

CompuCell3D Training Workshop

NIMBioS,

University of Tennessee

Knoxville,

May 18-21 2011



Maciej Swat

James Glazier

Randy Heiland

Julio Belmonte

Mitja Hmeljak

What you will learn during the workshop?

1. What is CompuCell3D?
2. Why use CompuCell3D?
3. Demo simulations
4. Glazier-Graner-Hogeweg (GGH) model – an overview
5. CompuCell3D architecture and terminology
6. XML 101. CC3DML-intro
7. Building first CompuCell3D simulation
8. Visualization package – CompuCell Player
9. Python scripting inside CompuCell3D
10. Building C++ CompuCell3D extension modules – for interested participants

What Is CompuCell3D?

1. CompuCell3D is a modeling environment used to build, test, run and visualize GGH-based simulations
2. CompuCell3D has built-in scripting language (Python) that allows users to quite easily write extension modules that are essential for building sophisticated biological models.
3. CompuCell3D thus is NOT a specialized software
4. Running CompuCell3D simulations DOES NOT require recompilation
5. CompuCell3D model is described using CompuCell3D XML and Python script(s)
6. CompuCell3D platform is distributed with a GUI front end – CompuCell Player. The Player provides 2- and 3-D visualization and simulation replay capabilities.
7. CompuCell3D is a cross platform application that runs on Linux/Unix, Windows, Mac OSX. CompuCell3D simulations can be easily shared

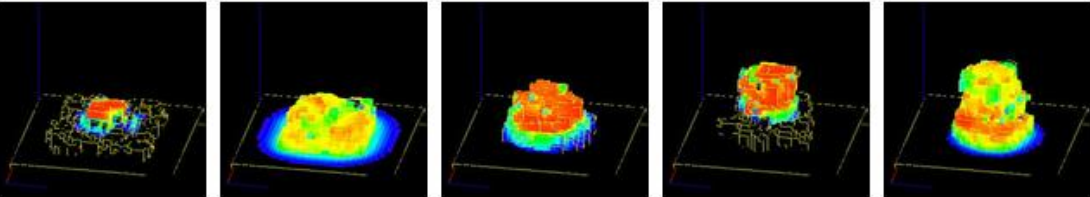
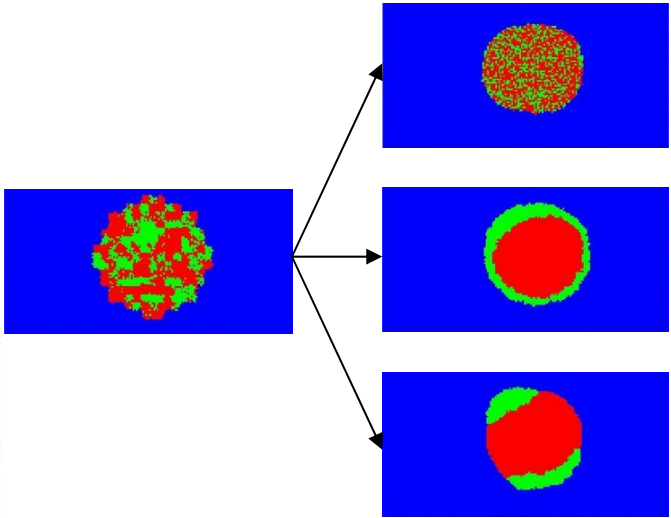
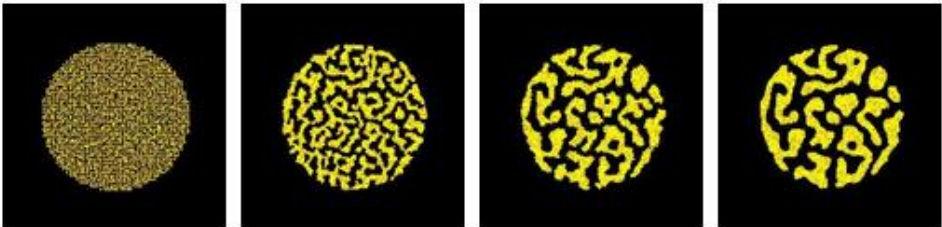
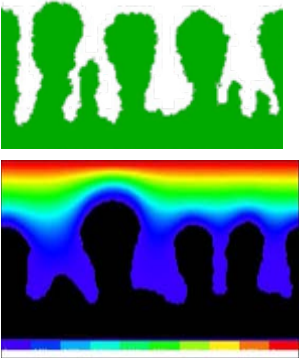
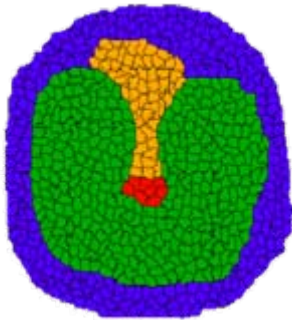
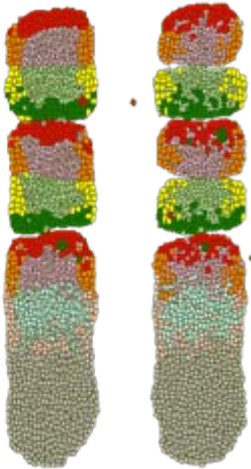
Why Use CompuCell3D? What Are the Alternatives?

1. CompuCell3D allows users to set up and run their simulations within minutes, maybe hours. A typical development of a specialized GGH code takes orders of magnitudes longer time.
2. CompuCell3D simulations **DO NOT** need to be recompiled. If you want to change parameters (in XML or Python scripts) or logic (in Python scripts) you just make the changes and re-run the simulation. With hand-compiled simulations there is much more to do. Recompilation of every simulation is also error prone and often limits users to those who have significant programming background.
3. CompuCell3D is actively developed , maintained and supported. On www.compuCell3d.org website users can download manuals, tutorials and developer documentation. CompuCell3D has approx. 4 releases each year – some of which are bug-fix releases and some are major
4. CompuCell3D has many users around the world. This makes it easier to collaborate or exchange modules and results saving time spent on developing new model.
5. The Biocomplexity Institute organizes training workshops and mentorship programs. Those are great opportunities to visit Bloomington and learn biological modeling using CompuCell3D. For more info see www.compuCell3d.org

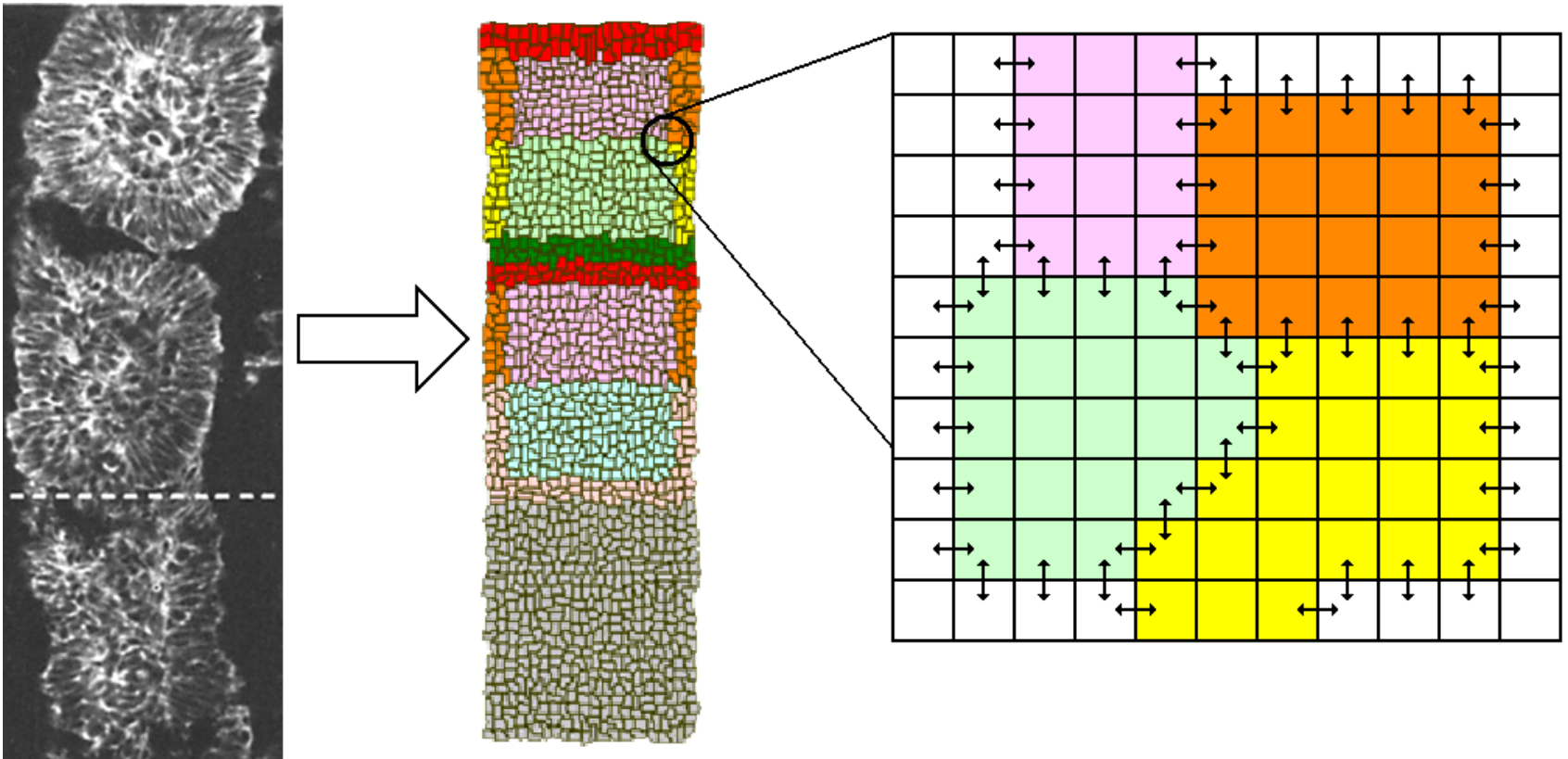
Why model sharing and standards are important?

