CompuCell3D Reference Manual

Version 3.7.4

Maciej H. Swat, Julio Belmonte, Randy W. Heiland, Benjamin L. Zaitlen, James A. Glazier, Abbas Shirinifard

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

1 CC3DML Syntax of CompuCell3D modules

This CompuCell3D reference material provides users with fairly detaild description of CC3DML syntax (CC3DML is XML-based model description format) of CC3D modules. The presented material is indended for users who are already familiar with CompuCell3D basics and know how to build and run simple simulations such as cellsorting, bacterium macrophage or cell-type-oscillator. Since we often show CC3DML syntax and accompanying Python syntax for CC3D scripting we assume that users are familiar with Python. For readers who are using CompuCell3D for the first time we strongly recommend reading "Introduction To CompuCell3D". Complete description of CC3D Python scripting can be found in "CC3D Python Scripting Manual". Both manuals are available from www.compucell3d.org or at your nearest bookstore.

1.1 Potts Section

The first section of the .xml file defines the global parameters of the lattice and the simulation.

```
<Potts>
<Dimensions x="101" y="101" z="1"/>
<Anneal>0</Anneal>
<Steps>1000</Steps>
<FluctuationAmplitude>5</ FluctuationAmplitude >
<Flip2DimRatio>1</Flip2DimRatio>
<Boundary_y>Periodic</Boundary_y>
<Boundary_x>Periodic</Boundary_x>
<NeighborOrder>2</NeighborOrder>
<DebugOutputFrequency>20</DebugOutputFrequency>
<RandomSeed>167473</RandomSeed>
  <EnergyFunctionCalculator Type="Statistics">
     <OutputFileName Frequency="10">statData.txt</OutputFileName>
      <OutputCoreFileNameSpinFlips Frequency="1" GatherResults=""
        OutputAccepted="" OutputRejected="" OutputTotal="">
        statDataSingleFlip
      </OutputCoreFileNameSpinFlips>
  </EnergyFunctionCalculator>
</Potts>
```

This section appears at the beginning of the configuration file. Line `<Dimensions x="101" y="101" z="1"/>` declares the dimensions of the lattice to be 101 x 101 x 1, *i.e.*, the lattice is two-dimensional and extends in the xy plane. The basis of the lattice is 0 in each direction, so the 101 lattice sites in the x and y directions have indices ranging from 0 to 100. `<Steps>1000</Steps>` tells CompuCell how long the simulation lasts in MCS. After executing this number of steps, CompuCell can run simulation at zero temperature for an additional period. In our case it will run for `<Anneal>10</Anneal>` extra steps. FluctuationAmplitude parameter determines intrinsic fluctuation or motility of cell membrane. Fluctuation amplitude is a temperature parameter in classical GGH model formulation. We have decided to use FluctuationAmplitude term instead of temperature because using word "temperature" to describe intrinsic motility of cell membrane was quite confusing.

In the above example, fluctuation amplitude applies to all cells in the simulation. To define fluctuation amplitude separately for each cell type we use the following syntax:

```
<FluctuationAmplitude>
      <FluctuationAmplitudeParameters CellType="Condensing"\
      FluctuationAmplitude="10"/>
      <FluctuationAmplitudeParameters CellType="NonCondensing"\
      FluctuationAmplitude="5"/>
</FluctuationAmplitude>
```

When CompuCell3D encounters expanded definition of FluctuationAmplitude it will use it in place of a global definition –

```
<FluctuationAmplitude>5</ FluctuationAmplitude >
```

To complete the picture CompUCell3D allows users to set fluctuation amplitude individually for each cell. Using Python scripting we write:

```
        for cell in self.cellList:
            if cell.type==1:
                cell.fluctAmpl=20
```

When determining which value of fluctuation amplitude to use, CompuCell first checks if `fluctAmpl` is non-negative. If this is the case it will use this value as fluctuation amplitude. Otherwise it will check if users defined fluctuation amplitude for cell types using expanded CC3DML definition and if so it will use those values as fluctuation amplitudes. Lastly it will resort to globally defined fluctuation amplitude (Temperature). Thus, it is perfectly fine to use FluctuationAmplitude CC3DML tags and set `fluctAmpl` for certain cells. In such a case CompuCell3D will use `fluctAmpl` for cells for which users defined it and for all other cells it will use values defined in the CC3DML.

In GGH model, the fluctuation amplitude is determined taking into account fluctuation amplitude of "source" (expanding) cell and "destination" (being overwritten) cell. Currently CompuCell3D supports 3 type functions used to calculate resultant fluctuation amplitude (those functions take as argument fluctuation amplitude of "source" and "destination" cells and return fluctuation amplitude that is used in calculation of pixel-copy acceptance). The 3 functions are `Min`, `Max`, and `ArithmeticAverage` and we can set them using the following option of the Potts section:

```
<Potts>
 <FluctuationAmplitudeFunctionName>
 Min
 </FluctuationAmplitudeFunctionName>
…
</Potts>
```

By default we use `Min` function. Notice that if you use global fluctuation amplitude definition (Temperature) it does not really matter which function you use. The differences arise when "source" and "destination" cells have different fluctuation amplitudes.
The above concepts are best illustrated by the following example:

```
<PythonScript>Demos/FluctuationAmplitude/FluctuationAmplitude.py\
</PythonScript>
```

```
 <Potts>
   <Dimensions x="100" y="100" z="1"/>
   <Steps>10000</Steps>
   <FluctuationAmplitude>5</FluctuationAmplitude>
   <FluctuationAmplitudeFunctionName>ArithmeticAverage\
   </FluctuationAmplitudeFunctionName>
   <NeighborOrder>2</NeighborOrder>
 </Potts>
```

Where in the CC3DML section we define global fluctuation amplitude and we also use ArithmeticAverage function to determine resultant fluctuation amplitude for the pixel copy.

In python script we will periodically set higher fluctuation amplitude for lattice quadrants so that when running the simulation we can see that cells belonging to different lattice quadrants have different membrane fluctuations:

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

        self.quarters=[[0,0,50,50],[0,50,50,100],\
        [50,50,100,100],[50,0,100,50]]

        self.steppableCallCounter=0

    def step(self, mcs):

        quarterIndex=self.steppableCallCounter % 4
        quarter=self.quarters[quarterIndex]

        for cell in self.cellList:

            if cell.xCOM>=quarter[0] and cell.yCOM>=quarter[1] and\
               cell.xCOM<quarter[2] and cell.yCOM<quarter[3]:

                cell.fluctAmpl=50
            else:

                    #this means CompuCell3D will use globally defined FluctuationAmplitude
                cell.fluctAmpl=-1

        self.steppableCallCounter+=1
```

Assigning negative fluctuationAmplitude `cell.fluctAmpl=-1` is interpreted by CompuCell3D as a hint to use fluctuation amplitude defined in the CC3DML.

The section below describes Temperature and CellMotility tags which are beibng deprecated (however cor compatibility reasons we still support those):

The first section of the .xml file defines the global parameters of the lattice and the simulation.

```
 <Potts>
 <Dimensions x="101" y="101" z="1"/>
 <Anneal>0</Anneal>
 <Steps>1000</Steps>
 <Temperature>5</Temperature>
 <Flip2DimRatio>1</Flip2DimRatio>
 <Boundary_y>Periodic</Boundary_y>
 <Boundary_x>Periodic</Boundary_x>
 <NeighborOrder>2</NeighborOrder>
```

```
<DebugOutputFrequency>20</DebugOutputFrequency>
<RandomSeed>167473</RandomSeed>
  <EnergyFunctionCalculator Type="Statistics">
     <OutputFileName Frequency="10">statData.txt</OutputFileName>
      <OutputCoreFileNameSpinFlips Frequency="1" GatherResults=""
         OutputAccepted="" OutputRejected="" OutputTotal="">
         statDataSingleFlip
       </OutputCoreFileNameSpinFlips>
  </EnergyFunctionCalculator>
</Potts>
```

This section appears at the beginning of the configuration file. Line `<Dimensions x="101" y="101" z="1"/>` declares the dimensions of the lattice to be 101 x 101 x 1, i.e., the lattice is two-dimensional and extends in the xy plane. The basis of the lattice is 0 in each direction, so the 101 lattice sites in the x and y directions have indices ranging from 0 to 100. `<Steps>1000</Steps>` tells CompuCell how long the simulation lasts in MCS. After executing this number of steps, CompuCell can run simulation at zero temperature for an additional period. In our case it will run for `<Anneal>10</Anneal>` extra steps. Setting the temperature is as easyas writing `<Temperature>5</Temperature>`.

We can also set temperature (or in other words cell motility) individually for each cell type. The syntax to do this is following:

```
<CellMotility>
    <MotilityParameters CellType="Condensing" Motility="10"/>
    <MotilityParameters CellType="NonCondensing" Motility="5"/>
</CellMotility>
```

You may use it in the Potts section in place of `<Temperature>`.

Based on discussion about the difference between pixel-flip attempts and MCS (see "Introduction to CompuCell3D") we can specify how many pixel copies should be attempted in every MCS. We specify this number indirectly by specifying the `Flip2DimRatio` - `<Flip2DimRatio>1</Flip2DimRatio>`, which tells CompuCell that it should make 1 x number of lattice sites attempts per MCS – in our case one MCS is 101x101x1 pixel-copy attempts. To set 2.5x101x101x1 pixel-copy attempts per MCS you would write `<Flip2DimRatio>2.5</Flip2DimRatio>`.

The next line specifies the neighbor order. The higher neighbor order the longer the Euclidian distance from a given pixel. In previous versions of CompuCell3D we have been using `<FlipNeighborMaxDistance>` or `<Depth>` (in Contact energy plugins) flag to accomplish same task. Since now CompuCell3D supports two kinds of lattices it would be inconvenient to change distances. It is much easier to think in terms n-th nearest neighbors. For the backwards compatibility we still support old flags but we discourage its use, especially that in the future we might support more than just two lattice types. Using nearest neighbor interactions may cause artifacts due to lattice anisotropy. The longer the interaction range, the more isotropic the simulation and the slower it runs. In addition, if the interaction range is comparable to the cell size, you may generate unexpected effects, since non-adjacent cells will contact each other.

On hex lattice those problems seem to be less seveare and there 1st or 2nd nearest neighbor usually are sufficient.

The Potts section also contains tags called `<Boundary_y>` and `<Boundary_x>`. These tags impose boundary conditions on the lattice. In this case the x and y axes are **periodic** (`<Boundary_x>Periodic</Boundary_x>`) so that *e.g.* the pixel with x=0, y=1, z=1 will neighbor the pixel with x=100, y=1, z=1. If you do not specify boundary conditions CompuCell will assume them to be of type **no-flux**, *i.e.* lattice will not be extended. The conditions are independent in each direction, so you can specify any combination of boundary conditions you like.

`DebugOutputFrequency` is used to tell CompuCell3D how often it should output text information about the status of the simulation. This tag is optional.

`RandomSeed` is used to initialize random number generator. If you do not do this all simulations will use same sequence of random numbers. Something you may want to avoid in the real simulations but is very useful while debugging your models.

`EnergyFunctionCalculator` is another option of Potts object that allows users to output statistical data from the simulation for further analysis.

> **Important:** CC3D has the option to run in the parallel mode but output from energy calculator will only work when running in a single CPU mode.

The `OutputFileName` tag is used to specify the name of the file to which CompuCell3D will write average changes in energies returned by each plugins with corresponding standard deviations for those MCS whose values are divisible by the Frequency argument. Here it will write these data every 10 MCS.

A second line with `OutputCoreFileNameSpinFlips` tag is used to tell CompuCell3D to output energy change for every plugin, every pixel-copy for MCS' divisible by the frequency. Option `GatherResults=""` will ensure that there is only one file written for accepted (`OutputAccepted`), rejected (`OutputRejected`) and accepted and rejected (`OutputTotal`) pixel copies. If you will not specify `GatherResults` CompuCell3D will output separate files for different MCS's and depending on the `Frequency` you may end up with many files in your directory.

One option of the Potts section that we have not used here is the ability to customize acceptance function for Metropolis algorithm:

```
<Offset>-0.1</Offset>
<KBoltzman>1.2</KBoltzman>
```

This ensures that pixel copies attempts that increase the energy of the system are accepted with probability
$P = e^{-(\Delta E - \delta)/kT}$ where δ and *k* are specified by `Offset` and `KBoltzman` tags respectively. By default $\delta=0$ and *k=1*.

As an alternative to exponential acceptance function you may use a simplified version which is essentially 1 order expansion of the exponential:

$$P = 1 - \frac{E - \delta}{kT}$$

To be able to use this function all you need to do is to add the following line in the Pots section:

```
<AcceptanceFunctionName>FirstOrderExpansion</AcceptanceFunctionName>
```

1.1.1 Lattice Type

Early versions of CompuCell3D allowed users to use only square lattice. Most recent versions however, allow the simulation to be run on hexagonal lattice as well.

> **Important:** Full description of hexagonal lattice including detailed derivations can be found in "Introduction to Hexagonal Lattices" available from www.compucell3d.org

To enable hexagonal lattice you need to put

```
<LatticeType>Hexagonal</LatticeType>
```

in the Potts section of the CC3DML configuration file.

There are few things to be aware of. When using hexagonal lattice. Obviously your pixels are hexagons (2D) or rhombic dodecahedrons (3D) but what is more important is that surface or perimeter of the pixel (depending whether in 2D or 3D) is different than in the case of sqaure pixel. The way CompuCell3D hex lattice implementation was done was that the volume of the pixel was constrained to be 1 regardless of the lattice type. There is also one to one correspondence between pixels of the square lattice and pixels of the hex lattice. Consequently we can come up with transformation equations which give positions of hex pixels as a function of square lattice pixel position:

$$[x_{hex}, y_{hex}, z_{hex}] = \left[\left(x_{cart} + \frac{1}{2}\right)L, \frac{\sqrt{3}}{2}y_{cart}L, \frac{\sqrt{6}}{3}z_{cart}L\right] \quad \text{for } y\%2 = 0 \text{ and } z\%3 = 0$$

$$[x_{hex}, y_{hex}, z_{hex}] = \left[x_{cart}L, \frac{\sqrt{3}}{2}y_{cart}L, \frac{\sqrt{6}}{3}z_{cart}L\right] \quad \text{for } y\%2 = 1 \text{ and } z\%3 = 0$$

$$[x_{hex}, y_{hex}, z_{hex}] = \left[x_{cart}L, \left(\frac{\sqrt{3}}{2}y_{cart} + \frac{\sqrt{3}}{6}\right)L, \frac{\sqrt{6}}{3}z_{cart}L\right] \quad \text{for } y\%2 = 0 \text{ and } z\%3 = 1$$

$$[x_{hex}, y_{hex}, z_{hex}] = \left[\left(x_{cart} + \frac{1}{2}\right)L, \left(\frac{\sqrt{3}}{2}y_{cart} + \frac{\sqrt{3}}{6}\right)L, \frac{\sqrt{6}}{3}z_{cart}L\right] \quad \text{for } y\%2 = 1 \text{ and } z\%3 = 1$$

$$[x_{hex}, y_{hex}, z_{hex}] = \left[x_{cart}L, \left(\frac{\sqrt{3}}{2}y_{cart} - \frac{\sqrt{3}}{6}\right)L, \frac{\sqrt{6}}{3}z_{cart}L\right] \quad \text{for } y\%2 = 0 \text{ and } z\%3 = 2$$

$$[x_{hex}, y_{hex}, z_{hex}] = \left[\left(x_{cart} + \frac{1}{2}\right)L, \left(\frac{\sqrt{3}}{2}y_{cart} - \frac{\sqrt{3}}{6}\right)L, \frac{\sqrt{6}}{3}z_{cart}L\right] \quad \text{for } y\%2 = 1 \text{ and } z\%3 = 2$$

Based on the above facts one can work out how unit length and unit surface transform to the hex lattice. The conversion factors are given below:
For the 2D case, assuming that each pixel has unit volume, we get:

$$S_{hex-unit} = \sqrt{\frac{2}{3\sqrt{3}}} \approx 0.6204$$

$$L_{hex-unit} = \sqrt{\frac{2}{\sqrt{3}}} \approx 1.075$$

where $S_{hex-unit}$ denotes length of the hexagon and $L_{hex-unit}$ denotes a distance between centers of the hexagons. Notice that unit surface in 2D is simply a length of the hexagon side and surface area of the hexagon with side 'a' is:

$$S = 6\frac{\sqrt{3}}{4}a^2$$

In 3D we can derive the corresponding unit quantities starting with the formulae for Volume and surface of rhombic dodecahedron (12 hedra)

$$V = \frac{16}{9}\sqrt{3}a^3$$

$$S = 8\sqrt{2}a^2$$

where 'a' denotes length of dodecahedron edge.
Constraining the volume to be one we get

$$a = 3\sqrt{\dfrac{9V}{16\sqrt{3}}}$$

and thus unit surface is given by:

$$S_{unit-hex} = \frac{S}{12} = \frac{8\sqrt{2}}{12} 3\sqrt{\frac{9V}{16\sqrt{3}}}^{2} \approx 0.445$$

and unit length by:

$$L_{unit-hex} = 2\frac{\sqrt{2}}{\sqrt{3}} a = 2\frac{\sqrt{2}}{\sqrt{3}} 3\sqrt{\frac{9V}{16\sqrt{3}}} \approx 1.122$$

1.2 Plugins Section

In this section we overview CC3DML syntax for all the plugins available in CompuCell3D. Plugins are either energy functions, lattice monitors or store user assigned data that CompuCell3D uses internally to configure simulation before it is run.

1.3 CellType Plugin

An example of the plugin that stores user assigned data that is used to configure simulation before it is run is a CellType Plugin. This plugin is responsible for defining cell types and storing cell type information. It is a basic plugin used by virtually every CompuCell simulation. The syntax is straight forward as can be seen in the example below:

```
<Plugin Name="CellType">
  <CellType TypeName="Medium" TypeId="0"/>
  <CellType TypeName="Fluid" TypeId="1"/>
  <CellType TypeName="Wall" TypeId="2" Freeze=""/>
</Plugin>
```

Here we have defined three cell types that will be present in the simulation: `Medium`, `Fluid`, `Wall`. Notice that we assign a number – `TypeId` – to every cell type. It is strongly recommended that `TypeId`'s are consecutive positive integers (e.g. 0,1,2,3...). `Medium` is traditionally given `TypeId=0` but this is not a requirement.

Important: Every CC3D simulation mut define CellType Plugin and include at least `Medium` specification.

Notice that in the example above cell type "Wall" has extra attribute `Freeze=""`. This attribute tells CompuCell that cells of "frozen" type will not be altered by pixel copies. Freezing certain cell types is a very useful technique in constructing different geometries for simulations or for restricting ways in which cells can move. In the example below we have frozen cell types wall to create tube geometry for fluid flow studies.

1.4 Simple Volume and Surface Constraints

One of the most commonly used energy term in the GGH Hamiltonian is a term that restricts variation of single cell volume. Its simplest form can be coded as show below:

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

By analogy we may define a term which will put similar constraint regarding the surface of the cell:

```
<Plugin Name="Surface">
  <TargetSurface>20</TargetSurface>
  <LambdaSurface>1.5</LambdaSurface>
</Plugin>
```

These two plugins inform CompuCell that the Hamiltonian will have two additional terms associated with volume and surface conservation. That is when pixel copy is attempted one cell will increase its volume and another cell will decrease. Thus overall energy of the system may or will change. Volume constraint essentially ensures that cells maintain the volume which close (this depends on thermal fluctuations) to target volume. The role of surface plugin is analogous to volume, that is to "preserve" surface. Note that surface plugin is commented out in the example above.

Energy terms for volume and surface constraints have the form:

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

$$E_{surface} = \lambda_{surface}(S_{cell} - S_{target})^2$$

Remark: Notice that copying a single pixel may cause surface change in more that two cells – this is especially true in 3D.

1.5 VolumeTracker and SurfaceTracker plugins

These two plugins monitor lattice and update volume and surface of the cells once pixel copy occurs. In most cases users will not call those plugins directly. They will be called automatically when either Volume (calls Volume Tracker) or Surface (calls Surface Tracker) or CenterOfMass (calls VolumeTracker) plugins are requested. However one should be aware that in some situations, for example when doing foam coarsening, where neither Volume nor Surface plugins are called, one may still want to track changes in surface or volume of cells. In such situations we explicitly invoke VolumeTracker or Surface Tracker plugin with the following syntax:

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

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

1.6 VolumeFlex Plugin

VolumeFlex plugin is more sophisticated version of Volume Plugin. While Volume Plugin treats all cell types the same i.e. they all have the same target volume and lambda coefficient, VolumeFlex plugin allows you to assign different lambda and different target volume to different cell types. The syntax for this plugin is straightforward and essentially mimics the example below.

```
<Plugin Name="Volume">
  <VolumeEnergyParameters CellType="Prestalk" TargetVolume="68" LambdaVolume="15"/>
  <VolumeEnergyParameters CellType="Prespore" TargetVolume="69" LambdaVolume="12"/>
  <VolumeEnergyParameters CellType="Autocycling" TargetVolume="80" LambdaVolume="10"/>
  <VolumeEnergyParameters CellType="Ground" TargetVolume="0" LambdaVolume="0"/>
  <VolumeEnergyParameters CellType="Wall" TargetVolume="0" LambdaVolume="0"/>
</Plugin>
```

We can also replace first line in the above listing with the following, old style syntax:
```
<Plugin Name="VolumeFlex">
```

Remark: Almost all CompuCell3D modules which have options Flex or LocalFlex are implemented as a single C++ module and CC3D, based on CC3DML syntax used, figures out which functionality to load at the run time. As a result for the reminder of this reference manual we will stick to the convention that all Flex and LocalFlex modules will be invoked using core name of the module only.

Notice that in the example above cell types `Wall` and `Ground` have target volume and coefficient lambda set to 0 – very unusual. That's because in this particular case those cells are frozen so the parameters specified for these cells do not matter. In fact it is safe to remove specifications for these cell types, but just for the illustration purposes we left them.

Using VolumeFlex Plugin you can effectively freeze certain cell types. All you need to do is to put very high lambda coefficient for the cell type you wish to freeze. You have to be careful though , because if initial volume of the cell of a given type is different from target volume for this cell type the cells will either shrink or expand to match target volume and only after this initial volume adjustment will they remain frozen provided `LambdaVolume` is high enough. Since rapid changes in the cell volume are uncontrolled (e.g. they can destroy many neighboring cells) you should opt for more gradual changes. In any case, we do not recommend this way of freezing cells because it is difficult to use, and also not efficient in terms of speed of simulation run.

1.7 SurfaceFlex Plugin

SurfaceFlex plugin is more sophisticated version of Surface Plugin. Everything that was said with respect to VolumeFlex plugin applies to SurfaceFlex. For syntax see example below:

```
<Plugin Name="Surface">
```

```
 <SurfaceEnergyParameters CellType="Prestalk" TargetSurface="90" LambdaSurface="0.15"/>
 <SurfaceEnergyParameters CellType="Prespore" TargetSurface="98" LambdaSurface="0.15"/>
 <SurfaceEnergyParameters CellType="Autocycling" TargetSurface="92" LambdaSurface="0.1"/>
 <SurfaceEnergyParameters CellType="Ground" TargetSurface="0" LambdaSurface="0"/>
 <SurfaceEnergyParameters CellType="Wall" TargetSurface="0" LambdaSurface="0"/>
 </Plugin>
```

1.8 VolumeLocalFlex Plugin

VolumeLocalFlex Plugin is very similar to Volume plugin. You specify both lambda coefficient and target volume, but as opposed to Volume Plugin the energy is calculated using target volume and lambda volume that are specified individually for each cell. In the course of simulation you can change this target volume depending on e.g. concentration of FGF in the particular cell. This way you can specify which cells grow faster, which slower based on a state of the simulation. This plugin requires you to develop a module (plugin or steppable) which will alter target volume for each cell. You can do it either in C++ or even better in Python.

Example syntax:

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

1.9 SurfaceLocalFlex Plugin

This plugin is analogous to VolumeLocalFlex but operates on cell surface.

Example syntax:

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

1.10 NeighborTracker Plugin

This plugin, as its name suggests, tracks neighbors of every cell. In addition it calculates common contact area between cell and its neighbors. We consider a neighbor this cell that has at least one common pixel side with a given cell. This means that cells that touch each other either "by edge" or by "corner" are not considered neighbors. See the drawing below:

5	5	5	4	4
5	5	5	4	4
5	5	4	4	4
1	1	2	2	2
1	1	2	2	2

Figure 1. Cells 5,4,1 are considered neighbors as they have non-zero common surface area. Same applies to pair of cells 4 ,2 and to 1 and 2. However, cells 2 and 5 are not neighbors because they touch each other "by corner". Notice that cell 5 has 8 pixels cell 4 , 7 pixels, cell 1 4 pixels and cell 2 6 pixels.

Example syntax:

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

This plugin is used as a helper module by other plugins and steppables e.g. Elasticity and AdvectionDiffusionSolver use NeighborTracker plugin.

1.11 Chemotaxis

Chemotaxis plugin, as its name suggests is used to simulate chemotaxis of cells. For every pixel copy, this plugin calculates change of energy associated with pixel move. There are several methods to define a change in energy due to chemotaxis. By default we define a chemotaxis using the following formula:

$$\Delta E_{chem} = \lambda \left(c\left(\vec{x}_{neighbor} \right) - c\left(\vec{x} \right) \right)$$

where

$c\left(\vec{x}_{neighbor} \right)$, $c\left(\vec{x} \right)$ denote chemical concentration at the pixel-copy-source and pixel-copy-destination pixel, respectively.

We also support a slight modification of the above formula in the Chemotaxis plugin where ΔE is non-zero only if the cell located at $\vec{x}$ after the pixel copy is non-medium. to enable such mode users need to include `<Algorithm="Regular"/>` tag in the body of CC3DML plugin.

Let's look at the syntax by studying the example usage of the Chemotaxis plugin:

```
<Plugin Name="Chemotaxis">
   <ChemicalField Source="FlexibleDiffusionSolverFE" Name="FGF">
        <ChemotaxisByType Type="Amoeba" Lambda="300"/>
```

```
            <ChemotaxisByType Type="Bacteria" Lambda="200"/>
    </ChemicalField>
 </Plugin>
```

The body of the chemotaxis plugin description contains sections called `ChemicalField`. In this section we tell CompuCell3D which module contains chemical field that we wish to use for chemotaxis. In our case it is `FlexibleDiffusionSolverFE`. Next we specify the name of the field - `FGF`. Subsequently, we specify lambda for each cell type so that cells of different type may respond differently to a given chemical. In particular types not listed will not respond to chemotaxis at all. Older versions of CompuCell3D allowed for different syntaxes as well. Despite the fact that those syntaxes are still supported for backward compatibility reasons, we discourage their use, because, they are somewhat confusing.

Ocassionally we may want to use different formula for the chemotaxis than the one presented above. Current version of CompCell3D supports the following definitions of change in chemotaxis energy (Saturation and SaturationLinear respectively):

$$\Delta E_{chem} = \lambda \left[\frac{c\left(\vec{x}_{neighbor}\right)}{\left(s + c\left(\vec{x}_{neighbor}\right)\right)} - \frac{c(\vec{x})}{\left(s + c(\vec{x})\right)} \right]$$

or

$$\Delta E_{chem} = \lambda \left[\frac{c\left(\vec{x}_{neighbor}\right)}{\left(s \cdot c\left(\vec{x}_{neighbor}\right) + 1\right)} - \frac{c(\vec{x})}{\left(s \cdot c(\vec{x}) + 1\right)} \right]$$

where '*s*' denotes saturation constant. To use first of the above formulas we set the value of the saturation coefficient:

```
<Plugin Name="Chemotaxis">
   <ChemicalField Source="FlexibleDiffusionSolverFE" Name="FGF">
        <ChemotaxisByType Type="Amoeba" Lambda="0"/>
        <ChemotaxisByType Type="Bacteria" Lambda="2000000" SaturationCoef="1"/>
   </ChemicalField>
 </Plugin>
```

Notice that this only requires small change in line where you previously specified only lambda.

