taken to the next level in terms of what you can accomplish with CompuCell3D.

Note that in many cases you might be better of using PyDictAdder instead of PyListAdder. PyDictAdder will attach a dictionary (hash table) as an additional attribute of the cell. Here is a fragment of example code (for full source code see ExtraAtrib_player_dictionary.py in the main installation directory):

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

Now the steppable that manipulates extra attribute looks as below (see also PySteppablesExamples.py for full listing):

File: *cellsort_2D_steppables.py*

```
class ModifyDictAttribute(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:
         print " MODIFY ATTRIB CELL ID=",cell.id
         dict_attrib=CompuCell.getPyAttrib(cell)
         dict_attrib['cell.id']=cell.id*4
         print " dict_attrib['cell.id']=", dict_attrib['cell.id']
      print "CHECK ITERATION"
      for cell in self.cellList:
         print " CHECK ATTRIB CELL ID=",cell.id
         dict_attrib=CompuCell.getPyAttrib(cell)
         print " dict_attrib['cell.id']=", dict_attrib['cell.id']

   def step(self,mcs):
      for cell in self.cellList:
         dict_attrib=CompuCell.getPyAttrib(cell)
```

```
        dict_attrib['cell.id']+=1
        if not mcs % 20: # this construct runs the body of the if statement every 20 MCS
           print "cellid=",cell.id
           print "CELL ID modified=", dict_attrib['cell.id']," true id=",cell.id
```

As you can see from the script above the structure of it is essentially the same as the structure of the previous example with PyListAdder. The difference now is that when you access cell attribute (`dict_attrib`) you do it by specifying attribute name not a number of the list element (`dict_attrib['cell.id']+=1`). This usually makes your program easier to read and maintain.

Important: Current implementation allows users to attach list or dictionary to a cell but not both in the way presented above. However, from Python level you may 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.

Example 3

This example is a task for you to figure out what this steppable does.

File: *cellsort_2D_steppables.py*

```
class TypeSwitcherSteppable(SteppablePy):
   def __init__(self,_simulator,_frequency=100):
      SteppablePy.__init__(self,_frequency)
      self.simulator=_simulator
      self.inventory=self.simulator.getPotts().getCellInventory()
      self.cellList=CellList(self.inventory)
   def step(self,mcs):
       for cell in self.cellList:
         if cell.type==1:
            cell.type=2
         elif (cell.type==2):
            cell.type=1
         else:
            print "Unknown type. In cellsort simulation there should”,
            print “ only be two types 1 and 2"
```

Try running this example and see what happens.

Example 4 – Cell's Neighbors

This example is a little bit more complicated. We will show you how to iterate over the list of cell neighbors. So for each cell in the cell inventory we will print a list of cell neighbors. A neighbor is a cell that has non-zero common surface area with a given cell. Since we will be using neighbor tracker plugin we need to declare it in the xml file. Check cellsort_2D_neighbor_tracker.xml file for details. The Python file to be used with the xml file is cellsort_2D_neighbor_tracker.py.

First let's look at how the steppable looks like:

File: *cellsort_2D_steppables.py*

```
class NeighborTrackerPrinterSteppable(SteppablePy):
   def __init__(self,_simulator,_frequency=100):
      SteppablePy.__init__(self,_frequency)
      self.simulator=_simulator
      self.nTrackerPlugin=CompuCell.getNeighborTrackerPlugin()

      self.inventory=self.simulator.getPotts().getCellInventory()
      self.cellList=CellList(self.inventory)

   def start(self):pass

   def step(self,mcs):
      self.cellList=CellList(self.inventory)
      for cell in self.cellList:
         cellNeighborList=CellNeighborListAuto(self.nTrackerPlugin,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
```

As you recognize this is another example of steppable where we iterate over cell inventory. However this time we iterate over the list of neighbors as well. Since it is a new thing let's go over steps necessary to get cell neighbors being accessible from Python. First notice that in the constructor (`__init__`) we initialize a reference to the NeighborTracker plugin:

```
self.nTrackerPlugin=CompuCell.getNeighborTrackerPlugin()
```

This is necessary because this plugins gives us access to data structures that store cell neighbors. The bottom line here is that we need to include this line in the constructor. Now, in step function things get a little bit more complicated (but we assure you, in C++ it would be even more scary). Let us give you some hints how things should be set up.

For each cell we visit we construct a temporary list of cell's neighbors:

```
cellNeighborList=CellNeighborListAuto(self.nTrackerPlugin,cell)
```

In the very next line we walk over all neighbor:

```
for neighborSurfaceData in cellNeighborList:
```

We then check if the neighbor is a Medium (NULL pointer):

```
if neighborSurfaceData.neighborAddress:
```

and if the neighbor cell (`neighborSurfaceData.neighborAddress`) is different than Medium we print its id (`neighborSurfaceData.neighborAddress.id`) and common surface area (`neighborSurfaceData.commonSurfaceArea`) between `cell` (the one from for cell in `self.cellList`) and neighbor cell (`neighborSurfaceData.neighborAddress`).

Example 5 – *Concentration Field*

In this example we will show you how you may extract values of the concentration field from the simulation. We will use diffusion_2D.xml which is a simple simulation where you only solve diffusion equation. There are no cells there, to make things simple. We will also use diffusion_2D.py file which contains actual code for simulation in Python. in the xml file we declare a concentration field FGF.

Let us show how to develop how to access FGF from Python level. We will develop a steppable that will write values of the concentration field to the file.

File: *cellsort_2D_steppables.py*

```
class ConcentrationFieldDumperSteppable(SteppablePy):
   def __init__(self,_simulator,_frequency=1):
      SteppablePy.__init__(self,_frequency)
      self.simulator=_simulator
      self.dim=self.simulator.getPotts().getCellFieldG().getDim()

   def setFieldName(self,_fieldName):
      self.fieldName=_fieldName

   def step(self,mcs):
      fileName=self.fieldName+"_"+str(mcs)+".dat"
      self.outputField(self.fieldName,fileName)

   def outputField(self,_fieldName,_fileName):
      field=CompuCell.getConcentrationField(self.simulator,_fieldName)
      pt=CompuCell.Point3D()
      if field:
         try:
            fileHandle=open(_fileName,"w")
         except IOError:
            print "Could not open file ", _fileName," for writing. Check if you have
necessary permissions"
         print "dim.x=",self.dim.x
         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)))
```

This steppable is not as complicated as the one in Example 4 but, still, we will walk through the code to explain everything.