1. 99% of modeling done with custom written code is very hard/impossible to reproduce or verify. Even in best quality publications authors may forget to describe small details which are actually essential to reproduce the described work.
2. Using standard modeling tools instead of writing your own code improves chances of your research being reused or improved by other scientists. Note: in certain situations people might be interested in, precisely, the opposite.
3. When people spend most of their time working on new ideas rather than struggling to reproduce old results it greatly improves research efficiency
4. Bug tracking/bug detection is much more efficient with shared tools than with custom written ones. Bugs are also better documented for shared software.
5. Developing and sharing modules with other researchers is best way of improving software modeling tools used by community of researchers

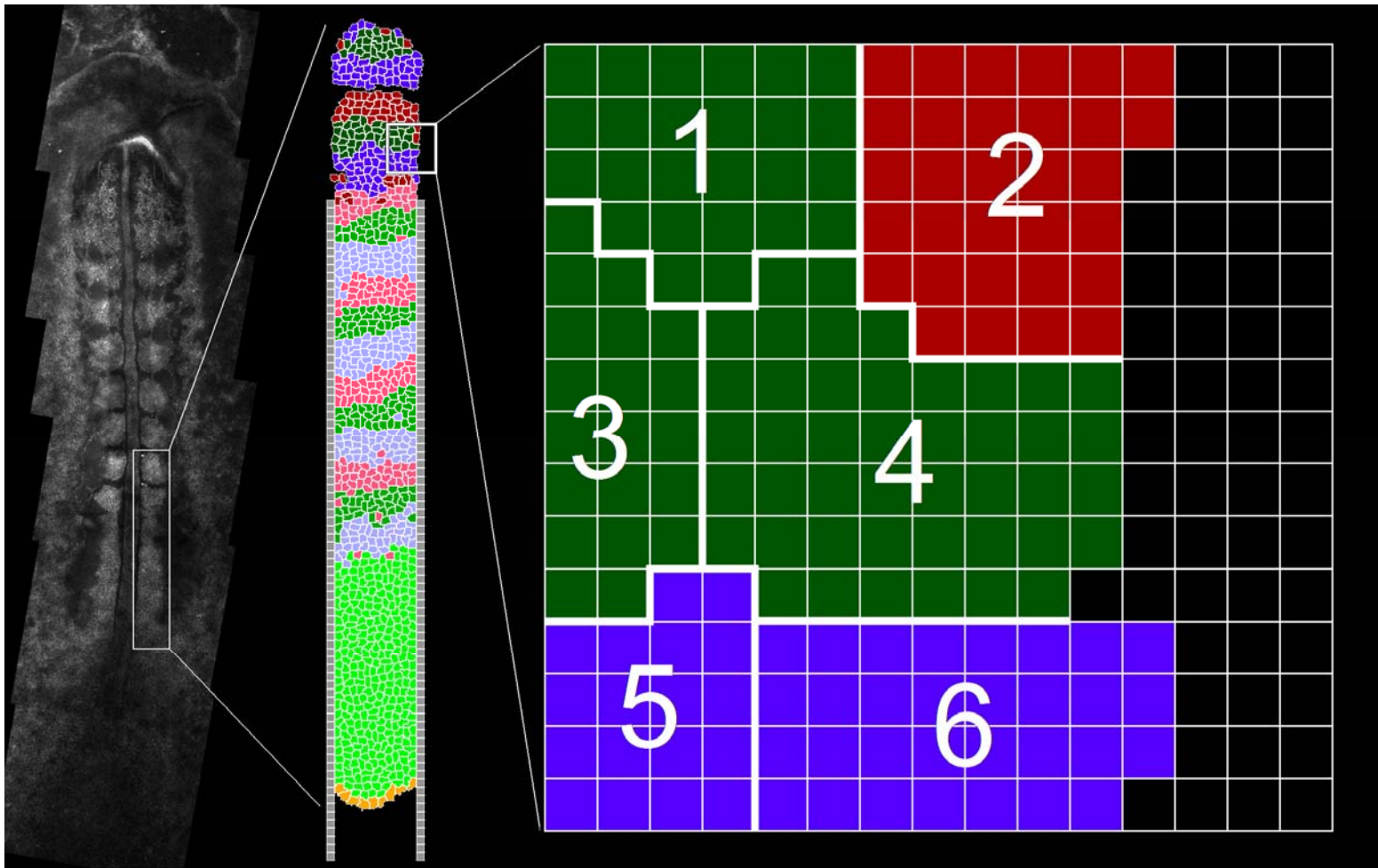
Demo Simulations



GGH(Glazier Graner Hogeweg) Model also known as CPM(Cellular Potts Model)

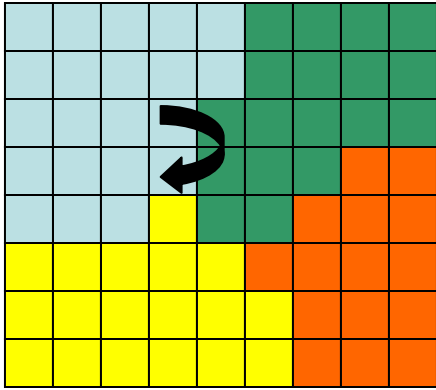


$$\begin{aligned}
 E = & \sum_{x,x'} J_{\tau(\sigma(x)),\tau(\sigma(x'))} (1 - \delta_{\tau(\sigma(x)),\tau(\sigma(x'))}) \\
 & + \lambda_s (s_\sigma - S_\sigma)^2 + \\
 & \lambda_v (v_\sigma - V_\sigma)^2
 \end{aligned}$$

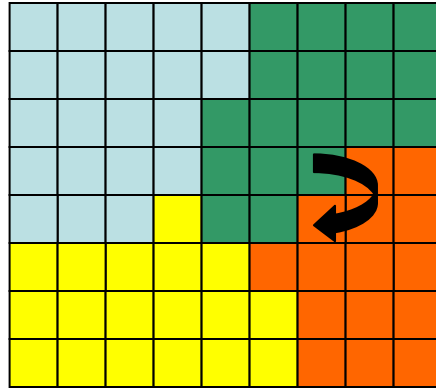


$$\begin{aligned}
 E = & \sum_{x,x'} J_{\tau(\sigma(x)),\tau(\sigma(x'))} (1 - \delta_{\tau(\sigma(x)),\tau(\sigma(x'))}) \\
 & + \lambda_s (s_\sigma - S_\sigma)^2 + \\
 & \lambda_v (v_\sigma - V_\sigma)^2
 \end{aligned}$$

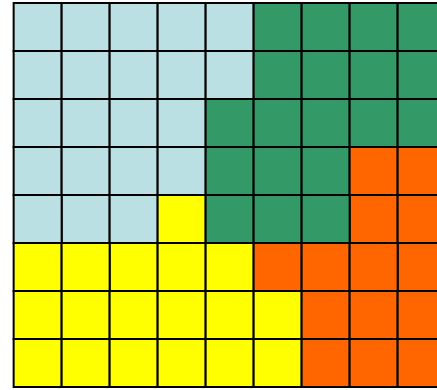
invalid attempt



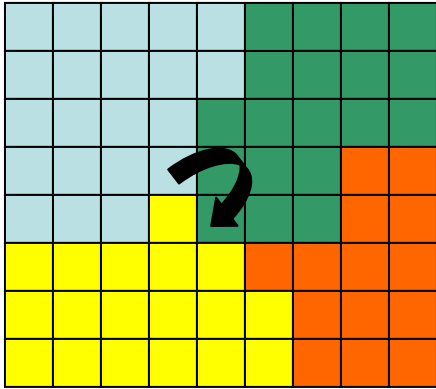
valid attempt



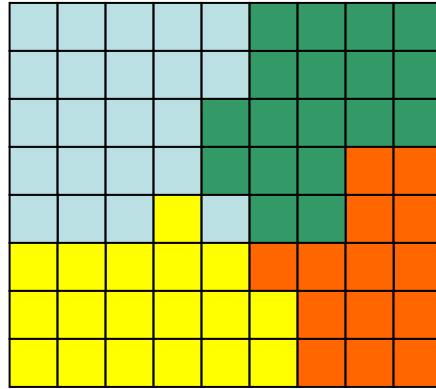
accept



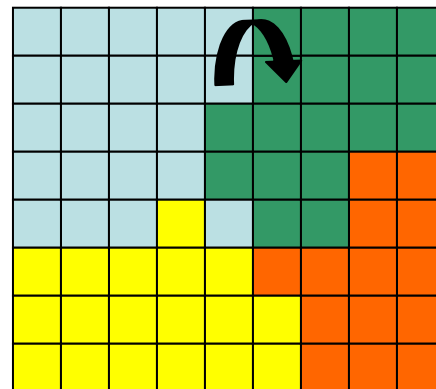
valid attempt



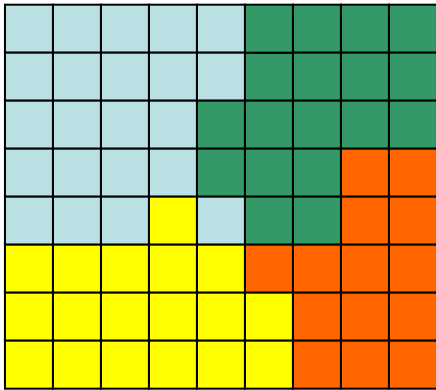
accept



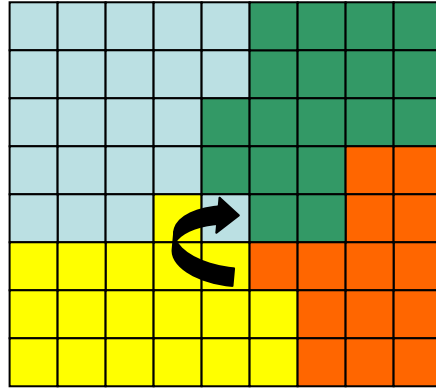
valid attempt



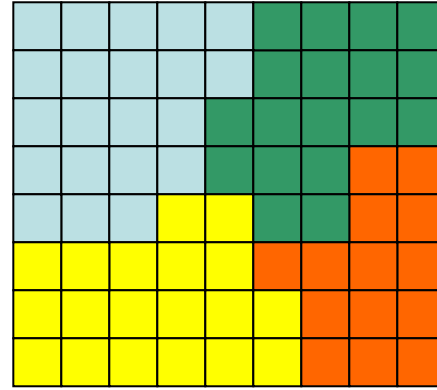
reject



valid attempt



accept



The GGH Model Formalism Overview

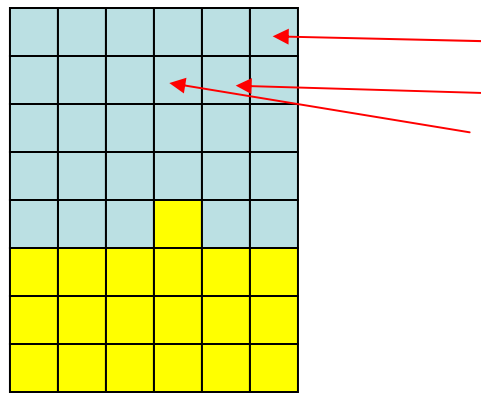
- Energy minimization formalism
 - extended by Graner and Glazier, 1992
- DAH: Contact energy depending on cell types (differentiated cells)

$$E = \sum_{x,x'} J_{\tau(\sigma(x)),\tau(\sigma(x'))} (1 - \delta_{\sigma(x),\sigma(x')}) +$$
$$\lambda_s (s_\sigma - S_\sigma)^2 + \lambda_v (v_\sigma - V_\sigma)^2 +$$
$$E_{chem} + E_{hapt} + \dots$$