```
 <ChemotaxisByType Type="Bacteria" Lambda="2000000" SaturationCoef="1"/>
```

To use second of the above formulas use `SaturationLinearCoef` instead of `SaturationCoef`:

```
<Plugin Name="Chemotaxis">
   <ChemicalField Source="FlexibleDiffusionSolverFE" Name="FGF">
     <ChemotaxisByType Type="Amoeba" Lambda="0"/>
```

```
      <ChemotaxisByType Type="Bacteria" Lambda="2000000" SaturationLinearCoef="1"/>
    </ChemicalField>
 </Plugin>
```

Sometimes it is desirable to have chemotaxis **at the interface between** only certain types of cells **and not between** other cell-type-pairs. In sucha case we augment `ChemotaxisByType` element with the following attribute:

```
<ChemotaxisByType Type="Amoeba" Lambda="100 "ChemotactTowards="Medium" />
```

This will cause that the change in chemotaxis energy will be non-zero only for those pixel copy attempts that happen between pixels belonging to `Amoeba` and `Medium`.

The definitions of chemotaxis presented so far do not allow specification of chemotaxis parameters individually for each cell. To do this we will use Python scripting. We still need to specify in the CC3DML which fields are important from chamotaxis stand point. Only fields listed in the CC3DML will be used to calculate chemotaxis energy:

```
…
<Plugin Name="CellType">
    <CellType TypeName="Medium" TypeId="0"/>
    <CellType TypeName="Bacterium" TypeId="1" />
    <CellType TypeName="Macrophage" TypeId="2"/>
    <CellType TypeName="Wall" TypeId="3" Freeze=""/>
 </Plugin>
…
<Plugin Name="Chemotaxis">
   <ChemicalField Source="FlexibleDiffusionSolverFE" Name="ATTR">
    <ChemotaxisByType Type="Macrophage" Lambda="20"/>
   </ChemicalField>
 </Plugin>
…
```

In the above excerpt from the CC3DML configuration file we see that cells of type `Macrophage` will chemotax in response to `ATTR` gradient.

Using Python scripting we can modify chemotaxis properties of individual cells as follows:

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


    def start(self):

        for cell in self.cellList:
            if cell.type==2:
                cd=self.chemotaxisPlugin.addChemotaxisData(cell,"ATTR")
                cd.setLambda(20.0)

                # cd.initializeChemotactTowardsVectorTypes("Bacterium,Medium")
                cd.assignChemotactTowardsVectorTypes([0,1])

                break

    def step(self,mcs):
        for cell in self.cellList:
            if cell.type==2:

                cd=self.chemotaxisPlugin.getChemotaxisData(cell,"ATTR")
                if cd:
                    l=cd.getLambda()-3
```

```
            cd.setLambda(l)
          break
```

In the `start` function for first encountered cell of type `Macrophage` (`type==2`) we insert `ChemotaxisData` object (it determines chemotaxing properties) and initialize λ parameter to 20. We also initialize vector of cell types towards which `Macrophage` cell will chemotax (it will chemotax towards `Medium` and `Bacterium` cells). Notice the break statement inside the `if` statement, inside the loop. It ensures that only first encountered `Macrophage` cell will have chemotaxing properties altered.

In the `step` function we decrease lambda chemotaxis by 3 units every 100 MCS. In effect we turn a cell from chemotaxing up ATTR gradient to being chemorepelled.

In the above example we have more than one macrophage but only one of them has altered chemotaxing properties. The other macrophages have chemotaxing properties set in the CC3DML section. CompuCell3D first checks if local definitions of chemotaxis are available (i.e. for individual cells) and if so it uses those. Otherwise it will use definitions from from the CC3DML.

The ChemotaxisData structure has additional functions which allo to set chemotaxis formula used. For example we may type:

```
def start(self):

    for cell in self.cellList:
        if cell.type==2:
            cd=self.chemotaxisPlugin.addChemotaxisData(cell,"ATTR")
            cd.setLambda(20.0)
                cd.setSaturationCoef(200.0)

            # cd.initializeChemotactTowardsVectorTypes("Bacterium,Medium")
            cd.assignChemotactTowardsVectorTypes([0,1])

            break
```

to activate Saturation formula. To activate SaturationLinear formula we would use:

```
cd.setSaturationLinearCoef(2.0)
```

Caution: when you use chemotaxis plugin you have to make sure that fields that you refer to and module that contains this fields are declared in the CC3DML file. Otherwise you will most likely cause either program crash (which is not as bad as it sounds) or unpredicted behavior (much worse scenario, although unlikely as we made sure that in the case of undefined symbols, CompuCell3D exits)

1.12 ExternalPotential plugin

Chemotaxis plugin is used to cause directional cell movement in response to chemical gradient. Another way to achieve directional movement is to use ExternalPotential plugin. This plugin is responsible for imposing a directed pressure (or rather force) on cells. It is used to implement persistent motion of cells and its applications can be very diverse.

Example usage of this plugin looks as follows:

```
  <Plugin Name="ExternalPotential">
     <Lambda x="-0.5" y="0.0" z="0.0"/>
  </Plugin>
```

`Lambda` is a vector quantity and determines components of force along three axes. In this case we apply force along x pointing in the positive direction.

Remark: positive component of Lambda vector pushes cell in the negative direction and negative component pushes cell in the positive direction

We can also apply external potential to specific cell types:

```
<Plugin Name="ExternalPotential">
 <ExternalPotentialParameters CellType="Body1" x="-10" y="0" z="0"/>
 <ExternalPotentialParameters CellType="Body2" x="0" y="0" z="0"/>
 <ExternalPotentialParameters CellType="Body3" x="0" y="0" z="0"/>
</Plugin>
```

Where in `ExternalPotentialParameters` we specity which cell type is subject to external potential (`Lambda` is specified using x,y,z attributes).

We can also apply external potential to individual cells. In that case, in the CC3DML section we only need to specify:

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

and in the Python file we change `lambdaVecX`, `lambdaVecY`, `lambdaVecZ`, which are properties of cell. For example in Python we could write:

```
cell.lambdaVecX=-10
```

Calculations done by ExternalPotential Plugin are by default based on direction of pixel copy (similarly as in chemotaxis plugin). One can however force CC3D to do calculations based on movement of center of mass of cell. To use algorithm based on center of mass movement we use the following CC3DML syntax:

```
<Plugin Name="ExternalPotential">
 <Algorithm>CenterOfMassBased</Algorithm>
…
</Plugin>
```

Remark: Note that in the pixel-based algorithm the typical value of pixel displacement used in calculations is of the order of 1 (pixel) whereas typical displacement of center of mass of cell due to single pixel copy is of the order of 1/cell volume (pixels) – ~ 0.1 pixel. This implies that to achieve compatible behavior of cells when using center of mass algorithm we need to multiply lambda's by appropriate factor, typically of the order of 10.

1.13 CellOrientation Plugin

Similarly as ExternalPotential plugin, this plugin gives preference to those pixel copies whose direction aligns with polarization vector (which is a property of each cell):

$$\Delta E = -\lambda(\sigma(i)) * \vec{p}(\sigma(i)) \cdot \vec{c} ,$$

where $\sigma(i)$ denotes cell at site i, $\vec{p}$ is polarization vector for cell at site i and $\vec{c}$ pixel copy vector. Because two cell participate in the pixel copy process the net energy change is simply a sum of above expressions: one for growing cell and one for shrinking cell. To set lambda we have two options: use global setting in the CC3DML:

```
<Plugin Name="CellOrientation">
    <LambdaCellOrientation>0.5</LambdaCellOrientation>
 </Plugin>
```

Or set λ individually for each cell and manage values of λ from Python. In this case we use the following CC3DML syntax:

```
<Plugin Name="CellOrientation">
    <LambdaFlex/>
 </Plugin>
```

or equivalently the shorter version:

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

If we manage λ values in Python we would use the following syntax to acces and modify values of lambda:

```
self.cellOrientationPlugin.getLambdaCellOrientation(cell)
```

```
self.cellOrientationPlugin.setLambdaCellOrientation(cell,0.5)
```

Calculations done by CellOrientation Plugin are by default based on direction of pixel copy (similarly as in chemotaxis plugin). One can however force CC3D to do calculations based on movement of center of mass of cell. To use algorithm based on center of mass movement we use the following CC3DML syntax:

```
<Plugin Name="CellOrientation">
 <Algorithm>CenterOfMassBased</Algorithm>
…
</Plugin>
```

See remark in External potential description about rescaling of parameters when changing algorithm to Center Of Mass–based.

Remark: In many (perhaps all) cases CellOrientation plugin can replaced by ExternalPotential plugin and we strongly recommend you do so.

1.14 PolarizationVector Plugin

PolarizationVector plugin is a simple plugin whose only task is to ensure that each cell in CompuCell3D simulation has as its attribute 3-component vector of floating point numbers. This plugin is normally used in together with CellOrientation but it also can be

reused in other applications, assuming that we do not use CellOrientation plugin at the same time. The CC3DML syntax is very simple:

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

To access or modify polarization vector requires use of Python scripting.

```
self.polarizationPlugin.getPolarizationVector(cell)
```

or to change values of the polarization vector:

```
self.polarizationPlugin.getPolarizationVector(cell,0.1,0.2,0.3)
```

1.15 CenterOfMass Plugin

This plugin monitors changes n the lattice and updates centroids of the cell:
$$x_{CM} = \sum_i x_i \quad , \quad y_{CM} = \sum_i y_i \quad , \quad z_{CM} = \sum_i z_i$$ where i denotes pixels belonging to a given cell. To obtain coordinates of a center of mass of a given cell we divide centroids by cell volume:

$$X_{CM} = \frac{x_{CM}}{V} \quad , \quad Y_{CM} = \frac{y_{CM}}{V} \quad , \quad Z_{CM} = \frac{z_{CM}}{V}$$

This plugin is aware of boundary conditions and centroids are calculated properly regardless which boundary conditions are used. The CC3DML syntax is very simple:

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

To access center of mass coordinates from Python we use the following syntax:

```
print 'x-component of COM is:', cell.xCOM
print 'y-component of COM is:', cell.yCOM
print 'z-component of COM is:', cell.zCOM
```

Remark: center of mass parameters in Python are read only. Any attempt to modify them will likely mess up the simulation.

1.16 Contact Energy

Energy calculations for the foam simulation are based on the boundary or contact energy between cells (or surface tension, if you prefer).
Together with volume constraint contact energy is one of the most commonly used energy terms in the GGH Hamiltonian. In essence it describes how cells "stick" to each other.

The explicit formula for the energy is:

$$E_{adhesion} = \sum_{i,j,neighbors} J(\tau_{\sigma(i)}, \tau_{\sigma(j)})(1 - \delta_{\sigma(i),\sigma(j)}),$$

where i and j label two neighboring lattice sites , σ's denote cell Ids, τ's denote cell types . In the case of foam simulation the total energy of the foam is simply the total boundary length times the surface tension (here defined to be 2J).

In the above formula, we need to differentiate between cell types and cell Ids. This formula shows that cell types and cell Ids **are not the same**. The Contact plugin in the .xml file, defines the energy per unit area of contact between cells of different types ($J(\tau_{\sigma(i)}, \tau_{\sigma(j)})$) and the interaction range (`NeighborOrder`) of the contact:

```
<Plugin Name="Contact">
  <Energy Type1="Foam" Type2="Foam">3</Energy>
  <Energy Type1="Medium" Type2="Medium">0</Energy>
  <Energy Type1="Medium" Type2="Foam">0</Energy>
  <NeighborOrder>2</NeighborOrder>
</Plugin>
```

In this case, the interaction range is 2, thus only up to second nearest neighbor pixels of a pixel undergoing a change or closer will be used to calculate contact energy change. `Foam` cells have contact energy per unit area of 3 and `Foam` and `Medium` as well as `Medium` and `Medium` have contact energy of 0 per unit area. For more information about contact energy calculations see "Introduction to CompuCell3D"

1.17 ContactLocalProduct Plugin

This plugin calculates contact energy based on local (i.e. per cell) cadhering expression levels. This plugin has to be used in conjunction with a steppable that assigns cadherin expression levels to the cell. Such steppables are usually written in Python – see ContactLocalProductExample in Demos directory.

We use the following formulas to calculate energy for this plugin:

$$E = \sum_{i,j-neighbors} \left(E_{offset} - k_{\sigma(i),\sigma(j)} f\left(N(i), N(j) \right) \right) \quad if \quad \sigma(i) \quad \wedge \quad \sigma(j) \quad \neq \quad medium$$

$$E = \sum_{i,j-neighbors} \left(E_{offset} - k_{\sigma(i),\sigma(j)} \right) \quad if \quad \sigma(i) \quad \vee \quad \sigma(j) \quad = \quad medium$$

By default $E_{offset} = 0$. $f\left(N(i), N(j) \right)$ is a function of cadherins and can be either a simple product $N(i)N(j)$, a product of squared expression levels $N(i)^2 N(j)^2$ or a $min\left(N(i), N(j) \right)$.

In the case of the second formula $E_{offset} - k_{\sigma(i),\sigma(j)}$ plays the role of "regular" contact energy between cell and medium.

The syntax of this plugin is as follows:

```
 <Plugin Name="ContactLocalProduct">
  <ContactSpecificity Type1="Medium" Type2="Medium">0</ContactSpecificity>
  <ContactSpecificity Type1="Medium" Type2="CadExpLevel1">-16</ContactSpecificity>
  <ContactSpecificity Type1="Medium" Type2="CadExpLevel2">-16</ContactSpecificity>
  <ContactSpecificity Type1="CadExpLevel1" Type2="CadExpLevel1">-2</ContactSpecificity>
  <ContactSpecificity Type1="CadExpLevel1" Type2="CadExpLevel2">2.75</ContactSpecificity>
  <ContactSpecificity Type1="CadExpLevel2" Type2="CadExpLevel2">-1</ContactSpecificity>
  <ContactFunctionType>Quadratic</ContactFunctionType>
  <EnergyOffset>0.0</EnergyOffset>
  <NeighborOrder>2</NeighborOrder>
 </Plugin>
```

Users need to specify ContactSpecificity ($k_{\sigma(i),\sigma(j)}$) between different cell types ContactFunctionType (by default it is set to Linear - $N(i)N(j)$ but other allowed key words are Quadratic - $N(i)^2 N(j)^2$ and Min - $min(N(i),N(j))$). EnergyOffset can be set to user specified value using above syntax. NeighborOrder has the same meaning as for "regular" Contact plugin.

Alternatively one can write customized function of the two cadherins and use it instead of the 3 choices given above. To do this, simply use the following syntax:

```
<Plugin Name="ContactLocalProduct">
  <ContactSpecificity Type1="Medium" Type2="Medium">0</ContactSpecificity>
  <ContactSpecificity Type1="Medium" Type2="CadExpLevel1">-16</ContactSpecificity>
  <ContactSpecificity Type1="Medium" Type2="CadExpLevel2">-16</ContactSpecificity>
  <ContactSpecificity Type1="CadExpLevel1" Type2="CadExpLevel1">-2</ContactSpecificity>
  <ContactSpecificity Type1="CadExpLevel1" Type2="CadExpLevel2">2.75</ContactSpecificity>
  <ContactSpecificity Type1="CadExpLevel2" Type2="CadExpLevel2">-1</ContactSpecificity>
  <ContactFunctionType>Quadratic</ContactFunctionType>
  <EnergyOffset>0.0</EnergyOffset>
  <NeighborOrder>2</NeighborOrder>
  <CustomFunction>
    <Variable>J1</Variable>
    <Variable>J2</Variable>
    <Expression>sin(J1*J2)</Expression>
  </CustomFunction>
 </Plugin>
```

Here we define variable names for cadherins in interacting cells (J1 denotes cadherin for one of the cells and cell2 denotes cadherin for another cell). Then in the Expression tag we give mathematical expression involving the two cadherin levels. The expression syntax has to follow syntax of the muParser - http://muparser.sourceforge.net/mup_features.html#idDef2.

1.18 AdhesionFlex Plugin

Adhesion Flex is a generalization of ContactLocalProduct plugin. It allows setting individual adhesivity properties for each cell. Users can use either CC3DML syntax or Python scripting to initialize adhesion molecule density for each cell. In addition,

Medium can also carry its own adhesion molecules. We use the following formula to calculate Contact energy in AdhesionFlex plugin:

$$E = \sum_{i,j-neighbors} \left(-\sum_{m,n} k_{mn} F(N_m(i), N_n(j)) \right) \left(1 - \delta_{\sigma(i),\sigma(j)} \right)$$

where indexes i, j label pixels, $J\left(\sigma(i),\sigma(j)\right)$ denotes contact energy between cell types $\sigma(i)$ and $\sigma(j)$, exactly as in "regular" contact plugin and indexes m,n label cadherins in cells composed f pixels i and j respectively. F denotes user-defined function of N_m and N_n. Altohugh this may look a bit complex, the basic idea is simple: each cell has certain number of cadherins on its surface. When cells touch each other the resultant energy is simpy a "product" - $k_{mn} F(N_m(i), N_n(j))$ - of every cadherin from one cell with every cadherin from another cell.The CC3DML syntax for this plugin is given below:

```
  <Plugin Name="AdhesionFlex">
   <AdhesionMolecule Molecule="NCad"/>
   <AdhesionMolecule Molecule="NCam"/>
   <AdhesionMolecule Molecule="Int"/>
   <AdhesionMoleculeDensity CellType="Cell1" Molecule="NCad"
   Density="6.1"/>
   <AdhesionMoleculeDensity CellType="Cell1" Molecule="NCam"
   Density="4.1"/>
   <AdhesionMoleculeDensity CellType="Cell1" Molecule="Int"
   Density="8.1"/>
   <AdhesionMoleculeDensity CellType="Medium" Molecule="Int"
   Density="3.1"/>
   <AdhesionMoleculeDensity CellType="Cell2" Molecule="NCad"
   Density="2.1"/>
   <AdhesionMoleculeDensity CellType="Cell2" Molecule="NCam"
   Density="3.1"/>

   <BindingFormula Name="Binary">
    <Formula> min(Molecule1,Molecule2)</Formula>
   <Variables>
      <AdhesionInteractionMatrix>
       <BindingParameter Molecule1="NCad" Molecule2="NCad" >
       -1.0</BindingParameter>
       <BindingParameter Molecule1="NCam" Molecule2="NCam">
       2.0</BindingParameter>
       <BindingParameter Molecule1="NCad" Molecule2="NCam" >
       -10.0</BindingParameter>
       <BindingParameter Molecule1="Int" Molecule2="Int"   >
       -10.0</BindingParameter>
      </AdhesionInteractionMatrix>
   </Variables>
   </BindingFormula>

   <NeighborOrder>2</NeighborOrder>
</Plugin>
```

k_{mn} matrix is specified within the AdhesionInteractionMatrix tag – the elements are listed using BindingParameter tags. The AdhesionMoleculeDensity tag specifies initial concentration of adhesion molecules. Even if you are going to modify those from Python (in the start function of the steppable) you are still required to specify the names of adhesion molecules and associate them with appropriate cell types. Failure to do so may result in simulation crash or undefined behaviors. The user-defined function F is specified using Formula tag where the arguments of the function are called Molecule1

and `Molecule2`. The syntax has to follow syntax of the muParser -
http://muparser.sourceforge.net/mup_features.html#idDef2 .

CompuCell3D example – *Demos/AdhesionFlex* - demonstrates how to manipulate
concentration of adhesion molecules:

```
self.adhesionFlexPlugin.getAdhesionMoleculeDensity(cell,"NCad")
```

allows to access adhesion molecule concentration using its name (as given in the
CC3DML above using `AdhesionMoleculeDensity` tag).

```
self.adhesionFlexPlugin.getAdhesionMoleculeDensityByIndex(cell,1)
```

allows to access adhesion molecule concentration using its index in the adhesion
molecule density vector. The order of the adhesion molecule densities in the vector is the
same as the order in which they were declared in the CC3DML above -
`AdhesionMoleculeDensity` tags.

```
self.adhesionFlexPlugin.getAdhesionMoleculeDensityVector(cell)
```

allows access to entire adhesion molecule density vector.

Each of these functions has its corresponding function whith operates on `Medium`. In this
case we do not give `cell` as first argument:

```
self.adhesionFlexPlugin.getMediumAdhesionMoleculeDensity("Int")
```

```
self.adhesionFlexPlugin.getMediumAdhesionMoleculeDensityByIndex (0)
```

```
self.adhesionFlexPlugin.getMediumAdhesionMoleculeDensityVector(cell)
```

To change the value of the adhesion molecule density we use `set` functions:

```
self.adhesionFlexPlugin.setAdhesionMoleculeDensity(cell,"NCad",0.1)
```

```
self.adhesionFlexPlugin.setAdhesionMoleculeDensityByIndex(cell,1,1.02)
```

```
self.adhesionFlexPlugin.setAdhesionMoleculeDensityVector(cell,\
[3.4,2.1,12.1])
```

Notice that in this las function we passed entire Python list as the argument. CC3D will
check if the number of entries in this vector is the same as the number of entries in the
currently used vector. If so the values from the passed vector will be copied, otherwise
they will be ignored.

Important: during mitosis we create new cell (`childCell`) and the adhesion molecule
vector of this cell will have no components. However in order for simulation to continue
we have to initialize this vector with number of cadherins appropriate to childCell type.

We know that `setAdhesionMoleculeDensityVector` is not appropriate for this task so we have to use:

```
self.adhesionFlexPlugin.assignNewAdhesionMoleculeDensityVector(cell,\
[3.4,2.1,12.1])
```

which will ensure that the content of passed vector is copied entirely into cell's vector (making size adjustments as necessry).

Important: You have to make sure that the number of newly assigned adhesion molecules is exactly the same as the number of adhesion molecules declared for the cell of this particular type.

All of `get` functions has corresponding `set` function which operates on `Medium`:

```
self.adhesionFlexPlugin.setMediumAdhesionMoleculeDensity("NCam",2.8)
```

```
self.adhesionFlexPlugin.setMediumAdhesionMoleculeDensityByIndex(2,16.8)
```

```
self.adhesionFlexPlugin.setMediumAdhesionMoleculeDensityVector(\
[1.4,3.1,18.1])
```

```
self.adhesionFlexPlugin.assignNewMediumAdhesionMoleculeDensityVector(\
[1.4,3.1,18.1])
```

1.19 ContactMultiCad Plugin

Remark: This plugin is deprecated . Please consider using AdhesionFlex.

