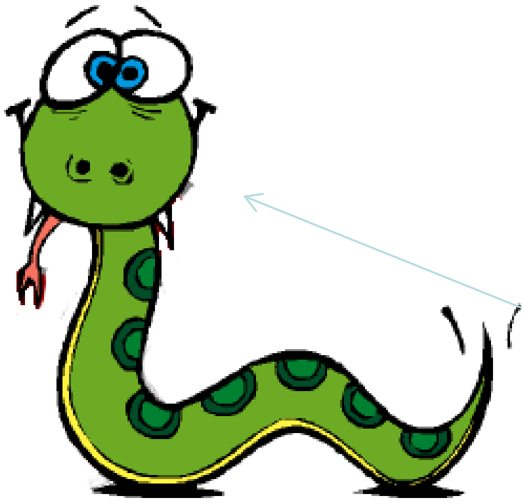


scripting inside CompuCell3D

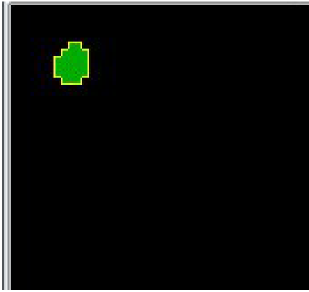
*Python, not Jararaca

- XML gives you the ability to change simulation parameters using human-readable syntax but does not allow users to implement more complex cell behaviors, sophisticated cell type transition rules, inter-cell signaling or connecting to intracellular models
- Python scripting capabilities in CompuCell3D allow users to accomplish abovementioned tasks (and much more) and are the reasons why CompuCell3D is called simulation environment, not simulation application.
- Python scripting capabilities allow users to use rich set of Python modules and third party libraries and provide level flexibility comparable with packages such as Matlab or Mathematica

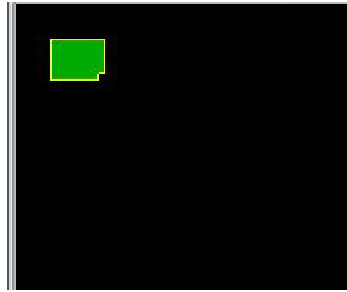
Python Scripting Prerequisites

- To make full use of Python scripting users should be familiar with Python programming language. They do not need to be experts though.
- CompuCell3D comes with template and example codes that make Python scripting inside CompuCell3D quite easy even for beginners.
- Python scripting in CompuCell3D typically requires users to develop a class that implements required logic. If you are unfamiliar with concept of class , think of it as a type that has several data members (such as floats, integers, other classes) and set of functions that operate on those internal members but can also take external arguments. Class is a generalization of "C" structure or "Pascal" record.
- Fortunately CompuCell3D comes with plenty of examples that users can adapt to serve their needs. This does not require thorough programming knowledge. If you are unfamiliar with Python scripting, reading (and doing) "CompuCell3D Python Scripting Tutorials" should quickly get you up-to-speed.

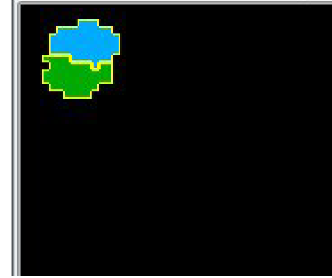
Typical example when Python proves to be very useful



Start with a small cell that grows



It reaches "doubling volume"



… and divides into two cells

After mitosis you want to specify types of parent and daughter cells. You may want to change target surface and target volume of daughter. And target volume is a function of a FGF concentration at the center of mass of the daughter cell.

How would you do it from just XML?

Python seems to be the best solution for problems like this one

Where Do You Begin?

- Early version of Python scripting in CompuCell3D required users to provide CC3DML configuration file. **This is no longer true. You can describe entire simulation from Python level. However, you may still use XML and Python if you want. The choice is up to you.**
- **You will need to write Python script that implements main CompuCell3D logic** i.e. reads CC3DML file (if you are using CC3DML file), initializes modules and executes calls in a loop Metropolis algorithm. This file will also call set up and initialize your modules written in Python. CompuCell3D comes with many examples of such files so in fact preparing one is reduced to minor modification of existing one.
- Once you have Python script (and optionally CC3DML file) ready, you open them up in the Player and start running simulations .

What Can You Do in Python?

- **You may implement any CompuCell3D module using Python** – energy functions, lattice monitors, steppers, steppables, fixed steppers.
- You need to remember that **Python is an interpreted language** and thus executes orders of magnitudes slower than, for example, C++. This means that although you can easily develop energy functions (remember, they are the most frequently called modules in CompuCell3D) in Python, you will probably want to avoid using them with your "production runs". In this it makes sense to implement those functions in C++ , which is not too difficult and we provide comprehensive Developers documentation.
- Since lattice monitors are called less frequently than energy functions, **the performance degradation due to lattice monitor being implemented in Python is much smaller**. That same is true for steppers, fixed steppers and steppables.
- Notice that CompuCell3D kernel that is called from Python, as well as other core CompuCell3D modules called from Python run at native speeds, as they are implemented in C++ with only their API exposed to Python. Therefore **if you run CompuCell3D through Python script but decide not to implement new Python modules, your speed of run will be essentially identical as if you ran CompuCell3D using just CC3DML file.**

What are the advantages of using Python inside CompuCell3D

•**Rapid development** – no compilation is necessary. Write or modify your script and run

•**Portability** – script developed by you on one machine (e.g. Mac) is ready to use under linux

•**Model integration** - you can quite easily implement hooks to subcellular models. We have been able to set up CompuCell3D simulation that was using SBW network intracell simulators within few minutes. T

•**Rich set of external libraries** – you may tap into rich Python library repositories that exist for essentially any task

•**Agile development** – developing and refining your model is an iterative process. Working at the compiled language stage will force you to spend significant portion of your time waiting for the program to compile. With Python you eliminate this step thus increase productivity. Users should first prototype their models in Python and once they are ready to be used for production runs, rewrite the ones causing significant slowdown in C++.

You need to remember this:

Python distinguishes blocks of codes by their indentation. Therefore

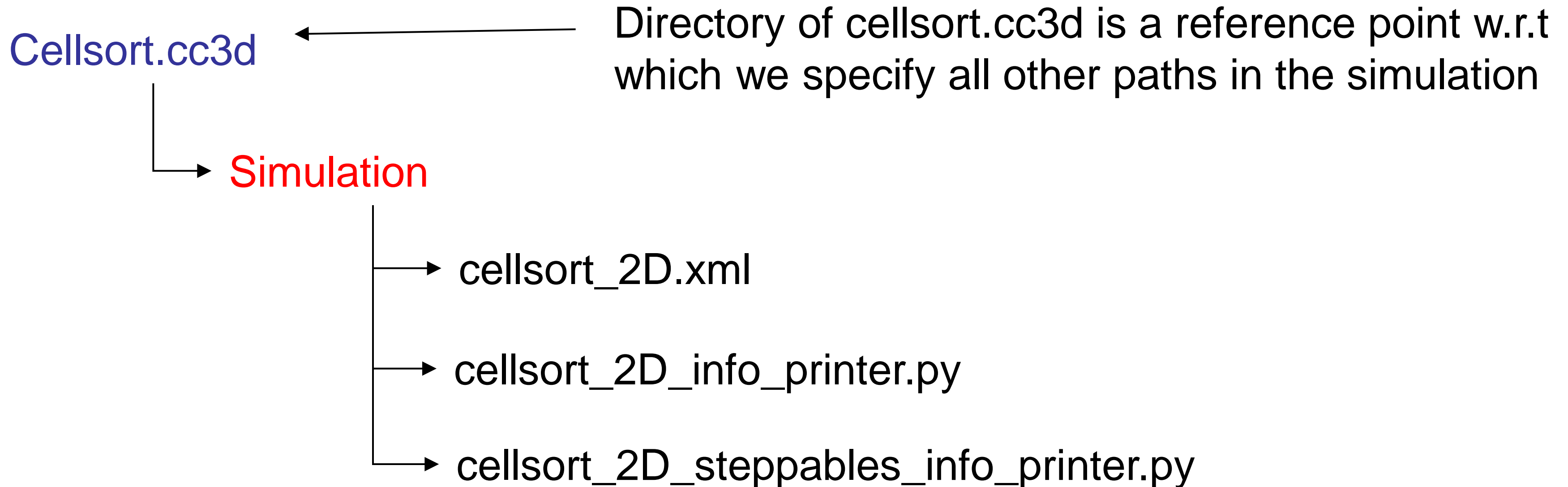
```
for a in xrange(0,5):
        print "variable a=",a
print " Final value of variable a is …" , a
```

would result in an error because the line print " Final value of variable a=",a has different indentation than other print statement and thus does belong to the "for" loop. Python will attempt executing this line once after the "for" loop is finished and will return an error that global object **"a"** was not found. It was found because **"a"** name is valid only inside the "for" loop body. Since the last line was not in the body, you get an error.