- Metropolis algorithm: probability of configuration change

$$P(\Delta E) = 1, \Delta E \leq 0$$
$$P(\Delta E) = e^{-\Delta E/kT}, \Delta E > 0$$

Brief Explanation of Equation Symbols



$\sigma(\mathbf{x})$ –denotes **id of the cell** occupying position \mathbf{x} . All pixels pointed by arrow have same **cell id** , thus they belong to the same cell

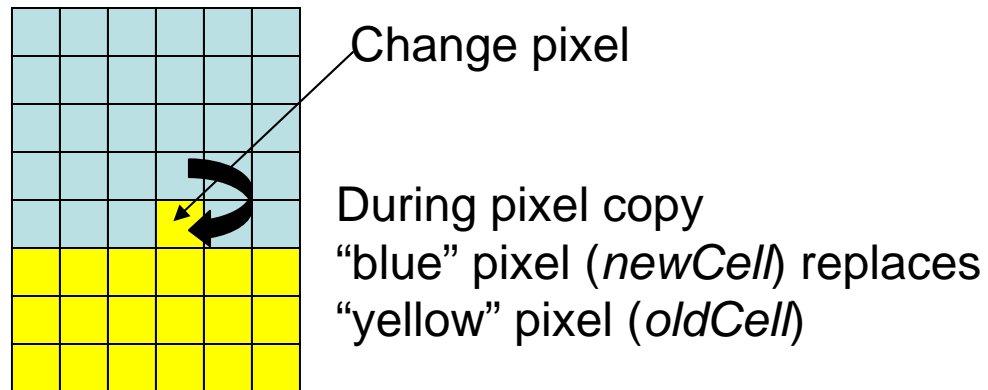
$\tau(\sigma(\mathbf{x}))$ denotes **cell type** of cell with id $\sigma(\mathbf{x})$. In the picture above blue and yellow cells have **different cell types and different cell id**. Arrows mark different cell types

Notice that in your model **you may (will) have many cells of the same type but with different id**. For example in a simple cellsorting simulation there will be many cells of type “Condensing” and many cells with type “NonCondensing”

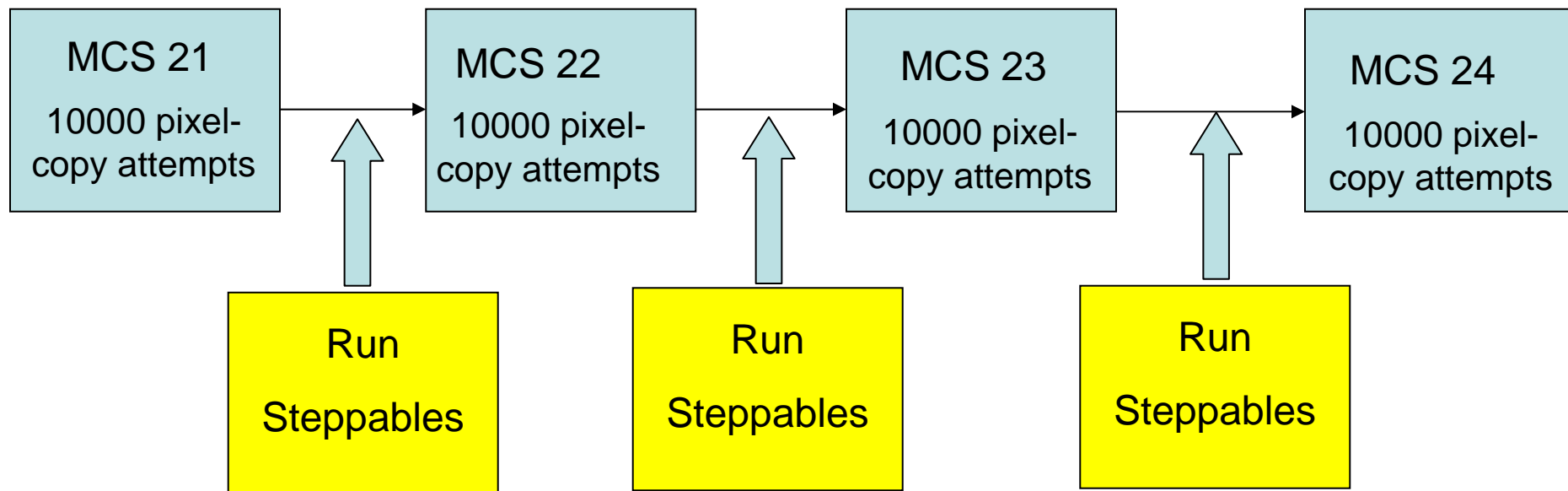
CompuCell3D terminology

1. **Pixel-copy attempt** is an event where program randomly picks a lattice site in an attempt to copy the pixel to a neighboring lattice site.
2. **Monte Carlo Step (MCS)** consists of series pixel-copy attempts. Usually the number of pixel copy-attempts in single MCS is equal to the number of lattice sites, but this is can be customized
3. **CompuCell3D Plugin** is a software module that either calculates an energy term in a Hamiltonian or implements action in response to pixel copy (lattice monitors). Note that not all spin-copy attempts will trigger lattice monitors to run.
4. **Steppables** are CompuCell3D modules that are run every MCS after all pixel-copy attempts for a given MCS have been exhausted. Most of Steppables are implemented in Python. Most customizations of CompuCell3D simulations is done through Steppables
5. **Steppers** are modules that are run for those spin-copy attempts that actually resulted in energy calculation. They are run regardless whether actual pixel-copy occurred or not. For example cell mitosis is implemented in the form of stepper.
6. **Fixed Steppers are** modules that are run every pixel-copy attempt.

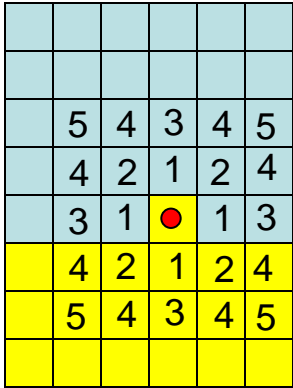
CompuCell3D Terminology – Visual Guide



100x100x1 square lattice = 10000 lattice sites (pixels)



Nearest neighbors in 2D and their Euclidian distances from the central pixel



Nearest Neighbor Order	Number of nearest neighbors	Euclidian distance – square lattice
1	4	1
2	4	$\sqrt{2}$
3	4	2
4	8	$\sqrt{5}$
5	4	$\sqrt{8}$

Pixel copy can take place between any order nearest neighbor (although in practice we limit ourselves to only few first orders).

`<NeighborOrder>2</NeighborOrder>` 2nd nearest neighbor

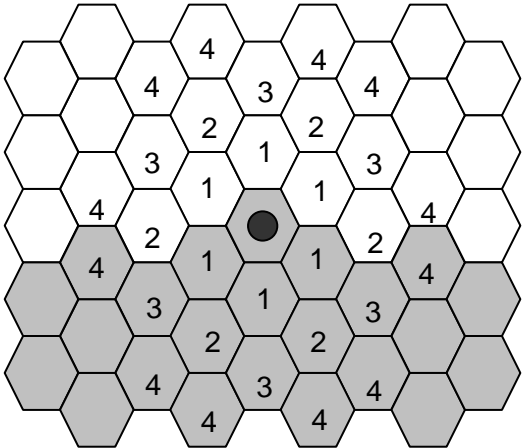
Contact energy calculation (see further slides) are also done up to certain order of nearest neighbors (default is 1)

`<NeighborOrder>2</NeighborOrder>`

Note: older tags still work but we encourage using new ones - they make more sense

Nearest neighbors in 2D and their Euclidian distances from the central pixel

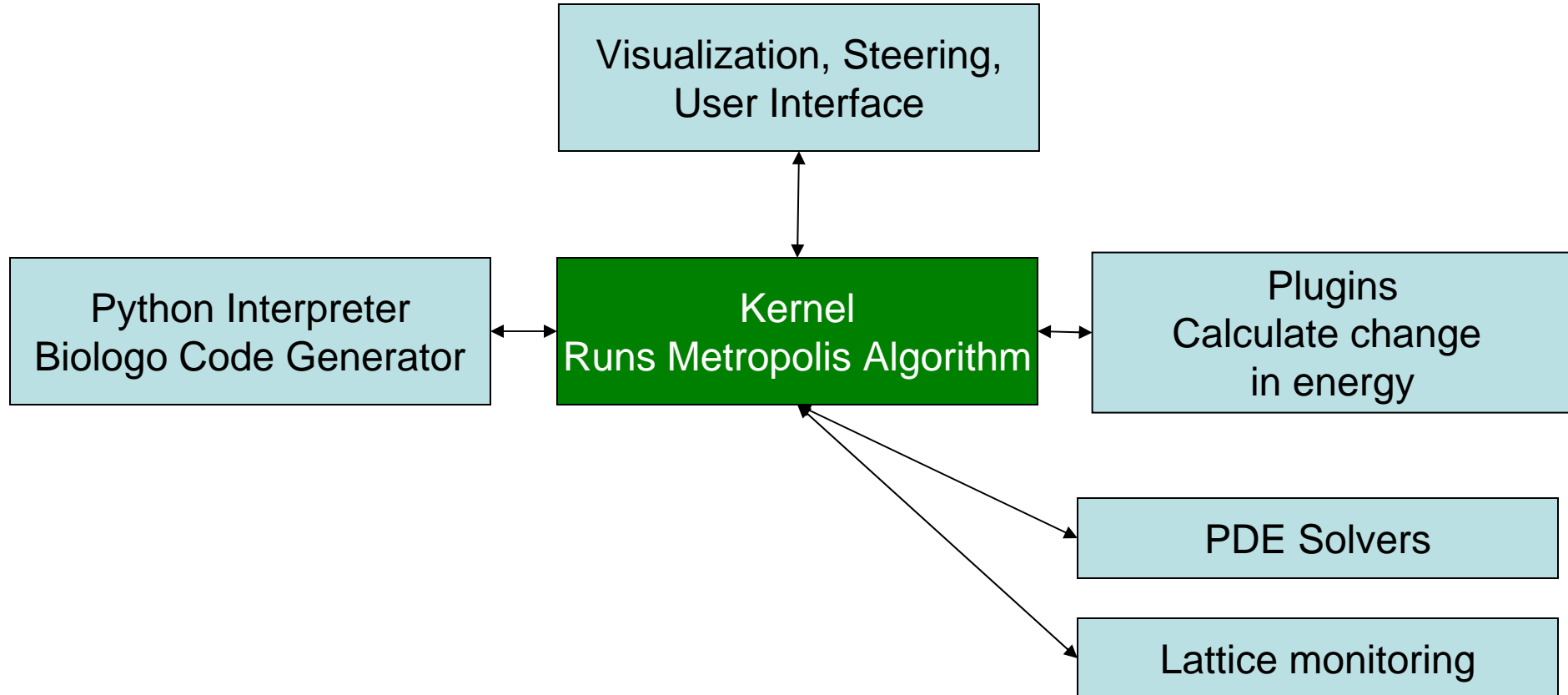
		4	3	4	
	4	2	1	2	4
	3	1	●	1	3
	4	2	1	2	4
		4	3	4	