In the constructor (`__init__`) we set references to simulator object and get dimension of the lattice:

```
self.dim=self.simulator.getPotts().getCellFieldG().getDim()
```

`self.dim` has three components x, y and z (`self.dim.x, self.dim.y, self.dim.z`) each denoting lattice dimension in the three directions.

Next we have a function that allows us to set field name that we wish to dump to the file. This function is invoked in the following way from the diffusion_2D.py file:

```
from cellsort_2D_steppables import ConcentrationFieldDumperSteppable
concentrationFieldDumperSteppable=ConcentrationFieldDumperSteppable\
(_simulator=sim,_frequency=100)
concentrationFieldDumperSteppable.setFieldName("FGF")
steppableRegistry.registerSteppable(concentrationFieldDumperSteppable)
```

So as you can see once you have instantiated `concentrationFieldDumperSteppable` object you set the concentration field name. Pretty straightforward. Now, the `step` function actually outsources all the work to the `outputField` function. In the first two lines of this function we get a reference to the concentration field. This field exists in "C++ space" but is accessible from Python using syntax as shown below. The next line creates a local object Point3D which is a C++ class that stores coordinates of the lattice. Again it is accessible from Python.

```
field=CompuCell.getConcentrationField(self.simulator,_fieldName)
pt=CompuCell.Point3D()
```

The next few lines

```
if field:
    try:
       fileHandle=open(_fileName,"w")
    except IOError:
       print "Could not open file ", \
         _fileName," for writing. Check if you have necessary permissions"
```

are to check if the reference to the field is non-null (which it would be if you typed something like "dgster44345" as a field name) and then we open a file and a reference to file handle is stored in `fileHandle` variable. Notice that we use exceptions to signal any error during file open operation.
The actual write to the file is a triple `for` loop. It is pretty straightforward to follow. Notice that we extract value of the concentration field using `field.get(pt)` construct. That's why we were creating Point3D object earlier in this function.

This example demonstrates how using Python you may extract information from the running CompuCell3D simulation. Needles to say, following this example you may perform complicated analyses of simulation result and write them to the file while simulation is running.

Note again how how our formatting string looks like in the write to file statement:

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

`%d` denotes integer `%t` denotes tabulator character and `%f` denotes floating point number. This will get familiar once you develop your own script that writes to a file.

Example 6 – *Extra Player Field*

In this example we will show you how to add concentration field to the Player from the Python level.

This feature is just a kind of proof of concept thing and the details of implementation may change in the future however it is useful to go over this example to show you that from Python you may also influence what Player is doing and do your own visualizations. We will use diffusion_2D_extra_player_field.py file together with diffusion_2D.xml (unmodified from the previous examples) file.

To insert additional field to the Player we modify diffusion_2D_extra_player_field.py file by inserting the following line (note that we show more lines to show you the exact place where you need to insert the line which adds new concentration field to the Player):

File: *diffusion_2D_extra_player_field.py*

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

#Create extra player fields here or add attributes

dim=sim.getPotts().getCellFieldG().getDim()

# the following line inserts to the Player additional scalar field "ExtraField"
# with dimension given by variable dim
extraPlayerField=simthread.createFloatFieldPy(dim,"ExtraField")
```

Now, you have to be aware that the field we inserted exists only in the Player. This means that C++ part CompuCell3D does not know about its existence. nevertheless this field is controllable from Python level.

Now, let's see how we defined our steppable that will be updating values of the "ExtraField" during the simulation run.

File: *cellsort_2D_steppables.py*

```
from PlayerPython import *
from math import *

class ExtraFieldVisualizationSteppable(SteppablePy):
   def __init__(self,_simulator,_frequency=10):
      SteppablePy.__init__(self,_frequency)
      self.simulator=_simulator
      self.cellFieldG=self.simulator.getPotts().getCellFieldG()
      self.dim=self.cellFieldG.getDim()

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

   def start(self):pass

   def step(self,mcs):
      for x in xrange(self.dim.x):
         for y in xrange(self.dim.y):
            for z in xrange(self.dim.z):
               pt=CompuCell.Point3D(x,y,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 may see this steppable is quite easy to understand. In the constructor (`__init__`) we get references to simulator object and we also extract the dimension of the lattice. `setScalarField` function is used to pass to the steppable a reference to the field that we have just allocated (we will show later). `step` function is actually very simple iteration over all pixels of the `self.scalarField`. As you may see if the number of MCS is divisible by 20 we set value of the `self.scalarField` to a product of x and z coordinates and if it is not divisible by 20 we set it to `sin(x*y)`. Notice that we need to use `fillScalarValue` function to set a value of the Player field. In particular, a construct

```
self.scalarField.set(pt,value)
```

would not work because it is valid for fields in the C++ part of the CompuCell3D and fields available in Player have different interface so the conclusion is that you need to use `fillScalarValue` function. Notice also that in order to get access to `fillScalarValue` and `sin` we need to import `PlayerPython` and `math` Python modules.

Now let's look at the way in which we invoke this steppable from main Python program:

```
from cellsort_2D_steppables import ExtraFieldVisualizationSteppable
extraFieldVisualizationSteppable=ExtraFieldVisualizationSteppable \
(_simulator=sim,_frequency=10)
extraFieldVisualizationSteppable.setScalarField(extraPlayerField)
steppableRegistry.registerSteppable(extraFieldVisualizationSteppable)
```

You can see that the steppable is called every 10 MCS and that in order to properly initialize it we need to pass reference to just allocated Player field

```
extraFieldVisualizationSteppable.setScalarField(extraPlayerField)
```

This example demonstrates that you have basic tools to extend Player visualization capabilities from Python level. We however anticipate that this feature may look somewhat different in the future and should be more user friendly. This is just to let you know that next version of the Player will have much greater possibilities as far as controlling your visualization from Python level.

At this point you should be able to develop steppables in Python so now let's see what else we can accomplish from the Python level. We will now show you how to develop Python plugins it is the objects that will be called either every spin flip or every spin flip attempt (those will be primarily energy functions). Here however you need to be careful as you might end up producing performance bottle neck. Yes, there are some things for which Python is not the best solution and plugins might be a good example for that. However, as we will show, it is still worth learning how to implement plugins in Python. Also, the slight slowdown might be not a high price to pay for much faster set up of the complex simulation using Python.

Example 7 – *Energy Functions*

First let's take a look how to develop an energy function that calculates a change in volume energy. With the knowledge you already have it should be easy to follow this example.