We are using 4 spaces to indent block of codes, you may choose differently, but need to be consistent. **HOWEVER 4 space indentation is a *de-facto* standard among Python programers so we strongly recommend you adhere to it**

*.cc3d Files

```
<Simulation version="3.5.1">
    <XMLScript>Simulation/cellsort_2D.xml</XMLScript><
    <PythonScript>Simulation/cellsort_2D_info_printer.py</PythonScript>
    <ResourceType="Python">Simulation/cellsort_2D_steppables_info_printer.py</Resource>
</Simulation>
```



.cc3d simulations can be easily moved from one place to another – not true with e.g. xml +piff based simulations

Your first CompuCell3D Python script. Make sure you have your copy of Python Scripting Tutorials

Begin with template code (the file will be called **cellsort_2D.py**)

```
#import useful modules
import sys
from os import environ
from os import getcwd
import string
#setup search paths
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)
```

Bacterium and macrophage simulation

```
def configureSimulation(sim):
  import CompuCellSetup
  from XMLUtils import ElementCC3D
  cc3d=ElementCC3D("CompuCell3D")
  potts=cc3d.ElementCC3D("Potts")
  potts.ElementCC3D("Dimensions",{"x":100,"y":100,"z":1})
  potts.ElementCC3D("Steps",{},10000)
  potts.ElementCC3D("Temperature",{},15)
  potts.ElementCC3D("NeighborOrder",{},2)

  cellType=cc3d.ElementCC3D("Plugin",{"Name":"CellType"})
  cellType.ElementCC3D("CellType", {"TypeName":"Medium", "TypeId":"0"})
  cellType.ElementCC3D("CellType", {"TypeName":"Bacterium", "TypeId":"1"})
  cellType.ElementCC3D("CellType", {"TypeName":"Macrophage", "TypeId":"2"})
  cellType.ElementCC3D("CellType", {"TypeName":"Wall", "TypeId":"3" , "Freeze":""})

  volume=cc3d.ElementCC3D("Plugin",{"Name":"Volume"})
  volume.ElementCC3D("TargetVolume",{},25)
  volume.ElementCC3D("LambdaVolume",{},15.0)

  surface=cc3d.ElementCC3D("Plugin",{"Name":"Surface"})
  surface.ElementCC3D("TargetSurface",{},20)
  surface.ElementCC3D("LambdaSurface",{},4.0)
```

Continued…

```
contact=cc3d.ElementCC3D("Plugin",{"Name":"Contact"})
contact.ElementCC3D("Energy", {"Type1":"Medium", "Type2":"Medium"},0)
contact.ElementCC3D("Energy", {"Type1":"Macrophage", "Type2":"Macrophage"},15)
contact.ElementCC3D("Energy", {"Type1":"Macrophage", "Type2":"Medium"},8)
contact.ElementCC3D("Energy",{"Type1":"Bacterium", "Type2":"Bacterium"},15)
contact.ElementCC3D("Energy", {"Type1":"Bacterium", "Type2":"Macrophage"},15)
contact.ElementCC3D("Energy", {"Type1":"Bacterium", "Type2":"Medium"},8)
contact.ElementCC3D("Energy", {"Type1":"Wall", "Type2":"Wall"},0)
contact.ElementCC3D("Energy", {"Type1":"Wall", "Type2":"Medium"},0)
contact.ElementCC3D("Energy", {"Type1":"Wall", "Type2":"Bacterium"},50)
contact.ElementCC3D("Energy", {"Type1":"Wall", "Type2":"Macrophage"},50)

chemotaxis=cc3d.ElementCC3D("Plugin",{"Name":"Chemotaxis"})
chemicalField=chemotaxis.ElementCC3D("ChemicalField", {"Source":"FlexibleDiffusionSolverFE", "Name":"ATTR"})
chemicalField.ElementCC3D("ChemotaxisByType", {"Type":"Macrophage" ,"Lambda":200})
```

Continued…

```
flexDiffSolver=cc3d.ElementCC3D("Steppable",{"Type":"FlexibleDiffusionSolverFE"})
diffusionField=flexDiffSolver.ElementCC3D("DiffusionField")
diffusionData=diffusionField.ElementCC3D("DiffusionData")
diffusionData.ElementCC3D("FieldName",{},"ATTR")
diffusionData.ElementCC3D("DiffusionConstant",{},0.10)
diffusionData.ElementCC3D("DecayConstant",{},0.0)
diffusionData.ElementCC3D("DoNotDiffuseTo",{},"Wall")
secretionData=diffusionField.ElementCC3D("SecretionData")
secretionData.ElementCC3D("Secretion", {"Type":"Bacterium"},200)

pifInitializer=cc3d.ElementCC3D("Steppable",{"Type":"PIFInitializer"})
pifInitializer.ElementCC3D("PIFName",{},"Demos/PythonOnlySimulationsExamples/bacterium_macrophage_2D_wall.pif")

# next line is very important and very easy to forget about. It registers XML description and points
# CC3D to the right XML file (or XML tree data structure in this case)
CompuCellSetup.setSimulationXMLDescription(cc3d)
```

Beyond XML - Developing Python Steppables

Examples presented above showed how to run Python based simulations and how to replace XML with Python. However, the true power of Python is demonstrated in the case when you develop **your own modules**. We will first teach you how to develop a **steppable** because **steppables** are most likely to be developed in Python anyway.

Let's take a look at the module that prints cell id, cell type and cell volume for every cell in the simulation. Iterating over all cells is probably most frequently used task in steppables:

```
class InfoPrinterSteppable(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 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
```

Python Steppable

Each Python Steppable should have three functions:

start()

step(mcs)

finish()

It is OK to leave out the implementation of any of above functions empty (or simply pretend they do not exist). An empty function will be then called.

In addition to this, because Python steppables are implemented as classes they need to define __init__ function that acts as a constructor.

Steppable Template:

```
class YourPythonSteppable(SteppableBasePy):
  def __init__(self,_simulator,_frequency=10):
    #your code here
 def start(self):
    #your code here
  def step(self,mcs):
    #your code here
 def finish(self):
    #your code here
```

If you are non-programmer it may looks a bit strange, but imagine how much more would be required to write do the same in C/C++. Much more. Let's explain the code:

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

- First line defines our steppable class. Each class has to have __init__method that is called when object of this class is created. You can pass any arguments to this method, but the first argument must be "self". This is required by Python language.
- First line in __init__ method initializes Base class SteppableBasePy. Do not worry if you do not understand it. Treat it as a boiler plate code.
- SteppableBasePy brings a lot of functionality e.g. self.cellList, self.dim, self.neighborTrackerPlugin etc… Please refer to Python scripting manual for more details.

```
def step(self,mcs):
    print "This function is called every", self.frequency, " MCS"
    for cell in self.cellList:
      print "CELL ID=",cell.id, " CELL TYPE=",cell.type," volume=",cell.volume
```

•Above function implements core functionality of our steppable. It informs that it is called every 10 MCS – see how we set frequency parameter in the __init__ function.

•The last two lines do actual iteration over each cell in the cell inventory

•Notice that it is really easy to do the iteration:

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

•Now you can see how storing CellType object as self.cellList comes handy. All we need to do is to pass iterable cell list (self.cellList) to the “for” loop.

Actual printing is done in line

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

•For each cell in inventory “cell” variable of the for loop will be initialized with different cell from inventory. All you need to do is to print cell.id, cell.type, and cell.volume. It is pretty simple.

Now save the file with the steppable as , **cellsort_2D_steppables.py** . All you need to do is to provide hooks to your steppable in the main Python script:

```
steppableRegistry=CompuCellSetup.getSteppableRegistry()

  ########## Steppable Registration ###########
  from cellsort_2D_steppables import InfoPrinterSteppable
  infoPrinter= InfoPrinterSteppable(sim)
  steppableRegistry.registerSteppable(infoPrinter)
  ##########End of  Steppable Registration ###########

  steppableRegistry.init(sim)
```

Notice that registering steppable requires importing your steppable from the file:

```
from cellsort_2D_stepables import InfoPrinterSteppable
```

creating steppable object:

```
infoPrinter= InfoPrinterSteppable(sim)
```

registering it with steppable registry:

```
steppableRegistry.registerSteppable(infoPrinter)
```

Full Main Script (examples_PythonTutorial/cellsort_2D_info_printer/cellsort_2D_info_printer.py):

```
#import useful modules
import sys
from os import environ
from os import getcwd
import string
#setup search paths
sys.path.append(environ["PYTHON_MODULE_PATH"])
sys.path.append(getcwd()+"/examples_PythonTutorial") #add search path

import CompuCellSetup
# tell CC3D that you will use CC3DML file together with current Python script
CompuCellSetup.setSimulationXMLFileName\
("examples_PythonTutorial/cellsort_2D_info_printer/cellsort_2D.xml")

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

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

Useful shortcut – simplifying steppable definition:

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

Notice that we have used as a base class **SteppableBasePy** instead of **SteppablePy**.

SteppableBasePy already contains members and initializations for:

self.cellList

self.simulator

self.potts

self.cellField

self.dim

self.inventory

SteppableBasePy:

```
class SteppableBasePy(SteppablePy):
   def __init__(self,_simulator,_frequency=1):
      SteppablePy.__init__(self,_frequency)
      self.simulator=_simulator
      self.potts=_simulator.getPotts()
      self.cellField=self.potts.getCellFieldG()
      self.dim=self.cellField.getDim()
      self.inventory=self.simulator.getPotts().getCellInventory()
      self.cellList=CellList(self.inventory)
```

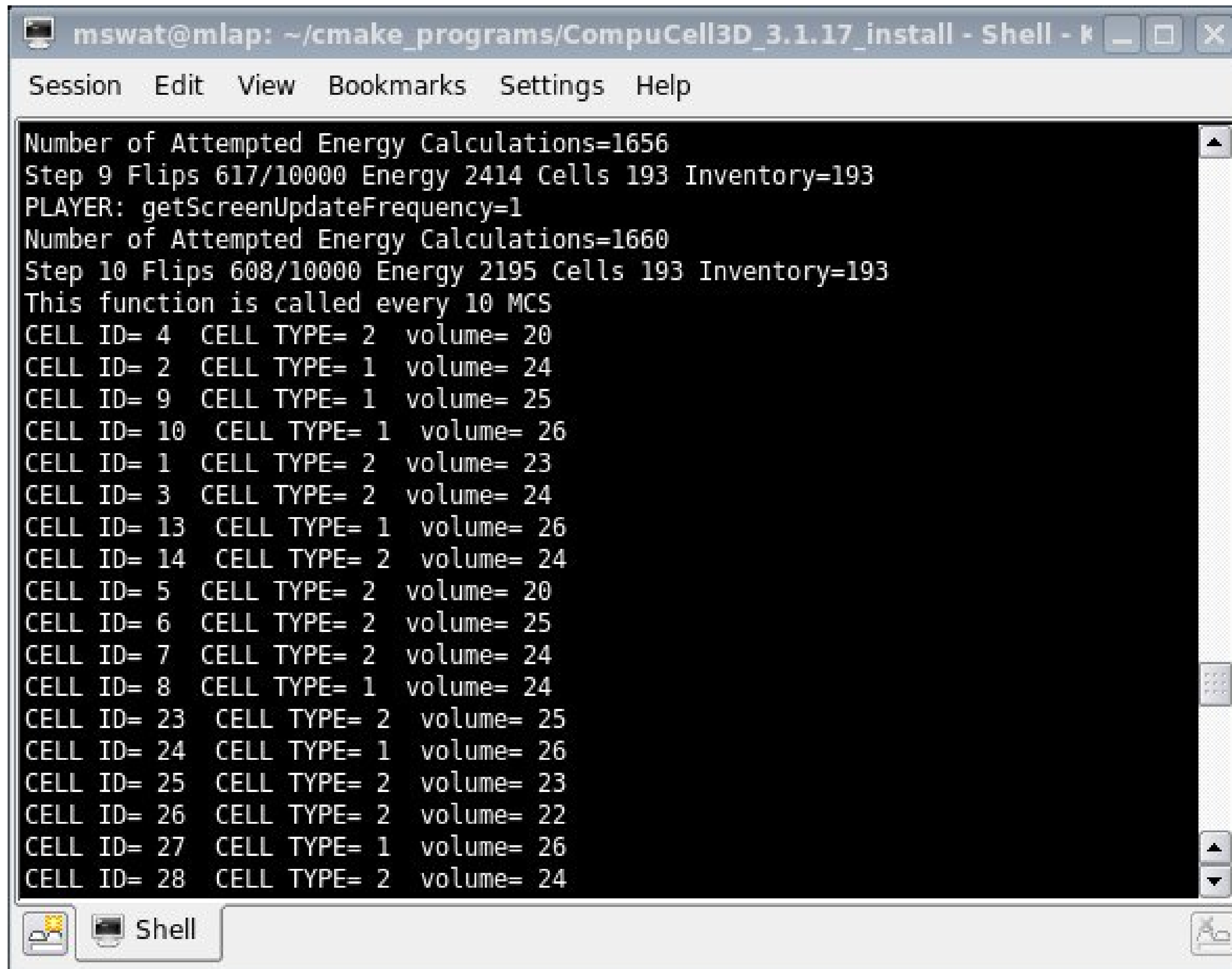
Now, all you need to do is to open in the Player newly created cellsort_2D_info_printer.py. Notice that you are not loading directly **cellsort_2D_steppables.py** file. The module you stored to this file will be called from **cellsort_2D_info_printer.py**.

Try running the simulation and see if you got any performance degradation. Probably not, but by using Python you have saved yourself a lot of tedious C++ coding, not to mention that you do not need to care about dependencies, compilation, etc..

Writing your next Python steppable will require much less effort as well, as you will quickly discover that you will be using same basic code template over and over again. Instead of thinking how the code you are writing fits in the overall framework you will just concentrate on it's core functionality and leave the rest to CompuCell3D.

In case you wonder how this is all possible , it is due to Object Oriented programming. Hopefully this short tutorial will encourage you to learn more of object oriented programming. It is really worth the effort.

Info Printer results



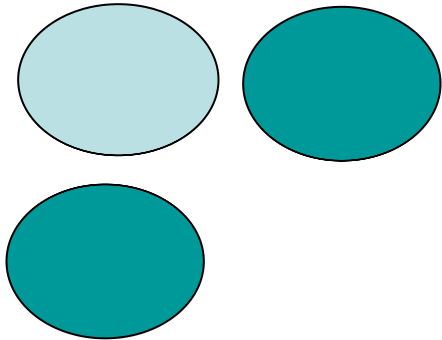
Python Scripting Checklist:

1. Write main Python script (modify or reuse existing one)

2. Write Python modules in a separate file. You will import these modules from main Python script

3. Provide CC3DML configuration file or describe entire simulation in Python skipping CC3DML entirely

Note: when using Python scripting your simulation may consists of many files. Make sure you keep track of them



More Complicated Simulations – Adding Extra Attribute To a Cell

In CompuCell3D simulations each cell by default will have several attributes such as volume, surface, centroids , target volume, cell id etc.

One can write a plugin that attaches additional attributes to a cell during run time. Doing so avoids recompilation of entire CompuCell3D but requires to write and compile the C++ plugin.

It is by far the easiest to attach additional cell attribute in Python.

Starting with version 3.7.0 we also attach to each cell a Python dictionary

We can access this dictionary very easily from Python level. Python ictionary is dynamic data structure can store Python object and the dictionary lookup is done using keywords

Full listing of simulation where each cell gets extra attribute – a list:

```
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

CompuCellSetup.setSimulationXMLFileName("examples_PythonTutorial/cellsort_2D_extra_attrib/cellsort_2D.xml")

sim,simthread = CompuCellSetup.getCoreSimulationObjects()


CompuCellSetup.initializeSimulationObjects(sim,simthread)

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

#here we will add ExtraAttributeCellsort steppable
from cellsort_2D_steppables import ExtraAttributeCellsort
extraAttributeCellsort=ExtraAttributeCellsort(_simulator=sim,_frequency=10)
steppableRegistry.registerSteppable(extraAttributeCellsort)

from cellsort_2D_steppables import TypeSwitcherSteppable
typeSwitcherSteppable=TypeSwitcherSteppable(sim,100)
steppableRegistry.registerSteppable(typeSwitcherSteppable)

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

ExtraAttributeCellsort

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

  def step(self,mcs):
    for cell in self.cellList:
      cellDict=self.getDictionaryAttribute(cell)
      cellDict["Double_MCS_ID"]=mcs*2*cell.id
      print "CELL DICT=",cellDict
```

Notice, you may also attach a **list to a cell as one of the dictionary elements**. See Python Scripting Tutorials for more information.

ExtraAttrib results

```
FAST numberOfAttempts=10000
Number of Attempted Energy Calculations=370
Step 2 Flips 124/10000 Energy 466 Cells 45 Inventory=45
cell.id= 1  dict= {'Double_MCS_ID': 4}
cell.id= 2  dict= {'Double_MCS_ID': 8}
cell.id= 3  dict= {'Double_MCS_ID': 12}
cell.id= 4  dict= {'Double_MCS_ID': 16}
cell.id= 5  dict= {'Double_MCS_ID': 20}
cell.id= 6  dict= {'Double_MCS_ID': 24}
cell.id= 7  dict= {'Double_MCS_ID': 28}
cell.id= 8  dict= {'Double_MCS_ID': 32}
cell.id= 9  dict= {'Double_MCS_ID': 36}
```

TypeSwitcherSteppable

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

  def step(self,mcs):
    for cell in self.cellList:
      if cell.type==self.CONDENSING:
        cell.type= self.NONCONDENSING
      elif cell.type==self.NONCONDENSING:
        cell.type=self. CONDENSING
      else:
        print "Unknown type. In cellsort simulation there should only be two types ",\
        “1 and 2"
```

Line continuation in Python

Accessing NeighborTracker from Python

As you remember from lectures on CC3DML configuration files, CompuCell3D can track cell neighbors. You can access information about cell neighbors directly from Python:

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

  def step(self,mcs):

    for cell in self.cellList:
      for neighbor , commonSurfaceArea in self.getCellNeighborDataList(cell):
        if neighbor:  #check if neighbor is non-Medium
          #access common surface area and id
          print "neighbor.id", neighbor.id," commonSurfaceArea=",commonSurfaceArea
        else:
          print "Medium commonSurfaceArea=", commonSurfaceArea
```

Understanding iteration over cell neighbors

def step(self,mcs):

```
for cell in self.cellList:
    for neighbor , commonSurfaceArea in self.getCellNeighborDataList(cell):
      if neighbor:  #check if neighbor is non-Medium
        #access common surface area and id
        print "neighbor.id", neighbor.id," commonSurfaceArea=",commonSurfaceArea
      else:
        print "Medium commonSurfaceArea=", commonSurfaceArea
```

Iterating over cell neighbors . The loop returns neighbor cell object and common surface area

```
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
CompuCellSetup.setSimulationXMLFileName("examples_PythonTutorial/cellsort_2D_ne
ighbor_tracker/cellsort_2D_neighbor_tracker.xml")
sim,simthread = CompuCellSetup.getCoreSimulationObjects()
#Create extra player fields here or add attributes
pyAttributeAdder,listAdder=CompuCellSetup.attachListToCells(sim)
CompuCellSetup.initializeSimulationObjects(sim,simthread)

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

from cellsort_2D_steppables import NeighborTrackerPrinterSteppable
neighborTrackerPrinterSteppable=NeighborTrackerPrinterSteppable(sim,100)
steppableRegistry.registerSteppable(neighborTrackerPrinterSteppable)