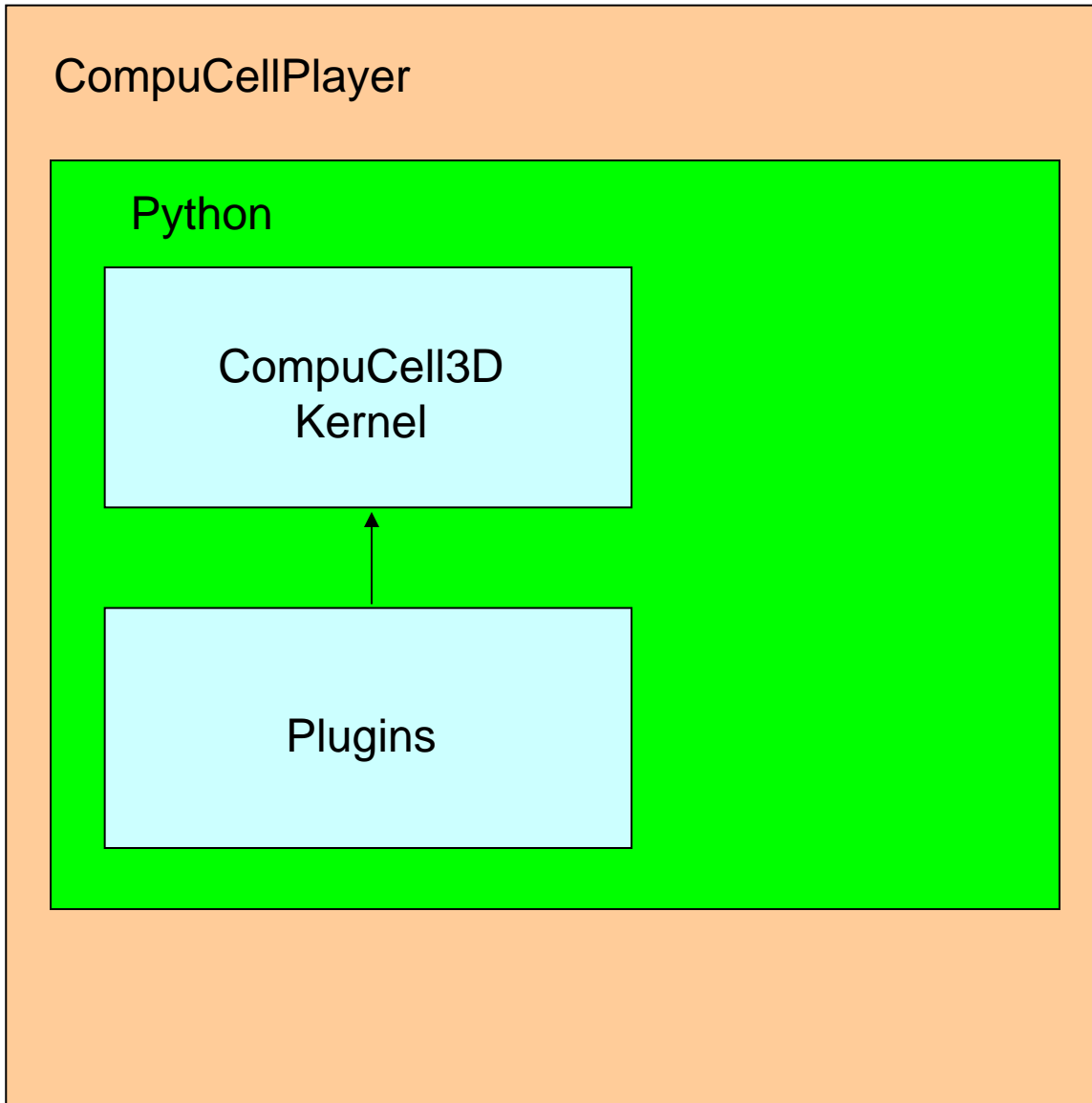
	2D Square Lattice		2D Hexagonal Lattice	
Neighbor Order	Number of Neighbors	Euclidian Distance	Number of Neighbors	Euclidian Distance
1	4	1	6	$\sqrt{2/\sqrt{3}}$
2	4	$\sqrt{2}$	6	$\sqrt{6/\sqrt{3}}$
3	4	2	6	$\sqrt{8/\sqrt{3}}$
4	8	$\sqrt{5}$	12	$\sqrt{14/\sqrt{3}}$

CompuCell3D Architecture

Object oriented implementation in C++ and Python



Typical “Run-Time” Architecture of CompuCell



CompuCell can be run in a variety of ways:

- Through the Player with or without Python interpreter
- As a Python script
- As a stand alone computational kernel+plugins

XML 101

XML stands for eXtensible Markup Language. **It is NOT a programming language.** Its main purpose is to standardize information exchange between different applications.

XML Example:

```
<Sentence>  
  <Text>It is too early to be in class</Text>  
  <FontType>TimesNewRoman</FontType>  
  <FontSize>12</FontSize>  
  <DisplayHint Hint="AddFrameAround"/>  
</Sentence>
```

```
def configureSimulation(sim):  
  Snt=ElementCC3D("Sentence")  
  Txt=Snt.ElementCC3D("Text",{,"It is to  
  Fnt=Snt.ElementCC3D("FontType",{,"T  
  fntSize=Snt.ElementCC3D("FontSize",{  
  Disp=Snt.ElementCC3D("DisplayHint",  
  {"Hint": "AddFrameAround"})
```

XML is essentially a definition of hierarchical (tree-like) data structure

```
<Computer>
```

```
  <CPU>Pentium
```

```
    <Frequency Unit="GHz">2.4</Frequency>
  </CPU>
```

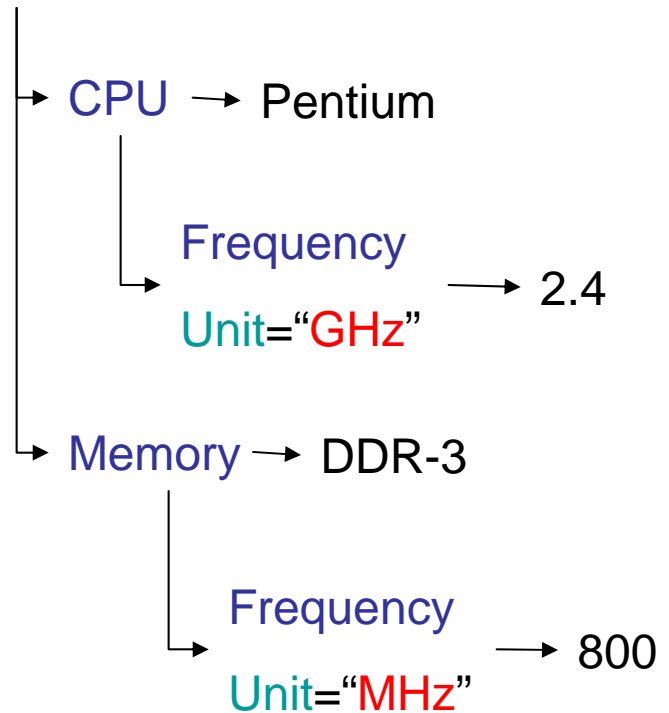
```
  <Memory>DDR-3
```

```
    <Frequency Unit="MHz">800</Frequency>
  </Memory>
```

```
  ...
```

```
</Computer>
```

Computer



CompuCell Related Example

Defining basic properties of the simulation like **lattice dimension, number of Monte Carlo Steps, Temperature and ratio of pixel-copy attempts to number of lattice sites (Flip2DimRatio)**. `<Potts>` section has to be included in every CompuCell3D simulation

```
<Potts>
  <Dimensions x="71" y="36" z="211"/>
  <Steps>10</Steps>
  <Temperature>2</Temperature>
  <Flip2DimRatio>2</Flip2DimRatio>
</Potts>
```

Defining properties of **Volume Energy** term – cell target volume and lambda parameter:

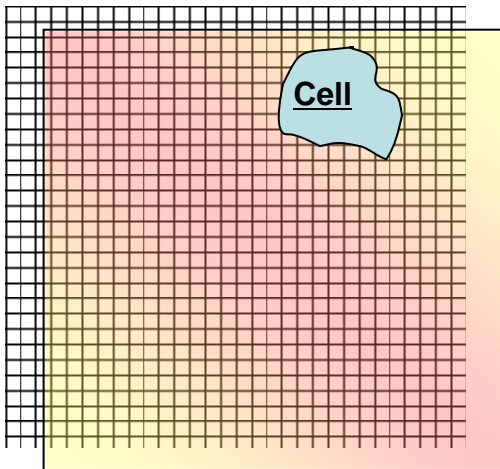
```
<Plugin Name="Volume">
  <TargetVolume>25</TargetVolume>
  <LambdaVolume>2.0</LambdaVolume>
</Plugin>
```

...

Building Your First CompuCell3D Simulation

All simulation parameters are controlled by the config file. The config file allows you to only add those features needed for your current simulation, enabling better use of system resources.