First, lets modify cellsort_2D.xml and save changes as cellsort_2D_py_plugin.xml. The most notable change is that we are not using VolumePlugin but VolumeTracker. The latter plugin tracks changes in volume but does not calculate energy change. In other words it keeps cell volume up to date.
This is the line you need to replace VolumeEnergy entry with:

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

Now let us show the implementation of the the volume plugin in Python. We will store it in cellsort_2D_plugins.py file. The Python simulation file will be cellsort_2D_with_py_plugin.py.

The implementation of the plugin looks as follows:

File: *cellsort_2D_with_py_plugin.py*

```
from PyPlugins import *
# here we had to import some definitions of the base classes for plugins

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=0.0
      if(self.energyWrapper.newCell):
         energy+=self.lambda_v*(1+2*(self.energyWrapper.newCell.volume-self.vt))
      if(self.energyWrapper.oldCell):
         energy+=self.lambda_v*(1-2*(self.energyWrapper.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 Python main 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:

$$E_{volume} = \lambda \left(V - V_{target} \right)^2$$

Consequently the change in the volume energy for newCell (the one whose volume will increase due to spin flip) is:

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

for the old cell (the one whose volume will decrease after spin flip) the corresponding formula is:

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

And overall change of energy is:

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

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.:

```
if(self.energyWrapper.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 spin flip attempt. It also stores Point3D object that contains coordinates of the lattice location at which a given spin flip 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:

File: *cellsort_2D_with_py_plugin.py*

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

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

from cellsort_2D_plugins import VolumeEnergyFunctionPlugin
volumeEnergy=VolumeEnergyFunctionPlugin(energyFunctionRegistry)
volumeEnergy.setParams(2.0,25.0)

energyFunctionRegistry.registerPyEnergyFunction(volumeEnergy)
```

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 `EenergyFunctionRegistr`:

```
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, although there are situations (e.g. testing the simulation) when speed of implementing the actual plugin matters more than actual speed of run and that's when Python comes handy.

Example 8 – *Plugins*

In this example we will show you how to build a plugin that will be called every spin flip, as opposed to every spin flip attempt. This plugin will invoke a changeField3D function (implemented in C++) from Mitosis plugin. After this call we will set a new cell types for the two cells that we got after mitosis but we will do it from Python level. This technique is quite useful because you are still calling C++ function to do real work (in this case C++ function will run mitosis algorithm) and at the Python level we do only finishing tasks, such as type reassignment, and setting targetVolume which usually does not take too much time. So performance-wise we should be OK, although some speed penalty is unavoidable. OK, let's see how this is implemented. We will be using growingcells_fast.py, growingcells_fast.xml and growingcells.pif simulation files.

In growingcells_fast.xml we are using VolumeLocalFlex plugin. This plugin calculates change of energy based on targetVolume and lambdaVolume which are attributes of the cell (i.e. it might be different for different cells). For this reason first, we need to initialize volumeTarget and lambdaVolume for every cell before we start the actual simulation and second, after the mitosis happens we need to set targetVolume for the childCell and we will set it to match targetVolume of the parentCell. Every 10 MCS we will be increasing by 1 targetVolume of every cell. This task as well as setting up initial targetVolume will be managed by VolumeParamSteppable:

File: *cellsort_2D_steppables.py*

```
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
```

We hope you are able to decipher this steppable without any problems. Now, let's go to the implementation of the mitosis plugin in Python. This task will be very easy. All we are required to do is to implement updateAttributes member functions where we assign parent and daughter cell properties after mitosis took place:

File: *cellsort_2D_plugins.py*

```
from PyPluginsExamples import MitosisPyPluginBase
class MitosisPyPlugin(MitosisPyPluginBase):
   def __init__(self , _simulator , _changeWatcherRegistry , _stepperRegistry):
      MitosisPyPluginBase.__init__(self,_simulator,_changeWatcherRegistry,
_stepperRegistry)
```

```
   def updateAttributes(self):
      self.childCell.targetVolume=self.parentCell.targetVolume
      self.childCell.lambdaVolume=self.parentCell.lambdaVolume

      if self.parentCell.type==1:
         self.childCell.type=2
      else:
         self.childCell.type=1
```

This function is run after mitosis happened in the `step` function. As you can see, first we set targetVolume of the childCell to targetVolume of the parentCell. Next we do type switching so that child cell has different cell than parent cell. Of course, we are using only two cell types here (1 and 2).

The actual `MitosisPyPlugin` class is derived from `MitosisPyPluginBase` class which resides in `PyPluginsExamples` module and that's why we use import statement:

```
from PyPluginsExamples import MitosisPyPluginBase
```

Finally let's look at how you set up this plugin in the main Python simulation file (see comments in the code snippet below):

File: *growingcells_fast.py*

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

sys.path.append(environ["PYTHON_MODULE_PATH"])
sys.path.append(getcwd()+"/examples_PythonTutorial")
import SystemUtils
SystemUtils.initializeSystemResources()

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

import CompuCell #notice importing CompuCell to main script has to be done after call to
getCoreSimulationObjects()
from cellsort_2D_plugins import MitosisPyPlugin

changeWatcherRegistry=CompuCellSetup.getChangeWatcherRegistry(sim)

stepperRegistry=CompuCellSetup.getStepperRegistry(sim)

mitPy=MitosisPyPlugin(sim,changeWatcherRegistry,stepperRegistry)
mitPy.setDoublingVolume(50)

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

from cellsort_2D_steppables import VolumeParamSteppable
volumeParamSteppable=VolumeParamSteppable(sim,10)
steppableRegistry.registerSteppable(volumeParamSteppable)

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

If you try running thins simulation you will notice that the slowdown in the speed of run is not as dramatic as in the case of simulation with VolumePlugin implemented in Python. This is due to the fact that there are much less actual spin flip than spin flip attempts, and also that in mitosis plugin most of the work is done by a C++ function. Python written Mitosis plugin serves as a wrapper for a C++ counterpart. This suggests that plugins that "react" to the spin flip are a decent candidates for being implemented in Python. However one needs to be careful about the details of implementation. In this case all the most of the work has been done indirectly in C++ anyway.

Important: Objects like changeWatcherRegistry, stepperRegistry, energyFunctionRegistry that we have encountered before serve two purposes:

1) They store references to change watchers, steppers, energy functions and run them at appropriate times
2) They provide change watchers, steppers, energy functions all the information from C++ code that is necessary for those objects to function properly

After studying all of the above examples you should be able to gain enough information to develop your own modules that will make your simulations be more complex. At the same time you will avoid the hassle of coding in C++ so you will be able to set up your simulations much faster.

