

Python for CC3D v3.6.0 – Quick Reference Guide

Julio M. Belmonte, Maciej H. Swat

http://www.compuCell3d.org/BinDoc/cc3d_binaries/Manuals/Quick_Reference_Guide.pdf

List of cell properties:

Name	Command	Fixed?	Comments
Cell identity	cell.id	Yes	Unique number that identifies each cell in the simulation
Cell type	cell.type	No	Specifies which type of cell it is
Cell volume	cell.volume	Yes	Volume of the cell at a specific MCS
	cell.targetVolume	No	Target/average volume of the cell
	cell.lambdaVolume	No	Specifies the strength of volume constraint
Cell surface ¹	cell.surface	Yes	Surface area of the cell at a specific MCS
	cell.targetSurface	No	Target/average surface area of the cell
	cell.lambdaSurface	No	Specifies the strength of surface constraint
Center of mass ²	cell.xCOM	Yes	Cartesian coordinate x of the center of mass
	cell.yCOM	Yes	Cartesian coordinate y of the center of mass
	cell.zCOM	Yes	Cartesian coordinate z of the center of mass
Eccentricity ³	cell.ecc	Yes	Eccentricity of cell
Inertia tensor ³	cell.iXX	Yes	Moment of inertia around x-axis when rotated around x-axis
	cell.iYY	Yes	Moment of inertia around y-axis when rotated around y-axis
	cell.iZZ	Yes	Moment of inertia around z-axis when rotated around z-axis
	cell.iXY	Yes	Moment of inertia around x-axis when rotated around y-axis
	cell.iXZ	Yes	Moment of inertia around x-axis when rotated around z-axis
	cell.iYZ	Yes	Moment of inertia around y-axis when rotated around z-axis
Minor axis vector ³	cell.lX	Yes	Component x of the vector pointing along semiminor axis
	cell.lY	Yes	Component y of the vector pointing along semiminor axis
	cell.lZ	Yes	Component z of the vector pointing along semiminor axis
Directional force ⁴	cell.lambdaVecX	No	Force component acting on the x-direction
	cell.lambdaVecY	No	Force component acting on the y-direction
	cell.lambdaVecZ	No	Force component acting on the z-direction

¹ Only available when Surface , SurfaceTracker, SurfaceFlex OR SurfaceLocalFlex plugins are called.

² Only available when CenterOfMass plugin is called.

³ Only available when MomentOfInertia plugin is called.

⁴ Only available when ExternalPotential plugin is called.

How to call a class:

In the main Python file *<mainFile.py>*, after the first 2 lines shown below:

```
# Add Python steppables here
stepperRegistry=CompuCellSetup.getStepperRegistry(sim)

from <steppablesFile> import <NameOfClass>
<nameOfClass>=<NameOfClass>(_simulator=sim,_frequency=1)
steppableRegistry.registerSteppable(<nameOfClass>)
```

NOTE: The strings *<NameOfClass>* and *<nameOfClass>* should not be necessarily equal, but as a standard we name them equally with the former starting with an uppercase letter and the latter with a lowercase letter.

How to do a loop over the cells:

```
for cell in self.cellList:
    <commands for all cells, excluding Medium>
```

How to do a loop over the cells' neighbors:

```
for cell in self.cellList:
    cellNeighborList=self.getCellNeighbors(cell)
    for neighbor in cellNeighborList:
        <commands for all neighbors, including Medium, if present>
        if neighbor.neighborAddress:
            <commands for all neighbors, excluding Medium>
```

NOTE: Make sure to call NeighborTracker plugin from either the .xml or the .py file. Neighbor cells have the same properties listed before for cells. To access them, substitute cell by neighbor.neighborAddress.

How to do a loop over the lattice sites:

```
for x in range(self.dim.x):
    for y in range(self.dim.y):
        for z in range(self.dim.z):
            <commands>
```

How to do a loop over the cell's lattice sites:

```
pixelList=CellPixelList(self.pixelTrackerPlugin,cell)
for pixelData in pixelList:
    pt=pixelData.pixel
    <commands>
```

NOTE: Make sure to call PixelTracker plugin from either the .xml or the .py file.

How to do a loop over the cell's boundary sites:

```
boundaryPixelList=self.getCellBoundaryPixelList(cell)
for boundaryPixelData in boundaryPixelList:
    pt=boundaryPixelData.pixel
    <commands>
```

NOTE: Make sure to call the BoundaryPixelTracker plugin from either the .xml or the .py file.

How to attach/access/modify a dictionary to a cell:

In the main Python file *<mainFile.py>*, after the commented line shown below:

```
#Create extra player fields here or add attributes
pyAttributeAdder,dictAdder=CompuCellSetup.attachDictionaryToCells(sim)
```

In the steppable Python file:

```
<cellDict>=CompuCell.getPyAttrib(cell)
<cellDict>[<key1>]=0.1
<cellDict>[<key2>]=2
<variable>=cellDict[<key1>]*cellDict[<key2>] #result = 0.2
```

NOTE: *<key1>* and *<key2>* can be integers (1, 2,...), reals (0.1, 3.14, ...) or strings ("b1", "age", ...).