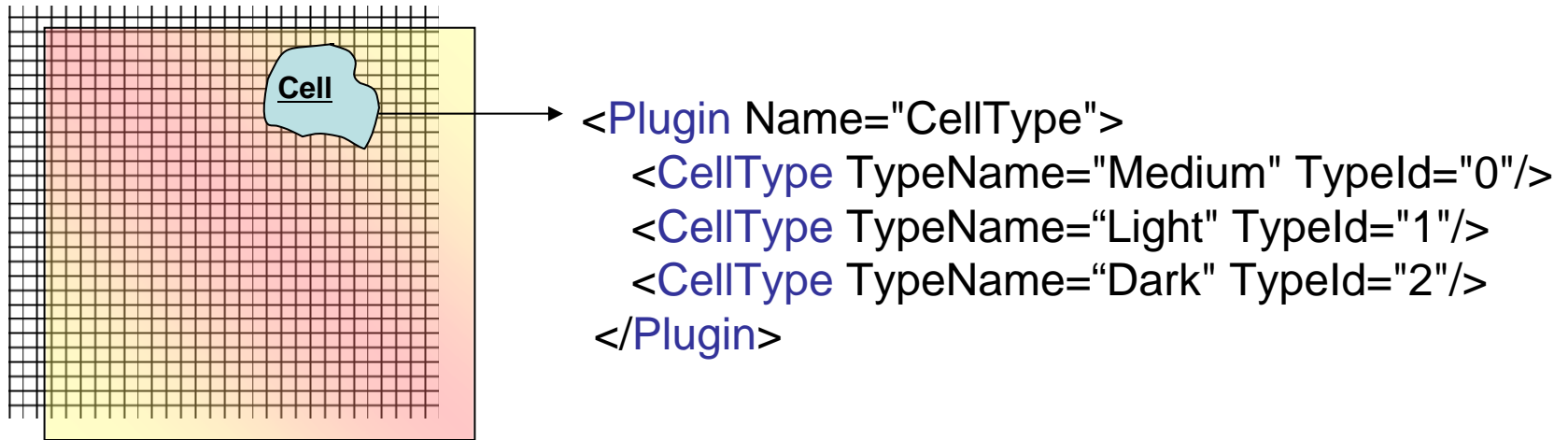
Define Lattice and Simulation Parameters



```
< CompuCell3D>  
<Potts>  
  <Dimensions x="100" y="100" z="1"/>  
  <Steps>10</Steps>  
  <Temperature>2</Temperature>  
  <Flip2DimRatio>1</Flip2DimRatio>  
</Potts>  
...  
</CompuCell3D>
```

Define Cell Types Used in the Simulation

Each CompuCell3D xml file must list all cell types that will be used in the simulation



Notice that Medium is listed with `TypeId = 0`. This is both convention and a **REQUIREMENT** in CompuCell3D. Reassigning Medium to a different `TypeId` may give undefined results. This limitation will be fixed in one of the next CompuCell3D releases

Define Energy Terms of the Hamiltonian and Their Parameters



Cell

Volume

volume
volumeEnergy(cell)

```
<Plugin Name="Volume">  
<TargetVolume>25</TargetVolume>  
<LambdaVolume>1.0</LambdaVolume>  
</Plugin>
```

Surface

area
surfaceEnergy(cell)

```
<Plugin Name="Surface">  
<TargetSurface>21</TargetSurface>  
<LambdaSurface>0.5</LambdaSurface>  
</Plugin>
```

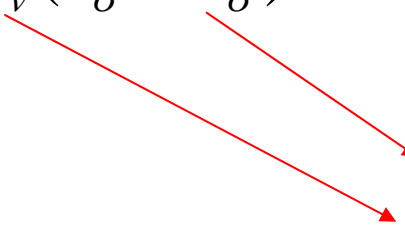
Contact

*contactEnergy(
cell1, cell2)*

```
<Plugin Name="Contact">  
<Energy Type1="Medium" Type2="Medium">0  
</Energy>  
<Energy Type1="Light" Type2="Medium">16  
</Energy>  
<Energy Type1="Dark" Type2="Medium">16  
</Energy>  
<Energy Type1="Light" Type2="Light">16.0  
</Energy>  
<Energy Type1="Dark" Type2="Dark">2.0  
</Energy>  
<Energy Type1="Light" Type2="Dark">11.0  
</Energy>  
</Plugin>
```

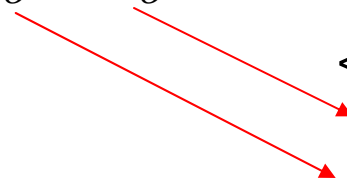
Plugin XML Syntax

$$E = \dots + \lambda_v (v_\sigma - V_\sigma)^2 + \dots$$



```
<Plugin Name="Volume">  
<TargetVolume>25</TargetVolume>  
<LambdaVolume>1.0</LambdaVolume>  
</Plugin>
```

$$E = \dots + \lambda_s (s_\sigma - S_\sigma)^2 + \dots$$



```
<Plugin Name="Surface">  
<TargetSurface>21</TargetSurface>  
<LambdaSurface>0.5</LambdaSurface>  
</Plugin>
```

Plugin XML Syntax – Contact Energy

$$E = \dots + \sum_{x,x'} J_{\tau(\sigma(x)),\tau(\sigma(x'))} (1 - \delta_{\sigma(x),\sigma(x')}) + \dots$$

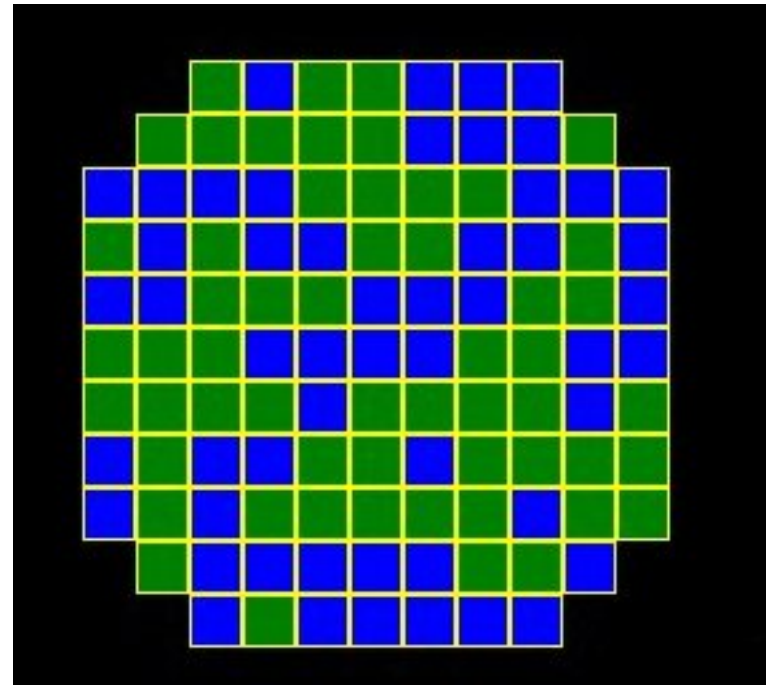
```
<Plugin Name="Contact">
  <Energy Type1="Medium" Type2="Medium">0
</Energy>
  <Energy Type1="Light" Type2="Medium">16.0
</Energy>
  <Energy Type1="Dark" Type2="Medium">16.0
</Energy>
  <Energy Type1="Light" Type2="Light">16
</Energy>
  <Energy Type1="Dark" Type2="Dark">2.0
</Energy>
  <Energy Type1="Light" Type2="Dark">11.0
</Energy>
</Plugin>
```

1- δ term ensures that pixels belonging to the same cell do not contribute to contact energy

Laying Out Cells on the Lattice

Using built-in cell field initializer:

```
<Steppable Type="BlobInitializer">  
  <Region>  
    <Radius>30</Radius>  
    <Center x="40" y="40" z="0"/>  
    <Gap>0</Gap>  
    <Width>5</Width>  
    <Types>Dark,Light</Types>  
  </Region>  
</Steppable>
```



This is just an example of cell field initializer. More general ways of cell field initialization will be discussed later.

NOTE: In actual example **Dark** cells are called **Condensing** and **Light** cells **NonCondensing**

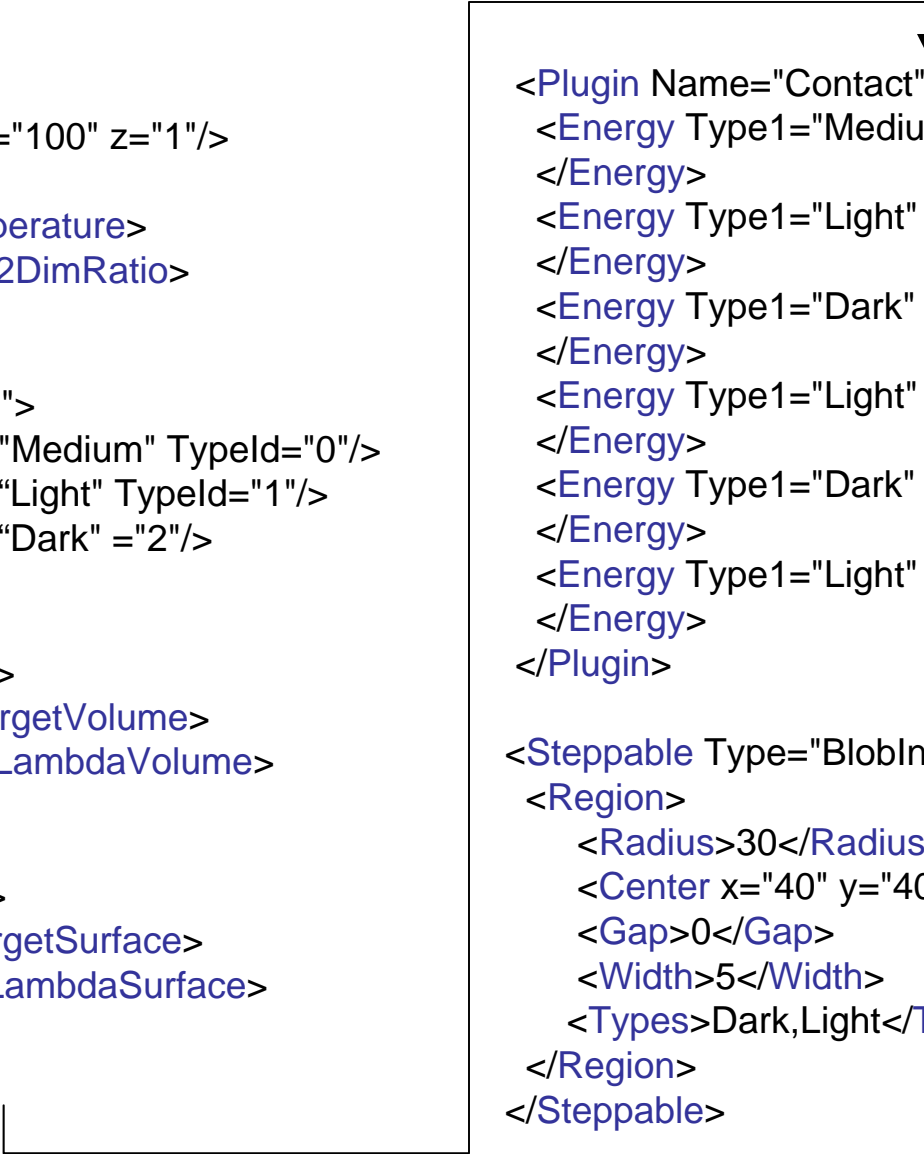
Putting It All Together - cellsort_2D.xml

```
<CompuCell3D>
  <Potts>
    <Dimensions x="100" y="100" z="1"/>
    <Steps>10</Steps>
    <Temperature>2</Temperature>
    <Flip2DimRatio>1</Flip2DimRatio>
  </Potts>

  <Plugin Name="CellType">
    <CellType TypeName="Medium" TypeId="0"/>
    <CellType TypeName="Light" TypeId="1"/>
    <CellType TypeName="Dark" TypeId="2"/>
  </Plugin>

  <Plugin Name="Volume">
    <TargetVolume>25</TargetVolume>
    <LambdaVolume>1.0</LambdaVolume>
  </Plugin>

  <Plugin Name="Surface">
    <TargetSurface>21</TargetSurface>
    <LambdaSurface>0.5</LambdaSurface>
  </Plugin>
```



```
<Plugin Name="Contact">
  <Energy Type1="Medium" Type2="Medium">0
</Energy>
  <Energy Type1="Light" Type2="Medium">16
</Energy>
  <Energy Type1="Dark" Type2="Medium">16
</Energy>
  <Energy Type1="Light" Type2="Light">16
</Energy>
  <Energy Type1="Dark" Type2="Dark">2.0
</Energy>
  <Energy Type1="Light" Type2="Dark">11
</Energy>
</Plugin>

<Steppable Type="BlobInitializer">
  <Region>
    <Radius>30</Radius>
    <Center x="40" y="40" z="0"/>
    <Gap>0</Gap>
    <Width>5</Width>
    <Types>Dark,Light</Types>
  </Region>
</Steppable>

</CompuCell3D>
```

Coding the same simulation in C/C++/Java/Fortran would take you at least 1000 lines of code...