ContactMultiCad plugin is a modified version of ContactLocalProduct plugin. In this case users can use several cadherins and describe how they translate into contact energy. The energy formula used by this plugin is given below:

$$E = \sum_{i,j-neighbors} \left(E_{offset} + J\left(\sigma(i),\sigma(j)\right) - \sum_{m,n} k_{mn} N_m(i) N_n(j) \right)$$

where indexes i, j label pixels, $J\left(\sigma(i),\sigma(j)\right)$ denotes contact energy between cell types $\sigma(i)$ and $\sigma(j)$, exactly as in "regular" contact plugin and indexes m,n label cadherins in cells composed f pixels i and j respectively.

The syntax for this plugin is as follows:

```
<Plugin Name="ContactMultiCad">

  <Energy Type1="Medium" Type2="CadExpLevel1">0</Energy>
  <Energy Type1="Medium" Type2="CadExpLevel2">0</Energy>
  <Energy Type1="CadExpLevel1" Type2="CadExpLevel1">0</Energy>
  <Energy Type1="CadExpLevel1" Type2="CadExpLevel2">0</Energy>
  <Energy Type1="CadExpLevel2" Type2="CadExpLevel2">0</Energy>
```

```
   <SpecificityCadherin>
    <Specificity Cadherin1="NCad1" Cadherin2="NCad1">-10</Specificity>
    <Specificity Cadherin1="NCad0" Cadherin2="NCad0">-12</Specificity>
    <Specificity Cadherin1="NCad1" Cadherin2="NCad0">-1</Specificity>
   </SpecificityCadherin>

   <EnergyOffset>0.0</EnergyOffset>
   </NeighborOrder>2</NeighborOrder>
 </Plugin>
```
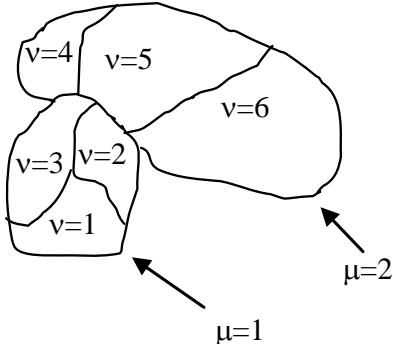
Entries of the type `<Energy Type1="Medium" Type2="CadExpLevel1">0</Energy>`
have the same meaning as in "regular" contact energy. Specificity parameters
specification k_{mn} are enclosed between tags `<SpecificityCadherin>` and

`<SpecificityCadherin>`. The names `NCad0` and `Ncad1` are arbitrary. However the
matrix k_{mn} will be ordered according to lexographic order of Cadherin names. For that
reason we recommend that you name cadherins in such a way that makes it easy what the
order will be. As in the example above using *NameNumber*
(e.g. NCad0, NCad1) makes it easy to figure out what the order will be (NCad0 will get
index 0 and NCad1 will get index 1). This is important because cadherins will be set in
Python and if you won't keep track of the ordering of the specificity you might wrongly
assign cadherins in Python and get unexpected results. In the example the order of
cadherins is clear based on the definition of cadherin specificity parameters.

1.20 Compartmentalized cells. ContactInternal Plugin

Calculating contact energies between compartmentalized cells is analogous to the non-
compartmentalized case. The energy expression takes the following form:

$$E_{contactcompartment} = \sum_{i,j-neighbors} J\left(\sigma\left(\mu_i, \nu_i\right), \sigma\left(\mu_j, \nu_j\right)\right)$$

where i and j denote pixels, $\sigma(\mu, \nu)$
denotes, as before, a cell type of a
cell with μ cluster id and ν cell id. In
compartmental cell models a cell is a
collection of subcells. Each subcell
has a unique id (cell id). In addition
to that each subcell will have
additional attribute, a cluster id that
determines to which cluster of
subcells a given subcell belongs.
(think of a cluster as a cell with
nonhomogenous cytoskeleton) The
idea here is to have different contact
energies between subcells belonging
to the same cluster and different
energies for cells belonging to
different clusters. Technically
subcells of a cluster are "regular"
CompuCell3D cells. By giving them

Figure 2. Two compartmental cells (cluster id $\mu=1$ and
cluster id $\mu=2$) Compartmentalized cell $\mu=1$ consists of
subcells with cell id $\nu=1,2,3$ and compartmentalized cell
$\mu=2$ consists of subcells with cell id $\nu=4,5,6$

an extra attribute cluster id we can introduce a concept of compartmental cells. In our convention $\sigma(0,0)$ denotes medium

Introduction of cluster id and cell id are essential for the definition of $J\left(\sigma\left(\mu_i,v_i\right),\sigma\left(\mu_j,v_j\right)\right)$.

$$J\left(\sigma\left(\mu_i,v_i\right),\sigma\left(\mu_j,v_j\right)\right) = J^{external}\left(\sigma\left(\mu_i,v_i\right),\sigma\left(\mu_j,v_j\right)\right) \quad if \quad \mu_i \neq \mu_i$$
$$J^{internal}\left(\sigma\left(\mu_i,v_i\right),\sigma\left(\mu_j,v_j\right)\right) \quad if \quad \mu_i = \mu_i$$

As we can see from above there are two hierarchies of contact energies – external and internal. To describe adhesive interactions between different compartmentalized cells we use two plugins: Contact and ContactInernal. Contact plugin calculates energy between two cells belonging to different clusters and ContactInternal calculates energies between cells belonging to the same cluster. An example syntax is shown below

```
<Plugin Name="Contact">

  <Energy Type1="Base" Type2="Base">0</Energy>
  <Energy Type1="Top" Type2="Base">25</Energy>
  <Energy Type1="Center" Type2="Base">30</Energy>
  <Energy Type1="Bottom" Type2="Base">-2</Energy>
  <Energy Type1="Side1" Type2="Base">25</Energy>
  <Energy Type1="Side2" Type2="Base">25</Energy>
  <Energy Type1="Medium" Type2="Base">0</Energy>


  <Energy Type1="Medium" Type2="Medium">0</Energy>
  <Energy Type1="Top" Type2="Medium">30</Energy>
  <Energy Type1="Bottom" Type2="Medium">20</Energy>
  <Energy Type1="Side1" Type2="Medium">30</Energy>
  <Energy Type1="Side2" Type2="Medium">30</Energy>
  <Energy Type1="Center" Type2="Medium">45</Energy>

  <Energy Type1="Top" Type2="Top">2</Energy>
  <Energy Type1="Top" Type2="Bottom">100</Energy>
  <Energy Type1="Top" Type2="Side1">25</Energy>
  <Energy Type1="Top" Type2="Side2">25</Energy>
  <Energy Type1="Top" Type2="Center">35</Energy>

  <Energy Type1="Bottom" Type2="Bottom">10</Energy>
  <Energy Type1="Bottom" Type2="Side1">25</Energy>
  <Energy Type1="Bottom" Type2="Side2">25</Energy>
  <Energy Type1="Bottom" Type2="Center">35</Energy>

  <Energy Type1="Side1" Type2="Side1">25</Energy>
  <Energy Type1="Side1" Type2="Center">25</Energy>
  <Energy Type1="Side2" Type2="Side2">25</Energy>
  <Energy Type1="Side2" Type2="Center">25</Energy>
  <Energy Type1="Side1" Type2="Side2">15</Energy>

  <Energy Type1="Center" Type2="Center">20</Energy>

  <NeighborOrder>2</NeighborOrder>
 </Plugin>
```

```
<Plugin Name="ContactInernal">
```

```
 <Energy Type1="Base" Type2="Base">0</Energy>
 <Energy Type1="Base" Type2="Bottom">0</Energy>
 <Energy Type1="Base" Type2="Side1">0</Energy>
 <Energy Type1="Base" Type2="Side2">0</Energy>
 <Energy Type1="Base" Type2="Center">0</Energy>

 <Energy Type1="Top" Type2="Top">4</Energy>
 <Energy Type1="Top" Type2="Bottom">25</Energy>
 <Energy Type1="Top" Type2="Side1">22</Energy>
 <Energy Type1="Top" Type2="Side2">22</Energy>
 <Energy Type1="Top" Type2="Center">15</Energy>

 <Energy Type1="Bottom" Type2="Bottom">4</Energy>
 <Energy Type1="Bottom" Type2="Side1">15</Energy>
 <Energy Type1="Bottom" Type2="Side2">15</Energy>
 <Energy Type1="Bottom" Type2="Center">10</Energy>

 <Energy Type1="Side1" Type2="Side1">11</Energy>
 <Energy Type1="Side2" Type2="Side2">11</Energy>
 <Energy Type1="Side1" Type2="Side2">11</Energy>

 <Energy Type1="Side2" Type2="Center">10</Energy>
 <Energy Type1="Side1" Type2="Center">10</Energy>

 <Energy Type1="Center" Type2="Center">2</Energy>

  <NeighborOrder>2</NeighborOrder>
 </Plugin>
```

Depending whether pixels for which we calculate contact energies belong to the same cluster or not we will use internal or external contact energies respectively.

1.21 LengthConstraint Plugin

This plugin imposes elongation constraint on the cell. Effectively it "measures" a cell along its "axis of elongation" and ensures that cell length along the elongation axis is close to target length. For detailed description of this algorithm in 2D see Roeland Merks' paper "Cell elongation is a key to in silico replication of in vitro vasculogenesis and subsequent remodeling" Developmental Biology **289** (2006) 44-54). This plugin is usually used in conjunction with Connectivity Plugin or ConnectivityGlobal Plugin. The syntax is as follows:

```
<Plugin Name="LengthConstraint">
 <LengthEnergyParameters CellType="Body1" TargetLength="30" LambdaLength="5"/>
</Plugin>
```

LambdaLength determines the degree of cell length oscillation around TargetLength parameter. The higher LambdaLength the less freedom a cell will have to deviate from TargetLength.
In the 3D case we use the following syntax:

```
<Plugin Name="LengthConstraint">
 <LengthEnergyParameters CellType="Body1" TargetLength="20"
 MinorTargetLength="5" LambdaLength="100" />
</Plugin>
```

Notice new attribute called MinorTargetLength. In 3D it is not sufficient to constrain the "length" of the cell you also need to constrain "width" of the cell along axis

perpendicular to the major axis of the cell. This "width" is referred to as
`MinorTargetLength`.

The parameters are assigned using Python – see *Demos\elongationFlexTest* example.

To control length constraint individually for each cell we may use Python scripting to assign LambdaLength, TartgetLength and in 3D MinorTargetLength. In Python steppable we typically would write the following code:

```
self.lengthConstraintPlugin.setLengthConstraintData(cell,10,20)
```

which enables length constraint for cell `cell` with `LambdaLength=10` and TargetLength=20. In 3D we may specify MinorTargetLength (we set it to 5) by adding 4th parameter to the above call:

```
self.lengthConstraintPlugin.setLengthConstraintData(cell,10,20,5)
```

Remark: If we use CC3DML specification of length constraint for certain cell types and in Python we set this constraint individually for a single cell then the local definition of the constraint has priority over definitions for the cell type.

If, in the simulation, we will be setting length constraint for only few individual cells then it is best to manipulate the constraint parameters from the Python script. In this case in the CC3DML we only have to declare that we will use length constraint plugin and we may skip the definition by-type definitions:

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

Remark: When using target length plugins it is important to use connectivity constraint. This constraint will check if a given pixel copy breaks cell connectivity. If it does, the plugin will add large energy penalty (defined by a user) to change of energy effectively prohibiting such pixel copy. In the case of 2D on square lattice checking cell connectivity can be done locally and thus is very fast. Unfortunately on hex lattice and in 3D on either lattice we don't have an algorithm of performing such check locally and therefore we do it globally using breadth first search algorithm and comparing volumes of cells calculated this way with actual volume of the cell. If they agree we conclude that cell connectivity is preserved. However, the computational cost of running such algorithm can be quite high. Therefore if one does need extremely elongated cells (it is when connectivity algorithm has to do a lot of work) one may neglect connectivity constraint and use Length constrain only. For slight cells elongations the connectivity should be preserved however, occasionally cells may fragment.

1.22 Connectivity Plugins

The basic Connectivity plugin works **only in 2D and only on square lattice** and is used to ensure that cells are connected or in other words to prevent separation of the cell into pieces. The detailed algorithm for this plugin is described in Roeland Merks' paper "Cell elongation is a key to *in-silico* replication of in vitro vasculogenesis and subsequent remodeling" Developmental Biology **289** (2006) 44-54). There was one modification of the algorithm as compared to the paper. Namely, to ensure proper connectivity we had to

reject all pixel copies that resulted in more that two collisions. (see the paper for detailed explanation what this means).
The syntax of the plugin is straightforward:

```
<Plugin Name="Connectivity">
   <Penalty>100000</Penalty>
</Plugin>
```

`Penalty` denotes energy that will be added to overall change of energy if attempted pixel copy would violate connectivity constraints. If the penalty is positive and much larger than the absolute value of other energy changes in the simulation this has the effect of preventing a pixel copy from occurring.
A more general type of connectivity constraint is implemented in ConnectivityGlobal plugin. In this case we calculate volume of a cell using breadth first search algorithm and compare it with actual volume of the cell. If they agree we conclude that cell connectivity is preserved. This plugin works both in 2D and 3D and on either type of lattice. However, the computational cost of running such algorithm can be quite high so it is best to limit this plugin to cell types for which connectivity of cell is really essential:

```
<Plugin Name="ConnectivityGlobal">
   <Penalty Type="Body1">1000000000</Penalty>
</Plugin>
```

In certain types of simulation it may happen that at some point cells change cell types. If a cell that was not subject to connectivity constraint, changes type to the cell that is constrained by global connectivity and this cell is fragmented before type change this situation normally would result in simulation freeze. However CompuCell3D, first before applying constraint it will check if the cell is fragmented. If it is, there is no constraint. Global connectivity constraint is only applied when cell is non-fragmented. The numerical value of `Penalty` in the CC3DML syntax above does not really matter as long as it is greater than 0. CompuCell3D guarantees that cells for which penalty is greater than 0 will remain connected.

Quite often in the simulation we don't need to impose connectivity constraint on all cells or on all cells of given type. Usually only select cell types or select cells are elongated and therefore need connectivity constraint. In such a case we use ConnectivityLocalFlex plugin and assign connectivity constraints to particular cells in Python

In CC3DML we only declare:

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

In Python we manipulate/access connectivity parameters for individual cells using the following syntax:

```
self.connectivityLocalFlexPlugin.setConnectivityStrength(cell,20.7)
```

```
self.connectivityLocalFlexPlugin.getConnectivityStrength(cell)
```

See also example in *Demos\elongationLocalFlexTest.*

ConnectivityLocalFlex plugin works only in 2D and on a square lattice. We may also use ConnectivityGlobal plugin to set connectivity constraint individually for each cell. Analogously, as in the case of ConnectivityLocalFlex , in the CC3DML we declare

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

and in Python we manipulate/access connectivity parameters for individual cells using the following syntax:

```
self.connectivityGlobalPlugin.setConnectivityStrength(cell,10000000)
```

```
self.connectivityGlobalPlugin.getConnectivityStrength(cell)
```

1.23 Mitosis Plugin

Remark: This is obsolete plugin and we strongly recommend you replace it with MitosisSteppable.

Mitosis plugin carries out cell division into two cells once the parent cell reaches critical volume (DoublingVolume). The two cells after mitosis will have approximately the same volume although it cannot be guaranteed in general case if the parent cell is fragmented. One major problem with Mitosis plugin is that after mitosis the attributes of the offspring cell might not be initialized properly. By default cell type of the offspring cell will be the same as cell type of parent and they will also share target volume. All other parameters for the new cell remain uninitialized.

Remark: For this reason we strongly recommend using Mitosis plugin through Python interface as there users can quite easily customize what happens to parent and offspring cells after mitosis. An example of the use of Mitosis plugin through Python scripting is provided in CompuCell3D's Python Scripting Manual. The syntax of the "standard" mitosis plugin is the following:

```
<Plugin Name="Mitosis">
    <DoublingVolume>50</DoublingVolume>
</Plugin>
```

Every time a cell reaches DoublingVolume it will undergo the mitosis and the offspring cell will inherit type and target volume of the parent. If this simple behavior is unsatisfactory consider use Python scripting to implement proper mitotic divisions of cells.

1.24 Secretion / SecretionLocalFlex Plugin

Secretion "by cell type" can and should be handled by the appropriate PDE solver. To implement secretion in individual cells using Python we can use secretion plugin defined in the CC3DML as:

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

or as:

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

The inclusion of the above code in the CC3DML will allow users to implement secretion
for individual cells from Python. **Note**:Secretion for individual cells invoked via Python
will be called only once per MCS.

> **Important:** Secretion plugin can be used to implement secretion by cell type however **we
> strongly advise against doing so**. Defining secretion by cell-type in the Secretion plugin
> will lead to performance degradation on multi-core machines. Please see section below
> for more information if you are still interested in using secretion by cell-type inside
> Secretion plugin

Typical use of secretion from Python is dempnstrated best in the example below:

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

    def step(self,mcs):
        attrSecretor=self.getFieldSecretor("ATTR")
        for cell in self.cellList:
            if cell.type==3:
                attrSecretor.secreteInsideCell(cell,300)
                attrSecretor.secreteInsideCellAtBoundary(cell,300)
                attrSecretor.secreteOutsideCellAtBoundary(cell,500)
                attrSecretor.secreteInsideCellAtCOM(cell,300)
            elif cell.type==2:
                attrSecretor.secreteInsideCellConstantConcentration(cell,300)
```

> **Remark**: Instead of using `SteppableBasePy` class we are using `SecretionBasePy` class.
> The reason for this is that in order for secretion plugin with secretion modes accessible
> from Python to behave exactly as previous versions of PDE solvers (where secretion was
> done first followed by "diffusion" step) we have to ensure that secretion steppable
> implemented in Python is called **before** each Monte Carlo Step, which implies that it will
> be also called before "diffusing" function of the PDE solvers. `SecretionBasePy` sets
> extra flag which ensures that steppable which inherits from SecretionBasePy is called
> before MCS (and before all "regular' Python steppables).

There is no magic to `SecretionBasePy` - if you still want to use `SteppableBasePy` as a
base class for secretion (or for that matter `SteppablePy`) do so, but remember that you
need to set flag:

self.runBeforeMCS=1

to ensure that your new steppable will run before each MCS. See example below for
alternative implementation of `SecretionSteppable` using `SteppableBasePy` as a base
class:

```
class SecretionSteppable(SteppableBasePy):
    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator, _frequency)
        self.runBeforeMCS=1
    def step(self,mcs):
```

```
        attrSecretor=self.getFieldSecretor("ATTR")
        for cell in self.cellList:
            if cell.type==3:
                attrSecretor.secreteInsideCell(cell,300)
                attrSecretor.secreteInsideCellAtBoundary(cell,300)
                attrSecretor.secreteOutsideCellAtBoundary(cell,500)
                attrSecretor.secreteOutsideCellAtBoundaryOnContactwith(cell,500,[2,3])
                attrSecretor.secreteInsideCellAtCOM(cell,300)
                attrSecretor.uptakeInsideCellAtCOM(cell,300,0.2)
            elif cell.type==2:
                attrSecretor.secreteInsideCellConstantConcentration(cell,300)
```

The secretion of individual cells is handled through Field Secretor objects. Field Secretor concenpt is quite convenient because the amoun of Python coding is quite small. To secrete chemical (this is now done for individual cell) we first create field secretor object, `attrSecretor=self.getFieldSecretor("ATTR")`, which allows us to secrete into field called ATTR.

Then we pick a cell and using field secretor we simulate secretion of chemical ATTR by a cell:

`attrSecretor.secreteInsideCell(cell,300)`

Currently we support 6 secretion modes for individual cells:
1. `secreteInsideCell` – this is equivalent to secretion in every pixel belonging to a cell
2. `secreteInsideCellConstantConcentration` – this is equivalent to secretion in every pixel belonging to a cell and setting concentration to fixed, constant level
3. `secreteInsideCellAtBoundary` – secretion takes place in the pixels belonging to the cell boundary
4. `secreteInsideCellAtBoundaryOnContactWith` - secretion takes place in the pixels belonging to the cell boundary that touches any of the cells listed as the last argument of the function call
5. `secreteOutsideCellAtBoundary` – secretion takes place in pixels which are outside the cell but in contact with cell boundary pixels
6. `secreteOutsideCellAtBoundaryOnContactWith` - secretion takes place in pixels which are outside the cell but in contact with cell boundary pixels and in contact with cells listed the last argument of the function call
7. `secreteInsideCellAtCOM` – secretion at the center of mass of the cell

and 6 uptake modes:
1. `uptakeInsideCell` – this is equivalent to uptake in every pixel belonging to a cell
2. `uptakeInsideCellAtBoundary` – uptake takes place in the pixels belonging to the cell boundary
3. `uptakeInsideCellAtBoundaryOnContactWith` - uptake takes place in the pixels belonging to the cell boundary that touches any of the cells listed as the last argument of the function call
4. `uptakeOutsideCellAtBoundary` – uptake takes place in pixels which are outside the cell but in contact with cell boundary pixels

5. `uptakeOutsideCellAtBoundaryOnContactWith` - uptake takes place in pixels which are outside the cell but in contact with cell boundary pixels and in contact with cells listed the last argument of the function call
6. `uptakeInsideCellAtCOM` – uptake at the center of mass of the cell

Secretion functions use the following syntax

```
secrete*(cell,amount,list_of_cell_types)
```

Note that `list_of_cell_types` is used only for function which implement such functionality i.e. (`secreteInsideCellAtBoundaryOnContactWith` and `secreteOutsideCellAtBoundaryOnContactWith`)

Uptake functions use the following syntax

```
uptake*(cell,max_amount,relative_uptake,list_of_cell_types)
```

Note that `list_of_cell_types` is used only for function which implement such functionality i.e. (`uptakeInsideCellAtBoundaryOnContactWith` and `uptakeOutsideCellAtBoundaryOnContactWith`)

Important: The uptake works as follows: when available concentratin is greater than `max_amount`, then `max_amount` is subtracted from `current_concentration`, otherwise we subtract `relative_uptake*current_concentration`.

As you may infer from above, the modes 1-5 require tracking of pixels belonging to cell and pixels belonging to cell boundary. If you are not using modes 1-3 you may disable pipxel tracking by including

`<DisablePixelTracker/>` and/or `<DisableBoundaryPixelTracker/>` tags – as shown in the example below:

```
<Plugin Name="Secretion">

 <DisablePixelTracker/>
 <DisableBoundaryPixelTracker/>

 <Field Name="ATTR" ExtraTimesPerMC="2">
  <Secretion Type="Bacterium">200</Secretion>
  <SecretionOnContact Type="Medium" SecreteOnContactWith="B">300</SecretionOnContact>
  <ConstantConcentration Type="Bacterium">500</ConstantConcentration>
 </Field>
</Plugin>
```

Remark: Make sure that fields into which you will be secreting chemicals exist. They are usually fields defined in PDE solvers. When using secretion plugin you do not need to specify SecretionData section for the PDE solvers.

When implementing e.g. secretion inside cell when the cell is incontact with other cell we use neighbor tracker and a short script in the spirit of the below snippet:

```
for cell in self.cellList:
    attrSecretor=self.getFieldSecretor("ATTR")
    for neighbor , commonSurfaceArea in self.getCellNeighborDataList(cell):
        if neighbor.type in [self.WALL]:
            attrSecretor.secreteInsideCell(cell,300)
```

1.25 Secretion Plugin (legacy version)

Important: While we still support Secretion plugin in as described in this section we observed performance degradation when when declaring `<Field>` elements inside the

plugin. To resolve this issue we encourage users to implement secretion "by cell type" in the PDE solver and keep using secretion plugin to implement secretion on a per-cell basis using Python scripting. Please see section "Secretion / SecretionLocalFlex Plugin" above for more information and **please consider using Secretion plugin in a way described in the section above.**

Remark: In version 3.6.2 Secretion plugin should not be used with DiffusionSolverFE or any of the GPU-based solvers. We are working to resolve this issue.

In earlier version os of CC3D secretion was part of PDE solvers. We still support this mode of model description however, starting in 3.5.0 we developed separate plugin which handles secretion only. Via secretion plugin we can simulate cellular secretion of various chemicals. The secretion plugin allows users to specify various secretion modes in the CC3DML file – CC3DML syntax is practically identical to the SecretionData syntax of PDE solvers. In addition to this Secretion plugin allows users to maniupulate secretion properties of individual cells from Python level. To account for possibility of PDE solver being called multiple times during each MCS, the Secretion plugin can be called multiple times in each MCS as well. We leave it up to user the rescaling of secretion constants when using multiple secretion calls in each MCS. **Note**:Secretion for individual cells invoked via Python will be called only once per MCS.

Typical CC3DML xyntax for Secretion plugin is presented below:

```
<Plugin Name="Secretion">
 <Field Name="ATTR" ExtraTimesPerMC="2">
  <Secretion Type="Bacterium">200</Secretion>
  <SecretionOnContact Type="Medium" SecreteOnContactWith="B">300</SecretionOnContact>
  <ConstantConcentration Type="Bacterium">500</ConstantConcentration>
 </Field>
</Plugin>
```

By default `ExtraTimesPerMC` is set to 0 - meaning no extra calls to Secretion plugin per MCS.

Typical use of secretion from Python is dempnstrated best in the example below:

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

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

Remark: Instead of using `SteppableBasePy` class we are using `SecretionBasePy` class. The reason for this is that in order for secretion plugin with secretion modes accessible from Python to behave exactly as previous versions of PDE solvers (where secretion was done first followed by "diffusion" step) we have to ensure that secretion steppable

implemented in Python is called **before** each Monte Carlo Step, which implies that it will be also called before "diffusing" function of the PDE solvers. SecretionBasePy sets extra flag which ensures that steppable which inherits from SecretionBasePy is called before MCS (and before all "regular' Python steppables).

There is no magic to SecretionBasePy - if you still want to use SteppableBasePy as a base class for secretion (or for that matter SteppablePy) do so, but remember that you need to set flag:

self.runBeforeMCS=1

to ensure that your new stoppable will run before each MCS. See example below for alternative implementation of SecretionSteppable using SteppableBasePy as a base class:

```
class SecretionSteppable(SteppableBasePy):
    def __init__(self,_simulator,_frequency=1):
        SteppableBasePy.__init__(self,_simulator, _frequency)
        self.runBeforeMCS=1
    def step(self,mcs):
        attrSecretor=self.getFieldSecretor("ATTR")
        for cell in self.cellList:
            if cell.type==3:
                attrSecretor.secreteInsideCell(cell,300)
                attrSecretor.secreteInsideCellAtBoundary(cell,300)
                attrSecretor.secreteOutsideCellAtBoundary(cell,500)
                attrSecretor.secreteInsideCellAtCOM(cell,300)
```

The secretion of individual cells is handled through Field Secretor objects. Field Secretor concenpt is quite convenient because the amoun of Python coding is quite small. To secrete chemical (this is now done for individual cell) we first create field secretor object, attrSecretor=self.getFieldSecretor("ATTR"), which allows us to secrete into field called ATTR.
Then we pick a cell and using field secretor we simulate secretion of chemical ATTR by a cell:

attrSecretor.secreteInsideCell(cell,300)

Currently we support 4 secretion modes for individual cells:
8. secreteInsideCell – this is equivalent to secretion in every pixel belonging to a cell
9. secreteInsideCellAtBoundary – secretion takes place in the pixels belonging to the cell boundary
10. secreteInsideCellAtBoundary – secretion takes place in pixels which are outide the cell but in contact with cell boundary pixels
11. secreteInsideCellAtCOM – secretion at the center of mass of the cell

As you may infer from above modes 1, 2 and 3 require tracking of pixels belonging to cell and pixels belonging to cell boundary. If you are not using modes 1-3 you may disable pipxel tracking by including

`<DisablePixelTracker/>` and/or `<DisableBoundaryPixelTracker/>` tags – as shown in the example below:

```
<Plugin Name="Secretion">

 <DisablePixelTracker/>
 <DisableBoundaryPixelTracker/>

 <Field Name="ATTR" ExtraTimesPerMC="2">
  <Secretion Type="Bacterium">200</Secretion>
  <SecretionOnContact Type="Medium" SecreteOnContactWith="B">300</SecretionOnContact>
  <ConstantConcentration Type="Bacterium">500</ConstantConcentration>
 </Field>
</Plugin>
```

Remark: Make sure that fields into which you will be secreting chemicals exist. They are usually fields defined in PDE solvers. When using secretion plugin you do not need to specify SecretionData section for the PDE solvers.

1.26 PDESolverCaller Plugin

Remark: In most cases you can specify extra calls to PDE solvers in the solver itself. Thus this plugin is being deprecated.

PDE solvers in CompuCell3D are implemented as steppables . This means that by default they are called every MCS. In many cases this is insufficient. For example if diffusion constant is large, then explicit finite difference method will become unstable and the numerical solution will have no sense. To fix this problem one could call PDE solver many times during single MCS. This is precisely the task taken care of by PDESolverCaller plugin. The syntax is straightforward:

```
<Plugin Name="PDESolverCaller">
 <CallPDE PDESolverName="FlexibleDiffusionSolverFE"ExtraTimesPerMC="8"/>
</Plugin>
```

All you need to do is to give the name of the steppable that implements a given PDE solver and pass let CompCell3D know how many extra times per MCS this solver is to be called (here `FlexibleDiffusionSolverFE` was 8 extra times per MCS).

1.27 Elasticity Plugin and ElasticityTracker Plugin

Remark: Users may consider FocalPointPoasticityPlugin as it often offers more flexibility than Elasticity plugin.

This plugin is responsible for handling the following energy term:

$$E = \sum_{i,j-cellneighbors} \lambda_{ij} \left(l_{ij} - L_{ij} \right)^2$$

where l_{ij} is a distance between center of masses of cells i and j and L_{ij} is a target length corresponding to l_{ij} .

The syntax of this plugin is the following

```
<Plugin Name="ElasticityEnergy">
   <LambdaElasticity>200.0</LambdaElasticity>
   <TargetLengthElasticity>6</TargetLengthElasticity>
</Plugin>
```

In this case λ_{ij} and L_{ij} are the same for all participating cells types.
By adding extra attribute `<Local/>` to the above plugin:

```
<Plugin Name="ElasticityEnergy">
   <Local/>
   <LambdaElasticity>200.0</LambdaElasticity>
   <TargetLengthElasticity>6</TargetLengthElasticity>
</Plugin>
```

we tell CompuCell3D to use λ_{ij} and L_{ij} defined on per pair of cells basis. The initialization of λ_{ij} and L_{ij} usually takes place in Python script and users must make sure that $l_{ij} = l_{ji}$ and $\lambda_{ij} = \lambda_{ji}$ or else one can get unexpected results. We provide example Python and CC3DML files that demo the use of plasticity plugin.

Users have to specify which cell types participate in the plasticity calculations. This is done by including ElasticityTracker plugin **before** Elasticity plugin in the CC3DML file. The syntax is very simple:

```
<Plugin Name="ElasticityTracker">
   <IncludeType>Body1</IncludeType>
   <IncludeType>Body2</IncludeType>
   <IncludeType>Body3</IncludeType>
</Plugin>
```

All is required is a list of participating cell types. Here cells of type `Body1`, `Body2` and `Body3` will be taken into account for elasticity energy calculation purposes.
The way in which CompuCell3D determines which cells are to be included in the elasticity energy calculations is by examining which cells are in contact with each other before simulation begins.
If the types of cells touching each other are listed in the list of `IncudeTypes` of ElasticityTracker then such cells are being taken into account when calculating elastic constraint. Cells which initially are not touching **will not participate in calculations** even if their type is included in the list of "ElasticityTracker". However, in some cases it is desirable to add elasticity pair even for cells that do not touch each other or do it once simulation has started. To do this ElasticityTracker plugin defines two function :

```
self.elasticityTrackerPlugin.assignElasticityPair(_cell1 , _cell2)
```

```
self.elasticityTrackerPlugin.removeElasticityPair(_cell1 , _cell2)
```

where `_cell1` and `_cell2` denote pointers to cell objects.
These functions add or remove two cell links to or from elastic constraint. Typically they are called from Python level.

FocalPointPlasticity handles link creation and deletion in a much more convenient way so please take a look at it.

1.28 FocalPointPlasticity Plugin

Similarly as Elasticity plugin, FocalPointPlasticity pust constrains the distance between cells' center of masses. The main difference is that the list of "focal point plasticity neighbors" can change as the simulation goes and user specifies the maximum number of "focal point plasticity neighbors" a given cell can have. Let's look at relatively simple CC3DML syntax of FocalPointPlasticityPlugin (see *Demos/FocalPointPlasticity* example and we will show more complex examples later):

```
  <Plugin Name="FocalPointPlasticity">
     <Parameters Type1="Condensing" Type2="NonCondensing">
       <Lambda>10.0</Lambda>
       <ActivationEnergy>-50.0</ActivationEnergy>
       <TargetDistance>7</TargetDistance>
       <MaxDistance>20.0</MaxDistance>
       <MaxNumberOfJunctions>2</MaxNumberOfJunctions>
    </Parameters>

    <Parameters Type1="Condensing" Type2="Condensing">
       <Lambda>10.0</Lambda>
       <ActivationEnergy>-50.0</ActivationEnergy>
       <TargetDistance>7</TargetDistance>
       <MaxDistance>20.0</MaxDistance>
       <MaxNumberOfJunctions>2</MaxNumberOfJunctions>
    </Parameters>
 <NeighborOrder>1</NeighborOrder>
 </Plugin>
```

`Parameters` section describes properties of links between cells. `MaxNumberOfJunctions`, `ActivationEnergy`, `MaxDistance` and `NeighborOrder` are responsible for establishing connections between cells. CC3D constantly monitors pixel copies and during pixel copy between two neighboring cells/subcells it checks if those cells are already participating in focal point plasticity constraint. If they are not, CC3D will check if connection can be made (e.g. `Condensing` cells can have up to two connections with `Condensing` cells and up to 2 connections with `NonCondensing` cells – see first line of `Parameters` section and `MaxNumberOfJunctions` tag). The `NeighborOrder` parameter determines the pixel vicinity of the pixel that is about to be overwritten which CC3D will scan in search of the new link between cells. `NeighborOrder` 1 (which is default value if you do not specify this parameter) means that only nearest pixel neighbors will be visited. The `ActivationEnergy` parameter is added to overall energy in order to increase the odds of pixel copy which would lead to new connection.

Once cells are linked the energy calculation is carried out in a very similar way as for the Elasticity plugin:

$$E = \sum_{i,j-cellneighbors} \lambda_{ij} \left(l_{ij} - L_{ij} \right)^2$$

where l_{ij} is a distance between center of masses of cells *i* and *j* and L_{ij} is a target length corresponding to l_{ij} .

λ_{ij} and L_{ij} between different cell types are determined using `Lambda` and `TargetDistance` tags. The `MaxDistance` determines the distance between cells' center of masses when the link between those cells break. When the link breaks, then in order for the two cells to reconnect they would need to come in contact (in order to reconnect). However it is usually more likely that there will be other cells in the vicinity of separated cells so it is more likely to establish new link than restore broken one.

The above example was one of the simplest examples of use of FocalPointPlasticity. A more complicated one involves compartmental cells. In this case each cell has separate "internal" list of links between cells belonging to the same cluster and another list between cells belonging to different clusters. The energy contributions from both lists are summed up and everything that we have said when discussing example above applies to compartmental cells. Sample syntax of the FocalPointPlasticity plugin which includes compartmental cells is shown below. We use `InternalParameters` tag/section to describe links between cells of the same cluster (see *Demos/FocalPointPlasticity* example):

```
  <Plugin Name="FocalPointPlasticity">

    <Parameters Type1="Top" Type2="Top">
       <Lambda>10.0</Lambda>
       <ActivationEnergy>-50.0</ActivationEnergy>
       <TargetDistance>7</TargetDistance>
       <MaxDistance>20.0</MaxDistance>
       <MaxNumberOfJunctions NeighborOrder="1">1</MaxNumberOfJunctions>
    </Parameters>

    <Parameters Type1="Bottom" Type2="Bottom">
       <Lambda>10.0</Lambda>
       <ActivationEnergy>-50.0</ActivationEnergy>
       <TargetDistance>7</TargetDistance>
       <MaxDistance>20.0</MaxDistance>
       <MaxNumberOfJunctions NeighborOrder="1">1</MaxNumberOfJunctions>
    </Parameters>

    <InternalParameters Type1="Top" Type2="Center">
       <Lambda>10.0</Lambda>
       <ActivationEnergy>-50.0</ActivationEnergy>
       <TargetDistance>7</TargetDistance>
       <MaxDistance>20.0</MaxDistance>
       <MaxNumberOfJunctions>1</MaxNumberOfJunctions>
    </InternalParameters>

    <InternalParameters Type1="Bottom" Type2="Center">
       <Lambda>10.0</Lambda>
       <ActivationEnergy>-50.0</ActivationEnergy>
       <TargetDistance>7</TargetDistance>
       <MaxDistance>20.0</MaxDistance>
       <MaxNumberOfJunctions>1</MaxNumberOfJunctions>
    </InternalParameters>

    <NeighborOrder>1</NeighborOrder>

  </Plugin>
```

We can also specify link constituent law and change it to different form that "spring relation". To do this we use the following syntax inside FocalPointPlasticity CC3DML plugin:

```
    <LinkConstituentLaw>
     <!--The following variables lare defined by default: Lambda,Length,TargetLength-->
        <Variable Name='LambdaExtra' Value='1.0'/>
        <Formula>LambdaExtra*Lambda*(Length-TargetLength)^2</Formula>
    </LinkConstituentLaw>
```

By default CC3D defines 3 variables (Lambda, Length, TargetLength) which correspond to l_{ij}, λ_{ij} and L_{ij} from the formula above. . We can also define extra variables in the CC3DML (e.g. LambdaExtra). The actual link constituent law obeys muParser syntax convention. Once link constituent law is defined it is applied to all focal point plasticity links. The example demonstrating the use of custom link constituent law can be found in *Demos/FocalPointPlasticityCustom*.

Sometimes it is necessary to modify link parameters individually for every cell pair. In this case we would manipulate FocalPointPlasticity links using Python scripting. Example *Demos/FocalPointPlasticityCompartments* demonstrates exactly this situation. Still, you need to include CC3DML section as the one shown above for compartmental cells, because we need to tell CC3D how to link cells. The only notable difference is that in the CC3DML we have to include <Local/> tag to signal that we will set link parameters (Lambda, TaretDistance, MaxDistance) individually for each cell pair:

```
  <Plugin Name="FocalPointPlasticity">
   <Local/>
    <Parameters Type1="Top" Type2="Top">
       <Lambda>10.0</Lambda>
       <ActivationEnergy>-50.0</ActivationEnergy>
       <TargetDistance>7</TargetDistance>
       <MaxDistance>20.0</MaxDistance>
       <MaxNumberOfJunctions NeighborOrder="1">1</MaxNumberOfJunctions>
    </Parameters>
……..
</Plugin>
```

Python steppable where we manipulate cell-cell focal point plasticity link properties is shown below:

```
class FocalPointPlasticityCompartmentsParams(SteppablePy):
    def __init__(self,_simulator,_frequency=10):
        SteppablePy.__init__(self,_frequency)
        self.simulator=_simulator
        self.focalPointPlasticityPlugin=CompuCell.\
        getFocalPointPlasticityPlugin()
        self.inventory=self.simulator.getPotts().\
        getCellInventory()
        self.cellList=CellList(self.inventory)

    def step(self,mcs):
        for cell in self.cellList:
            for fppd in InternalFocalPointPlasticityDataList\
            (self.focalPointPlasticityPlugin,cell):
                self.focalPointPlasticityPlugin.\
                setInternalFocalPointPlasticityParameters\
                (cell,fppd.neighborAddress,0.0,0.0,0.0)
```

The syntax to change focal point plasticity parameters (or as here internal parameters) is as follows:

```
setFocalPointPlasticityParameters(cell1, cell2, lambda,\ targetDistance, maxDistance)
```

```
setInternalFocalPointPlasticityParameters(cell1, cell2, lambda,\ targetDistance,
maxDistance)
```

Similarly to inspect current values of the focal point plasticity parameters we would use the following Python construct:

```
for cell in self.cellList:
    for fppd in InternalFocalPointPlasticityDataList\
    (self.focalPointPlasticityPlugin,cell):
        print "fppd.neighborId",fppd.neighborAddress.id
        " lambda=",fppd.lambdaDistance
```

For non-internal parameters we simply use `FocalPointPlasticityDataList` instead of `InternalFocalPointPlasticityDataList` .

Examples *Demos/FocalPointPlasticity...* show in relatively simple way how to use FocalPointPlasticity plugin. Those examples also contain useful comments.

When using FocalPointPlasticity Plugin from mitosis module one might need to break or create focal point plasticity links. To do so FocalPointPlasticity Plugin provides 4 convenience functions which can be invoked from the Python level:

```
deleteFocalPointPlasticityLink(cell1,cell2)

deleteInternalFocalPointPlasticityLink(cell1,cell2)

createFocalPointPlasticityLink(\
cell1,cell2,lambda,targetDistance,maxDistance)

createInternalFocalPointPlasticityLink(\
cell1,cell2,lambda,targetDistance,maxDistance)
```

1.29 Curvature Plugin

This plugin implements energy term for compartmental cells. It is based on "A New Mechanism for Collective Migration in *Myxococcus xanthus*", J. Starruß, Th. Bley, L. Søgaard-Andersen and A. Deutsch, *Journal of Statistical Physics*, DOI: **10.1007/s10955-007-9298-9**, (2007). For a "long" compartmental cell composed of many subcells it imposes constraint on curvature of cells. The syntax is slightly complex:

```
   <Plugin Name="Curvature">

   <InternalParameters Type1="Top" Type2="Center">
      <Lambda>100.0</Lambda>
      <ActivationEnergy>-50.0</ActivationEnergy>
   </InternalParameters>

   <InternalParameters Type1="Center" Type2="Center">
      <Lambda>100.0</Lambda>
      <ActivationEnergy>-50.0</ActivationEnergy>
```

```
        </InternalParameters>


    <InternalParameters Type1="Bottom" Type2="Center">
        <Lambda>100.0</Lambda>
        <ActivationEnergy>-50.0</ActivationEnergy>
    </InternalParameters>

     <InternalTypeSpecificParameters>
        <Parameters TypeName="Top" MaxNumberOfJunctions="1"
        NeighborOrder="1"/>
        <Parameters TypeName="Center" MaxNumberOfJunctions="2"
        NeighborOrder="1"/>
        <Parameters TypeName="Bottom" MaxNumberOfJunctions="1"
        NeighborOrder="1"/>
    </InternalTypeSpecificParameters>


 </Plugin>
```

The `InternalTypeSpecificParameter` tells Curvature Plugin how many neighbors a cell of given type will have. In this case, numbers which make sense are 1 and 2. The middle segment will have 2 connection and head and tail segments will have only one connection with neighboring segmens (subcells). The connections are established dymamically. The way it happens is that during simulation CC3D constantly monitors pixel copies and during pixel copy between two neighboring cells/subcells it checks if those cells are already "connected" using curvature constraint. If they are not, CC3D will check if connection can be made (e.g. `Center` cells can have up to two connections and `Top` and `Bottom` only one connection). Usually establishing connections takes place at the beginning if the simulation and often happens within first Monte Carlo Step (depending on actual initial configuration, of course, but if segments touch each other connections are established almost immediately). The `ActivationEnergy` parameter is added to overall energy in order to increase the odds of pixel copy which would lead to new connection. `Lambda` tag/parameter determines "the strength" of curvature constraint. The higher the `Lambda` the more "stiff" cells will be i.e. they will tend to align along straight line.

1.30 PlayerSettings Plugin

Remark: We are working on reoplacing global Player setting with settings which will apply to current instance of the Polayer. Therefore some of the features in this plugin may not work properly. Please let us know about any problems you have with this plugin

This plugin allows users to specify or configure Player settings directly from CC3DML, without s single click. Some users might prefer this way of setting configuring Player. In addition to this if users want to run two different simulations at the same time on the same machine but with different , say, cell colors, then doing it with "regular" Player configuration file might be tricky. The solution is to use PlayerSetting Plugin. The syntax of this plugin is as follows:

```
<Plugin Name="PlayerSettings">
    <Project2D XZProj="50"/>
    <Concentration LegendEnable="true" NumberOfLegendBoxes="3"/>
```

```
    <VisualControl ScreenshotFrequency="200" ScreenUpdateFrequency="10"
    NoOutput="true" ClosePlayerAfterSimulationDone="true" />
    <Border BorderColor="red" BorderOn="false"/>
    <TypesInvisibleIn3D Types="0,2,4,5"/>
     <Cell Type="1" Color="red"/>
    <Cell Type="2" Color="yellow"/>
    <!-- Note: SaveSettings flag is unimportant for the new Player
    because whenever settings are changed from XML script
    they are written by default to disk
    This seems to be default behavior of most modern applications.
    We may implement this feature later
    <Settings SaveSettings="false"/>
    -->
</Plugin>
```

All the keywords in the above CC3DML correspond to an action in the Player.
`Project2D` sets up the values of the projection on the Player steering bar. Here we set the
player to start 2D display in the `xz` projection with `y` coordinate set to 50. Borders and
contours properties are handled using `Border` and `Contour` elements. Specifying cell
colors is done using `Cell` element. `VisualControl` element allows users to specify zoom
factor and screen update and screenshot frequencies. Screen update frequency migh not
work properly when using Python script. In this case CompuCell will use whatever
screen update frequency was stored in the config file (by default 1). We may also change
things such as screen update frequency or screenshot frequency and choose whether or
not to close the player after the simulation.
To start Player in the 3D view instead of adding `<Project2D>` tag we add `<View3D>`
section:

```
<View3D>
 <CameraClippingRange Max="388.363853764" Min="182.272762471"/>
 <CameraFocalPoint x="50.0" y="50.0" z="0.75"/>
 <CameraPosition x="150.062764552" y="-88.9777652942" z="213.639720537"/>
 <CameraViewUp x="0.112255891114" y="0.855400655565" z="0.505656339196"/>
</View3D>
```

The camera settings stored here position 3D camera. The best way to get those settings is
to run a simulation, add 2D screenshot using camera button, switch to 3D and position
camera (using mouse) however you like and subsequently add another screenshot using
camera button, save screenshot descrition file (`File->Save Screenshot`
`Description`…) and open up in text editor newly saved screenshot description file (with
.sdfml extension) and copy camera setting from there into PlayerSettings `<View3D>`
section. An example of using Player settings is shown in
Demos\cellsort_2D\cellsort_2D_PlayerSettings.xml.
Although the set of allowed changes of player settings is fairly small at the moment we
believe that the options that users have right now are quite sufficient for configuring the
Player from the CC3DML or python level. We will continue adding new options though.

1.31 BoundaryPixelTracker Plugin

This plugin allows storing list of boundary pixels for each cell. The syntax is as follows:

```
<Plugin Name="BoundaryPixelTracker">
    <NeighborOrder>1</NeighborOrder>
 </Plugin>
```

This plugin is also used by other plugins as a helper module. Example use of this plugin is found in *Demos/BoundaryPixelTrackerExample*.

1.32 GlobalBoundaryPixelTracker

This plugin tracks boundary pixels of all the cells including `Medium`. It is used in a Boundary Walker algorithm where instead of blindly picking pixel copy candidate we pick it from the set of pixels comprising boundaries of non frozen cells. In situations when lattice is large and there are not that many cells it makes sense to use BoundaryWalker algorithm to limit number of "wrong" pixel picks when perfming pixel copy attempts.

Remark: BoundaryWalkerAlgorithm does not really work with OpenMP versionof CC3D which includes all versions starting with 3.6.0.

Take a look at the following example:

```
<Potts>
   <Dimensions x="100" y="100" z="1"/>
   <Anneal>10</Anneal>
   <Steps>10000</Steps>
   <Temperature>5</Temperature>
   <Flip2DimRatio>1</Flip2DimRatio>
   <NeighborOrder>2</NeighborOrder>
   <MetropolisAlgorithm>BoundaryWalker</MetropolisAlgorithm>
   <Boundary_x>Periodic</Boundary_x>
 </Potts>

 <Plugin Name="GlobalBoundaryPixelTracker">
    <NeighborOrder>2</NeighborOrder>
 </Plugin>
```

Here we are using `BoundaryWalker` algorithm (`Potts` section) and subsequently we list GlobalBoundaryTracker plugin where we set neighbor order to match that in the Potts section. The neighbor order determines how "thick" the overall boundary of cells will be. The higher this number the more pixels will belong to the boundary.

1.33 PixelTracker Plugin

This plugin allows storing list of all pixels belonging to a given cell. The syntax is as follows:

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

This plugin is also used by other plugins (e.g. Mitosis) as a helper module. Simple example can be found in *Demos/PixelTrackerExample*.

1.34 MomentOfInertia Plugin

This plugin updates tensor of inertia for every cell. Internally it uses parallel axis theorem to calculate most up-to-date tensor of inertia. It can be called directly:

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

However, most commonly it is called indirectly by other plugins like Elongation plugin.

MomentOfInertia plugin gives users access (via Python scripting) to current lengths of cell's semiaxes. Examples in `Demos/MomentOfInertia` demonstrate how to get lengths of semiaxes:

```
axes=self.momentOfInertiaPlugin.getSemiaxes(cell)
```

axes is a 3-component vector with 0th element being length of minor axis, 1st – length of median axis (which is set to 0 in 2D) and 2nd element indicating the length of major semiaxis.

Important: Because calculating lengths of semiaxes involves quite a few of floating point operations it may happen (usually on hexagonal lattice) that for cells composed of 1, 2, or 3 pixels one moment the square of one of the semiaxes may end up being slightly negative leadind to `NaN` (not a number)length. This is due to roundoff error and whenever CC3D detects very small absolute value of square of the length of semiaxes (10^{-6}) it sets length of this semiaxes to 0.0 regardless whether the squared value is positive or negative. However it is a good practice to test whether the length of semiaxis is sane by doing simple if (here we show how to test for a `NaN`)

```
jf length!=length:
        print "length is NaN":
else:
        print "length is a proper floating point number"
```

1.35 SimpleClock plugin

This plugin adds an integer as a cell attribute:

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

1.36 ConvergentExtension plugin

Remark: This is very specialized plugin which currently is in Tier 2 plugins in terms of support. It attempts to implement energy term described in "Simulating Convergent Extension by Way of Anisotropic Differemtial Adhesion," *Zajac* M, *Jones* GL, and *Glazier* JA, Journal of Theoretical Biology **222** (2), 2003. However due to certain ambiguities in the plugin description we had difficulties to getting it to work properly.

CC3D's ConvergentExtension plugin is a somewhat simplified version of energy term described in the paper.

This plugin uses the following syntax:

```
  <Plugin Name="ConvergentExtension">
   <Alpha Type="Condensing" >0.99</Alpha>
```

```
  <Alpha Type="NonCondensing" >0.99</Alpha>
  <NeighborOrder>2</NeighborOrder>
</Plugin>
```

The `Alpha` tag represents numerical value of α parameter from the paper.

2 Steppable Section

Steppables are CompuCell modules that are called every Monte Carlo Step (MCS). More precisely, they are called after all the pixel copy attempts in a given MCS have been carried out. Steppables may have various functions - for example solving PDE's, checking if critical concentration threshold have been reached, updating target volume or target surface given the concentration of come growth factor, initializing cell field, writing numerical results to a file etc... In general, steppables perform all functions that need to be done every MCS. In the reminder of this section we will present steppables currently available in the CompuCell3D and describe their usage.

2.1 *UniformInitializer Steppable*

This steppable lays out cells on the lattice. It allows users to specify rectangular regions of field with square (or cube in 3D) cells of user defined types (or random types). Cells can be touching each other or can be separated by a gap.

The syntax of the plugin is as follows:

```
<Steppable Type="UniformInitializer">
  <Region>
    <BoxMin x="35" y="0" z="30"/>
      <BoxMax x="135" y="1" z="430"/>
      <Gap>0</Gap>
      <Width>5</Width>
    <Types>psm</Types>
  </Region>
</Steppable>
```

Above we have defined a 2D rectangular box filled with 5x5 cells touching each other (`Gap`=0) and having type `psm`. Notice that if you want to initialize 2D box in `xz` plane as above then `y_min` and `y_max` have to be 0 and 1 respectively.

Users can include as many regions as they want. The regions can overlap each other and as expected region defined later in the code overshadows the one defined earlier. As a result cells from 'earlier' regions may get overwritten by cells from regions defined later in the code. Cells that are overwritten will either disappear or be truncated.

Additionally users can initialize region with random cell types chosen from provided list of cell types:

```
<Steppable Type="UniformInitializer">
  <Region>
    <BoxMin x="35" y="0" z="30"/>
    <BoxMax x="135" y="1" z="430"/>
    <Gap>0</Gap>
```

```
      <Width>5</Width>
      <Types>psm,ncad,ncam</Types>
    </Region>
</Steppable>
```

When user specifies more than one cell type between `<Types>` tags then cells for this region will be initialized with types chosen randomly from the provided list (here the choices would be `psm, ncad, ncam`).

Remark: The types have to be separated with ',' and there should be no spaces)

Remark: If one of the type names is repeated inside `<Types>` element this type will get greater weighting means probability of assigning this type to a cell will be greater. So for example `<Types>psm,ncad,ncam,ncam,ncam</Types>` ncam will assigned to a cell with probability 3/5 and psm and `ncad` with probability 1/5.

2.2 BlobInitializer Steppable

This steppable is used to lay out circular blob of cells on the lattice. Original (**and deprecated**) syntax of this plugin looks as follows:

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

The blob is centered in the middle of th lattice and has radius given by `<Radius>` parameter all cells are initially squares (or cubes in 3D) - `<Width>` determines the length of the cube or square side and `<Gap>` determines space between squares or cubes. `<CellSortInit>` tag and value `yes` is used to initialize cells randomly with type id being either 1 or 2. Otherwise all cells will have type id 1. This can be easily modified in Python .

The **most recent syntax** for this plugin gives users additional flexibility in initializing cell field using BlobFieldInitializer:

```
<Steppable Type="BlobInitializer">
   <Region>
     <Gap>0</Gap>
     <Width>5</Width>
     <Radius>40</Radius>
     <Center x="100" y="100" z="0"/>
     <Types>Condensing,NonCondensing</Types>
   </Region>
</Steppable Type="BlobInitializer">
```

Similarly as for the UniformFieldInitializer users can define many regions each of which is a blob of a particular center point, radius and list of cell types that will be assigned to cells forming the blob. Listing types in the `<Types>` tag follows same rules as in the UniformInitializer.

2.3 PIF Initializer

To initialize the configuration of the simulation lattice we can write your own **lattice initialization file**. Our experience suggests that you will probably have to write your own initialization files rather than relying on built-in initializers. The reason is simple: the built-in initializers implement very simple cell layouts, and if you want to study more complicated cell arrangements, the built-in initializers will not be very helpful. Therefore we encourage you to learn how to prepare lattice initialization files. We have developed CellDraw tool which is a part of CC3D suite and it allows users to draw initial cell layout in a very intuitive way. We encourage you to read "Introduction to CellDraw" to familiarize yourself with this tool.

To import custom cell layouts , CompuCell3D uses very simple **Potts Initial File** (PIF) file format.It tells CompuCell3D how to lay out assign the simulation lattice pixels to cells.

The PIF consists of multiple lines of the following format:

```
cell# celltype x1 x2 y1 y2 z1 z2
```

Where `cell#` is the unique integer index of a cell, celltype is a string representing the cell's initial type, and `x1` and `x2` specify a *range* of x-coordinates contained in the cell (similarly `y1` and `y2` specify a range of y-coordinates and `z1` and `z2` specify a range of z-coordinates). Thus each line assigns a rectangular volume to a cell. If a cell is not perfectly rectangular, multiple lines can be used to build up the cell out of rectangular sub-volumes (just by reusing the `cell#` and `celltype`).

A PIF can be provided to CompuCell3D by including the steppable object **PIFInitializer.**

Let's look at a PIF example for foams:

```
0 Medium 0 101 0 101 0 0
1 Foam 13 25 0 5 0 0
2 Foam 25 39 0 5 0 0
3 Foam 39 46 0 5 0 0
4 Foam 46 57 0 5 0 0
5 Foam 57 65 0 5 0 0
6 Foam 65 76 0 5 0 0
7 Foam 76 89 0 5 0 0
```

These lines define a background of `Medium` which fills the whole lattice and is then overwritten by seven rectangular cells of type `Foam` numbered 1 through 7. Notice that these cells lie in the `xy` plane (z1=0 z2=0 implies that cells have thickness =1) so this example is a two-dimensional initialization.

You can write the PIF file manually, but using a script or program that will write PIF file for you in the language of your choice (Perl, Python, Matlab, Mathematica, C, C++, Java or any other programming language) will save a great deal of typing. You may also use tools like PIFTracer which allow you to "paint" the lattice by tracing regions of the experimental pictures.

Notice, that for compartmental cell model the format of the PIF file is different:

```
Include Clusters
cluster # cell# celltype x1 x2 y1 y2 z1 z2
```

For example:

```
Include Clusters
1 1 Side1 23 25 47 56 10 14
1 2 Center 26 30 50 54 10 14
1 3 Side2 31 33 47 56 10 14
1 4 Top 26 30 55 59 10 14
1 5 Bottom 26 30 45 49 10 14
2 6 Side1 35 37 47 56 10 14
2 7 Center 38 42 50 54 10 14
2 8 Side2 43 45 47 56 10 14
2 9 Top 38 42 55 59 10 14
2 10 Bottom 38 42 45 49 10 14
```
Remark: An easy way to generate PIF file from the current simulation snapshot is to use
Player Tools->Generate PIF file from current snapshot… menu option. Alternatively we
can use PIFDumper steppable discussed next.

2.4 PIFDumper Steppable

This steppable does the opposite to PIFItializer – it writes PIF file of current lattice
configuration. The syntax similar to the syntax of PIFInitializer:

```
<Steppable Type="PIFDumper" Frequency="100">
  <PIFName>line</PIFName>
</Steppable>
```

Notice that we used `Frequency` attribute of steppable to ensure that PIF files are written
every 100 MCS. Without it they would be written every MCS. The file names will have
the following format:

```
PIFName.MCS.pif
```

In our case they would be `line.0.pif`, `line.100.pif`, `line.200.pif` etc...

This plugin is actually quite useful. For example, if we want to start simulation from a
more configuration of cells (not rectangular cells as this is the case when we use Uniform
or Blob initializers). In such a case we would run a simulation with a PIFDumper
included and once the cell configuration reaches desired shape we would stop and use
PIF file corresponding to this state. Once we have PIF initial configuration we may run
many simulation starting from the same, realistic initial condition.
Remark: You can also generate PIF file from the current simulation snapshot by using
Player tool: Tools->Generate PIF file from current snapshot…

2.5 Mitosis Steppabe.

This steppable is described in great detail in Python tutorial but because of its importance
we are including a copy of that description here.

In developmental simulations we often need to simulate cells which grow and divide. In earlier versions of CompuCell3D we had to write quite complicated plugin to do that which was quite cumbersome and unintuitive. The only advantage of the plugin was that mitosis was taking place immediately after the pixel copy which had triggered mitosis condition. This guaranteed that any cell which was supposed divide at any instance in the simulation, actually did. However, because state of the simulation is normally observed after completion of full a Monte Carlo Step, and not in the middle of MCS it makes actually more sense to implement Mitosis as a steppable. Let us examine the simplest simulation which involves mitosis. We start with a single cell and grow it. When cell reaches critical (doubling) volume it divides. We check if the cell has reached doubling volume at the end of each MCS. The folder containing this simulation is *examples_PythonTutorial/steppableBasedMitosis*. The mitosis algorithm is implemented in *examples_PythonTutorial/steppableBasedMitosis/steppableBasedMitosisSteppables.py*

File:
examples_PythonTutorial/steppableBasedMitosis/steppableBasedMitosisSteppables.py

```
from PySteppables import *
from PySteppablesExamples import MitosisSteppableBase
import CompuCell
import sys

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

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: # mitosis condition
                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
        childCell.targetVolume=parentCell.targetVolume
        childCell.lambdaVolume=parentCell.lambdaVolume
        if parentCell.type==1:
            childCell.type=2
```

```
        else:
            childCell.type=1
```

Two steppables: `VolumeParamSteppable` and `MitosisSteppable` are the essence of the above simulation. The first steppable initializes volume constraint for all the cells present at T=0 MCS (only one cell) and then every 10 MCS (see the frequency with which VolumeParamSteppable in initialized to run - *examples_PythonTutorial/steppableBasedMitosis/steppableBasedMitosis.py*) it increases target volume of cells, effectively causing cells to grow.

The second steppable checks every 10 MCS (we can, of course, run it every MCS) if cell has reached doubling volume of 50. If it did such cell is added to the list `cells_to_divide`. After construction of `cells_to_divide` is complete we iterate over this list and divide all the cells in it.

Remark: It is important to divide cells outside the loop where we iterate over entire cell inventory. If we keep dividing cells in this loop we are adding elements to the list over which we iterate over and this might have unwanted side effects. The solution is to use use list of cells to divide as we did in the example.

Notice that we call `self.divideCellRandomOrientation(cell)` function to divide cells. Other modes of division are available as well and they are shown in *examples_PythonTutorial/steppableBasedMitosis/steppableBasedMitosisSteppables.py* as commented line with appropriate explanation.

Notice `MitosisSteppable` inherits `MitosisSteppableBase` class (defined in *PySteppablesExamples.py*).It is is the base class which ensures that after we call any of the cell dividing function (e.g. `divideCellRandomOrientation`) CompuCell3D will automatically call `updateAttributes` function as well. `updateAttributes` function is very important and we must call it in order to ensure integrity and sanity of the simulation. During mitosis a new cell is created (accessed in Python as childCell – defined in `MitosisSteppableBase` - `self.mitosisSteppable.childCell`) and as such this cell is uninitialized. It does have default attributes (read-only) of a cell such as volume, surface (if we decide to use surface constraint or SurfaceTracker plugin) but all other parameters of such cell are set to default values. In our simulation we have been setting `targetVolume` and `lambdaVolume` individually for each cell. After mitosis `childCell` will need those parameters to be set as well. To make things more interesting, in our simulation we decided to change type of cell to be different than type of parent cell. In more complex simulations where cells have more attributes which are used in the simulation, we have to make sure that in the `updateAttributes` function `childCell` and its attributes get properly initialized. It is also very common practice to change attributes of `parentCell` after mitosis as well to account for the fact that `parentCell` is not the original `parentCell` from before the mitosis.

Important: If you specify orientation vector for the mitosis the actual division will take place along the line/plane **perpendicular to this vector**.

Important: the name of the function where we update attributes after mitosis has to be exactly `updateAtttributes`. If it is called differently CC3D will not call it

automatically. We can of course call such function by hand, immediately we do the
mitosis but this is not very elegant solution.

2.6 PDESolvers in CompuCell3D

One of the most important and time consuming parts of the CC3D simulation is to solve
all sorts of Partial Differential Equations which describe behavior of certain simulation
objects (usually chemical fields). Most of the CC3D PDE solvers solve PDE with
diffusive terms. Because we are dealing with moving boundary condition problems it was
easiest and probably most practical to use explicit scheme of Finite Difference method.
Most of CC3D PDE solvers run on multi core architectures and we also have GPU
solvers which ruun and high performance GPU's and they also provide biggest speedups
in terms of performance. Because CC3D solvers were implemented at different CC3D
life cycle and often in response to particular user requests, CC3DML specification may
differ from solver to solver. However, the basic structure of CC3DML PDE solver code
follows the same pattern .

2.7 FlexibleDiffusionSolver

This steppable is one of the basic and most important modules in CompuCell3D
simulations.

Remark: starting from version 3.6.2 we developed DiffusionSolverFE which eliminates
several inconveninces of FlexibleDiffusionSolver.

As the name suggests it is responsible for solving diffusion equation but in addition to
this it also handles chemical secretion which maybe thought of as being part of general
diffusion equation.

$$\frac{\partial c}{\partial t} = D\nabla^2 c + kc + secretion$$

where k is a decay constant of concentration c and D is the diffusion constant. The term
called *secretion* has the meaning as described below.

Example syntax for FlexibleDiffusionSolverFE looks as follows:

```
<Steppable Type="FlexibleDiffusionSolverFE">
    <AutoscaleDiffusion/>
    <DiffusionField Name="FGF8">
       <DiffusionData>
           <FieldName>FGF8</FieldName>
           <DiffusionConstant>0.1</DiffusionConstant>
           <DecayConstant>0.002</DecayConstant>
              <ExtraTimesPerMCS>5</ExtraTimesPerMCS>
           <DeltaT>0.1</DeltaT>
           <DeltaX>1.0</DeltaX>
           <DoNotDiffuseTo>Bacteria</DoNotDiffuseTo>

              <InitialConcentrationExpression>x*y
              </InitialConcentrationExpression>
```

```
        </DiffusionData>

        <SecretionData>
         <Secretion Type="Amoeba">0.1</Secretion>
        </SecretionData>

        <BoundaryConditions>
         <Plane Axis="X">
          <ConstantValue PlanePosition="Min" Value="10.0"/>
          <ConstantValue PlanePosition="Max"  Value="10.0"/>
         </Plane>

         <Plane Axis="Y">
          <ConstantDerivative PlanePosition="Min" Value="10.0"/>
          <ConstantDerivative PlanePosition="Max"  Value="10.0"/>
         </Plane>
        </BoundaryConditions>

    </DiffusionField>

    <DiffusionField Name="FGF">
        <DiffusionData>
            <FieldName>FGF</FieldName>
            <DiffusionConstant>0.02</DiffusionConstant>
            <DecayConstant>0.001</DecayConstant>
            <DeltaT>0.01</DeltaT>
            <DeltaX>0.1</DeltaX>
        <DoNotDiffuseTo>Bacteria</DoNotDiffuseTo>
        </DiffusionData>
        <SecretionData>
            <SecretionOnContact Type="Medium"
             SecreteOnContactWith="Amoeba">0.1</SecretionOnContact>
            <Secretion Type="Amoeba">0.1</Secretion>
        </SecretionData>
    </DiffusionField>
 </Steppable>
```

We define sections that describe a field on which the steppable is to operate. In our case we declare just two diffusion fields.

Important: When you want to solve more than one field with the same solver field definitions have to declared inside `<Steppable Type="SolverName">` tag. Do not create multiple tags for the same solver – it will simply not work.

Inside the diffusion field we specify sections describing diffusion and secretion. Let's take a look at `DiffusionData` section first:

```
    <DiffusionField Name="FGF8">
        <DiffusionData>
            <FieldName>FGF8</FieldName>
            <DiffusionConstant>0.1</DiffusionConstant>
            <DecayConstant>0.002</DecayConstant>
           <ExtraTimesPerMCS>5</ExtraTimesPerMCS>
            <DeltaT>0.1</DeltaT>
            <DeltaX>1.0</DeltaX>
            <DoNotDiffuseTo>Bacteria</DoNotDiffuseTo>

               <InitialConcentrationExpression>x*y
               </InitialConcentrationExpression>

        </DiffusionData>
```

We give a name (`FGF8`) to the diffusion field – this is required as we will refer to this field in other modules.

> **Notice** that field name is repeated twice once in the `<DiffusionField Name="FGF8">` element and once in the `<FieldName>FGF8</FieldName>` element. The rule is that the name defined in the `<DiffusionField Name="FIELD_NAME">` element trumps the latter definition. The latter definition was used for all versions of CC3D until 3.7.2 therefore to keep old code compatible we still maintain possibility that field name will be devined using `<FieldName>FIELD_NAME</FieldName>` only.

Next we specify diffusion constant and decay constant.

> **Important:** We use Forward Euler Method to solve these equations. This is not a stable method for solving diffusion equation and we do not perform stability checks. If you enter too high diffusion constant for example you may end up with unstable (wrong) solution. Always test your parameters to make sure you are not in the unstable region.

We may also specify cells which will not participate in the diffusion. You do it using `<DoNotDiffuseTo>` tag. In this example you do not let any FGF diffuse into `Bacteria` cells. You may of course use as many as necessary `<DoNotDiffuseTo>` tags. To prevent decay of a chemical in certain cells we use syntax:

```
<DoNotDecayIn>Medium</DoNotDecayIn>
```

In addition to diffusion parameters we may specify how secretion should proceed. SecretionData section contains all the necessary information to tell CompuCell how to handle secretion. Let's study the example:

```
<SecretionData>
 <SecretionOnContact Type="Medium" SecreteOnContactWith="Amoeba">0.1</SecretionOnContact>
 <Secretion Type="Amoeba">0.1</Secretion>
</SecretionData>
```

Here we have a definition two major secretion modes. Line:

```
<Secretion Type="Amoeba">0.1</Secretion>
```

ensures that every cell of type `Amoeba` will get 0.1 increase in concentration every MCS. Line:

```
<SecretionOnContact Type="Medium" SecreteOnContactWith="Amoeba">0.1</SecretionOnContact>
```

means that cells of type `Medium` will get additional 0.1 increase in concentration but only when they touch cell of type `Amoeba`. This mode of secretion is called `SecretionOnContact`.

We can also see new CC3DML tags `<DeltaT>` and `<DeltaX>`. Their values determine the correspondence between MCS and actual time and between lattice spacing and actual spacing size. In this example for the first diffusion field one MCS corresponds to 0.1 units of actual time and lattice spacing is equal 1 unit of actual length. What is happening here is that the diffusion constant gets multiplied by:

```
DeltaT/(DeltaX* DeltaX)
```

provided the decay constant is set to 0. If the decay constant is not zero `DeltaT` appears additionally in the term (in the explicit numerical approximation of the diffusion equation solution) containing decay constant so in this case it is more than simple diffusion constant rescaling.

`DeltaT` and `DeltaX` settings are closely related to `ExtraTimesPerMCS` setting which allows calling of diffusion (and only diffusion) more than once per MCS. The number of extra calls per MCS is specified by the user on a per-field basis using `ExtraTimesPerMCS` tag.

Important: When using `ExtraTimesPerMCS` secretion functions will called only once per MCS. This is different than using PDESolverCaller where entire module is called multiple times (this include diffusion and secretion for all fields).

Remark: We recommend that you stay away from redefining DeltaX and DeltaT and assume that your diffusion/decay coefficients are expressed in units of pixel (distance) and MCS (time). This way when you assing physical time and distance usnits to MCS and pixels you can easily obtain diffusion and decay constants. DeltaX and DeltaT introduce unnecessary complications.

The `AutoscaleDiffusion` tag tells CC3D to automatically rescale diffusion constant when switching between sqaure and hex lattices. In previous versions of CC3D such scaling had to be done manually to ensure that solutions diffusion of equation on different lattices match. Here we introduced for user convenience a simple tag that does rescaling automatically. The rescaling factor comes from the fact that the discretization of the divergence term in the diffusion equation has factors such as unit lengths, using surface are and pixel/voxel volume in it. On square lattice all those values have numerical value of 1.0. On hex lattice, and for that matter of non-square latticeses, only pixel/voxel volume has numerical value of 1. All other quantities have values different than 1.0 which causes the necessity to rescale diffusion constant. The detail of the hex lattice derivation will be presented in the "Introduction to Hexagonal Lattices in CompuCell3D".

2.7.1 Instabilities of the Forward Euler Method

Most of the PDE soplvers in CC3D use Forward Euler exmplicit numerical scheme. This method is unstable for large diffusioni constant. As a matter of fact using D=0.25 with pulse initial condition will lead to instabilities in 2D. To deal with this you would normally use implicit solvers however due to moving boundary conditions that we have to deal with in CC3D simulations, memory requirements, perofmance and the fact that

most diffusion constants encountered in biology are quite low (unfortunately this is not for all chemicals e.g. oxygen) we decided to use explicit scheme. If you have to use large diffusion constants with explicit solvers you need to do rescaling:

1) Set D, Δt, Δx according to your model
2) If

$$D\frac{\Delta t}{\Delta x^2} > 0.16 \quad in\ 3D$$

you will need to call solver multiple times per MCS.

3) Set <ExtraTimesPerMCS> to N-1 where:

$$ND' = D$$

and

$$D\frac{\Delta t/N}{\Delta x^2} < 0.16 \quad in\ 3D$$

`SecretionData` sections are analogous to those defined in AdvectionDiffusionSolver. here however, the secretion is done done on per-pixel basis (as opposed to per cell basis for AdvectionDiffusionSolver). For example when we use the following CC3DML statement

```
<Secretion Type="Amoeba">0.1</Secretion>
```

this means that every pixel that belongs to cells of type `Amoebae` will get boost in concentration by 0.1. That is the secretion proceeds uniformly in the whole body of a cell. Alternative secretion mode would be the one described by the following line:

```
<SecretionOnContact Type="Medium" SecreteOnContactWith="Amoeba">0.1
</SecretionOnContact>
```

Here the secretion will take place in medium and only in those pixels belonging to `Medium` that touch directly `Amoeba`.
More secretion schemes will be added in the future.

2.7.2 Initial Conditions

We can specify initial concentration as a function of `x`, `y`, `z` coordinates using `<InitialConcentrationExpression>` tag we use muParser syntax to type the expression. Alternatively we may use `ConcentrationFileName` tag to specify a text file that contains values of concentration for every pixel:

```
<ConcentrationFileName>initialConcentration2D.txt</ConcentrationFileName>
```

 The value of concentration of the last pixel read for a given cell becomes an overall value of concentration for a cell. That is if cell has, say 8 pixels, and you specify different concentration at every pixel, then cell concentration will be the last one read from the file.

Concentration file format is as follows:

```
x y z c
```

where `x`,`y`,`z`, denote coordinate of the pixel. `c` is the value of the concentration.

Example:

```
0 0 0 1.2
0 0 1 1.4
...
```

The initial concentration can also be input from the Python script (typically in the `start` function of the steppable) but often it is more convenient to type one line of the CC3DML script than few lines in Python.

2.7.3 Boundary Conditions

All standard solvers (Flexible, Fast, and Reaction Diffusion) by default use the same boundary conditions as the GGH simulation (and those are specified in the Potts section of the CC3DML script). Users can, however, override those defaults and use customized boundary conditions for each field individually. Currently CompuCell3D supports the following boundary conditions for the diffusing fields: periodic, constant value (Dirichlet) and constant derivative (von Neumann). To specify custom boundary condition we include `<BoundaryCondition>` section inside `<DiffusionField>` tags. The `<BoundaryCondition>` section describes boundary conditions along particular axes. For example:

```
        <Plane Axis="X">
         <ConstantValue PlanePosition="Min" Value="10.0"/>
         <ConstantValue PlanePosition="Max"  Value="10.0"/>
        </Plane>
```

specifies boundary conditions along the X axis. They are Dirichlet-type boundary conditions. `PlanePosition="Min"` denotes plane parallel to `yz` plane passing through `x=0`. Similarly `PlanePosition="Min"` denotes plane parallel to `yz` plane passing through `x=fieldDimX-1` where `fieldDimX` is x dimensionof the lattice.

By analogy we specify constant derivative boundary conditions:

```
        <Plane Axis="Y">
         <ConstantDerivative PlanePosition="Min" Value="10.0"/>
         <ConstantDerivative PlanePosition="Max" Value="10.0"/>
        </Plane>
```

We can also mix types of boundary conditions along single axis:

```
        <Plane Axis="Y">
         <ConstantDerivative PlanePosition="Min" Value="10.0"/>
         <ConstantValue PlanePosition="Max" Value="0.0"/>
        </Plane>
```

Here in the `xz` plane at `y=0` we have von Neumann boundary conditions but at `y=fieldFimY-1` we have dirichlet boundary condition.

To specify periodic boundary conditions along, say `x` axis we use the following syntax:

```
        <Plane Axis="X">
            <Periodic/>
        </Plane>
```

Notice, that `<Periodic>` boundary condition specification applies to both "ends" of the axis i.e. we cannot have periodic boundary conditions at `x=0` and constant derivative at `x=fieldDimX-1`.

The FlexibleDiffusionSolver is also capable of solving simple coupled diffusion type PDE of the form:

$$\frac{\partial c}{\partial t} = D\nabla^2 c + kc + secretion + m_d cd + m_f cf$$

$$\frac{\partial d}{\partial t} = D\nabla^2 d + kd + secretion + n_c dc + n_f df$$

$$\frac{\partial f}{\partial t} = D\nabla^2 f + kf + secretion + p_c fc + p_d fd$$

where m_c , m_g , n_c , n_f , p_c , p_d are coupling coefficients. To code the above equations in CC3DML syntax you need to use the following syntax:

```
<Steppable Type="FlexibleDiffusionSolverFE">
  <DiffusionField>
    <DiffusionData>
      <FieldName>c</FieldName>
      <DiffusionConstant>0.1</DiffusionConstant>
      <DecayConstant>0.002</DecayConstant>
      <CouplingTerm InteractingFieldName="d" CouplingCoefficent="0.1"/>
      <CouplingTerm InteractingFieldName="f" CouplingCoefficent="0.2"/>
      <DeltaT>0.1</DeltaT>
      <DeltaX>1.0</DeltaX>
      <DoNotDiffuseTo>Bacteria</DoNotDiffuseTo>
    </DiffusionData>
    <SecretionData>
      <Secretion Type="Amoeba">0.1</Secretion>
    </SecretionData>
  </DiffusionField>

  <DiffusionField>
    <DiffusionData>
      <FieldName>d</FieldName>
      <DiffusionConstant>0.02</DiffusionConstant>
      <DecayConstant>0.001</DecayConstant>
      <CouplingTerm InteractingFieldName="c" CouplingCoefficent="-0.1"/>
      <CouplingTerm InteractingFieldName="f" CouplingCoefficent="-0.2"/>
      <DeltaT>0.01</DeltaT>
      <DeltaX>0.1</DeltaX>
      <DoNotDiffuseTo>Bacteria</DoNotDiffuseTo>
    </DiffusionData>
    <SecretionData>
      <Secretion Type="Amoeba">0.1</Secretion>
    </SecretionData>
  </DiffusionField>

  <DiffusionField>
    <DiffusionData>
      <FieldName>f</FieldName>
```

```
        <DiffusionConstant>0.02</DiffusionConstant>
        <DecayConstant>0.001</DecayConstant>
        <CouplingTerm InteractingFieldName="c" CouplingCoefficent="-0.2"/>
        <CouplingTerm InteractingFieldName="d" CouplingCoefficent="0.2"/>
        <DeltaT>0.01</DeltaT>
        <DeltaX>0.1</DeltaX>
        <DoNotDiffuseTo>Bacteria</DoNotDiffuseTo>
    </DiffusionData>
    <SecretionData>
    <Secretion Type="Amoeba">0.1</Secretion>
        </SecretionData>
  </DiffusionField>
</Steppable>
```

As one can see the only addition that is required to couple diffusion equations has simple
syntax:

```
<CouplingTerm InteractingFieldName="c" CouplingCoefficent="-0.1"/>
<CouplingTerm InteractingFieldName="f" CouplingCoefficent="-0.2"/>
```


2.8 DiffusionSolverFE

`DiffusionSolverFE` is new solver in 3.6.2 and is intended to fully replace
`FlexibleDiffusuionSolverFE`. It eliminates several limitations and inconveniences of
`FlexibleDiffusionSolverFE` and provides new features such as GPU implementation
or cell type dependent diffusion/decay coeeficcients. In addition it also eliminates the
need to rescale diffusion/decay/secretion constants. It checks stability condition of the
PDE and then rescales appropriately all coeeficients and computs how many extra times
per MCS the solver has to be called. It makes those extra calls automatically.

One of the key differences FlexibleDiffusionSolverFE and DiffusionSolverFE is the
way in which secretion is treated. In FlexibleDiffusionSolverFE all secretion amount is
done once followed by possibly multiple diffusion calls to diffusion (to avoid numerical
instabilities). In DiffusionSolverFE the default mode of operation is such that multiple
secretion and diffusion calls are interleaved. Meaning that instead of secreting full
amount for a given MCS and diffusing it, the DiffusionSolverFE secretes substance
gradually so that there is equal amount of secretion before each call of the diffusion. One
can change this behavior by adding `<DoNotScaleSecretion/>` to   definition of the
diffusion solver e.g.

```
   <Steppable Type="DiffusionSolverFE">
      <DoNotScaleSecretion/>
      <DiffusionField Name="ATTR">
         <DiffusionData>
       …
```

With such definition the DiffusionSolverFE will behave like FlexibleDiffusionSolverFE
as far as computation.

Remark: DiffusionSolverFE autoscales diffusion discretization depending on the lattice
so that `<AutoscaleDiffusion/>` we used in FlexibleDiffusionSolverFE is unnecessary.

This may result in slow performance so users have to be aware that those extra calls to the solver may be the cause. Typical syntax for the `DiffusionSolverFE` may look like example below:

```
    <Steppable Type="DiffusionSolverFE">
        <DiffusionField Name="ATTR">
            <DiffusionData>
                <FieldName>ATTR</FieldName>
                <GlobalDiffusionConstant>0.1</GlobalDiffusionConstant>
                <GlobalDecayConstant>5e-05</GlobalDecayConstant>
                <DiffusionCoefficient CellType="Red">0.0</DiffusionCoefficient>
            </DiffusionData>
            <SecretionData>
                <Secretion Type="Bacterium">100</Secretion>
            </SecretionData>
            <BoundaryConditions>
                <Plane Axis="X">
                    <Periodic/>
                </Plane>
                <Plane Axis="Y">
                    <Periodic/>
                </Plane>
            </BoundaryConditions>
        </DiffusionField>
    </Steppable>
```

The syntax resembles the syntax for FlexibleDiffusionSolverFE. We specify global diffusion constant by using `<GlobalDiffusionConstant>` tag. This specifies diffusion coeeficient which applies to entire region of the simulation. We can override this specification for regions occupied by certain cell types by using the following syntax:

```
<DiffusionCoefficient CellType="Red">0.0</DiffusionCoefficient>
```

Similar principles apply to decay constant and we use `<GlobalDecayConstant>` tag to specify global decay coeeficient and

```
<DecayCoefficient CellType="Red">0.0</DecayCoefficient>
```

to override global definition for regions occupied by `Red` cells.

We do not support `<DeltaX>`, `<DeltaT>` or `<ExtraTimesPerMCS>` tags.

Remark: DiffusionSolverFE autoscales diffusion discretization depending on the lattice so that `<AutoscaleDiffusion/>` we used in FlexibleDiffusionSolverFE is unnecessary.

2.8.1 GPU Solver

To run DiffusionSolverFE on GPU all we have to do (besides having OpenCL compatible GPU and correct drives installed) to replace first line of solver specification:

```
    <Steppable Type="DiffusionSolverFE">
```

with

```
<Steppable Type="DiffusionSolverFE_OpenCL">
```

2.9 AdvectionDiffusionSolver.

Remark: This is an experimental module and was not fully curated.

This steppable solves advection diffusion equation on a cell field as opposed to grid. Of course, the inaccuracies are bigger than in the case of PDE being solved on the grid but on the other hand solving the PDE on a cell field means that we associate cocentration with a given cell (not just with a lattice point). This means that as cells move so does the

concentration. In other words we get advection for free. The mathematical treatment of this kind of approximation was spelled out in Phys. Rev. E 72, 041909 (2005) paper by D.Dan et al.

The equation solved by this steppable is of the type:

$$\frac{\partial c}{\partial t} = D\nabla^2 c + kc + \vec{v} \cdot \vec{\nabla} c + secretion$$

where c denotes concentration , D is diffusion constant, k decay constant, $\vec{v}$ is velocity field.

In addition to just solving advection-diffusion equation this module allows users to specify secretion rates of the cells as well as different secretion modes. The syntax for this module follows same apttern as FlexibleDiffusionSolverFE.

Example syntax:

```
<Steppable Type="AdvectionDiffusionSolverFE">
  <DiffusionField Name="FGF">
    <DiffusionData>
      <FieldName>FGF</FieldName>
      <DiffusionConstant>0.05</DiffusionConstant>
      <DecayConstant>0.003</DecayConstant>
       <ConcentrationFileName>flowFieldConcentration2D.txt
       </ConcentrationFileName>
      <DoNotDiffuseTo>Wall</DoNotDiffuseTo>
    </DiffusionData>
    <SecretionData>
      <Secretion Type="Fluid">0.5</Secretion>
      <SecretionOnContact Type="Fluid"
      <SecreteOnContactWith="Wall">0.3</SecretionOnContact>
    </SecretionData>

  </DiffusionField>
</Steppable>
```

2.10 FastDiffusionSolver2D

FastDiffusionSolver2DFE steppable is a simplified version of the FlexibleDiffusionSolverFE steppable. It runs several times faster that flexible solver but lacks some of its features. Typical syntax is shown below:

```
<Steppable Type="FastDiffusionSolver2DFE">
   <DiffusionField Name="FGF">
     <DiffusionData>
       <UseBoxWatcher/>
       <FieldName>FGF</FieldName>
       <DiffusionConstant>0.010</DiffusionConstant>
       <DecayConstant>0.003</DecayConstant>
        <ExtraTimesPerMCS>2</ExtraTimesPerMCS>
       <DoNotDecayIn>Wall</DoNotDecay>
       <ConcentrationFileName>
       Demos/diffusion/diffusion_2D_fast_box.pulse.txt
       </ConcentrationFileName>
     </DiffusionData>
   </DiffusionField>
</Steppable>
```

In particular for fast solver you cannot specify cells into which diffusion is prohibited. However, you may specify cell types where diffusant decay is prohibited
For exmplanation how ExtraTimesPerMCS works see section on FlexibleDiffusionSolverFE.

2.11 KernelDiffusionSolver

This diffusion solver has the advantage over previous solvers that it can handle large diffusion constants. It is also stable. However, it does not accept options like `<DoNotDiffuseTo>` or `<DoNotDecayIn>`. It also requires periodic boundary conditions. Simply put KernelDiffusionSolver solves diffusion equation

$$\frac{\partial c}{\partial t} = D\nabla^2 c + kc + secretion$$

With fixed, periodic boundary conditions on the edges of the lattice. This is different from FlexibleDiffusionSolver where the boundary conditions evolve. You also need to choose a proper `Kernel` range (K) according to the value of diffusion constant. Usually when $K^2 e^{-(K^2/(4D))}$ is small (this is the main part of the integrand), the approximation convergers to the exact value.

The syntax for this solver is as follows:

```
<Steppable Type="KernelDiffusionSolver">
  <DiffusionField Name="FGF">
    <Kernel>4</Kernel>
    <DiffusionData>
      <FieldName>FGF</FieldName>
      <DiffusionConstant>1.0</DiffusionConstant>
      <DecayConstant>0.000</DecayConstant>
       <ConcentrationFileName>
      Demos/diffusion/diffusion_2D.pulse.txt
      </ConcentrationFileName>
    </DiffusionData>
  </DiffusionField>
</Steppable>
```

Inside `<DiffusionField>` tag one may also use option `<CoarseGrainFactor>` to For example:

```
<Steppable Type="KernelDiffusionSolver">
  <DiffusionField Name="FGF">
    <Kernel>4</Kernel>
    <CoarseGrainFactor>2</CoarseGrainFactor>
    <DiffusionData>
      <FieldName>FGF</FieldName>
      <DiffusionConstant>1.0</DiffusionConstant>
      <DecayConstant>0.000</DecayConstant>
      <ConcentrationFileName>
      Demos/diffusion/diffusion_2D.pulse.txt
      </ConcentrationFileName>
    </DiffusionData>
  </DiffusionField>
</Steppable>
```

2.12 ReactionDiffusionSolver

The reaction diffusion solver solves the following system of N reaction diffusion equations:

$$\frac{\partial c_1}{\partial t} = D_1 \nabla^2 c_1 + f_1\left(c_1, c_2, \ldots, c_N\right)$$

$$\frac{\partial c_2}{\partial t} = D_2 \nabla^2 c_2 + f_2\left(c_1, c_2, \ldots, c_N\right)$$

$$\vdots$$

$$\frac{\partial c_N}{\partial t} = D_N \nabla^2 c_N + f_N\left(c_1, c_2, \ldots, c_N\right)$$

Let's consider a simple example of such system:

$$\frac{\partial F}{\partial t} = 0.1 \nabla^2 F + (-0.1H)$$

$$\frac{\partial H}{\partial t} = 0.0 \nabla^2 H + 0.1F$$

It can be coded as follows:

```
<Steppable Type="ReactionDiffusionSolverFE">
  <AutoscaleDiffusion/>
  <DiffusionField Name="F">
    <DiffusionData>
      <FieldName>F</FieldName>
      <DiffusionConstant>0.010</DiffusionConstant>
      <ConcentrationFileName>
      Demos/diffusion/diffusion_2D.pulse.txt
      </ConcentrationFileName>
      <AdditionalTerm>-0.01*H</AdditionalTerm>
    </DiffusionData>
  </DiffusionField>

  <DiffusionField Name="H">
    <DiffusionData>
      <FieldName>H</FieldName>
      <DiffusionConstant>0.0</DiffusionConstant>
      <AdditionalTerm>0.01*F</AdditionalTerm>
    </DiffusionData>
  </DiffusionField>
</Steppable>
```

Notice how we implement functions *f* from the general system of reaction diffusion equations. We simply use `<AdditionalTerm>` tag and there we type arithmetic expression involving field names (tags `<FieldName>`). In addition to this we may include in those expression word `CellType`. For example:

```
<AdditionalTerm>0.01*F*CellType</AdditionalTerm>
```

This means that function *f* will depend also on `CellType` . `CellType` hodls the value of the type of the cell at particular location - `x`, `y`, `z` – of the lattice. The inclusion of the cell type might be useful if you want to use additional terms which may change depending of the cell type. Then all you have to do is to either use if statements inside

`<AdditionalTerm>` or form equivalent mathematical expression using functions allowed by muParser (http://muparser.sourceforge.net/mup_features.html#idDef2)

For example, let's assume that additional term for second equation is the following:

$$f_H = \begin{cases} 0.1*F & \text{if} \quad \text{CellType=1} \\ 0.15*F & \text{otherwise} \end{cases}$$

In such a case additional term would be coded as follows:

```
<AdditionalTerm>CellType==1 ? 0.01*F : 0.15*F</AdditionalTerm>
```

Notice that we have used here, so called ternary operator which might be familiar to you from other programing languages such as C or C++ and is equivalent to `if-then-else` statement

The syntax of the ternary (aka `if-then-else` statement) is as follows:

```
condition ? expression if condition is true : expression if condition false
```

Important: If change the above expression to

```
<AdditionalTerm>CellType<1 ? 0.01*F : 0.15*F</AdditionalTerm>
```

we will get an XML parsing error. Why? Because XML parser will think that `<1` is the beginning of the new XML elelement. To fix this you could use two approaches:
1.Present your expression as CDATA

```
<AdditionalTerm>
<![CDATA[
CellType<1 ? 0.01*F : 0.15*F
]]>
</AdditionalTerm>
```

In this case XML parser will correctly interpret the expression enclosed between `<![CDATA[` and `]]>` .

2. Replace XML using equivalent Python syntax, in which case you would code the above XML element as:

```
DiffusionDataElmnt_2.ElementCC3D("AdditionalTerm",{}," CellType<1 ? 0.01*F : 0.15*F ")
```

The moral from this story is that if like to use muParser in the XML file make sure to use this general syntax:

```
<AdditionalTerm>
<![CDATA[
YOUR EXPRESSION
]]>
</AdditionalTerm>
```

One thing to remember is that computing time of the additional term depends on the level of complexity of this term. Thus you might get some performance degradation for very complex expressions coded in muParser

Similarly as in the case of FlexibleDiffusionSolverwe may use `AutoscaleDiffusion` tag tells CC3D to automatically rescale diffusion constant. See section FlexibleDiffusionSolver or the Appendix for more information.

2.13 Steady State diffusion solver

Often in the multi-scale simulations we have to deal with chemicals which have drastically different diffusion constants. For slow diffusion fields we can use standard explicit solvers (e.g. `FlexibleDiffusionSolverFE`) but once the diffusion constant becomes large the number of extra calls to explicit solvers becomes so large that solving diffusion euation using Forward-Euler based solvers is simply impractical. In situations where the diffusion constant is so large that the solution of the diffusion equation is not that much different from the asymptotic solution (i.e. at $t = \infty$) it is often more convenient to use `SteadyStateDiffusion` solver which solves Helmholtz equation:

$$\nabla^2 c - kc = F$$

where *F* is a source function of the coordinates - it is an input to the equation, *k* is decay constant and *c* is the concentration. The F function in CC3D is either given implicitly by specifying cellular secretion or explicitly by specifying concentration *c* before solving Helmholtz equation.

The CC3D stead state diffusion solvers are stable and allow solutions for large values of diffusion constants.

The example syntax for the steady-state solver is shown below:

```
<Steppable Type="SteadyStateDiffusionSolver2D">
    <DiffusionField Name="INIT">
       <DiffusionData>
           <FieldName>INIT</FieldName>
           <DiffusionConstant>1.0</DiffusionConstant>
           <DecayConstant>0.01</DecayConstant>
       </DiffusionData>
    <SecretionData>
           <Secretion Type="Body1">1.0</Secretion>
       </SecretionData>

       <BoundaryConditions>

           <Plane Axis="X">
               <ConstantValue PlanePosition="Min" Value="10.0"/>
               <ConstantValue PlanePosition="Max"  Value="5.0"/>
           </Plane>

           <Plane Axis="Y">
               <ConstantDerivaive PlanePosition="Min" Value="0.0"/>
               <ConstantDerivaive PlanePosition="Max"  Value="0.0"/>
           </Plane>

       </BoundaryConditions>

    </DiffusionField>
```

```
   </Steppable>
```

The syntax is is similar (actually, almost identical) to the syntax of the `FlexibleDiffusionSolverFE`. The only difference is that while `FlexibleDiffusionSolverFE` works in in both 2D and 3D users need to specify the dimensionality of the steady state solver. We use

```
<Steppable Type="SteadyStateDiffusionSolver2D">
```

for 2D simulations when all the cells lie in the xy plane and

```
<Steppable Type="SteadyStateDiffusionSolver">
```

for simulations in 3D.

We can use Python to control secretion in the steady state solvers but it requires a little bit of low level coding. Implementing secretion in steady state diffusion solver is different from "regular" Forward Euler solvers. Steady state solver takes secretion rate that is specified at t=0 and returns the solution at t=∞. For alrge diffusion constants we approximate solution to the PDE during one MCS by using solution at t=∞. However that means that if at each MCS secretion changes we have to do three things 1) zero entire field, 2) set secretion rate 3) solve steady state solver. The reason we need to zero entire field is because any value left in the field at mcs=N is interpreted by the solver as a secretion constant at this location at mcs=N+1. **Moreover the the secretion constant needs to have negative value if we want to secrete positive amount of substance - this weird requirements comes from the fact that we re using 3rd party solver which inverts signs of the secretion constants.**

An example below demonstrates how we control secretion of the steady state in Python. First we need to include tag `<ManageSecretionInPython/>` in the XML definition of the solver:

```
<Steppable Type="SteadyStateDiffusionSolver2D">
    <DiffusionField>
        <ManageSecretionInPython/>
        <DiffusionData>
            <FieldName>FGF</FieldName>
            <DiffusionConstant>1.00</DiffusionConstant>
            <DecayConstant>0.00001</DecayConstant>
        </DiffusionData>
    </DiffusionField>
 </Steppable>
```

In Python the code to control the secretion involves iteration over every pixel and adjusting concentration (which as we mentioned will be interpreted by the solver as a secretion constant) and we have to make sure that we inherit from SecretionBasePy not SteppableBasePy to ensure proper ordering of calls to Python module and the C++

diffusion solver. **Important:** make sure you inherit from SecretionBasePy when you try to manage secretion in the steady state solver using Python. This will ensure proper ordering of calls to steppable and to C++ solver code.

Important: Once you use `<ManageSecretionInPython/>` tag in the XML all secretion tags in the SecretionData will be ignored. In other words, for this solver you cannot mix secretion specification in Python and secretion specification in the XML.

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


    def start(self):
        self.field=CompuCell.getConcentrationField\
        (self.simulator,"FGF")

        secrConst=10
        for x,y,z in self.everyPixel(1,1,1):
            cell=self.cellField[x,y,z]
            if cell and cell.type==1:
                self.field[x,y,z]=-secrConst
            else:
                self.field[x,y,z]=0.0

    def step(self, mcs):

        secrConst=mcs
        for x,y,z in self.everyPixel(1,1,1):
            cell=self.cellField[x,y,z]
            if cell and cell.type==1:
                self.field[x,y,z]=-secrConst
            else:
                self.field[x,y,z]=0.0
```

Notice that all the pixels that do not secrete have to be 0.0 as mentioned above. **If you don't initialize field values in the non-secreting pixels to 0.0 you will get wrong results**. The above code, with comments, is available in our Demo suite.

2.14 BoxWatcher Steppable

Remark: Functionality of this module has been reduced in CC3D versions which support parallel computations (3.6.0 and up). Main motivation for this module was to speed up computations but with parallel version the need for this module is somewhat smaller.

This steppable can potentially speed-up your simulation. Every MCS (or every `Frequency` MCS) it determines maximum and minimum coordinates of cells and then imposes slightly bigger box around cells and ensures that in the subsequent MCS pixel copy attempts take place only inside this box containing cells (plus some amount of medium on the sides). Thus instead of sweeping entire lattice and attempting random pixel copies CompuCell3D will only spend time trying flips inside the box. Depending on the simulation the performance gains are up to approx. 30%. The steppable will work best

if you have simulation with cells localized in one region of the lattice with lots of empty space. The steppable will adjust box every MCS (or every `Frequency` MCS) according to evolving cellular pattern.

The syntax is as follows:

```
<Steppable Type="BoxWatcher">
    <XMargin>5</XMargin>
    <YMargin>5</YMargin>
    <ZMargin>5</ZMargin>
</Steppable>
```

All that is required is to specify amount of extra space (expressed in units of pixels) that needs to be added to a tight box i.e. the box whose sides just touch most peripheral cells' pixels.

2.15 muParser

CC3D uses muParser to allow users specify simple mathematical expressions in the XML (or XML-equivalent Python scripts). The following link points to full specification of the muParser: http://muparser.sourceforge.net/mup_features.html#idDef2. The general guideline to using muParser syntax inside XML is to enclose muParser expression between `<![CDATA[` and `]]>` :

```
<XML_ELEMENT_WITH_MUPARSER_EXPRESSION>
<![CDATA[
MUPARSER EXPRESSION
]]>
</ XML_ELEMENT_WITH_MUPARSER_EXPRESSION >
```

For example:

```
<AdditionalTerm>
<![CDATA[
CellType<1 ? 0.01*F : 0.15*F
]]>
</AdditionalTerm>
```

The reason for enclosing muParser expression between `<![CDATA[` and `]]>` is to prevent XML parser from interpreting `<` or `>` as beginning or end of the XML elements

Alternatively you may replace XML with equivalent Python syntax in which case things will look a bit simpler:

```
DiffusionDataElmnt_2.ElementCC3D("AdditionalTerm",{}," CellType<1 ? 0.01*F : 0.15*F ")
```

2.16 Additional Plugins and Modules

Besides the modules that were introduced above CompuCell3D contains other modules which were developed to solve particular problem. For example module called DictyFieldInitializer is used to prepare initial cell configuration for the simulation of *Dictyostelium discoideum* morphogenesis based on the paper by P.Hogeweg and N.Savill

Modelling morphogenesis: from single cells to crawling slugs. J. theor. Biol. 184, 229-235.

Such modules have limited area of applicability and are mostly used in a single simulation. For this reason we will not describe them in more detail here. Interested user may consult CompuCell3D manual 3.2.0 where all such modules were described. It is our goal however to eliminate a need to write customized modules as much as possible. For example, DictyFieldInitializer can be easily replaced by using UniformInitializer and defining several regions there. Similarly Reaction diffusion solver for this simulation can be replaced by a general Reaction Diffusion solver described above.

While we might run into performance issues when using general as opposed to customized, the flexibility and portability associated with using general use modules are worth extra run time.