Exercises

In this section you will find few exercises that might improve your CompuCell3D Python scripting skills.

Exercise 1

Set up cell sorting simulation using xml (simply use cellsort_2D.xml example included with CompuCell3D package). In the BlobInitializer steppable remove CellSortInit tag so that you begin with a blob of cells each of the same type.

Tasks for you:

1) Write Python steppable that initializes cell types so that they are random (with type id either 1 or
2) Write Python steppable that initializes cell types so that bottom half is of type 1 and upper half is of type 2. Make the steppable flexible so that users can define coordinate range for cells of type 1 and type 2.
3) Write Python steppable that initializes cell types so that different quarters of the blob have alternating cell type ids (1 or 2)

Exercise 2

Open up foam simulation (Foam_try.xml). In one of the exercises that dealt with foams you were asked to
find average area of n-sided bubble and produce a a histogram of $<A_n>$ as a function of *n.* You did this exercise by taking advantage of a CompuCell3D module that did this analysis for you. All you had to do was to use steppable FoamDataOutput (see Foam_try.xml). Now, your task would be to:

1) Write a Python steppable that writes a file where each line (one line per each cell) consists of

   cell id its surface number of neighbors

2) Modify the stappable so that in addition to writing the file it outputs values of the histogram to a text file. The file format would be as follows:

   n $<A_n>$

   for example

   3 23.1

4 34.2
5 50.1
...

Exercise 3

In this, slightly longer exercise, you will build a flowing foam simulation from, almost, scratch.

Open up an editor and start typing “regular” xml file for CompuCell3D simulation (call it e.g. foamair.xml). Make your lattice longer in x-direction so that foam bubbles will flow from left to right. For now let's assume that your lattice will be 100x50x1. After you get your simulation working we will increase the lattice size. Prepare “regular” Python script for CompuCell3D simulation (simply copy existing Python script, modify it and call it foamair.py). Here are the tasks for you:

1) Go to PySteppablesExamples.py and find BubbleNucleator steppable (class). Try to understand what it does. Now, use this steppable in the main Python script (foamair.py). Make sure you set up all its options like initial cell type, initial volume, initial lambda volume etc... In foamair.xml include <Plugin Name="VolumeLocalFlex"/> tag so that volume energy will be calculated based on local cell parameters. This is why you need to make sure that initial target volume and initial lambda volume are set in the bubble nucleator. You will also need Contact plugin. Copy the syntax from one of existing example xml files.

   Try running the simulation.

2) As you will notice cells are created on the left side of the lattice and as more and more cells gets created cells foam starts flowing from left to right. Simply cells on the left exert pressure on rightmost cells. Now, at some point you will notice that there is more and more cells and cells become squashed. We need to create sink for the rightmost cells. Get familiar with BubbleCellRemover and make sure you know how to use it. Pyt BubbleCellRemover into foamair.py, properly initialize it and run the simulation. Did you manage prevent cells from touching right wall of the lattice (occasional touching is OK).

3) Now you are ready to start injecting air to some of the bubbles. Read AirInjector class and see what is going on there. Add object of type AirInjector to foamair.py.

4) Increase the lattice size and watch the “full size” simulation.

Using Python to Replace XML Configuration File

In the material presented above we required that in order to run the CompuCell3D simulation one needs to supply XML configuration file which will list C++ plugins, steppables and properties of the overall simulation such as lattice dimension, temperature *etc*...

Most recent version of CompuCell3D allows users to script entire simulation using Python only. This way your main simulation script requires only one file instead of two. Of course you still may need other file if you defined Python steppables in separate file but the "main" simulation Python script will now allow users to store all the information that previously were stored in the XML configuration file.

The syntax of the Python replacement was based on actual XML syntax and many Python function names for parameter input have the same names as XML tags making it very easy for users to switch.

In general, to pass information from Python to CompuCell3D you need to instantiate (create) ParseData type objects in Python, fill out their values and register them with CompuCell3D kernel. Once registered, those objects are analyzed by CompuCell3D and appropriate CompuCell3D modules (plugins, steppables or kernel) are being initialized (also created if necessary) and simulation is then ready to run.

Perhaps the best way to show benefits of using CompuCell3D with just one Python configuration file is to show the example:

File: *cellsort-2D-player-new-syntax.py*

```
def configureSimulation(sim):
   import CompuCell
   import CompuCellSetup

   ppd=CompuCell.PottsParseData()
   ppd.DebugOutputFrequency(40)
   ppd.Steps(20000)
   ppd.Anneal(10)
   ppd.Temperature(5)
   ppd.Flip2DimRatio(1.0)
   ppd.FlipNeighborMaxDistance(1.75)
   ppd.Dimensions(CompuCell.Dim3D(100,100,1))

   ctpd=CompuCell.CellTypeParseData()
   ctpd.CellType("Medium",0)
   ctpd.CellType("Condensing",1)
   ctpd.CellType("NonCondensing",2)

   cpd=CompuCell.ContactParseData()
   cpd.Energy("Medium","Medium",0)
   cpd.Energy("NonCondensing","NonCondensing",16)
   cpd.Energy("Condensing","Condensing",2)
   cpd.Energy("NonCondensing","Condensing",11)
   cpd.Energy("NonCondensing","Medium",16)
   cpd.Energy("Condensing","Medium",16)
```

```
   vpd=CompuCell.VolumeParseData()
   vpd.LambdaVolume(1.0)
   vpd.TargetVolume(25.0)

   bipd=CompuCell.BlobInitializerParseData()
   region=bipd.Region()
   region.Center(CompuCell.Point3D(50,50,0))
   region.Radius(30)
   region.Types("Condensing")
   region.Types("NonCondensing")
   region.Width(5)

   CompuCellSetup.registerPotts(sim,ppd)
   CompuCellSetup.registerPlugin(sim,ctpd)
   CompuCellSetup.registerPlugin(sim,cpd)
   CompuCellSetup.registerPlugin(sim,vpd)

   CompuCellSetup.registerSteppable(sim,bipd)

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)
```

Notice that `configureSimulation` function is the one which replaces XML configuration file. Notice also how closely names of the function calls for the ParseData type objects resemble names of tags used in the XML file. The pattern here, as you may have already noticed it is that every plugin, steppable and Potts section has its own ParseData object responsible for passing parameters from Python to the CompuCell3D kernel. Once you have initialized ParseData objects all you need to do is to register them with CompuCell3D kernel using `CompuCellSetup.registerPotts` for Potts, `CompuCellSetup.registerPlugin` for plugins and `CompuCellSetup.registerSteppable` for steppables.