Putting It All Together - Avoiding Common Errors in XML code

1. First specify Potts section, then list all the plugins and finally list all the steppables. This is the correct order and if you mix e.g. plugins with steppables you will get an error. Remember the correct order is

- Potts
- Plugins
- Steppables

2. Remember to match every xml tag with a closing tag

```
<Plugin>
```

```
...
```

```
</Plugin>
```

3. Watch for typos – if there is an error in the XML syntax CC3D will give you an error pointing to the location of an offending line
4. Modify/reuse available examples rather than starting from scratch – saves a lot of time

Foam Coarsening simulation

```
<CompuCell3D>
  <Potts>
    <Dimensions x="101" y="101" z="1"/>
    <Steps>1000</Steps>
    <Temperature>5</Temperature>
    <Flip2DimRatio>1.0</Flip2DimRatio>
    <Boundary_y>Periodic</Boundary_y>
    <Boundary_x>Periodic</Boundary_x>
    <NeighborOrder>2</NeighborOrder>
  </Potts>

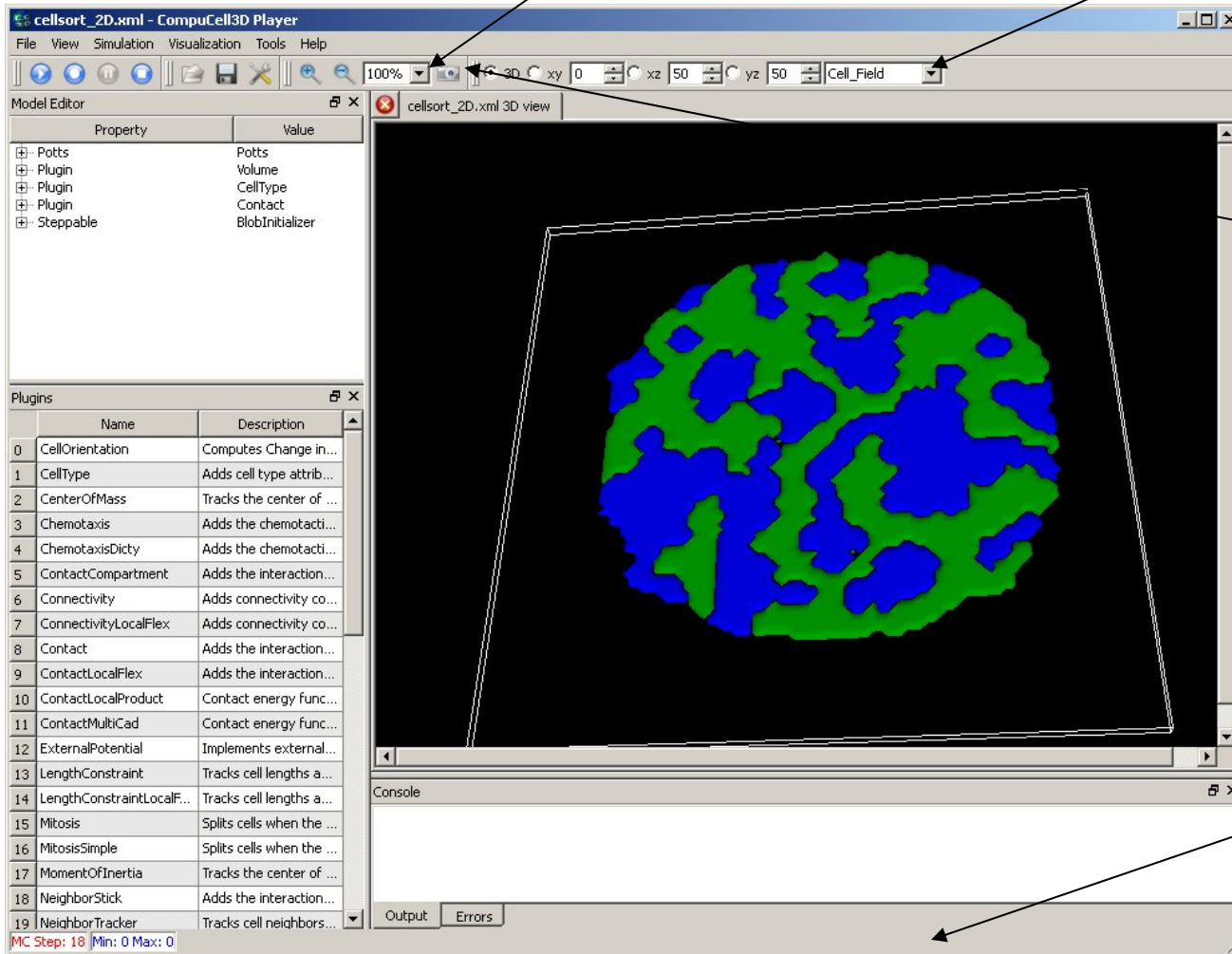
  <Plugin Name="CellType">
    <CellType TypeName="Medium" TypeId="0"/>
    <CellType TypeName="Foam" TypeId="1"/>
  </Plugin>

  <Plugin Name="Contact">
    <Energy Type1="Foam" Type2="Foam">50</Energy>
    <NeighborOrder>2</NeighborOrder>
  </Plugin>

  <Steppable Type="PIFInitializer">
    <PIFName>foaminit2D.pif</PIFName>
  </Steppable>
</CompuCell3D>
```


CompuCellPlayer – the Best Way To Run Simulations

Steering bar allows users to start or pause the simulation, zoom in , zoom out, to switch between **2D and 3D** visualization, change **view modes** (cell field, pressure field , chemical concentration field, velocity field etc..)

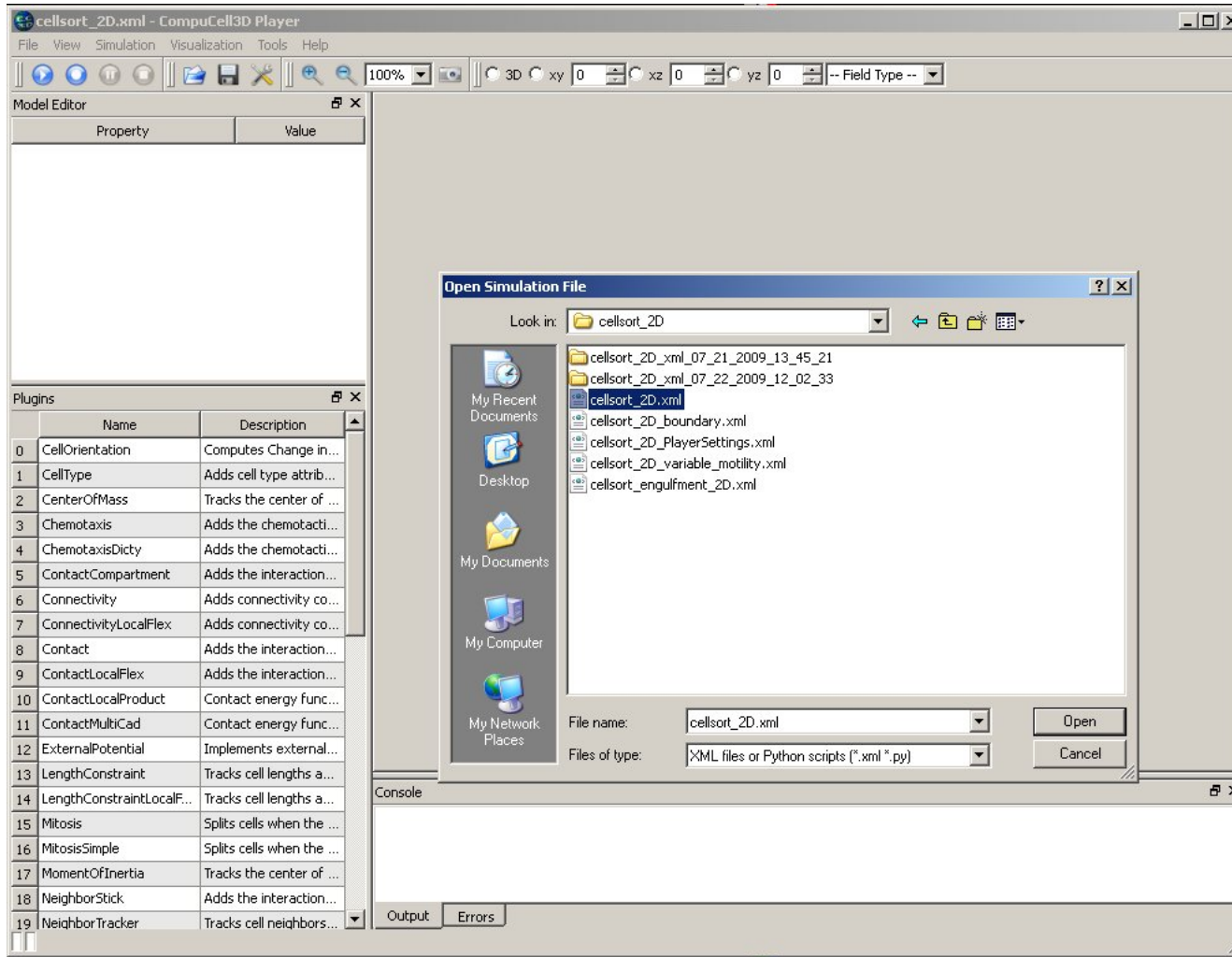


Player can output multiple views during single simulation run – **Add Screenshot** function

Information bar

Opening a Simulation in the Player

Go to File->Open Simulation File



Running Simulation From Command Line

You can simply start the simulation with or without Player straight from command line

Open up console (terminal) and type:

./compucell3d.command -i cellsort_2D.xml (on OSX)

./compucell3d.sh -i cellsort_2D.xml (on Linux)

compucell3d.bat -i cellsort_2D.xml (on Windows) – or simply double click Desktop icon

Running CompuCell3D from command line not only convenient, but sometimes (on clusters) the only option to run the simulation. For more information about command line options please see “Running CompuCell3D” manual available at www.compucell3d.org.