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

NeighborTracker printouts

CompuCell3D

```
*********NEIGHBORS OF CELL WITH ID  189  *********
Medium commonSurfaceArea= 6
neighbor.id 165  commonSurfaceArea= 6
neighbor.id 179  commonSurfaceArea= 3
neighbor.id 181  commonSurfaceArea= 2
neighbor.id 187  commonSurfaceArea= 3
*********NEIGHBORS OF CELL WITH ID  190  *********
neighbor.id 142  commonSurfaceArea= 4
neighbor.id 144  commonSurfaceArea= 4
neighbor.id 154  commonSurfaceArea= 3
neighbor.id 155  commonSurfaceArea= 5
neighbor.id 156  commonSurfaceArea= 13
neighbor.id 170  commonSurfaceArea= 3
neighbor.id 172  commonSurfaceArea= 9
neighbor.id 175  commonSurfaceArea= 1
*********NEIGHBORS OF CELL WITH ID  191  *********
neighbor.id 143  commonSurfaceArea= 17
neighbor.id 153  commonSurfaceArea= 3
neighbor.id 165  commonSurfaceArea= 1
neighbor.id 169  commonSurfaceArea= 3
neighbor.id 171  commonSurfaceArea= 6
neighbor.id 180  commonSurfaceArea= 1
neighbor.id 181  commonSurfaceArea= 7
neighbor.id 182  commonSurfaceArea= 2
*********NEIGHBORS OF CELL WITH ID  192  *********
Medium commonSurfaceArea= 9
neighbor.id 169  commonSurfaceArea= 2
neighbor.id 182  commonSurfaceArea= 7
neighbor.id 183  commonSurfaceArea= 3
neighbor.id 193  commonSurfaceArea= 3
*********NEIGHBORS OF CELL WITH ID  193  *********
Medium commonSurfaceArea= 12
neighbor.id 183  commonSurfaceArea= 8
neighbor.id 186  commonSurfaceArea= 3
neighbor.id 192  commonSurfaceArea= 3
SAVING: cellsort_2D_neighbor_tracker.xml.01000.png
```

cellsort_2D_neighbor_tracker.xml - Application

File Simulation Zoom Show Configure Help

Cross Section: 3D xy 0 xz 50 yz

MC Step: 1000

Printing values of the concentration to a file

```
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
CompuCellSetup.setSimulationXMLFileName("examples_PythonTutorial/diffusion/diffusion_2D.xml")
sim,simthread = CompuCellSetup.getCoreSimulationObjects()

#Create extra player fields here or add attributes
CompuCellSetup.initializeSimulationObjects(sim,simthread)

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

from cellsort_2D_steppables import ConcentrationFieldDumperSteppable
concentrationFieldDumperSteppable=ConcentrationFieldDumperSteppable(sim,_frequency=100)
steppableRegistry.registerSteppable(concentrationFieldDumperSteppable)

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

```
class ConcentrationFieldDumperSteppable(SteppableBasePy):
  def __init__(self,_simulator,_frequency=1):
    SteppableBasePy.__init__(self, _simulator_frequency)
    self.fieldName=“FGF”

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

      for I,j,k in self.everyPixel():
        print >>fileHandle,I,” ”,j,” ”,k,” “, field[I,j,k] #write to a file
```

Creating, initializing and manipulating a concentration field directly from Python

- Although in most cases concentration fields are created and manipulated by PDE solvers it is possible to accomplish all those tasks directly from Python.

- This can be very useful if you want to develop custom visualization that is not directly supported by the Player. For example you may want to color cells according to how many neighbors they have. Player does not offer such an option but you can implement it very easily in Python in less then 5 minutes. This is not a joke. I am sure that by combining two examples from this tutorial you will accomplish this task very fast.

The task of adding extra field to the Player and "managing" it consist of two steps

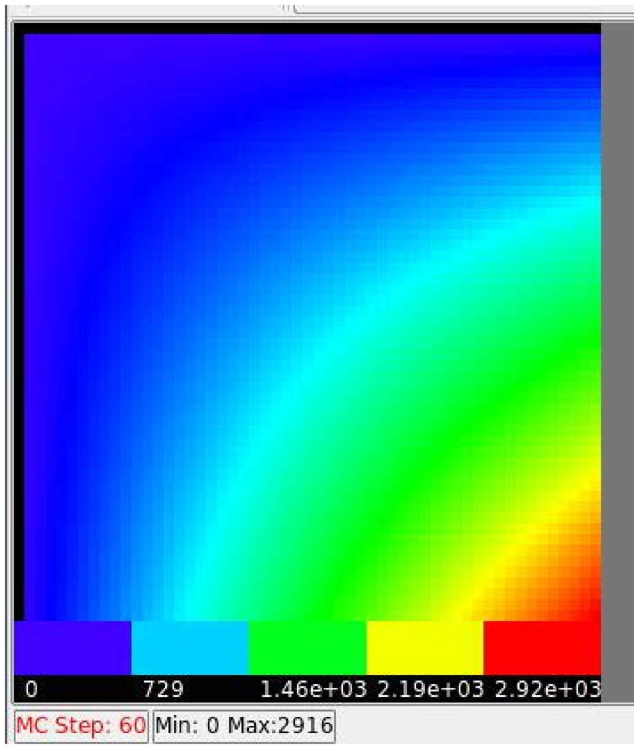
- Creating extra field and registering it with the Player and CompuCell3D kernel
- Writing steppable that manipulates values stored in the field

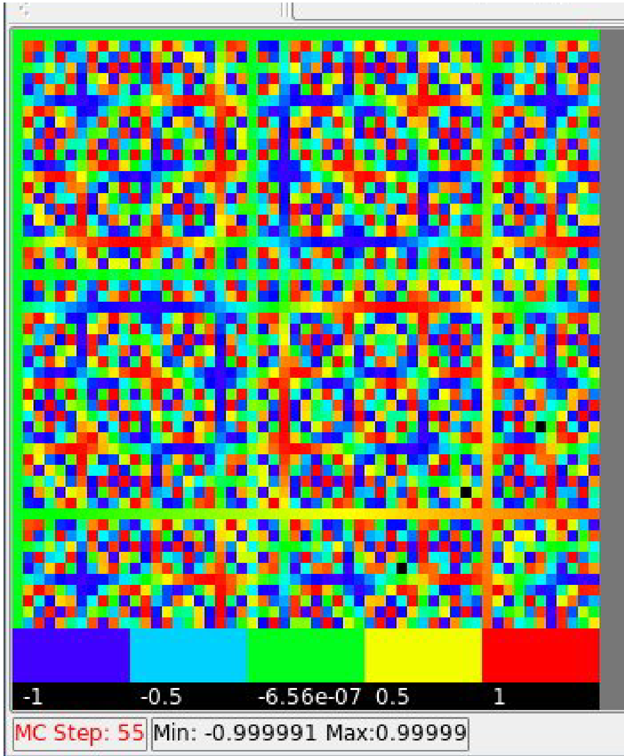
First let's look at the full listing:

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

sys.path.append(environ["PYTHON_MODULE_PATH"])
import SystemUtils
SystemUtils.initializeSystemResources()

import CompuCellSetup
sim,simthread = CompuCellSetup.getCoreSimulationObjects()
import CompuCell #notice importing CompuCell to main script has to be done after call to
sim,simthread =getCoreSimulationObjects()

#Create extra player fields here or add attributes
CompuCellSetup.initializeSimulationObjects(sim,simthread)

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

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

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

```
from PlayerPython import * # necessary to manipulate Player fields
from math import * # getting access to special functions from math module

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

  def setScalarField(self,_field): # getting access to newly created field
    self.scalarField=_field