Once is complete you use Python script introduced earlier and put a call to **`configureSimulation`** right after line that initializes sim and simthread objects - `sim,simthread = CompuCellSetup.getCoreSimulationObjects()`. One thing to be aware of is that if you are using new style of Python scripting does not prohibit you from using XML. As a matter of fact you can even mix where you initialize plugins – some of them may be handled directly through Python as above and some of them might be handled through XML. However, if you decide you want to mix the two styles, be prepared that strange errors may occur with cryptic messages. Therefore we strongly recommend that you pick Python scripting style and avoid strange mixes.

The reminder of this section is devoted to presenting specific examples of how to replace XML for various plugins and steppables available in the CompuCell3D

CellType Plugin:

```
ctpd=CompuCell.CellTypeParseData()
ctpd.CellType("Medium",0)
ctpd.CellType("Condensing",1)
ctpd.CellType("NonCondensing",2,True)
```

the format is as CellType is as follows CellType(type name, type id, Freeze or not (default False))

Contact Plugin:

```
cpd=CompuCell.ContactParseData()
cpd.Energy("Medium","Medium",0)
cpd.Energy("NonCondensing","NonCondensing",16)
cpd.Energy("Condensing","Condensing",2)
cpd.Energy("NonCondensing","Condensing",11)
cpd.Energy("NonCondensing","Medium",16)
cpd.Energy("Condensing","Medium",16)
cpd.NeighborOrder(2)
```

The format of the Energy function is Energy(type1 name ,type2 name, contact energy). You may also specify how many neighbors to include in the calculations by using NeighborOrder function.

Volume Plugin:

```
vpd=CompuCell.VolumeParseData()
vpd.LambdaVolume(1.0)
vpd.TargetVolume(25.0)
```

VolumeTracker Plugin:

```
vtpd=CompuCell.VolumeTrackerParseData()
```

The syntax of the ParseData object for this plugin mirrors XML syntax.

Surface Plugin:

```
spd=CompuCell.SurfaceParseData()
spd.LambdaSurface(1.0)
spd.TargetSurface(25.0)
```

The syntax of the ParseData object for this plugin mirrors XML syntax.

CenterOfMass Plugin:

```
comtpd=CompuCell.CenterOfMassTrackerParseData()
```

This is all that is required if you want to use this plugin.

PlasticityTracker Plugin:

```
ptpd=CompuCell.PlasticityTrackerParseData()
ptpd.IncludeType("Body1")
ptpd.IncludeType("Body2")
ptpd.IncludeType("Body3")
```

Similarly as in XML you use IncludeType function in place of the actual XML tag.

Plasticity Plugin:

```
plastpd=CompuCell.PlasticityParseData()
plastpd.Local(True)
plastpd.LambdaPlasticity(200.0)
plastpd.TargetLengthPlasticity(6.0)
```

The syntax of this plugin closely follows XML format.

ExternalPotential Plugin:

```
eppd=CompuCell.ExternalPotentialParseData()
eppd.Lambda(-10,0,0)
```

The syntax of this plugin closely follows XML format.

LengthConstraintLocalFlex Plugin:

```
lcpd=CompuCell.LengthConstraintLocalFlexParseData()
```

Connectivity Plugin:

```
connectpd=CompuCell.ConnectivityParseData()
connectpdPenalty(10000000)
```

LengthConstraint Plugin:

```
lcpd=CompuCell.LengthConstraintParseData()
lcpd.LengthEnergyParameters("Condensing",10,2.0)
```

where arguments of `LengthEnergyParameters()` function are: type name, target length, lambda length.

ConnectivityLocalFlex Plugin:

```
connectpd=CompuCell.ConnectivityLocalFlexParseData()
```

SimpleClock Plugin:

```
scpd=CompuCell.SimpleClockParseData()
```

SimpleArray Plugin:

```
sapd=CompuCell.SimpleArrayParseData()
sapd.Values("10")
```

or

```
sapd.test_string="10"
```

Depending on the particular version of the CompuCell3D you may need to use either

`sapd.Values("10")` or `sapd.test_string="10"` syntax.

ChemotaxisDicty Plugin:

```
cdpd=CompuCell.ChemotaxisDictyParseData()
cdpd.Lambda(200)
cdpd.ChemicalField("ReactionDiffusionSolverFE_SavHog","cAMP")
```

Here `ChemicalField` takes two arguments: the first one is name of the PDE solver (`ReactionDiffusionSolverFE_SavHog`) and the other is the name of the actual field causing chemotaxis (`cAMP`).

Chemotaxis Plugin:

```
chpd=CompuCell.ChemotaxisParseData()
chfield=chpd.ChemicalField()
chfield.Source("FastDiffusionSolver2DFE")
chfield.Name("ATTR")
chbt=chfield.ChemotaxisByType()
chbt.Type("Macrophage")
chbt.Lambda(2.0)
```

Notice that here `ChemicalField()` Python function returns an object that is equivalent to `ChemicalField` section of the XML Subsequently `ChemicalField` object is filled with `ChemotaxisByType` objects (again there is very close XML similarity here). You use `ChemotaxisByType()` function to get `ChemotaxisByType` structure to be filled out.

PDESolverCaller Plugin:

```
pscpd=CompuCell.PDESolverCallerParseData()
sd=pscpd.CallPDE("FlexibleDiffusionSolverFE",10)
```

The syntax of this plugin closely follows XML format.

Compartment Plugin:

```
cpd=CompuCell.CompartmentParseData()
cpd.Energy("Medium","Medium",0)
cpd.Energy("NonCondensing","NonCondensing",16)
cpd.Energy("Condensing","Condensing",2)
cpd.Energy("NonCondensing","Condensing",11)
cpd.Energy("NonCondensing","Medium",16)
cpd.Energy("Condensing","Medium",16)

cpd.InternalEnergy("Medium","Medium",0)
cpd.InternalEnergy("NonCondensing","NonCondensing",23)
cpd.InternalEnergy("Condensing","Condensing",20)
cpd.InternalEnergy("NonCondensing","Condensing",22)
cpd.InternalEnergy("NonCondensing","Medium",33)
cpd.InternalEnergy("Condensing","Medium",50)

cpd.NeighborOrder(2)
```

The format of the `Energy` function is `Energy(type1 name ,type2 name, contact energy)`. You may also specify how many neighbors to include in the calculations by using `NeighborOrder` function.