3 References

1.	Bassingthwaighte, J. B. (2000) Strategies for the Physiome project. *Annals of Biomedical Engineering* **28**, 1043-1058.
2.	Merks, R. M. H., Newman, S. A., and Glazier, J. A. (2004) Cell-oriented modeling of *in vitro* capillary development. *Lecture Notes in Computer Science* **3305**, 425-434.
3.	Turing, A. M. (1953) The Chemical Basis of Morphogenesis. *Philosophical Transactions of the Royal Society B* **237**, 37-72.
4.	Merks, R. M. H. and Glazier, J. A. (2005) A cell-centered approach to developmental biology. *Physica A* **352**, 113-130.
5.	Dormann, S. and Deutsch, A. (2002) Modeling of self-organized avascular tumor growth with a hybrid cellular automaton. *In Silico Biology* **2**, 1-14.
6.	dos Reis, A. N., Mombach, J. C. M., Walter, M., and de Avila, L. F. (2003) The interplay between cell adhesion and environment rigidity in the morphology of tumors. *Physica A* **322**, 546-554.
7.	Drasdo, D. and Hohme, S. (2003) Individual-based approaches to birth and death in avascular tumors. *Mathematical and Computer Modelling* **37**, 1163-1175.
8.	Holm, E. A., Glazier, J. A., Srolovitz, D. J., and Grest, G. S. (1991) Effects of Lattice Anisotropy and Temperature on Domain Growth in the Two-Dimensional Potts Model. *Physical Review A* **43**, 2662-2669.
9.	Turner, S. and Sherratt, J. A. (2002) Intercellular adhesion and cancer invasion: A discrete simulation using the extended Potts model. *Journal of Theoretical Biology* **216**, 85-100.
10.	Drasdo, D. and Forgacs, G. (2000) Modeling the interplay of generic and genetic mechanisms in cleavage, blastulation, and gastrulation. *Developmental Dynamics* **219**, 182-191.
11.	Drasdo, D., Kree, R., and McCaskill, J. S. (1995) Monte-Carlo approach to tissue-cell populations. *Physical Review E* **52**, 6635-6657.
12.	Longo, D., Peirce, S. M., Skalak, T. C., Davidson, L., Marsden, M., and Dzamba, B. (2004) Multicellular computer simulation of morphogenesis: blastocoel roof thinning and matrix assembly in *Xenopus laevis*. *Developmental Biology* **271**, 210-222.
13.	Collier, J. R., Monk, N. A. M., Maini, P. K., and Lewis, J. H. (1996) Pattern formation by lateral inhibition with feedback: A mathematical model of Delta-Notch intercellular signaling. *Journal of Theoretical Biology* **183**, 429-446.
14.	Honda, H. and Mochizuki, A. (2002) Formation and maintenance of distinctive cell patterns by coexpression of membrane-bound ligands and their receptors. *Developmental Dynamics* **223**, 180-192.
15.	Moreira, J. and Deutsch, A. (2005) Pigment pattern formation in zebrafish during late larval stages: A model based on local interactions. *Developmental Dynamics* **232**, 33-42.
16.	Wearing, H. J., Owen, M. R., and Sherratt, J. A. (2000) Mathematical modelling of juxtacrine patterning. *Bulletin of Mathematical Biology* **62**, 293-320.
17.	Zhdanov, V. P. and Kasemo, B. (2004) Simulation of the growth of neurospheres.

Europhysics Letters **68**, 134-140.

18. Ambrosi, D., Gamba, A., and Serini, G. (2005) Cell directional persistence and chemotaxis in vascular morphogenesis. *Bulletin of Mathematical Biology* **67**, 195-195.

19. Gamba, A., Ambrosi, D., Coniglio, A., de Candia, A., di Talia, S., Giraudo, E., Serini, G., Preziosi, L., and Bussolino, F. (2003) Percolation, morphogenesis, and Burgers dynamics in blood vessels formation. *Physical Review Letters* **90**, 118101.

20. Novak, B., Toth, A., Csikasz-Nagy, A., Gyorffy, B., Tyson, J. A., and Nasmyth, K. (1999) Finishing the cell cycle. *Journal of Theoretical Biology* **199**, 223-233.

21. Peirce, S. M., van Gieson, E. J., and Skalak, T. C. (2004) Multicellular simulation predicts microvascular patterning and *in silico* tissue assembly. *FASEB Journal* **18**, 731-733.

22. Merks, R. M. H., Brodsky, S. V., Goligorksy, M. S., Newman, S. A., and Glazier, J. A. (2006) Cell elongation is key to *in silico* replication of *in vitro* vasculogenesis and subsequent remodeling. *Developmental Biology* **289**, 44-54.

23. Merks, R. M. H. and Glazier, J. A. (2005) Contact-inhibited chemotactic motility can drive both vasculogenesis and sprouting angiogenesis. *q-bio/0505033*.

24. Kesmir, C. and de Boer., R. J. (2003) A spatial model of germinal center reactions: cellular adhesion based sorting of B cells results in efficient affinity maturation. *Journal of Theoretical Biology* **222**, 9-22.

25. Meyer-Hermann, M., Deutsch, A., and Or-Guil, M. (2001) Recycling probability and dynamical properties of germinal center reactions. *Journal of Theoretical Biology* **210**, 265-285.

26. Nguyen, B., Upadhyaya, A. van Oudenaarden, A., and Brenner, M. P. (2004) Elastic instability in growing yeast colonies. *Biophysical Journal* **86**, 2740-2747.

27. Walther, T., Reinsch, H., Grosse, A., Ostermann, K., Deutsch, A., and Bley, T. (2004) Mathematical modeling of regulatory mechanisms in yeast colony development. *Journal of Theoretical Biology* **229**, 327-338.

28. Borner, U., Deutsch, A., Reichenbach, H., and Bar, M. (2002) Rippling patterns in aggregates of *myxobacteria* arise from cell-cell collisions. *Physical Review Letters* **89**, 078101.

29. Bussemaker, H. J., Deutsch, A., and Geigant, E. (1997) Mean-field analysis of a dynamical phase transition in a cellular automaton model for collective motion. *Physical Review Letters* **78**, 5018-5021.

30. Dormann, S., Deutsch, A., and Lawniczak, A. T. (2001) Fourier analysis of Turing-like pattern formation in cellular automaton models. *Future Generation Computer Systems* **17**, 901-909.

31. Börner, U., Deutsch, A., Reichenbach, H., and Bär, M. (2002) Rippling patterns in aggregates of myxobacteria arise from cell-cell collisions. *Physical Review Letters* **89,** 078101.

32. Zhdanov, V. P. and Kasemo, B. (2004) Simulation of the growth and differentiation of stem cells on a heterogeneous scaffold. *Physical Chemistry Chemical Physics* **6**, 4347-4350.

33. Knewitz, M. A. and Mombach, J. C. (2006) Computer simulation of the influence of cellular adhesion on the morphology of the interface between tissues of

proliferating and quiescent cells. Computers in Biology and Medicine **36**, 59-69.

34. Marée, A. F. M. and Hogeweg, P. (2001) How amoeboids self-organize into a fruiting body: Multicellular coordination in Dictyostelium discoideum. _Proceedings of the National Academy of Sciences of the USA_ **98**, 3879-3883.

35. Marée, A. F. M. and Hogeweg, P. (2002) Modelling Dictyostelium discoideum morphogenesis: the culmination. _Bulletin of Mathematical Biology_ **64**, 327-353.

36. Marée, A. F. M., Panfilov, A. V., and Hogeweg, P. (1999) Migration and thermotaxis of _Dictyostelium discoideum_ slugs, a model study. _Journal of Theoretical Biology_ **199**, 297-309.

37. Savill, N. J. and Hogeweg, P. (1997) Modelling morphogenesis: From single cells to crawling slugs. _Journal of Theoretical Biology_ **184**, 229-235.

38. Hogeweg, P. (2000) Evolving mechanisms of morphogenesis: on the interplay between differential adhesion and cell differentiation. _Journal of Theoretical Biology_ **203**, 317-333.

39. Johnston, D. A. (1998) Thin animals. _Journal of Physics A_ **31**, 9405-9417.

40. Groenenboom, M. A. and Hogeweg, P. (2002) Space and the persistence of male-killing endosymbionts in insect populations. _Proceedings in Biological Sciences_ **269**, 2509-2518.

41. Groenenboom, M. A., Maree, A. F., and Hogeweg, P. (2005) The RNA silencing pathway: the bits and pieces that matter. _PLoS Computational Biology_ **1**, 155-165.

42. Kesmir, C., van Noort, V., de Boer, R. J., and Hogeweg, P. (2003) Bioinformatic analysis of functional differences between the immunoproteasome and the constitutive proteasome. _Immunogenetics_ **55**, 437-449.

43. Pagie, L. and Hogeweg, P. (2000) Individual- and population-based diversity in restriction-modification systems. _Bulletin of Mathematical Biology_ **62**, 759-774.

44. Silva, H. S. and Martins, M. L. (2003) A cellular automata model for cell differentiation. _Physica A_ **322**, 555-566.

45. Zajac, M., Jones, G. L., and Glazier, J. A. (2000) Model of convergent extension in animal morphogenesis. _Physical Review Letters_ **85**, 2022-2025.

46. Zajac, M., Jones, G. L., and Glazier, J. A. (2003) Simulating convergent extension by way of anisotropic differential adhesion. _Journal of Theoretical Biology_ **222**, 247-259.

47. Savill, N. J. and Sherratt, J. A. (2003) Control of epidermal stem cell clusters by Notch-mediated lateral induction. _Developmental Biology_ **258**, 141-153.

48. Mombach, J. C. M, de Almeida, R. M. C., Thomas, G. L., Upadhyaya, A., and Glazier, J. A. (2001) Bursts and cavity formation in Hydra cells aggregates: experiments and simulations. _Physica A_ **297**, 495-508.

49. Rieu, J. P., Upadhyaya, A., Glazier, J. A., Ouchi, N. B. and Sawada, Y. (2000) Diffusion and deformations of single hydra cells in cellular aggregates. _Biophysical Journal_ **79**, 1903-1914.

50. Mochizuki, A. (2002) Pattern formation of the cone mosaic in the zebrafish retina: A cell rearrangement model. _Journal of Theoretical Biology_ **215**, 345-361.

51. Takesue, A., Mochizuki, A., and Iwasa, Y. (1998) Cell-differentiation rules that generate regular mosaic patterns: Modelling motivated by cone mosaic formation in fish retina. _Journal of Theoretical Biology_ **194**, 575-586.

52. Dallon, J., Sherratt, J., Maini, P. K., and Ferguson, M. (2000) Biological

implications of a discrete mathematical model for collagen deposition and alignment in dermal wound repair. *IMA Journal of Mathematics Applied in Medicine and Biology* **17**, 379-393.

53. Maini, P. K., Olsen, L., and Sherratt, J. A. (2002) Mathematical models for cell-matrix interactions during dermal wound healing. *International Journal of Bifurcations and Chaos* **12**, 2021-2029.

54. Kreft, J. U., Picioreanu, C., Wimpenny, J. W. T., and van Loosdrecht, M. C. M. (2001) Individual-based modelling of biofilms. *Microbiology* **147**, 2897-2912.

55. Picioreanu, C., van Loosdrecht, M. C. M., and Heijnen, J. J. (2001) Two-dimensional model of biofilm detachment caused by internal stress from liquid flow. *Biotechnology and Bioengineering* **72**, 205-218.

56. van Loosdrecht, M. C. M., Heijnen, J. J., Eberl, H., Kreft, J., and Picioreanu, C. (2002) Mathematical modelling of biofilm structures. *Antonie Van Leeuwenhoek International Journal of General and Molecular Microbiology* **81**, 245-256.

57. Popławski, N. J., Shirinifard, A., Swat, M., and Glazier, J. A. (2008) Simulations of single-species bacterial-biofilm growth using the Glazier-Graner-Hogeweg model and the CompuCell3D modeling environment. *Mathematical Biosciences and Engineering* **5**, 355-388.

58. Chaturvedi, R., Huang, C., Izaguirre, J. A., Newman, S. A., Glazier, J. A., Alber, M. S. (2004) A hybrid discrete-continuum model for 3-D skeletogenesis of the vertebrate limb. *Lecture Notes in Computer Science* **3305**, 543-552.

59. Popławski, N. J., Swat, M., Gens, J. S., and Glazier, J. A. (2007) Adhesion between cells, diffusion of growth factors, and elasticity of the AER produce the paddle shape of the chick limb. *Physica A* **373**, 521-532.

60. Glazier, J. A. and Weaire, D. (1992) The Kinetics of Cellular Patterns. *Journal of Physics: Condensed Matter* **4**, 1867-1896.

61. Glazier, J. A. (1993) Grain Growth in Three Dimensions Depends on Grain Topology. *Physical Review Letters* **70**, 2170-2173.

62. Glazier, J. A., Grest, G. S., and Anderson, M. P. (1990) Ideal Two-Dimensional Grain Growth. In *Simulation and Theory of Evolving Microstructures*, M. P. Anderson and A. D. Rollett, editors. The Minerals, Metals and Materials Society, Warrendale, PA, pp. 41-54.

63. Glazier, J. A., Anderson, M. P., and Grest, G. S. (1990) Coarsening in the Two-Dimensional Soap Froth and the Large-Q Potts Model: A Detailed Comparison. *Philosophical Magazine B* **62**, 615-637.

64. Grest, G. S., Glazier, J. A., Anderson, M. P., Holm, E. A., and Srolovitz, D. J. (1992) Coarsening in Two-Dimensional Soap Froths and the Large-Q Potts Model. *Materials Research Society Symposium* **237**, 101-112.

65. Jiang, Y. and Glazier, J. A. (1996) Extended Large-Q Potts Model Simulation of Foam Drainage. *Philosophical Magazine Letters* **74**, 119-128.

66. Jiang, Y., Levine, H., and Glazier, J. A. (1998) Possible Cooperation of Differential Adhesion and Chemotaxis in Mound Formation of *Dictyostelium*. *Biophysical Journal* **75**, 2615-2625.

67. Jiang, Y., Mombach, J. C. M., and Glazier, J. A. (1995) Grain Growth from Homogeneous Initial Conditions: Anomalous Grain Growth and Special Scaling States. *Physical Review E* **52**, 3333-3336.

68. Jiang, Y., Swart, P. J., Saxena, A., Asipauskas, M., and Glazier, J. A. (1999) Hysteresis and Avalanches in Two-Dimensional Foam Rheology Simulations. *Physical Review E* **59**, 5819-5832.
69. Ling, S., Anderson, M. P., Grest, G. S., and Glazier, J. A. (1992) Comparison of Soap Froth and Simulation of Large-Q Potts Model. *Materials Science Forum* **94-96**, 39-47.
70. Mombach, J. C. M. (2000) Universality of the threshold in the dynamics of biological cell sorting. *Physica A* **276**, 391-400.
71. Weaire, D. and Glazier, J. A. (1992) Modelling Grain Growth and Soap Froth Coarsening: Past, Present and Future. *Materials Science Forum* **94-96**, 27-39.
72. Weaire, D., Bolton, F., Molho, P., and Glazier, J. A. (1991) Investigation of an Elementary Model for Magnetic Froth. *Journal of Physics: Condensed Matter* **3**, 2101-2113.
73. Glazer, J. A., Balter, A., Popławski, N. (2007) Magnetization to Morphogenesis: A Brief History of the Glazier-Graner-Hogeweg Model. In *Single-Cell-Based Models in Biology and Medicine*. Anderson, A. R. A., Chaplain, M. A. J., and Rejniak, K. A., editors. Birkhauser Verlag Basel, Switzerland. pp. 79-106.
74. Walther, T., Reinsch, H., Ostermann, K., Deutsch, A. and Bley, T. (2005) Coordinated growth of yeast colonies: experimental and mathematical analysis of possible regulatory mechanisms. *Engineering Life Sciences* **5**, 115-133.
75. Keller, E. F. and Segel., L. A. (1971) Model for chemotaxis. *Journal of Theoretical Biology* **30**, 225-234.
76. Glazier, J. A. and Upadhyaya, A. (1998) First Steps Towards a Comprehensive Model of Tissues, or: A Physicist Looks at Development. In *Dynamical Networks in Physics and Biology: At the Frontier of Physics and Biology*, D. Beysens and G. Forgacs editors. EDP Sciences/Springer Verlag, Berlin, pp. 149-160.
77. Glazier, J. A. and Graner, F. (1993) Simulation of the differential adhesion driven rearrangement of biological cells. *Physical Review E* **47**, 2128-2154.
78. Glazier, J. A. (1993) Cellular Patterns. *Bussei Kenkyu* **58**, 608-612.
79. Glazier, J. A. (1996) Thermodynamics of Cell Sorting. *Bussei Kenkyu* **65**, 691-700.
80. Glazier, J. A., Raphael, R. C., Graner, F., and Sawada, Y. (1995) The Energetics of Cell Sorting in Three Dimensions. In *Interplay of Genetic and Physical Processes in the Development of Biological Form*, D. Beysens, G. Forgacs, F. Gaill, editors. World Scientific Publishing Company, Singapore, pp. 54-66.
81. Graner, F. and Glazier, J. A. (1992) Simulation of biological cell sorting using a 2-dimensional extended Potts model. *Physical Review Letters* **69**, 2013-2016.
82. Mombach, J. C. M and Glazier, J. A. (1996) Single Cell Motion in Aggregates of Embryonic Cells. *Physical Review Letters* **76**, 3032-3035.
83. Mombach, J. C. M., Glazier, J. A., Raphael, R. C., and Zajac, M. (1995) Quantitative comparison between differential adhesion models and cell sorting in the presence and absence of fluctuations. *Physical Review Letters* **75**, 2244-2247.
84. Cipra, B. A. (1987) An Introduction to the Ising-Model. *American Mathematical Monthly* **94**, 937-959.
85. Metropolis, N., Rosenbluth, A., Rosenbluth, M. N., Teller, A. H., and Teller, E. (1953) Equation of state calculations by fast computing machines. *Journal of*

Chemical Physics **21**, 1087-1092.

86.   Forgacs, G. and Newman, S. A. (2005). *Biological Physics of the Developing Embryo*. Cambridge Univ. Press, Cambridge.

87.   Alber, M. S., Kiskowski, M. A., Glazier, J. A., and Jiang, Y. On cellular automation approaches to modeling biological cells. In *Mathematical Systems Theory in Biology, Communication and Finance*. J. Rosenthal, and D. S. Gilliam, editors. Springer-Verlag, New York, pp. 1-40.

88.   Alber, M. S., Jiang, Y., and Kiskowski, M. A. (2004) Lattice gas cellular automation model for rippling and aggregation in *myxobacteria*. *Physica D* **191**, 343-358.

89.   Novak, B., Toth, A., Csikasz-Nagy, A., Gyorffy, B., Tyson, J. A., and Nasmyth, K. (1999) Finishing the cell cycle. *Journal of Theoretical Biology* **199**, 223-233.

90.   Upadhyaya, A., Rieu, J. P., Glazier, J. A., and Sawada, Y. (2001) Anomalous Diffusion in Two-Dimensional Hydra Cell Aggregates. *Physica A* **293**, 549-558.

91.   Cickovski, T., Aras, K., Alber, M. S., Izaguirre, J. A., Swat, M., Glazier, J. A., Merks, R. M. H., Glimm, T., Hentschel, H. G. E., Newman, S. A. (2007) From genes to organisms via the cell: a problem-solving environment for multicellular development. *Computers in Science and Engineering* **9**, 50-60.

92.   Izaguirre, J.A., Chaturvedi, R., Huang, C., Cickovski, T., Coffland, J., Thomas, G., Forgacs, G., Alber, M., Hentschel, G., Newman, S. A., and Glazier, J. A. (2004) CompuCell, a multi-model framework for simulation of morphogenesis. *Bioinformatics* **20**, 1129-1137.

93.   Armstrong, P. B. and Armstrong, M. T. (1984) A role for fibronectin in cell sorting out. *Journal of Cell Science* **69**, 179-197.

94.   Armstrong, P. B. and Parenti, D. (1972) Cell sorting in the presence of cytochalasin B. *Journal of Cell Science* **55**, 542-553.

95.   Glazier, J. A. and Graner, F. (1993) Simulation of the differential adhesion driven rearrangement of biological cells. *Physical Review E* **47**, 2128-2154.

96.   Glazier, J. A. and Graner, F. (1992) Simulation of biological cell sorting using a two-dimensional extended Potts model. *Physical Review Letters* **69**, 2013-2016.

97.   Ward, P. A., Lepow, I. H., and Newman, L. J. (1968) Bacterial factors chemotactic for polymorphonuclear leukocytes. *American Journal of Pathology* **52**, 725-736.

98.   Lutz, M. (1999) *Learning Python*. Sebastopol, CA: O'Reilly & Associates, Inc.

99.   Balter, A. I., Glazier, J. A., and Perry, R. (2008) Probing soap-film friction with two-phase foam flow. *Philosophical Magazine, submitted.*

100.   Dvorak, P., Dvorakova, D., and Hampl, A. (2006) Fibroblast growth factor signaling in embryonic and cancer stem cells. *FEBS Letters* **580**, 2869-2287.

4 Appendix

4.1 Calculating Inertia Tensor in CompuCell3D.

For each cell the inertia tensor is defined as follows:

$$I = \begin{bmatrix} \sum_i y_i^2 + z_i^2 & -\sum_i x_i y_i & -\sum_i x_i z_i \\ -\sum_i x_i y_i & \sum_i x_i^2 + z_i^2 & -\sum_i y_i z_i \\ -\sum_i x_i z_i & -\sum_i y_i z_i & \sum_i x_i^2 + y_i^2 \end{bmatrix}$$

where index 'i' denotes i-th pixel of a given cell and x_i, y_i, z_i are coordinates of that pixel in a given coordinate frame.
where index 'i' denotes i-th pixel of a given cell and x_i, y_i, z_i are coordinates of that pixel in a given coordinate frame.

In Figure 4 we show one possible coordinate frame in which one can calculate inertia tensor. If the coordinate frame is fixed calculating components of inertia tensor for cell gaining or losing one pixel is quite easy. We will be adding and subtracting terms like $y_i^2 + z_i^2$ or $x_i y_i$. However, in CompuCell3D we are mostly interested in knowing tensor of inertia of a cell with respect to *xyz* coordinate frame with origin at the center of mass (*COM*) of a given cell as shown in Figure 3 Now, to calculate such tensor we cannot
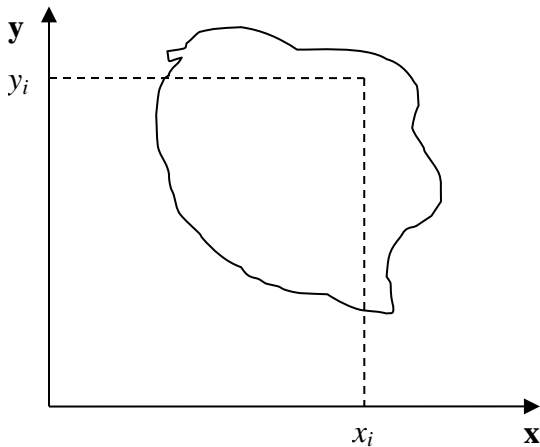
Figure 4. Cell and its coordinate frame in which we calculate inertia tensor

simply add or subtract terms like $y_i^2 + z_i^2$ or $x_i y_i$ to account for lost or gained pixel. If a cell gains or loses a pixel its COM coordinates change. If so then all the x_i, y_i, z_i coordinates that appear in the inertia tensor expression will have different value. Thus for each change in cell shape (gain or loss of pixel) we would have to recalculate inertia tensor from scratch. This would be quite time consuming and would require us to keep track of all the pixels belonging to a given cell. It turns out however that there is a better way of keeping track of inertia tensor for cells. We will be using
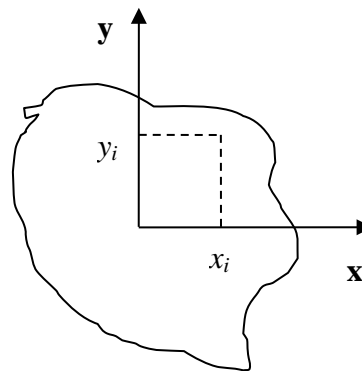
Figure 3 Cell and coordinate system passing through center of mass of a cell. Notice that as cell changes shape the position of center of mass moves.

parallel axis theorem to do the calculations. Paralel axis theorem states that if I_{COM} is a moment of inertia with respect to axis passing through center of mass then we can calculate moment of inertia with respect to any parallel axis to the one passin through the COM by using the following formula:

$$I_{x'x'} = I_{xx} + Md^2$$

where I_{xx} denotes moment of inertia with respect to x axis passing through center of mass, $I_{x'x'}$ is a moment of inertia with respect to axis parallel to the x axis passing through center of mass, d is the distance between the axes and M is mass of the cell.

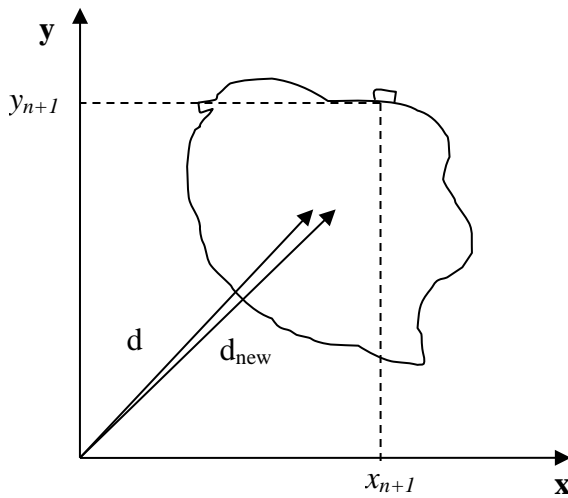Let us now draw a picture of a cell gaining one pixel:

Figure 5. Cell gaining one pixel.d denotes a distance from origin of a fixed fram of reference to a center of mass of a cell before cell gains new pixel. d_{new} denotes same distance but after cell gains new pixel

Now using parallel axis theorem we can write expression for the moment of inertia after cell gains one pixel the following that:

$$I^{new}_{xx} = I^{new}_{x'x'} - (V+1)d_{new}^2$$

where as before I^{new}_{xx} denotes moment of inertia of a cell with new pixel with respect to x axis passing through center of mass, $I^{new}_{x'x'}$ is a moment of inertia with respect to axis parallel to the x axis passing through center of mass, d_{new} is the distance between the axes and $V+1$ is volume of the cell after it gained one pixel. Now let us rewrite above equation by adding ad subtracting Vd^2 term:

$$I^{new}_{xx} = I^{old}_{x'x'} + y_{n+1}^2 + z_{n+1}^2 - Vd^2 + Vd^2 - (V+1)d_{new}^2$$

$$= I^{old}_{x'x'} - Vd^2 + Vd^2 - (V+1)d_{new}^2 + y_{n+1}^2 + z_{n+1}^2$$

$$= I^{old}_{xx} + Vd^2 - (V+1)d_{new}^2 + y_{n+1}^2 + z_{n+1}^2$$

Therefore we have found an expression for moment of inertia passing through the center of mass of the cell with additional pixel. Note that this expression involves moment of inertia but for the old cell (*i.e.* the original cell, not the one with extra pixel). When we add new pixel we know its coordinates and we can also easily calculate d_{new}. Thus when we need to calculate the moment of intertia for new cell instead of performing summation as given in the definition of the inertia tensor we can use much simpler expression.