  def step(self,mcs):
    for x,y,z in self.everyPixel: #iteration over each pixel
      if (not mcs%20): #filling the values of the concentration
        self.scalarField[x,y,z]=x*y # sometimes it is x*y
      else:
        self.scalarField[x,y,z]=sin(x*y) # sometimes it is x*y
```

Managing concentration field from Python - results





c(x,y)=x*y

c(x,y)=sin(x*y)

Mitosis in CompuCell3D simulations

Supporting cell division (mitosis) in CompuCell3D simulations is a prerequisite for building faithful biomedical simulations.

You can use mitosis module (Mitosis Plugin) directly from XML however, its use will be very limited because of the following fact:

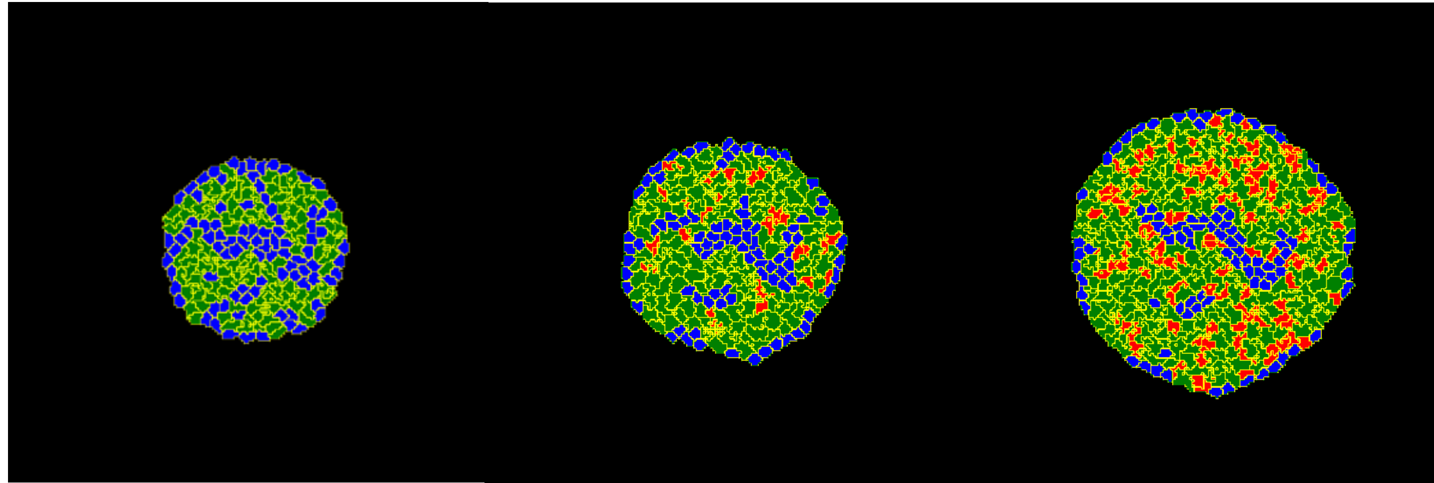
After cell division you end up with two cells. **What parameters should those two cells have (type, target volume etc.)? How do you modify the parameters?**

The best solution is to manage mitosis from Python and the example below will explain you how to do it.

There are two ways to implement mitosis – as a plugin or as a steppable. On older versions we have used plugin-based approach as this was the only option. However steppable based approach is much simpler to implement and we will focus on it first.

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

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

import CompuCellSetup

sim,simthread = CompuCellSetup.getCoreSimulationObjects()

CompuCellSetup.initializeSimulationObjects(sim,simthread)

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

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

from cellsort_2D_field_modules import VolumeConstraintSteppable
volumeConstraint=VolumeConstraintSteppable(sim)
steppableRegistry.registerSteppable(volumeConstraint)

from cellsort_2D_field_modules import MitosisSteppable
mitosisSteppable=MitosisSteppable(sim,1)
steppableRegistry.registerSteppable(mitosisSteppable)

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

Mitosis example results



t=200 MCS t=600 MCS t=1000 MCS

"Green" cells grow in response to diffusing FGF. Once they reach doubling volume they divide. They have 50% probability of differentiating into "red" cells. After 1500 MCS we gradually decrease target volume of each cell, effectively killing them.

Mitosis is implemented as a steppable class MitosisSteppable which inherits from MitosisSteppableBase

```
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:
            self.divideCellRandomOrientation(cell)

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

        parentCell.targetVolume/=2.0
        childCell.targetVolume=parentCell.targetVolume
        childCell.lambdaVolume=parentCell.lambdaVolume
        if (random()<0.5):
            childCell.type=parentCell.type
        else:
            childCell.type=3
```

Adding mitosis history to parent and child cells:

#Mitosis data has to have base class "object" otherwise if cell will be deleted CC3D
#may crash due to improper garbage collection

```
class MitosisData(object):
    def __init__(self, _MCS=-1, _parentId=-1, _parentType=-1,\
                    _offspringId=-1, _offspringType=-1):
        self.MCS=_MCS
        self.parentId=_parentId
        self.parentType=_parentType
        self.offspringId=_offspringId
        self.offspringType=_offspringType
    def __str__(self):
        return "Mitosis time="+str(self.MCS)+" parentId="\
                +str(self.parentId)+" offspringId="+str(self.offspringId)

def updateAttributes(self):
        parentCell=self.mitosisSteppable.parentCell
        childCell=self.mitosisSteppable.childCell
         .....
        #get a reference to lists storing Mitosis data
        parentCellDict=self.getDictionaryAttribute(parentCell)
        childCellDict=self.getDictionaryAttribute(childCell)
        ##will record mitosis data in parent and offspring cells
        mcs=self.simulator.getStep()
        mitData=MitosisData(mcs,parentCell.id,parentCell.type,childCell.id,childCell.type)
        parentCellDict[str(mcs)]=mitData
        childCellDict[str(mcs)]=mitData
```

Cell-attributes revisited - **VERY IMPORTANT (probably not relevant with 3.7.0)**:

- In the previous example – *examples_PythonTutorial/cellsort_2D_extra_attrib* - we were attaching integers as additional cell attributes.
- If we define our own class in the most straightforward way and try to append objects of this class to the list of attributes of a cell, it may happen that CompuCell3D will crash when such cell gets deleted (e.g. in a simulations where some of the cells die).
- It turns out that with exception of Python "core" objects such as integers or floating point numbers adding user defined class which DOES NOT inherit from Python "object" leads to improper garbage collection, and this causes overall CC3D crash
- The solution: Always inherit from "object" when you want to add objects of your custom class as cell attribute:

```
class CellPosition(object):
    def __init__(self,_x=0, _y=0, _z=0):
        self.x=_x
        self.y=_y
        self.z=_z
```

Directional Mitosis

So far we have divided cells using randomly chosen division axis/plane:

```
class MitosisSteppable(MitosisSteppableBase):
    .......

    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:
            self.divideCellRandomOrientation(cell)
```

However MitosisSteppableBase, and consequently any class which inherits from MitosisSteppableBase, allows additional modes of division:

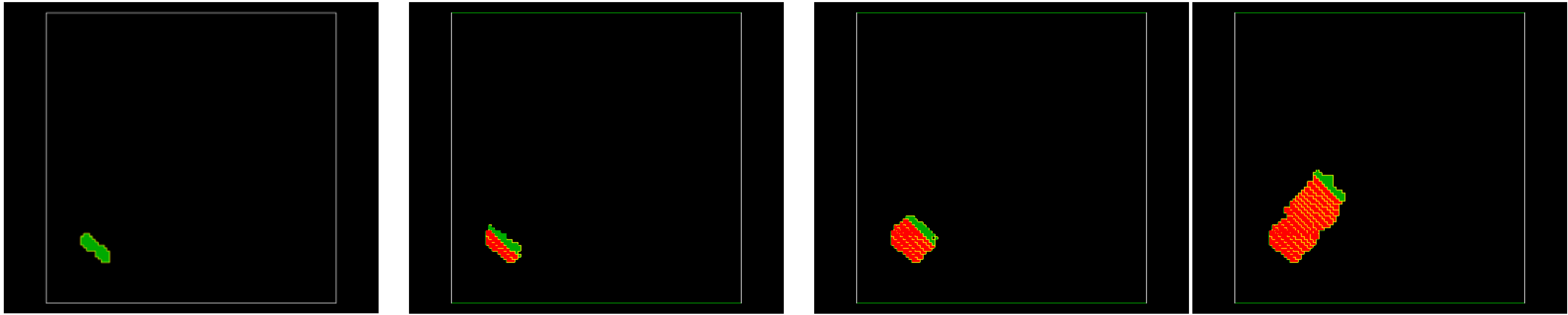
- Along major axis/plane of a cell
- Along minor axis/plane of a cell
- Along user specified axis/plane of the cell

By changing one function name in the Mitosis Steppable we can cause cells to divide along e.g. Major axis:

```
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:
            self. divideCellAlongMajorAxis(cell)
            # self.divideCellOrientationVectorBased(cell,1,1,0)
            # self. divideCellAlongMajorAxis(cell)
```

Commented lines show addition options in choosing orientation of division axis. Notice that when specifying user-defined orientation we simply specify vector along which the division will take place. The vector does not have to be normalized

Results of dividing cells along major axis



Try running *examples_PythonTutorial/steppableBasedMitosis* example and change division axis to major axis. Do you get similar results as the one shown above?

What do you have to modify to achieve similar picture?

Hint: look at *examples_PythonTutorial/growingcells_fast/growingcells_fast_directional.xml*

Plugin-based mitosis (deprecated)

- Several next slides demonstrate how to implement mitosis as a plugin (lattice monitor)

- The simulation results are “statistically” the same as in the case of steppable-based mitosis however:

- Plugin based mitosis is more complicated to implement, runs slower, is more error-prone, obsolete, does not run properly with periodic boundary conditions and **makes it significantly harder to implement custom mitosis-triggering events/conditions.**

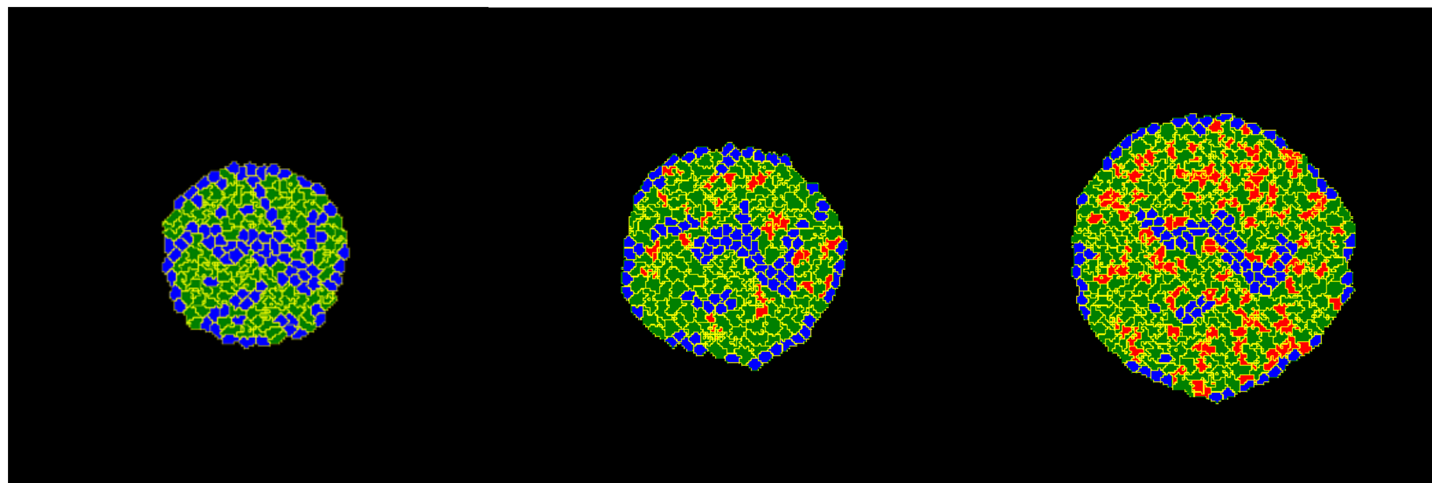
. If you have a working simulation which uses plugin-based mitosis you do not need to change it. Simply be aware that CC3D has more user-friendly way to implement the mitosis

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

sys.path.append(environ["PYTHON_MODULE_PATH"])
import CompuCellSetup
CompuCellSetup.setSimulationXMLFileName("Demos/cellsort_2D_growing_cells_mitosis/cellsort_2D_field.xml")
sim,simthread = CompuCellSetup.getCoreSimulationObjects()

#add additional attributes
pyAttributeAdder,listAdder=CompuCellSetup.attachListToCells(sim)

CompuCellSetup.initializeSimulationObjects(sim,simthread)

import CompuCell #notice importing CompuCell to main script has to be done after call to
getCoreSimulationObjects()
changeWatcherRegistry=CompuCellSetup.getChangeWatcherRegistry(sim)
stepperRegistry=CompuCellSetup.getStepperRegistry(sim)

from cellsort_2D_field_modules import CellsortMitosis
cellsortMitosis=CellsortMitosis(sim,changeWatcherRegistry,stepperRegistry)
cellsortMitosis.setDoublingVolume(50)

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

from cellsort_2D_field_modules import VolumeConstraintSteppable
volumeConstraint=VolumeConstraintSteppable(sim)
steppableRegistry.registerSteppable(volumeConstraint)

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

Mitosis function is a type of plugin that monitors lattice (field3DWatcher). Most of the mitosis setup is handled inside base class **MitosisPyPluginBase**