The `InternalEnergy` is used to enter internal energies following XML syntax. The order of argument is the same as in the case of Energy function.

OrientedContact Plugin:

```
cpd=CompuCell.CompartmentParseData()
cpd.Energy("Medium","Medium",0)
cpd.Energy("NonCondensing","NonCondensing",16)
cpd.Energy("Condensing","Condensing",2)
cpd.Energy("NonCondensing","Condensing",11)
cpd.Energy("NonCondensing","Medium",16)
cpd.Energy("Condensing","Medium",16)

cpd.NeighborOrder(2)
```

The format of the `Energy` function is `Energy(type1 name ,type2 name, contact energy)`.

Rearrangement Plugin:

```
rpd=CompuCell.RearrangementParseData()
rpd.FRearrangement(10)
rpd.LambdaRearrangement(20)
rpd.PercentageLossThreshold(0.50)
rpd.DefaultPenalty(100000000000)
```

The syntax of this plugin closely follows XML format.

Mitosis Plugin:

```
mppd=CompuCell.MitosisParseData()
mppd.DoublingVolume(50)
```

The syntax of this plugin closely follows XML format.

NeighborTracker Plugin:

```
ntpd=CompuCell.NeighborTrackerParseData()
```

All you need to do for this plugin is to create `NeighborTrackerParseData` object and of course register it.

PolarizationVector Plugin:

```
pvpd=CompuCell.PolarizationVectorParseData()
```

All you need to do for this plugin is to create `PolarizationVectorParseData` object and of course register it.

ContactMultiCad Plugin:

```
cpd=CompuCell.ContactMultiCadParseData()

cpd.Energy("Medium" ,"Medium",0)
cpd.Energy("Medium" ,"CadExpLevel1",0)
cpd.Energy("Medium" ,"CadExpLevel2",0)
cpd.Energy("CadExpLevel1" ,"CadExpLevel1",0)
cpd.Energy("CadExpLevel1" ,"CadExpLevel2",0)
```

```
cpd.Energy("CadExpLevel2" ,"CadExpLevel2",0)
sc=cpd.SpecificityCadherin()
sc.Specificity("NCad","NCad",-1)
sc.Specificity("NCam","NCam",-1)
sc.Specificity("NCad","NCam",-1)
cpd.EnergyOffset(0.0)
cpd.Depth(1.75)
```

The syntax of this plugin closely follows XML format.

ContactLocalProduct Plugin:

```
cpd=CompuCell.ContactLocalProductParseData()
cpd.ContactSpecificity("Medium" ,"Medium",0)
cpd.ContactSpecificity("Medium" ,"CadExpLevel1",-16)
cpd.ContactSpecificity("Medium" ,"CadExpLevel2",-16)
cpd.ContactSpecificity("CadExpLevel1" ,"CadExpLevel1",-2)
cpd.ContactSpecificity("CadExpLevel1" ,"CadExpLevel2",-2.75)
cpd.ContactSpecificity("CadExpLevel2" ,"CadExpLevel2",-1)
cpd.EnergyOffset(0.0)
cpd.Depth(1.75)
```

The syntax of this plugin closely follows XML format.

Notice that for the last two plugins you need to use Python scripting to initialize appropriate cells' expression levels.

ContactLocalFlex Plugin:

```
cpd=CompuCell.ContactLocalFlexParseData()
cpd.Energy("Medium","Medium",0)
cpd.Energy("NonCondensing","NonCondensing",16)
cpd.Energy("Condensing","Condensing",2)
cpd.Energy("NonCondensing","Condensing",11)
cpd.Energy("NonCondensing","Medium",16)
cpd.Energy("Condensing","Medium",16)
cpd.Depth(1.75)
```

The syntax of this plugin closely follows XML format.

BlobInitializer Steppable:

```
bipd=CompuCell.BlobInitializerParseData()
region=bipd.Region()
region.Center(CompuCell.Point3D(50,50,0))
region.Radius(30)
region.Types("Condensing")
region.Types("NonCondensing")
region.Width(5)
```

Similarly as in the XML you can include many regions. Each time you call `bipd.Region()` a reference to the new region is returned. Using this reference (region) you describe the region in the very similar way you would describe it in the XML.

UniformInitializer Steppable:

```
uipd=CompuCell.UniformInitializerParseData()
```

```
region=uipd.Region()
region.BoxMin(CompuCell.Point3D(20,20,0))
region.BoxMax(CompuCell.Point3D(80,80,0))
region.Types("Condensing")
region.Types("NonCondensing")
region.Width(5)
```

Similarly as in the XML you can include many regions. Each time you call `uipd.Region()` a reference to the new region is returned. Using this reference (region) you describe the region in the very similar way you would describe it in the XML.

PIFInitializer Steppable:

```
pifpd=CompuCell.PIFInitializerParseData()
pifpd.PIFName("plasticitytest.pif")
```

PIFDumper Steppable:

```
pdpd=CompuCell.PIFDumperParseData()
pdpd.PIFName("output")
```

The syntax of this steppable closely follows XML format.

BoxWatcher Steppable:

```
bwpd=CompuCell.BoxWatcherParseData()
bwpd.XMargin(1)
bwpd.YMargin(1)
bwpd.ZMargin(1)
```

The syntax of this steppable closely follows XML format.

DictyFieldInitializer Steppable:

```
dipd=CompuCell.DictyFieldInitializerParseData()
dipd.ZonePoint(CompuCell.Point3D(14,14,3),10)
dipd.PresporeRatio(0.8)
dipd.Gap(1)
dipd.Width(4)
```

The syntax of this steppable closely follows XML format.

DictyChemotaxis Steppable:

```
dcspd=CompuCell.DictyChemotaxisSteppableParseData()
dcspd.ClockReloadValue(850)
dcspd.ChemotactUntil(750)
dcspd.IgnoreFirstSteps(500)
dcspd.ChetmotaxisActivationThreshold(0.2)
dcspd.ChemicalField("ReactionDiffusionSolverFE_SavHog","cAMP")
```

The syntax of this steppable closely follows XML format.