This was diagonal term of the inertia tensor. What about off-diagonal terms? Let us write explicitly expression for I_{xy} :

$$I_{xy} = -\sum_i^N (x_i - x_{COM})(y_i - y_{COM}) = -\sum_i^N x_i y_i + x_{COM}\sum_i^N y_i + y_{COM}\sum_i^N x_i - x_{COM} y_{COM}\sum_i^N 1$$

$$= -\sum_i^N x_i y_i + x_{COM}V y_{COM} + y_{COM}V x_{COM} - x_{COM} y_{COM}V$$

$$= -\sum_i^N x_i y_i + V x_{COM} y_{COM}$$

where x_{COM} denotes x COM position of the cell, similarly y_{COM} denotes y COM position of cell and V denotes cell volume. In the above formula we have used the fact that

$$x_{COM} = \frac{\sum_i x_i}{V} \Rightarrow \sum_i x_i = x_{COM}V \text{ and similarly for the } y \text{ coordinate.}$$

Now, for the new cell with additional pixel we have the following relation:

$$I^{new}_{xy} = -\sum_i^{N+1} x_i y_i + (V+1)x^{new}_{COM} y^{new}_{COM}$$

$$= -\sum_i^N x_i y_i + V x_{COM} y_{COM} - x_{COM} V y_{COM} + (V+1)x^{new}_{COM} y^{new}_{COM} - x_{N+1}y_{N+1}$$

$$= I^{old}_{xy} - V x_{COM} y_{COM} + (V+1)x^{new}_{COM} y^{new}_{COM} - x_{N+1}y_{N+1}$$

where we have added and subtracted $V x_{COM} y_{COM}$ to be able to form

$$I^{old}_{xy} - \sum_i^N x_i y_i + V x_{COM} y_{COM} \text{ on the right hand side of the expression for } I^{new}_{xy}. \text{ As it was}$$

the case for diagonal element, calculating off-diagonal of the inertia tensor involves I^{old}_{xy} and positions of center of mass of the cell before and after gaining new pixel. All those quantities are either known a priori (I^{old}_{xy}) or can be easily calculated (center of mass position after gaining one pixel).

Therefore we have shown how we can calculate tensor of inertia for a given cell with respect to a coordinate frame with origin at cell's center of mass, without evaluating full sums. Such "local" calculations greatly speed up simulations

4.2 Calculating shape constraint of a cell – elongation term

The shape of single cell immersed in medium and not subject to too drastic surface or surface constraints will be spherical (circular in 2D). However in certain situation we may want to use cells which are elongated along one of their body axes. To facilitate this we can place constraint on principal lengths of cell. In 2D it is sufficient to constrain one of the principal lenghths of cell how ever in 3D we need to constrain 2 out of 3 principal lengths. Our first task is to diagonalize inertia tensor (i.e. find a coordinate frame transformation which brings inertia tensor to a giagonal form)

4.2.1 Diagonalizing inertia tensor

We will consider here more difficult 3D case. The 2D case is described in detail in M.Zajac, G.L.jones, J,A,Glazier "*Simulating convergent extension by way of anisotropic differential adhesion"* Journal of Theoretical Biology **222** (2003) 247–259.

In order to diagonalize inertia tensor we need to solve eigenvalue equation:
$\det(I - \lambda) = 0$ or in full form

$$\begin{vmatrix} \sum_i y_i^2 + z_i^2 - \lambda & -\sum_i x_i y_i & -\sum_i x_i z_i \\ -\sum_i x_i y_i & \sum_i x_i^2 + z_i^2 - \lambda & -\sum_i y_i z_i \\ -\sum_i x_i z_i & -\sum_i y_i z_i & \sum_i x_i^2 + y_i^2 - \lambda \end{vmatrix} = \begin{vmatrix} I_{xx} - \lambda & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} - \lambda & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} - \lambda \end{vmatrix} = 0$$

The eigenvalue equation will be in the form of 3rd order polynomial. The roots of it are guaranteed to be real. The polynomial itself can be found either by explicit derivation, using symbolic calculation or simply in Wikipedia (http://en.wikipedia.org/wiki/Eigenvalue_algorithm)

$$\det \begin{bmatrix} a - \lambda & b & c \\ d & e - \lambda & f \\ g & h & i - \lambda \end{bmatrix} = -\lambda^3 + \lambda^2(a+e+i) + \lambda(db+gc+fh-ae-ai-ei) + (aei-afh-dbi+dch+gbf-gce).$$

so in our case the eigenvalue equation takes the form:

$$-L^3 + L^2(\mathrm{I}_{xx} + \mathrm{I}_{yy} + \mathrm{I}_{zz}) + L(\mathrm{I}_{xy}^2 + \mathrm{I}_{yz}^2 + \mathrm{I}_{xz}^2 - \mathrm{I}_{xx}\mathrm{I}_{yy} - \mathrm{I}_{yy}\mathrm{I}_{zz} - \mathrm{I}_{xx}\mathrm{I}_{zz})$$
$$+\mathrm{I}_{xx}\mathrm{I}_{yy}\mathrm{I}_{zz} - \mathrm{I}_{xx}\mathrm{I}_{yz}^2 - \mathrm{I}_{yy}\mathrm{I}_{xz}^2 - \mathrm{I}_{zz}\mathrm{I}_{xy}^2 + 2\mathrm{I}_{xy}\mathrm{I}_{yz}\mathrm{I}_{xz} = 0$$

This equation can be solved analytically, again we may use Wikipedia (http://en.wikipedia.org/wiki/Cubic_function)
Now, the eigenvalues found that way are principal moments of inertia of a cell. That is they are components of inertia tensor in a coordinate frame rotated in such a way that off-diagonal elements of inertia tensor are 0:

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$$

In our cell shape constraint we will want to obtain ellipsoidal cells. Therefore the target tensor of inertia for the cell should be tensor if inertia for ellipsoid:

$$I = \begin{bmatrix} \frac{1}{5}\left(b^2 + c^2\right) & 0 & 0 \\ 0 & \frac{1}{5}\left(a^2 + c^2\right) & 0 \\ 0 & 0 & \frac{1}{5}\left(a^2 + b^2\right) \end{bmatrix}$$

where *a,b,c* are parameters describing the surface of an ellipsoid:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$$

In other words *a,b,c* are half lengths of principal axes (they are analogues of circle's radius)

Now we can determine semi axes lengths in terms of principal moments of inertia by inverting the following set of equations:

$$I_{xx} = \frac{1}{5}\left(b^2 + c^2\right)$$

$$I_{yy} = \frac{1}{5}\left(a^2 + c^2\right)$$

$$I_{zz} = \frac{1}{5}\left(a^2 + b^2\right)$$

Once we have calculated semiaxes lengths in terms of moments of inertia we can plug – in actual numbers for moment of inertia (the ones for actual cell) and obtain lengths of semiexes. Next we apply quadratic constraint on largest (semimajor) and smallest (seminimor axes). This is what elongation plugin does.

4.3 Forward Euler method for solving PDE's in CompuCell3D.

Remark: we present more complete derivations of explicit finite difference scheme for diffusion solver in "Introduction to Hexagonal Lattices in CompuCell3D".

In CompuCell3D most of the solvers uses explicit schemes (Forward Euler method) to obtain PDE solutions. Thus for the diffusion equation we have:

$$\frac{\partial c}{\partial t} = \frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2} + \frac{\partial^2 c}{\partial z^2}$$

In a discretetized form we may write:

$$\frac{c(x,t+\Delta t)-c(x,t)}{\Delta t} = \frac{c(x+\Delta x,t)-2c(x,t)+c(x-\Delta x,t)}{\Delta x^2} +$$

$$\frac{c(y+\Delta y,t)-2c(x,t)+c(y-\Delta y,t)}{\Delta y^2} +$$

$$\frac{c(z+\Delta z,t)-2c(z,t)+c(z-\Delta z,t)}{\Delta z^2}$$

where to save space we used shorthand notation:

$$c(x+\Delta x, y, z, t) \equiv c(x+\Delta x,t)$$

$$c(x, y, z, t) \equiv c(x,t)$$

and similarly for other coordinates.

After rearranging terms we get the following expression:

$$c(x,t+\Delta t) = \left[\frac{\Delta t}{\Delta x^2} \sum_{i=neighbors} \big(c(i,t)-c(x,t)\big) \right] - c(x,t)$$

where the sum over index 'i' goes over neighbors of point (x,y,z) and the neighbors will have the following concentrations: $c(x+\Delta x,t)$, $c(y+\Delta y,t)$, ..., $c(z+\Delta z,t)$.

4.4 Calculating center of mass when using periodic boundary conditions.

When you are running calculation with periodic boundary condition you may end up with situation like in the figure below:
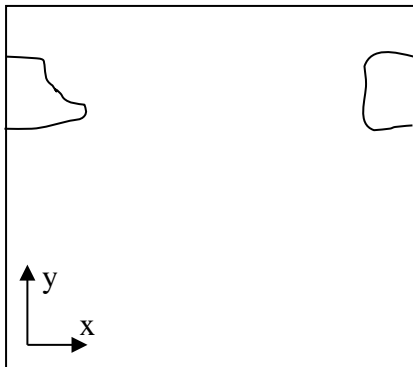

Figure 6 A connected cell in the lattice edge area – periodic boundary conditions are applied

Clearly what happens is that simply connected cell is wrapped around the lattice edge so part of it is in the region of high values of x coordinate and the other is in the region where x coordinates have low values. Consequently a naïve calculation of center of mass position according to :

$$x_{COM} = \frac{\sum_i x_i}{V}$$

would result in x_{COM} being somewhere in the middle of the lattice and abviously outside the cell.A better procedure could be as follows: Before calculating center of mass when new pixel is added or lost we "shift" a cell and new pixel (gained or lost)to the middle of the lattice do calculations "in the middle of the lattice" and shift back. Now if after shifting back it turns out that center of mass of a cell lies outside lattice position it in the center of mass by applygin a shift

equal to the length of the lattice and whose direction should be such that the center of mass of the cell ends up inside the lattice (there is only one such shift and it might be be equal to zero vector).

This is how we do it using mathematical formulas:

$$\vec{s} = \vec{x}_{COM} - \vec{c}$$

First we define shift vector $\vec{s}$ as a vector difference between vector pointing to center of mass of the lattice and vector pointing to (approximately) the middle of the lattice.

Next we shift cell to the middle of the lattice using :

$$\vec{x}'_{COM} = \vec{x}_{COM} - \vec{s}$$

where $\vec{x}'_{COM}$ denotes center of mass position of a cell after shifting but before adding or subtracting a pixel.

Next we take into account the new pixel (either gained or lost) and calculate center of mass position (for the shifted cell):

$$\vec{x}'_{COM}{}^{new} = \frac{\vec{x}'_{COM} V + \vec{x}_i}{V + 1}$$

Above we have assumed that we are adding one pixel.

Now all that we need to do is to shift back $\vec{x}'_{COM}{}^{new}$ by same vector $\vec{s}$ that brought cell to (approximately) center of the lattice.

$$\vec{x}_{COM}{}^{new} = \vec{x}'_{COM}{}^{new} + \vec{s}$$

We are almost done. We still have to check if $\vec{x}_{COM}{}^{new}$ is inside the lattice. If this is not the case we need to shift it back to the lattice but now we are allowed to use only a vector whose components are multiples of lattice dimensions (and we can safely restrict to +1 and -1 multiples of the lattice dimmensions) . For example we may have:

$$\vec{P} = (x_{max}, -y_{max}, 0)$$ where x_{max}, y_{max}, z_{max} are dimensions of the lattice.

There is no cheating here. In the lattice with periodic boundary conditions you are allowed to shift point coordinates a vector whose components are multiples of lattice dimensions.

All we need to do is to examine new center of mass position and form suitable vector $\vec{P}$.

4.5 Dividing cluster cells

While dividing non-clustered cells is straightforward, doing the same for clustered cells is more challenging. To divide non-cluster cell using directional mitosis algorithm we construct a line or a plane passing through center of mass of a cell and pixels of the cell (we are using PixelTracker plugin with mitosis) on one side of the line/plane end up in child cell and the rest stays in parent cell. The orientation of the line/plane can be either specified by the user or we can use CC3D built-in feature to calculate calculate orientation tion of principal axes and divide either along minor or major axis.

With compartmental cells, things get more complicated because: 1) Compartmental cells are composed of many subcells. 2) There can be different topologies of clusters. Some

clusters may look "snake-like" and some might be compactly packed blobs of subcells. The algorithm which we implemented in CC3D works in the following way:

1) We first construct a set of pixels containing every pixel belonging to a cluster cell. You may think of it as of a single "regular" cell.
2) We store volumes of compartments so that we know how big compartments shold be after mitosis (they will be half of original volume)
3) We calculate center of mass of entire cluster and calculate vector offsets between center of mass of a cluster and center of mass of particulat compartments as on the figure below:
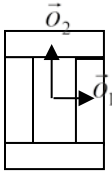

Figure 7. Vectors $\vec{o}_1$ and $\vec{o}_2$ show offsets between center of mass of a cluster and center of mass particular compartments.

4) We pick division line/plane and for parents and child cells we offsets between cluster center of mass (after mitosis) and center of masses of clusters. We do it according to the formula:

$$\vec{p} = \vec{o} - \frac{1}{2}(\vec{o} \cdot \vec{n})\vec{n}$$

where $\vec{p}$ denotes offset after mitosis from center of mass of child (parent) clusters, $\vec{o}$ is orientation vector before mitosis (see picture above) and $\vec{n}$ is a normalized vector perpendicular to division line/plane. If we try to divide the cluster along dashed line as on the picture below
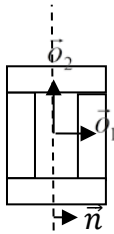

Figure 8. Division of cell along dashed line. Notice the orientation of $\vec{n}$

the offsets after the mitosis for child and parent cell will be $\vec{p}_1 = \frac{1}{2}\vec{o}_1$ and $\vec{p}_2 = \vec{o}_2$ as

expected because both parent and child cells will retain their heights but after mitosis will become twice narrower (cell with grey outer compartments is a parent cell):
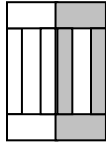
Figure 9.Child and parent (the one with grey outer compartments) cells after mitosis.

The formula given above is heuristic. It gives fairly simple way of assigning pixels of child/parent clusters to cellular compartments. It is not perfect but the idea is to get approximate shape of the cell after the mitosis and as simulation runs cell shape will readjust based on constraints such as adhesion of focal point plasticity. Before continuing with the mitosis we check if center of masses of compartments belong to child/parent clusters. If the center of masses are outside their target pixels we abandon mitosis and wait for readjustment of cell shape at which point mitosis algorithm will pass this sanity check. For certain "exotic" shapes of cluster shapes presented mitosis algorithm may not work well or at all . In this case we would have to write specialized mitosis algorithm.

5) We divide clusters and knowing offsets from child/parent cluster center of mass we assign pixels to particular compartments. The assignment is based on the distance of particular pixel to center of masses of clusters. Pixel is assigned to particular compartment if its distance to the center of mass of the compartment is the smallest as compared to distances between centroids of other compartments. If given compartment has reached its target volume and other compartmets are underpopulated we would assign pixels to other compartments based on the closest distance criterion. Altohugh this method may result in some deviation from perfect 50-50 division of compartment volume in most cases after few MCS cells will readjust their volume.

Figure 10. CC3D example of compartmental cell division. See also examples_PythonTutorial/clusterMitosis.

4.6 *Command line options of CompuCell3D*

Although most users run CC3D using Player GUI sometimes it is very convenient to run CC3D using command line options. CC3D allows to invoke Player directly from command line which is convenient because if saves several clicks and if you run many simulations this might be quite convenient.

Remark: On Windows we use .bat extension for run scripts and on Linux/OSX it is .sh. Otherwise all the material in this section applies to all the platforms.

4.6.1 CompuCell3D Player Command Line Options

The command line options for running simulation with the player are as follows:

```
compucell3d.bat  [options]
```

Options are:

`-i <simulation file>` - users specify simulation file they want to run. It can be either CC3DML (XML) configuration file or Python script.

`-s <screenshotDescriptionFileName>` - name of the file containing description of screenshots to be taken with the simulation. Usually this file is prepared using Player by switching to different views, clickin camera button and saving screenshot description file from the Player File menu.

`-o <customScreenshotDirectoryName>` - allows users to specify where screenshots will be written. Overrides default settings.

`--noOutput` - instructs CC3D not to store any screenshots. Overrides Player settings.

`--exitWhenDone` - instructs CC3D to exit at the end of simulation. Overrides Player settings.

`-h, --help` - prints command line usage on the screen

Example command may look like:

```
compucell3d.bat –i Demos\cellsort_2D\cellsort_2D.xml –s screenshotDescription.sdfml –o
Demos\cellsort_2D\screenshot
```

The frequency of the screenshots is read using Player settings so if you need to adjust it please use either GUI directly or change it using PlayerSettings plugin (see example *Demos\cellsort_2D\cellsort_2D_PlayerSettings.xml*)

4.6.2 Runnig CompuCell3D in a GUI-Less Mode - Command Line Options.

Sometimes when you want to run CC3D on a cluster you will have to use runScript.bat which allows running CC3D simulations without invoking GUI. However, all the screenshots will be still stored.

> **Remark:** current version of this script does not handle properly relative paths so it has to be run from the installation directory of CC3D i.e. you have to `cd` into this directory prior to runnit runScript.bat. Another solution is to use full paths.

The output of this script is in the form of vtk files which can be subsequently replayed in the Player (and one can take screenshots then). By default all fields present in the simulation are stored in the vtk file. If users want to remove some of the fields fro mbeing stored in the vtk format they have to pass this information in the Python script:

```
CompuCellSetup.doNotOutputField(_fieldName)
```

The best place to put such stetements is directly before steppable section in the Python main script. See also commented line (and try uncommenting it) in *examples_PythonTutorial\ diffusion_extra_player_field\ diffusion_2D_extra_player_field.py*.
Storing entire fields (as opposed to storing screenshots) preserves exact snapshots of the simulation and allows result postprocessing. In addition to the vtk files runScript stores lattice description file with .dml extension which users open in the Player (`File->Open Lattice Description Summary File…`) if they want to reply generated vtk files.

The format of the command is:

```
runScript.bat [options]
```

The command line options for runScript.bat are as follows:

`-i <simulation file>` - users specify simulation file they want to run. It can be either CC3DML (XML) configuration file or Python script. Remember about using full paths if you run the script from directory different than
`-c <outputFileCoreName>` - allows users to specify core name for the vtk files. The default name for vtk files is "Step"

`-o <customVtkDirectoryName>` - allows users to specify where vtk files and the .dml file will be written. Overrides default settings

`-f  <frequency>` or `–outputFrequency=<frequency>` - allows to specify how often vtk files are stored to the disk. Those files tend to be quite large for bigger simulations so storing them every single MCS (default setting) slows down simulation considerably and also uses a lot of disk space.

`--noOutput` - instructs CC3D not to store any output. This option makes little sense in most cases.

`-h, --help` - prints command line usage on the screen

Example command may look as follows:

```
runScript.bat –i examples_PythonTutorial\cellsort_2D_info_printer\
cellsort_2D_info_printer.py –f 10
–o examples_PythonTutorial\cellsort_2D_info_printer\screenshots
–c infoPrinter
```

4.7 Managing CompuCell3D simulations (CC3D project files)

Until version 3.6.0 CompuCell3D simulations were stored as a combination of Python, CC3DML (XML), and PIF files. This solution was working fine but there were significant problems with keeping track of simulations files. We still support this convention. However, starting with version 3.6.0 we introduced new way of managing CC3D simulations by enforcing that a single CC3D simulation is stored in a folder containing .cc3d project file describing simulation resources (.cc3d is in fact XML), such as CC3DML configuration file, Python scripts, PIF files, Concentration filets etc… and a directory called Simulation where all the resources reside. The structure of the new-style CC3D simulation is presented in the diagram below:

->**CellsortDemo**
       CellsortDemo.cc3d
            ->Simulation
                    Cellsort.xml
                    Cellsort.py
                    CellsortSteppables.py
                    Cellsort.piff
                    FGF.txt

Bold fonts denote folders. The benefit of using CC3D project files instead of loosely related files are as follows:

1) Previously users had to guess which file needs to be open in CC3D – CC3DML or Python. While in a well written simulation one can link the files together in a way that when user opens either one the simulation would work but, nevertheless, such approach was clumsy and unreliable. Starting with 3.6.0 users open .cc3d file and they don't have to stress out that CompUCell3D will complain with error message.

2) All the files specified in the .cc3d project files are copied to the result output directory along with simulation results (uncles you explicitly specify otherwise). Thus, when you run multiple simulations each one with different parameters, the copies of all CC3DML and Python files are stored eliminating guessing which parameters were associated with particular simulations.

3) All file paths appearing in the simulation files are relative paths with respect to main simulation folder. This makes simulations portable because all simulation resources are contained withing single folder. In the example above when referring to Cellsort.piff file from Cellsort.xml you use "Simulation/ Cellsort.piff". This effectively eliminates drawbacks of previous approach – when user one stores his simulations in Demos/cellsort and gives this simulation to his colleague who stores simulation in MySimulations/cellsort then second user will most likely see error message informing him that file

"Demos/cellsort/cellsort.piff" was not found (I assume here that initial condition is specified using cellsort.piff). With approach based on relative paths such problems do not exist. Second user can put the simulation anywhere he wants and it will run just fine.
4) New style of storing CC3D simulations has also another advantage – it makes graphical management of simulation content and simulation generation very easy. As amatetr of fact new component of CC3D suite – Twedit++ - CC3D edition has a graphical tool that allows for easy project file management and it also has new simulation wizadrd which allows users to build template of CC3D simulation within less than a minute.

Let's now look in detail at the structure of .cc3d files:

```
<Simulation version="3.6.0">
    <XMLScript>Simulation/Cellsort.xml</XMLScript>
    <PythonScript>Simulation/Cellsort.py</PythonScript>
    <Resource Type="Python">Simulation/CellsortSteppables.py</Resource>
   <PIFFile>Simulation/Cellsort.piff</PIFFile>
    <Resource Type="Field" Copy="No">Simulation/FGF.txt</Resource>
</Simulation>
```

As you can see the structure of the file is quite flat. All that we are storing there is names of files that are used in the simulation. Two files have special tags <XMLFile> which specifies name of the CC3DML file storing "CC3DML portion" of the simulation and <PythonScript> which specifies main Python script. We have also PIFFile tag which is used to designate PIF files. All other files used in the simulation are referred to as Resources. For example Python steppable file is a resource of type "Python". FGF.txt is aresource of type "Field". Notice that all the files are specified using paths relative to main simulation directory.

As we mentioned before, when you run .cc3d simulation all the files listed in the project file are copied to result folder. If for somereason oyu want to avoid coping of some of the files, simply add Copy="No" attribute in the tag with file name specification.

4.8 Keeping Track of Simulation Files (deprecated!)

CompuCell3D will store screenshots, vtk lattice snapshots and CC3DML file/Python main script in the output directory. However often simulations consist of several files: CC3DML, Python main script, Python steppable script, Python plugin script, PIF files etc. If you want those files to be archived with the rest of simulation output you need to use SimulationFileStorage steppable declared in *PySteppablesExamples*.py.

The usage is very simple (see also *examples_PythonTutorial\cellsort_2D_info_printer\cellsort_2D_info_printer.py*):

```
from PySteppablesExamples import SimulationFileStorage
sfs=SimulationFileStorage(_simulator=sim,_frequency=10)
sfs.addFileNameToStore("\
examples_PythonTutorial/cellsort_2D_info_printer/cellsort_2D.xml")
sfs.addFileNameToStore("\
examples_PythonTutorial/cellsort_2D_info_printer/cellsort_2D_info_printer.py")
sfs.addFileNameToStore("examples_PythonTutorial/cellsort_2D_info_printer\
/cellsort_2D_steppables_info_printer.py")
```

```
steppableRegistry.registerSteppable(sfs)
```

It wil ensure that files listed here will be writte to simulation directory. This way if you keep changing simulation files you will be able to easily recover entire simulation at