```
from random import random
from PyPluginsExamples import MitosisPyPluginBase

class CellsortMitosis(MitosisPyPluginBase): #inherit base class

  def __init__(self , _simulator , _changeWatcherRegistry , _stepperRegistry):
    MitosisPyPluginBase.__init__(self,_simulator,_changeWatcherRegistry,_stepperRegistry)

  def updateAttributes(self): #called after mitosis is done
    self.parentCell.targetVolume/=2.0

    self.childCell.targetVolume=self.parentCell.targetVolume
    self.childCell.lambdaVolume=self.parentCell.lambdaVolume

    if (random()<0.5):
      self.childCell.type=self.parentCell.type
    else:
      self.childCell.type=3
```

Mitosis example results



t=200 MCS t=600 MCS t=1000 MCS

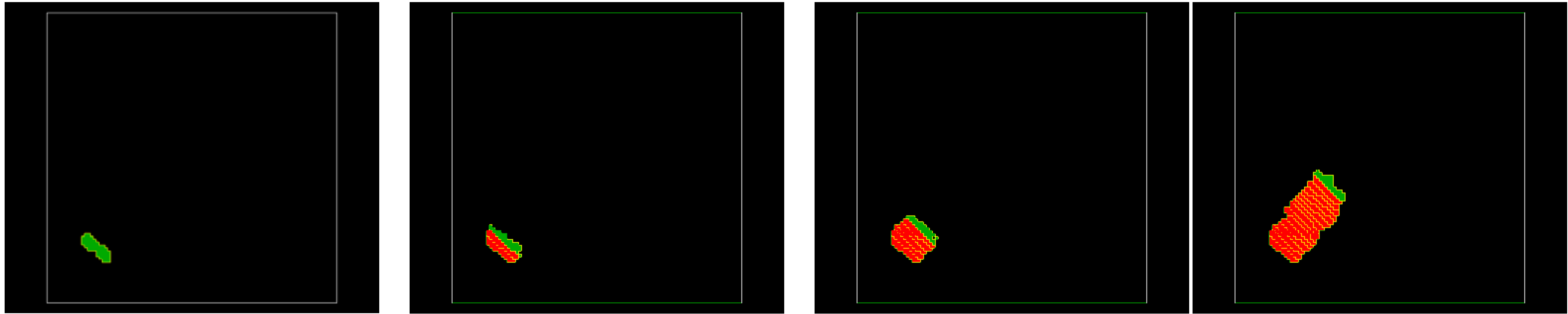
“Green” cells grow in response to diffusing FGF. Once they reach doubling volume they divide. They have 50% probability of differentiating into “red” cells.

Directional Mitosis

By default mitosis will split parent cells into two cells in a somewhat random fashion. If you need cell division that is carried out along a specified orientation axis you need to do a small modification to the mitosis plugin:

```
from PyPlugins import *
from PyPluginsExamples import MitosisPyPluginBase
class MitosisPyPlugin(MitosisPyPluginBase):
  def __init__(self , _simulator , _changeWatcherRegistry , _stepperRegistry):
    MitosisPyPluginBase.__init__(self,_simulator,_changeWatcherRegistry, _stepperRegistry)
    self.setDivisionAlongMajorAxis()
  def updateAttributes(self):
    self.parentCell.targetVolume=50.0
    self.childCell.targetVolume=self.parentCell.targetVolume
    self.childCell.lambdaVolume=self.parentCell.lambdaVolume
```

Results of dividing cells along major axis



Directional Mitosis with more customization

Sometimes you may want to have more control over mitosis. For example you may want to have cells of certain type to undergo mitosis or each time the mitosis is run you may want to specify cell division axis. Here is how you do it:

```
from PyPlugins import *
from PyPluginsExamples import MitosisPyPluginBase
class MitosisPyPlugin(MitosisPyPluginBase):
  def __init__(self , _simulator , _changeWatcherRegistry , _stepperRegistry):
    MitosisPyPluginBase.__init__(self,_simulator,_changeWatcherRegistry, _stepperRegistry)
  def field3DChange(self):
    if self.changeWatcher.newCell and self.changeWatcher.newCell.type==2 and \
    self.changeWatcher.newCell.volume>self.doublingVolume:
      self.mitosisPlugin.field3DChange(self.changeWatcher.changePoint, \
      self.changeWatcher.newCell, \
      self.changeWatcher.newCell)
      self.mitosisFlag=1
      self.setMitosisOrientationVector(1, self.changeWatcher.newCell.type*2,0)
  def updateAttributes(self):
    self.parentCell.targetVolume=50.0
    self.childCell.targetVolume=self.parentCell.targetVolume
    self.childCell.lambdaVolume=self.parentCell.lambdaVolume
   self.unsetMitosisOrientationVector()
```

Mitosis was our first example of a plugin implemented in Python. We can implement other plugins for example energy function in Python as well:

```
class VolumeEnergyFunctionPlugin(EnergyFunctionPy):

  def __init__(self,_energyWrapper):# proper initialization
    EnergyFunctionPy.__init__(self)
    self.energyWrapper=_energyWrapper
    self.vt=0.0
    self.lambda_v=0.0
  def setParams(self,_lambda,_targetVolume):# configuration of the plugin
    self.lambda_v=_lambda;
    self.vt=_targetVolume
  def changeEnergy(self): # core function of energy function plugin
    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
```

Full script:

```
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
CompuCellSetup.setSimulationXMLFileName\
("examples_PythonTutorial/cellsort_2D_with_py_plugin/cellsort_2D_py_plugin.xml")
sim,simthread = CompuCellSetup.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)
```

XML file

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

<!--Notice we eliminated Volume plugin but need to keep VolumeTracker Plugin→
<Plugin Name="VolumeTracker"/>

<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>
   …
</Plugin>

<Steppable Type="BlobInitializer">
  <Gap>0</Gap>
  <Width>5</Width>
  …
</Steppable>

</CompuCell3D>
```

Simulation Steering

•By steering we mean the ability to change *any* simulation parameter while the simulation is running

•Steering is essential to build realistic simulations because biological parameters do vary in time.

•Primitive way of steering would be to run a simulation stop it change parameters restart the simulation and so on.

•Starting with 3.2.1 version of CompuCell3D we can implement much more convenient way to steer the simulation. It requires developing simple steppable where we change simulation parameters.