ReactionDiffusionSolverFE_SavHog Steppable:

```
rdssh=CompuCell.ReactionDiffusionSolverFE_SavHogParseData()
rdssh.NumberOfFields(3)
rdssh.FieldName("cAMP")
rdssh.FieldName("Refractoriness")
```

```
rdssh.DeltaX(0.37)
rdssh.DeltaT(0.01)
rdssh.DiffusionConstant(1.0)
rdssh.DecayConstant(0.0)
rdssh.MaxDiffusionZ(8)
rdssh.IntervalParameters(0.0065,0.841)
rdssh.fFunctionParameters(20.0, 3.0,15.0, 0.15)
rdssh.epsFunctionParameters(0.5, 0.0589,0.5)
rdssh.RefractorinessParameters(3.5, 0.35)
rdssh.MinDiffusionBoxCorner(CompuCell.Point3D(0,0,0))
rdssh.MaxDiffusionBoxCorner(CompuCell.Point3D(40,40,40))
```

The syntax of this steppable follows XML format however it is important to put interval certain coefficients in the correct order:

`rdssh.IntervalParameters(0.0065,0.841)` – the argument order is c1, c2 (for more information see XML syntax description or XML file using this steppable)

`rdssh.fFunctionParameters(20.0, 3.0,15.0, 0.15)` - the argument order is C1, C2, C3, a

`rdssh.epsFunctionParameters(0.5, 0.0589,0.5)` - the argument order is eps1, eps2, eps3

`rdssh.epsFunctionParameters(0.5, 0.0589,0.5)` the argument order is k,b

DictyChemotaxis Steppable:

```
dcspd=CompuCell.DictyChemotaxisSteppableParseData()
dcspd.ClockReloadValue(850)
dcspd.ChemotactUntil(750)
dcspd.IgnoreFirstSteps(500)
dcspd.ChetmotaxisActivationThreshold(0.2)
dcspd.ChemicalField("ReactionDiffusionSolverFE_SavHog","cAMP")
```

The syntax of this steppable follows XML format. Here `ChemicalField` takes two arguments one is name of the PDE solver (`ReactionDiffusionSolverFE_SavHog`) another on is the name of the actual field causing chemotaxis (`cAMP`).

FastDiffusionSolver2DFE Steppable:

```
fdspd=CompuCell.FastDiffusionSolver2DFEParseData()
df=fdspd.DiffusionField()
diffData=df.DiffusionData()
secrData=df.SecretionData()
diffData.UseBoxWatcher(True)
diffData.DiffusionConstant(0.1)
diffData.FieldName("FGF")
diffData.ConcentrationFileName("diffusion_2D_fast_box.pulse.txt")
```

Again, the syntax follows that of the XML file. Notice that the following calls:

```
df=fdspd.DiffusionField()
diffData=df.DiffusionData()
secrData=df.SecretionData()
```

return equivalents of `Diffusion Field, DiffusionData` and `SecretionData` sections of the XML. Once you have these you feel the rest of the steppable parameters following XML pattern. The function names are analogous to the XML tags.

FlexibleDiffusionSolver3DFE Steppable:

```
fdspd=CompuCell.FlexibleDiffusionSolverFEParseData()
df=fdspd.DiffusionField()
diffData=df.DiffusionData()
secrData=df.SecretionData()
diffData.DiffusionConstant(0.1)
diffData.FieldName("ATTR")
diffData.DoNotDiffuseTo("Wall")
secrData.Secretion("Bacterium",200)
secrData.SecretionOnContact("Wall","Medium",200)
secrData.SecretionOnContact("Macrophage","Medium",500)
```

The syntax is very similar to that of FastDiffusionSolver2DFE. Notice how SecretionOnContact function is implemented:

```
secrData.SecretionOnContact("Macrophage","Medium",500)
```

The argument order is the following Secreting type name (Macrophage), type name of the cells that trigger secretion (“Medium”), secretion constant

Steering

3.2.1 release of CompuCell3D allows users to steer the simulation while it is running. By steering we mean altering model parameter while the simulation is running. For example you may imagine a cell-sorting simulation where at certain Monte Carlo Step you change adhesivity between Condensing and NonCondensing cells or want to change target volume of Condensing cells only. in earlier version of CompuCell3D you could use “flex” plugins such as VolumeFlex or ContactLocalFlex to control appropriate parameters for each cell individually so in this sense, steering was possible even with earlier versions of CompuCell3D however, it was not possible to alter parameters that were passed from XML configuration file. Current version removes this limitation and allows users to control pretty much all aspects of the simulation from Python script. things like diffusion or secretion constants, adhesivities, constraints can all be changed from Python regardless where they are initially defined.

We will now show you examples of steering implementation in Python. The full documentation of steering modules is under development but we will show you how to figure out appropriate function calls to steer CompuCell3D modules.

Let's look at the cell sorting simulation and see how we can change adhesivities while the simulation is running:

File: *cellsort-2D-player-new-syntax-steering.py*

```
def configureSimulation(sim):
```

```
   import CompuCell
   import CompuCellSetup

   ppd=CompuCell.PottsParseData()
   ppd.DebugOutputFrequency(40)
   ppd.Steps(20000)
   ppd.Anneal(10)
   ppd.Temperature(5)
   ppd.Flip2DimRatio(1.0)
   ppd.FlipNeighborMaxDistance(1.75)
   ppd.Dimensions(CompuCell.Dim3D(100,100,1))

   ctpd=CompuCell.CellTypeParseData()

   ctpd.CellType("Medium",0)
   ctpd.CellType("Condensing",1)
   ctpd.CellType("NonCondensing",2)

   cpd=CompuCell.ContactParseData()
   cpd.Energy("Medium","Medium",0)
   cpd.Energy("NonCondensing","NonCondensing",16)
   cpd.Energy("Condensing","Condensing",2)
   cpd.Energy("NonCondensing","Condensing",11)
   cpd.Energy("NonCondensing","Medium",16)
   cpd.Energy("Condensing","Medium",16)

   vfpd=CompuCell.VolumeFlexParseData()
   vfpd.VolumeEnergyParameters("Condensing",25.0,1.0)
   vfpd.VolumeEnergyParameters("NonCondensing",25.0,1.0)

   sfpd=CompuCell.SurfaceFlexParseData()
   sfpd.SurfaceEnergyParameters("Condensing",25.0,1.0)
   sfpd.SurfaceEnergyParameters("NonCondensing",25.0,1.0)

   bipd=CompuCell.BlobInitializerParseData()
   region=bipd.Region()
   region.Center(CompuCell.Point3D(50,50,0))
   region.Radius(30)
   region.Types("Condensing")
   region.Types("NonCondensing")
   region.Width(5)

   CompuCellSetup.registerPotts(sim,ppd)
   CompuCellSetup.registerPlugin(sim,ctpd)
   CompuCellSetup.registerPlugin(sim,cpd)
   CompuCellSetup.registerPlugin(sim,vfpd)
   CompuCellSetup.registerPlugin(sim,sfpd)

   CompuCellSetup.registerSteppable(sim,bipd)

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()

from steering_steppables_examples import PottsSteering
ps=PottsSteering(sim,100)
steppableRegistry.registerSteppable(ps)

from steering_steppables_examples import ContactSteering
cs=ContactSteering(sim,100)
steppableRegistry.registerSteppable(cs)

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

In the above example we initialize entire simulation in Python (no XML file is needed). Notice however, that despite the fact that entire initialization takes place in Python in previous versions of CompuCell3D we could not alter parameters of Contact plugin once the simulation was started. Now we will show you how this is done with the new version. The following lines of Python code call steppable where we do the actual steering:

```
from steering_steppables_examples import ContactSteering
cs=ContactSteering(sim,100)
steppableRegistry.registerSteppable(cs)
```

The ContactSteering module is quite simple:

File: *steering_steppables_examples.py*

```
class ContactSteering(SteppablePy):
   def __init__(self,_simulator,_frequency=1):
      SteppablePy.__init__(self,_frequency)
      self.simulator=_simulator
      self.contactParseData=CompuCellSetup.getModuleParseData("Contact")
   def step(self, _mcs):
      enTupple=self.contactParseData.getContactEnergyTupple("Condensing",\
      "NonCondensing")
      if enTupple:
         enTupple.energy+=1
         CompuCellSetup.steer(self.contactParseData)
```

In the constructor we fetch a reference to ParseData structure for Contact plugin:

```
self.contactParseData=CompuCellSetup.getModuleParseData("Contact")
```

In the step function we fetch a tupple that defines contact energy between Condensing and NonCondensing cells and increase the energy by 1 unit each time the steppable is called:

```
enTupple=self.contactParseData.getContactEnergyTupple("Condensing","NonCondensing")
if enTupple:
   enTupple.energy+=1
   CompuCellSetup.steer(self.contactParseData)
```

To make steering actually happen we must call steer function from CompuCellSetup module;

```
CompuCellSetup.steer(self.contactParseData)
```

As you can see implementing steering is quite simple and follows the following pattern:

1. Obtain ParseData class (here it was ContactParseData class) that is associated with each CompuCell3D module
2. Get access to parameters from ParseData that you wish to change
3. Call steer function from CompuCellSetup module

Now we will show you how to figure out function calls or parameter names that you need to use to get access to plugin parameters you wish to change.
Each plugin has a ParseData class associated with it. For example Volume plugin will have a class called VolumeParseData, Contact Plugin will have class ContactParseData and so on. You may find header files for these classes in the source code for CompuCell3D in the directory CompuCell3D/plugins/NameOfAPlugin. Same applies to steppables. Potts module has ParseData class that resides in CompuCell3D directory of the source code and is called PottsParseData.

Now lets take a look at the CompuCell3D/plugins/Contact/ContactParseData.h file where ContactParseData is defined:

File: *ContactParseData.h*

```
#ifndef CONTACTPARSEDATA_H
#define CONTACTPARSEDATA_H

#include <CompuCell3D/ParseData.h>


namespace CompuCell3D {

  class ContactEnergyTupple{
      public:
      ContactEnergyTupple():
         type1Name(""),
         type2Name(""),
         energy(0)

      {}

      ContactEnergyTupple(std::string _type1Name, std::string _type2Name,double _energy):
         type1Name(_type1Name),
         type2Name(_type2Name),
         energy(_energy)
      {}

      std::string type1Name;
      std::string type2Name;
      double energy;

  };

  class ContactParseData : public ParseData {
      public:
         ContactParseData():
            ParseData("Contact"),
```

```
            depth(1.1),
            depthFlag(false),
            weightDistance(false),
            neighborOrder(1)
            {}

         double depth;
         bool depthFlag;
         std::vector<ContactEnergyTupple> contactEnergyTuppleVec;
         bool weightDistance;

         unsigned int neighborOrder;


         void Energy(std::string _type1Name, std::string _type2Name,double _energy){
            contactEnergyTuppleVec.push_back(ContactEnergyTupple(_type1Name,_type2Name,_ene
rgy));
         }

           ContactEnergyTupple * getContactEnergyTupple(std::string _type1Name, std::string
_type2Name){
            for (int i = 0 ; i < contactEnergyTuppleVec.size() ; ++i){
                                    if(contactEnergyTuppleVec[i].type1Name==_type1Name   &&
contactEnergyTuppleVec[i].type2Name==_type2Name)
                  return &contactEnergyTuppleVec[i];
                              else  if  (contactEnergyTuppleVec[i].type2Name==_type1Name  &&
contactEnergyTuppleVec[i].type1Name==_type2Name)
                  return &contactEnergyTuppleVec[i];
            }
            return 0;
         }

         void Depth(double _depth){
            depthFlag=true;
            depth=_depth;
         }
         void Weight(bool _weightDistance){weightDistance=_weightDistance;}
         void NeighborOrder(unsigned int _neighborOrder=1){
            depthFlag=false;
            if(_neighborOrder>neighborOrder){
               neighborOrder=_neighborOrder;
            }
         }
   };
};
```

All you need to get from this file are the names of the functions that allow you to fetch ContactEnergyTupple class where actual energy between two cell types is defined:

```
getContactEnergyTupple(std::string _type1Name, std::string _type2Name)
```

Next you look at the ContactEnergyTupple class and see that it has parameter called energy. This is the parameter that you are changing from Python, recall:

```
if enTupple:
   enTupple.energy+=1
   CompuCellSetup.steer(self.contactParseData)
```

We realize that without at least basic knowledge of programming it might be hard to figure it out how to accomplish steering you might be interested in. To help you with this task we also provide fairly extensive example file steering_steppables_examples.py that shows how to steer various CompuCell3D modules.

The steering will make even more sense when it could be done through user interface. As a matter of fact the current implementation of steering will be used in building the user interface that will have steering widgets which will be much more convenient to use than scripting. On the other hand there is a class of users who prefer scripting to graphical user interfaces. Our implementation of steering will accommodate needs of various types of users.