Running the Simulation

- After typing the XML file in your favorite editor all you need to do to run the simulation is to open the XML file in the Player and hit “Play” button.
- Screenshots from the simulations are automatically stored in the directory with name composed of simulation file name and a time at which simulation was started
- As you can see, setting up CompuCell3D simulation was reasonably simple.
- It is quite likely that if you were to code entire simulation in C/C++/Java etc. you would need much more time.
- We hope that now you understand why using CompuCell3D saves you a lot of time and allows you to concentrate on biological modeling and not on writing low level computer code.
- During last year we have improved CompuCell3D performance so that it is on par with hand-written code. Yet, if you really to have the fastest GGH code in the world you should write code your own simulation directly in C or even better in assembly language. Before you do it, make sure you want to spend time rewriting the code that already exist...

Replacing CC3DML with Python

Choosing the Right Text Editor

Since developing CompuCell3D simulation requires typing some simple code it is important that you have the right tools to do that most effectively.

THE BEST EDITOR IS TWEDIT

- On Windows systems we highly recommend Notepad++ editor:

<http://notepad-plus.sourceforge.net/uk/site.htm>

- On Linux you have lots of choices: Kate (my favorite), gedit, mcedit etc.

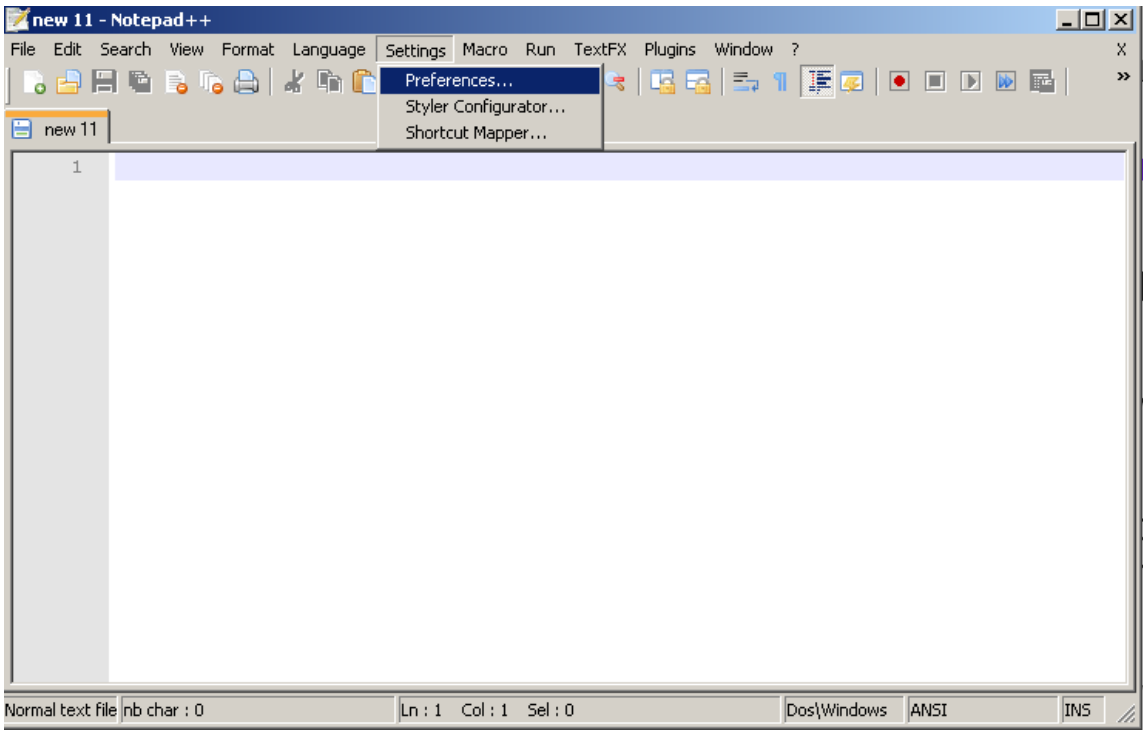
- On OSX situation gets a bit complicated, but there is one editor called Smultron which is good for programming

<http://sourceforge.net/projects/smultron/>

And as usual, if nothing else works there is always vi, and emacs

Configuring Notepad++ for use with Python

Go to Settings->Preferences...

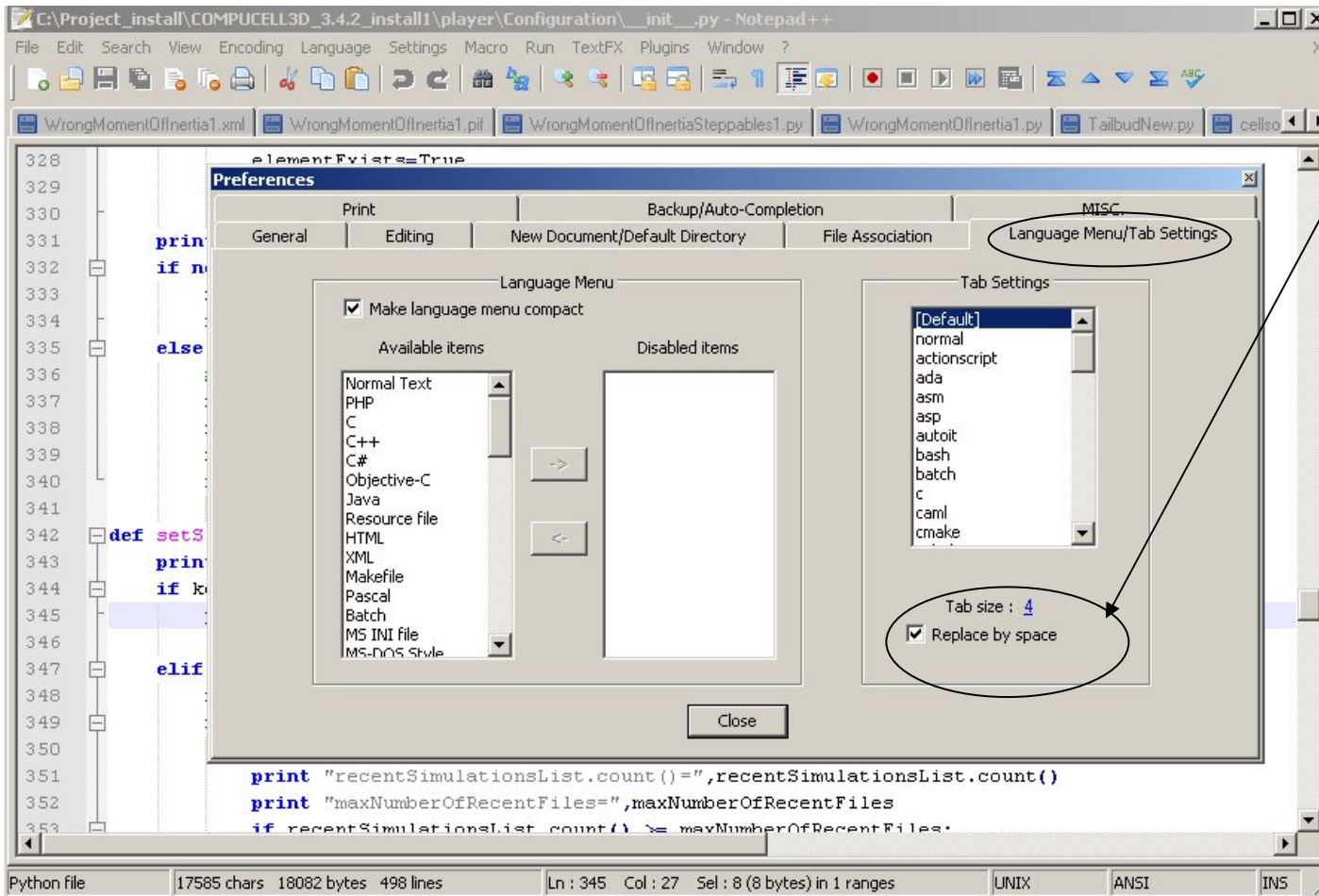


On the “Edit Components” tab change Tab Settings to :

Tab size: 4

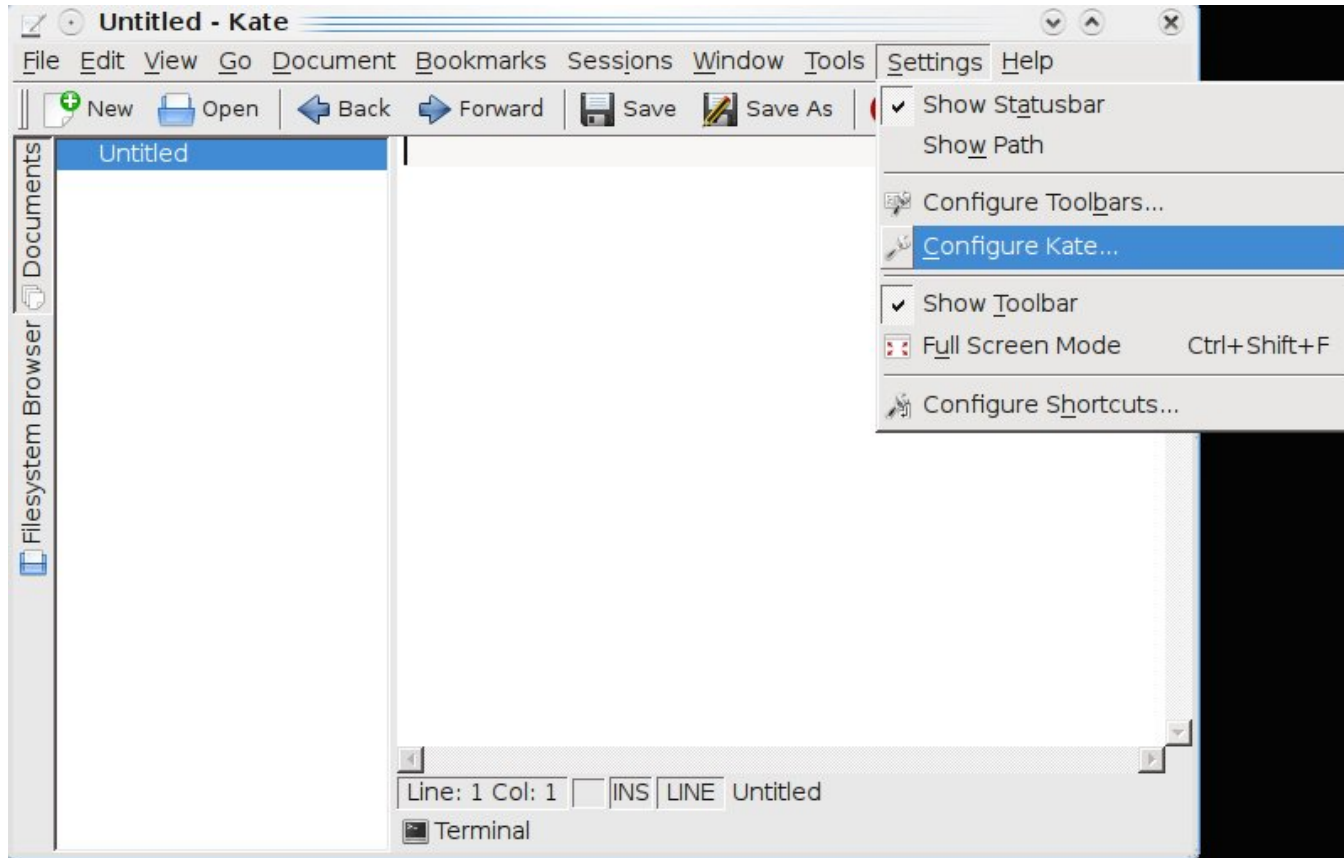
Replace by space: “checked”