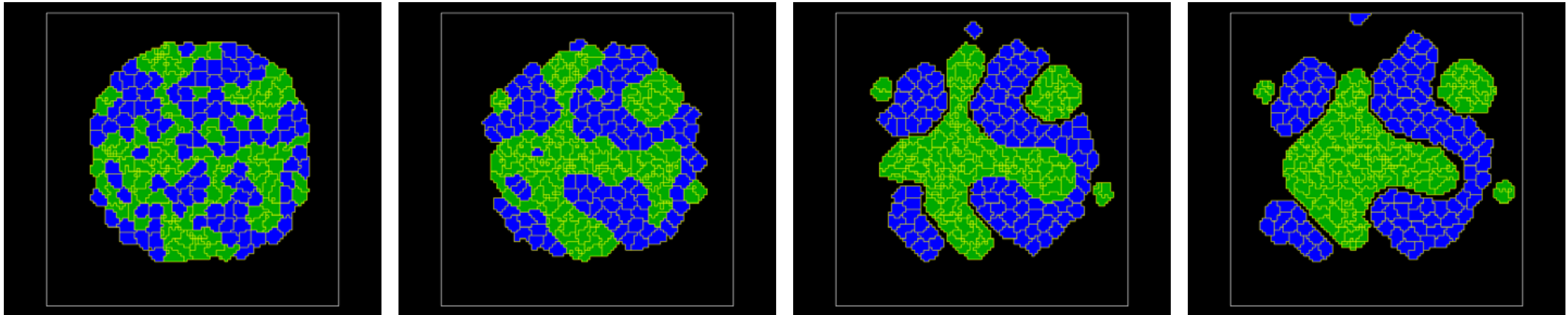
•Notice even with earlier versions of CompuCell3D you had an opportunity to partially steer the simulation whenever you were using, for example, VolumeLocalFlex, SurfaceLocalFlex or ContactLocalFlex plugins

Let's take a look at what is needed to have steerable CompuCell3D simulation

Simplest steering steppable – will increase contact energy between Condensing and NonCondensing cells by 1 unit every 10 MCS:

```
class ContactSteering(SteppablePy):
  def __init__(self,_simulator,_frequency=10):
    SteppablePy.__init__(self,_frequency)
    self.simulator=_simulator
  def step(self,mcs):
    # get <Plugin Name="Contact"> section of XML file
     contactXMLData=self.simulator.getCC3DModuleData("Plugin","Contact")
    # check if we were able to successfully get the section from simulator
    if contactXMLData:
      # get <Energy Type1="NonCondensing" Type2="Condensing"> element
      energyNonCondensingCondensingElement=contactXMLData.\
      getFirstElement("Energy",d2mss({"Type1":"NonCondensing","Type2":"Condensing"}))
      # check if the attempt was succesful
      if energyNonCondensingCondensingElement:
        # get value of the <Energy Type1="NonCondensing" Type2="Condensing"> element  and
        #convert it into float
        val=float(energyNonCondensingCondensingElement.getText())
        # increase the value by 1.0
        val+=1.0
        # update <Energy Type1="NonCondensing" Type2="Condensing"> element remembering
        #about converting the value back to string
        energyNonCondensingCondensingElement.updateElementValue(str(val))
     # finally call simulator.updateCC3DModule(contactXMLData) to tell simulator to update
     # model parameters - this is actual steering
    self.simulator.updateCC3DModule(contactXMLData)
```

Results of the simulation



Note: the ContactSteering steppable might a bit convoluted at first sight. However if we eliminate consistency checks we can write it in more compact form:

```
def step(self,mcs):

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

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

   val=float(energyNonCondensingCondensingElement.getText())
   val+=1.0

   energyNonCondensingCondensingElement.updateElementValue(str(val))

   self.simulator.updateCC3DModule(contactXMLData)
```

Steering Checklist

- Create Steering Steppable.
- In the step funuction implement your steering algorithm:
    - obtain XML data structure for module you wish to steer:
      contactXMLData=self.simulator.getCC3DModuleData("Plugin","Contact")
    - obtain and update XML parameter values
    - Make sure to update steered module

      self.simulator.updateCC3DModule(contactXMLData)

See more steering examples in the Demos directory

Steering Bacterium-Macrophage simulation – periodically (every 100 MCS) decrease chemotaxis constant for Macrophage. Macrophage will be attracted to secreting bacterium and then it will become repealed

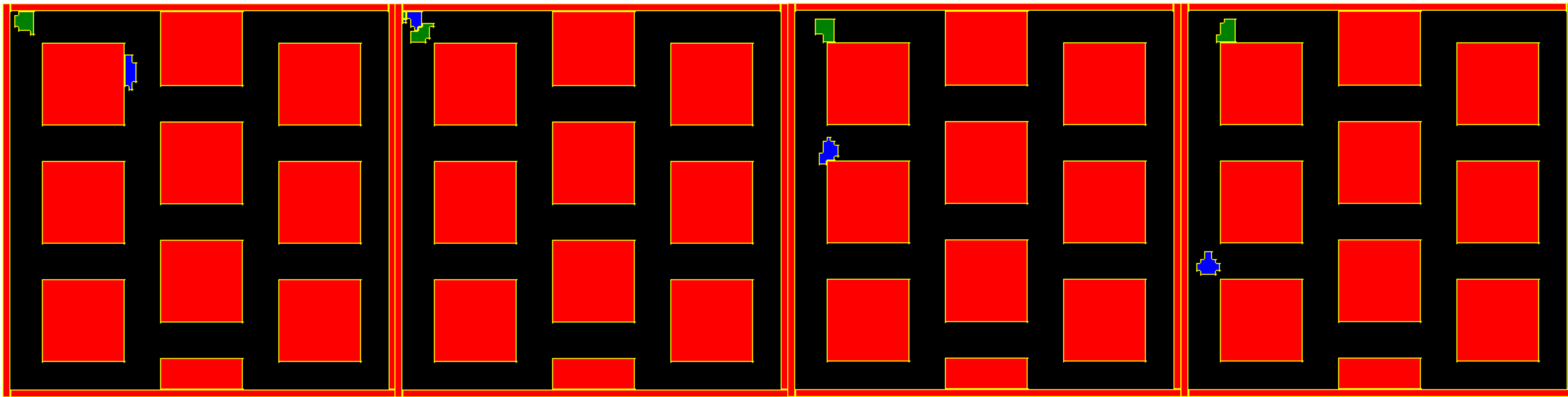
Steering Steppable

```
from XMLUtils import dictionaryToMapStrStr as d2mss


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

  def step(self,mcs):
    if mcs>100 and not mcs%100:
      chemicalField=chemotaxisXMLData.getFirstElement\
      ("ChemicalField",d2mss({"Source":"FlexibleDiffusionSolverFE", "Name":"ATTR"}))

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

      lambdaVal=chemotaxisByTypeMacrophageElement.getAttributeAsDouble("Lambda")

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

      self.simulator.updateCC3DModule(chemotaxisXMLData)
```

Let’s not forget to instantiate and register the newly created steppable:

```
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 bacterium_macrophage_2D_steering_steppables import ChemotaxisSteering
cs=ChemotaxisSteering(sim,100)
steppableRegistry.registerSteppable(cs)

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

Steerable Bacterium-Macrophage Simulation - Screenshots:



t=700 *MCS* | t=900 *MCS* | t=1400 *MCS* | t=1700 *MCS*

Macrophage is first attracted and then repealed from the bacterium. This behavior is altered during simulation runtime.

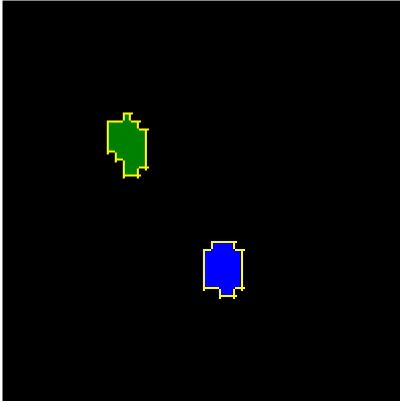
Steering LengthConstraint plugin:

```
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 LengthConstraintSteering
lcs=LengthConstraintSteering(sim,100)
steppableRegistry.registerSteppable(lcs)

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

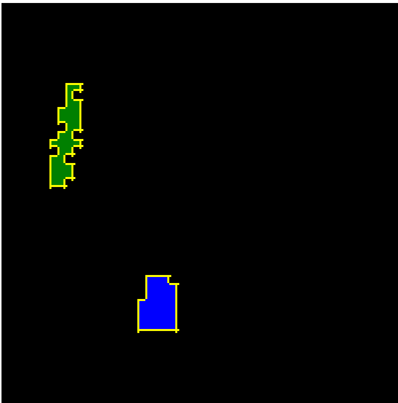
Steerable steppable will modify LengthConstraint and Connectivity plugins data:

```
class LengthConstraintSteering(SteppablePy):

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

  def step(self,mcs):

    if mcs>100 and not mcs%100:
      lengthConstraintXMLData=self.simulator.getCC3DModuleData("Plugin","LengthConstraint")
      lengthEnergyParametersBody1=lengthConstraintXMLData.\
      getFirstElement("LengthEnergyParameters",d2mss({"CellType":"Body1"}))
      targetLength=lengthEnergyParametersBody1.getAttributeAsDouble("TargetLength")

      targetLength+=0.5

      lengthEnergyParametersBody1.\
      updateElementAttributes(d2mss({"TargetLength":str(targetLength)}))
      self.simulator.updateCC3DModule(lengthConstraintXMLData)

    if mcs>3000:
       connectivityXMLData=self.simulator.getCC3DModuleData("Plugin","Connectivity")
       penaltyElement=connectivityXMLData.getFirstElement("Penalty")
        penaltyElement.updateElementValue(str(0))
        self.simulator.updateCC3DModule(connectivityXMLData)
```

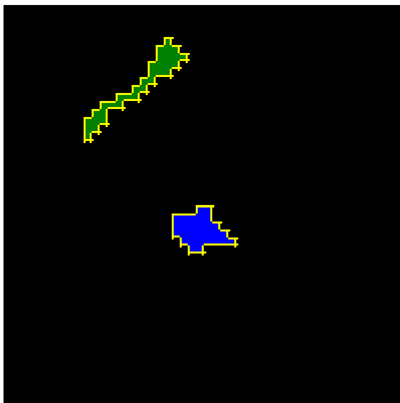
Screenshots of steerbaly length constraint simulations:



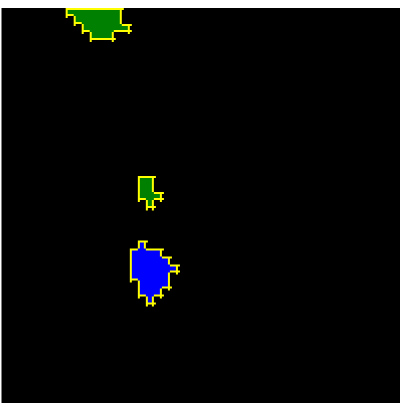
t=500 *MCS*



t=1500 *MCS*



t=2500 *MCS*



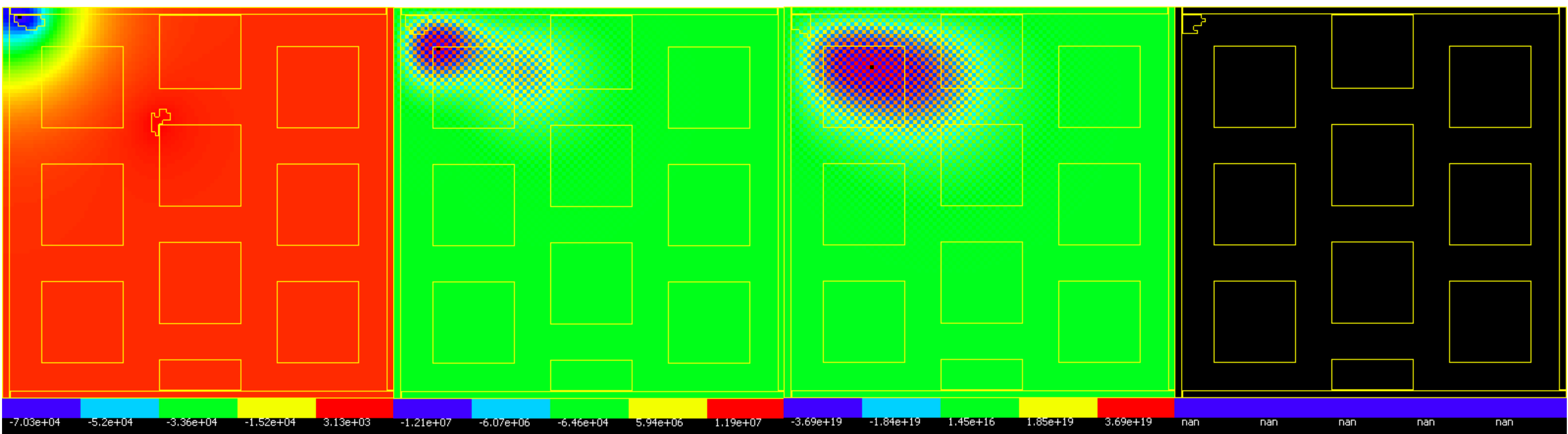
t=3200 *MCS*

Connectivity constraint released but length constraint still present. Green cell fragments into two pieces satisfying moment of inertia constraint of the LengthConstraint plugin

Steering PDE solver parameters

```
class DiffusionSolverSteering(SteppablePy):
  def __init__(self,_simulator,_frequency=100):
    SteppablePy.__init__(self,_frequency)
    self.simulator=_simulator
  def step(self, mcs):
    if mcs>100:
      flexDiffXMLData=self.simulator.getCC3DModuleData("Steppable","FlexibleDiffusionSolverFE")
      diffusionFieldsElementVec=CC3DXMLListPy(flexDiffXMLData.getElements("DiffusionField"))
        for diffusionFieldElement in diffusionFieldsElementVec:
          if diffusionFieldElement.getFirstElement("DiffusionData").\
          getFirstElement("FieldName").getText()=="FGF":
            diffConstElement=diffusionFieldElement.\
            getFirstElement("DiffusionData").getFirstElement("DiffusionConstant")
            diffConst=float(diffConstElement.getText())
            diffConst+=0.01
            diffConstElement.updateElementValue(str(diffConst))
            if mcs>500:
              secretionElement=diffusionFieldElement.getFirstElement("SecretionData").\
              getFirstElement("Secretion",d2mss({"Type":"Bacterium"}))
              secretionConst=float(secretionElement.getText())
              secretionConst+=2
              secretionElement.updateElementValue(str(secretionConst))
        self.simulator.updateCC3DModule(fiexDiffXMLData)
```

The result …



t=1000 *MCS*

t=1600 *MCS*

t=1700 *MCS*

t=1800 *MCS*

Numerical instabilities in Forward-Euler solver

When diffusion constant gets too large instabilities arise. The solution to this is to either use implicit solvers (might be time consuming) or use PDESolverCaller plugin which calls numerical algorithm user-specified times per MCS:

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

  def step(self,mcs):
    if _mcs>100 and not _mcs%100
      pdeCallerXMLData=self.simulator.getCC3DModuleData("Plugin","PDESolverCaller")

      callPDEXMLElement=pdeCallerXMLData.getFirstElement\
      ("CallPDE",d2mss({"PDESolverName":"FlexibleDiffusionSolverFE"}))

      extraTimesPerMC=callPDEXMLElement.getAttributeAsInt("ExtraTimesPerMC")
      extraTimesPerMC+=1

      callPDEXMLElement.updateElementAttributes\
     (d2mss({"ExtraTimesPerMC":str(extraTimesPerMC)}))

      self.simulator.updateCC3DModule(pdeCallerXMLData)
```

Dealing with diffusion constants – especially when they are too large…

- Diffusion constant has units of m^2/s
- Pick lattice spacing and the time scale i.e. how many seconds/MCS

$$\frac{\partial c}{\partial t} = D\frac{\partial^2 c}{\partial x^2} \quad \rightarrow \quad c_j^{k+1} = c_j^k + D\left(\frac{\Delta t}{(\Delta x)^2}\right)\left(c_{j+1}^k - 2c_j^k + c_{j-1}^k\right)$$

$$D\left(\frac{\Delta t}{(\Delta x)^2}\right) < 0.16 - \text{in 3D}$$

If the last condition is not satisfied you will get instabilities. In such a case decrease Δt and use PDESolverCaller to call PDE solver extra times.

For example:

$$n = 2 \quad \Rightarrow \quad \Delta t \rightarrow \Delta t / 3$$

$$c_j^{k+1} = c_j^k + D\left(\frac{\Delta t / 3}{(\Delta x)^2}\right)\left(c_{j+1}^k - 2c_j^k + c_{j-1}^k\right) \quad \text{will be called extra 2 times per MCS}$$

If you don't decrease time step Δt you get approximate relation between ExtraTimesPerMCS and diffusion constant:

$$\frac{\partial c}{\partial t} = D\frac{\partial^2 c}{\partial x^2} \quad \rightarrow \quad c_j^{k+1} = c_j^k + D\left(\frac{\Delta t}{(\Delta x)^2}\right)\left(c_{j+1}^k - 2c_j^k + c_{j-1}^k\right)$$

fixed between multiple calls

$$c_j^{k+1} = c_j\left(t + \frac{\Delta t}{n+1}\right)$$

$$c_j^{k+2} = c_j\left(t + 2\frac{\Delta t}{n+1}\right)$$

...

$$c_j^{k+n+1} = c_j\left(t + (n+1)\frac{\Delta t}{n+1}\right) = c_j\left(t + \Delta t\right)$$

Call PDE solver many times per MCS:

$$c_j^{k+1} = c_j^k + D\left(\frac{\Delta t}{(\Delta x)^2}\right)\left(c_{j+1}^k - 2c_j^k + c_{j-1}^k\right)$$

$$c_j^{k+2} = c_j^{k+1} + D\left(\frac{\Delta t}{(\Delta x)^2}\right)\left(c_{j+1}^{k+1} - 2c_j^{k+1} + c_{j-1}^{k+1}\right)$$

...

$$c_j^{k+n+1} = c_j^{k+n} + D\left(\frac{\Delta t}{(\Delta x)^2}\right)\left(c_{j+1}^{k+n} - 2c_j^{k+n} + c_{j-1}^{k+n}\right)$$

Assuming

$$c_{j+1}^{k} \approx c_{j+1}^{k+1} \approx \ldots \approx c_{j+1}^{k+n+1} \quad , \quad c_{j}^{k} \approx c_{j}^{k+1} \approx \ldots \approx c_{j}^{k+n+1} \quad , \quad c_{j-1}^{k} \approx c_{j-1}^{k+1} \approx \ldots \approx c_{j-1}^{k+n+1}$$

Substitute recursively:

$$c_{j}^{k+n+1} \approx c_{j}^{k} + (n+1) D \left(\frac{\Delta t}{(\Delta x)^2} \right) \left(c_{j+1}^{k} - 2c_{j}^{k} + c_{j-1}^{k} \right) \quad \rightarrow \quad \frac{\partial c}{\partial t} = \left[(n+1) D \right] \frac{\partial^2 c}{\partial x^2}$$

where n is ExtraTimesPerMCS parameter

Calling PDE solver n extra times per MCS we effectively increase the diffusion constant $n+1$ times

The relation derived above is approximate at best (for small *n*).

The correct way to deal with large diffusion constants is to manipulate Δt, Δx and extraTimesPerMCS parameters or even better us implicit diffusion solver when accuracy is needed.

Δx - <DeltaX> -CC3DML or deltaX Python

Δt - <DeltaT> -CC3DML or deltaT Python

Steering using GUI

There is only a prototype…

Benefits of using Python:

1. Removes limitations of CC3DML
2. Gives access to 3 party libraries
3. Makes CompuCell3D simulation environment
4. Future GUI will be implemented using Python – users will be able to write their own control panels
5. Steering is done from Python level
6. Enables Model Sharing – cross platform (but be careful here)
7. Allows for easy extensibility of CompuCell3D – *e.g.* parameter sweeps, output of results, analysis of results etc.