How to access/modify the cell lattice:

```
pt=CompuCell.Point3D() # defines a lattice vector
pt.x=3; pt.y=2; pt.z=0 # specifies its coordinates

# to access the cell (or Medium) occupying the point (3,2,0):
cell=self.cellField.get(pt) # or self.cellField.get(pt.x,pt.y,pt.z)

# to create an extension of that cell in another location:
pt.x=4; pt.y=6; pt.z=0
self.cellField.set(pt,cell)

# to create a brand new cell
pt.x=2; pt.y=8; pt.z=0
newcell=self.potts.createCellG(pt)
newcell.type=1 # don't forget to assign a type to the new cell
```

How to create a visualization field:

In the main Python file `<mainFile.py>`, after the 2 lines shown below, chose any of the 4 options:

```
# Create extra player fields here or add attributes
dim=sim.getPotts().getCellFieldG().getDim()

<VisField1>=simthread.createFloatFieldPy(dim,"<VisFieldScreenName1>")
<VisField2>=simthread.createScalarFieldCellLevelPy("<VisFieldScreenName2>")
<VisField3>=simthread.createVectorFieldPy(dim,"<VisFieldScreenName3>")
<VisField4>=simthread.createVectorFieldCellLevelPy("<VisFieldScreenName4>")

from <steppablesFile> import <NameOfFieldClass>
<nameOfFieldClass>=<NameOfFieldClass>(_simulator=sim,_frequency=1)
<nameOfFieldClass>.setScalarField(<VisField{1-2}>) #use this command for the first 2 choices
<nameOfFieldClass>.setVectorField(<VisField{3-4}>) #use this command for the last 2 choices
steppableRegistry.registerSteppable(<nameOfFieldClass>)
```

NOTE: As before, the strings `<NameOfFieldClass>` and `<nameOfFieldClass>` should not be necessarily equal, but as a standard we name the former starting with an uppercase letter and the latter with a lowercase letter.

In the appropriate class in the steppable Python file, the way to write into the visualization field is:

```
# for option 1
fillScalarValue(self.scalarField,x,y,z,<value>)
# for option 2
fillScalarValueCellLevel(self.scalarField,cell,<value>)
# for option 3
insertVectorIntoVectorField(self.vectorField,x,y,z,<valueX>,<valueY>,<valueZ>)
# for option 4
insertVectorIntoVectorCellLevelField(self.vectorField,cell,<valueX>,<valueY>,<valueZ>)
```

How to access/modify PDE field's values:

```
<fieldName>=CompuCell.getConcentrationField(self.simulator,"<fieldName>")
...
pt=CompuCell.Point3D() # defines a lattice vector
pt.x=3; pt.y=2; pt.z=0
<fieldName>.get(pt) # extracts the field value at point 'pt'
<fieldName>.set(pt,10) # sets the field at point 'pt' to value of 10
```

How to write output files inside a python class:

```
...
def start(self):
    self.<File>=open(<FileName.dat>,"w") # creates a new file
    # self.<File>=open(<FileName>,"a") # appends an existing file

def step(self,mcs):
    ...
    self.<File>.write("%s" % (str(mcs)+", "+str(<variable>))) # writes a line of strings
    self.<File>.write("%d" % (mcs) ) # writes a line of integers
    self.<File>.write("%f %f" % (mcs*0.1,<variable>) ) # writes a line of 2 reals
    self.<File>.write("%f\n" % (3.1416) ) # '\n' creates a new line

def finish(self):
    self.<File>.close() # close the file
```

NOTE: By default all files opened this way are stored in the `CompuCell3D\` directory unless a complete path is given in the string `<FileName.dat>`.

To save the file in the same directory as the output screenshots use the commands:

```
import CompuCellSetup
self.<File>=open(CompuCellSetup.getScreenshotDirectoryName() + <FileName.dat>,"w")
```

Make sure that the CompuCell Player has the option "Save Image every Nth MCS" checked.

How to load and run a subcellular SBML models:

In some class in the steppable Python file, add the indicated commands:

```
import bionetAPI # Import bionetAPI functions
class <someClass>(SteppableBasePy):
    def __init__(self, _simulator, _frequency=1):
        SteppableBasePy.__init__(self, _simulator, _frequency)
        bionetAPI.initializeBionetworkManager(self.simulator) # Initialize bionet inside class

    def start(self):
        # Load a specific subcellular SBML submodel
        ModelName = <sbmlModelName> # Name of the model
        ModelPath = <sbmlModelPath> # Path where the model is stored
        ModelKey = <modelKey> # Nickname of the model
        IntegrationStep = <timeStep> # Time step of integration
        bionetAPI.loadSBMLModel( ModelName, ModelPath, ModelKey, IntegrationStep )

        # Add SBML submodel to a group of cells or a single cell
        bionetAPI.addSBMLModelToTemplateLibrary(<sbmlModelName>, {<cellType> or <cellId>})
        ...
        # Modify the parameter value or molecular concentration of a cell (or group of cells)
        bionetAPI.setBionetworkValue(<molecule/parameter>, <value>, {<cellType> or <cellId>})
        ...
        # Initialize model
        bionetAPI.initializeBionetworks()

    def step(self, mcs):
        # Iterate the model (run it for the time step specified on the load command)
        bionetAPI.timestepBionetworks()
        ...
        # Get the parameter value or molecular concentration from a cell (or group of cells)
        <var>=bionetAPI.getBionetworkValue({<parameter> or <molecule>}, {<cellType> or <cellId>})
        ...
        # Modify the parameter value or molecular concentration of a cell (or group of cells)
        bionetAPI.setBionetworkValue(<molecule/parameter>, <value>, {<cellType> or <cellId>})
```

NOTE: The way to refer to parameters/molecules is by using the string “<sbmlModelName>_<variableName>” or “<modelKey>_<variableName>”.

If using mitosis, add the following commands in the `updateAttributes` function:

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

    # Make sure to assign a type to the child cell
    childCell.type=parentCell.type

    # Add this line to copy the current state of parentCell to childCell
    bionetAPI.copyBionetworkFromParent(parentCell, childCell)
```

NOTE: There is no need to call `bionetAPI.initializeBionetworkManager(self.simulator)` command inside this class.