Click on the number to change it



Configuring Kate for use with Python

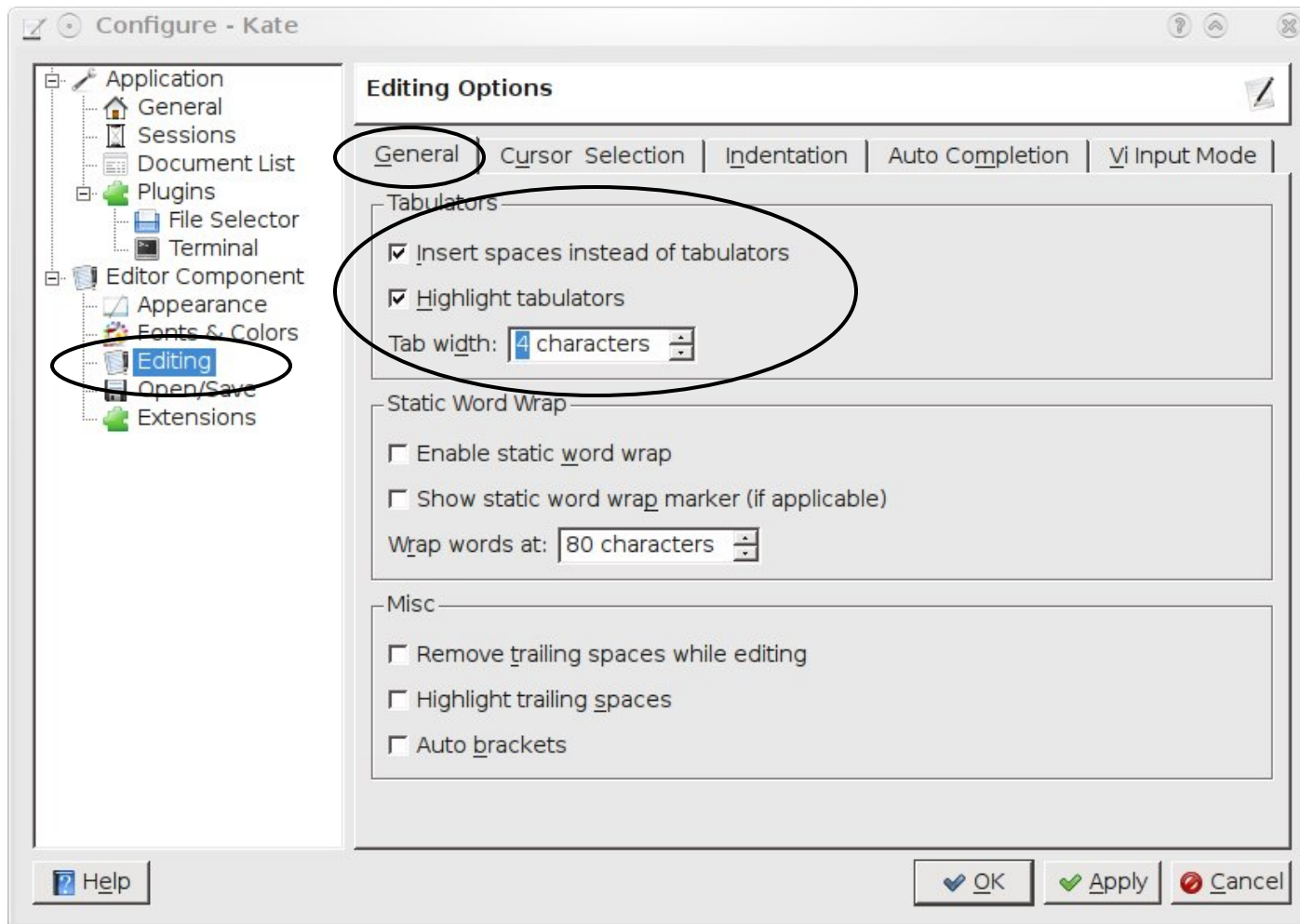
Go to Settings->Configure Kate ...



Click Editing and in the “General” Tab in “Tabulators” section set:

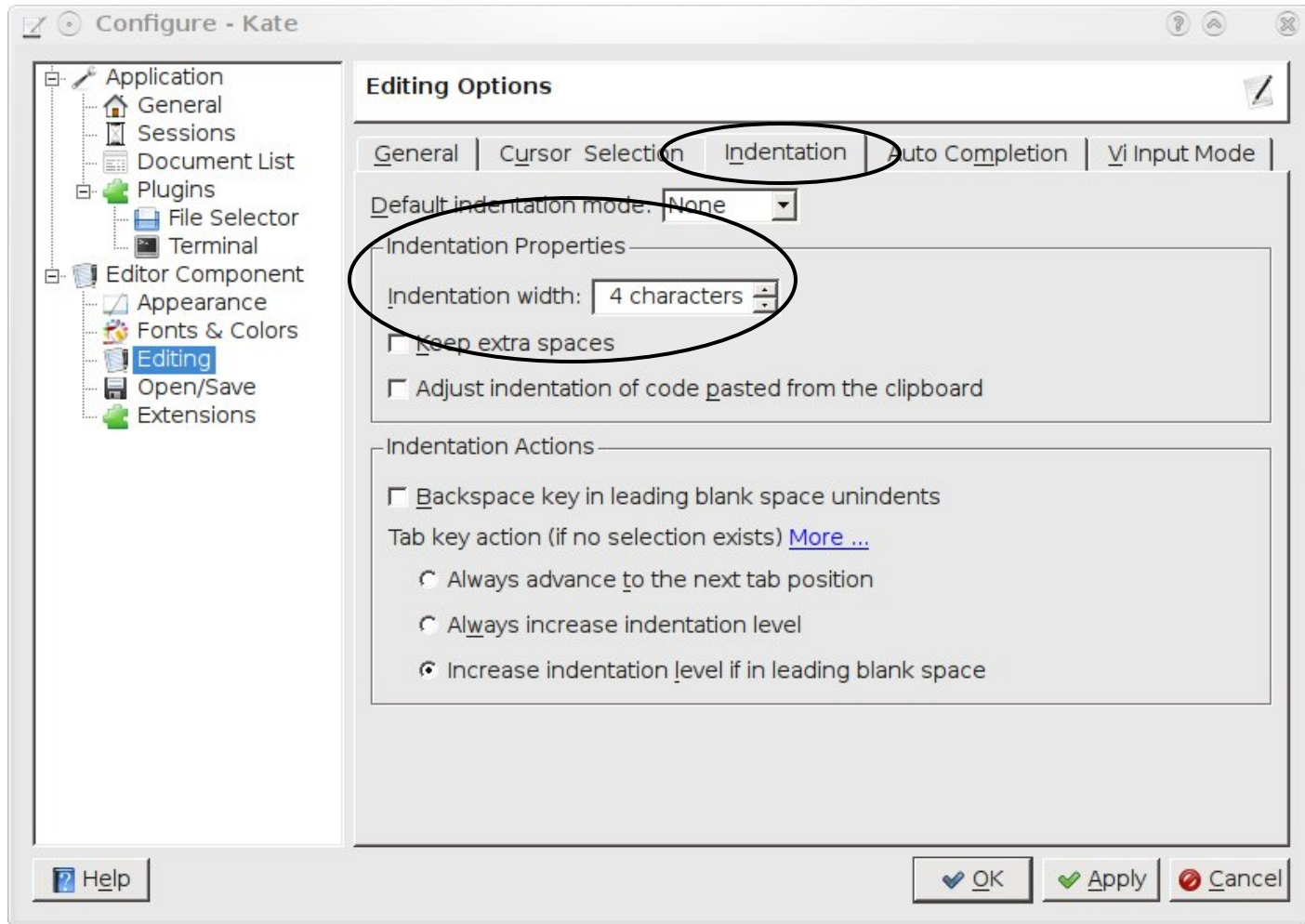
Insert spaces instead of tabulators: “checked”

Tab width: “4 characters”



On “Indentation” tab in “Indentation Properties” section set:

Indentation width: 4 characters



Using Python to describe entire simulations

- Starting with 3.2.0 versions you may get rid of XML file and use Python to describe entire simulation.
- The advantage of doing so is that you have one less file to worry about but also you may more easily manipulate simulation parameters. For example if you want contact energy between two cell types be twice as big as between two other cell types you could easily implement it in Python. Doing the same exercise with CC3DML is a bit harder (but not impossible).
- Python syntax used to describe simulation closely mimics CC3DML syntax. There are however certain differences and inconsistencies caused by the fact that we are using different languages to accomplish same task. Currently there is no documentation explaining in detail Python syntax that replaces CC3DML. It will be developed soon
- The most important reason for defining entire simulation in Python is the possibility of simulation steering i.e. the ability to dynamically change simulation parameters while simulation is running (available in 3.2.1)
- The way you replace XML in Python is purely mechanical and we will show it on a simple example

XML is essentially a definition of hierarchical (tree-like) data structure

```
<Computer>
```

```
  <CPU>Pentium
```

```
    <Frequency Unit="GHz">2.4</Frequency>
  </CPU>
```

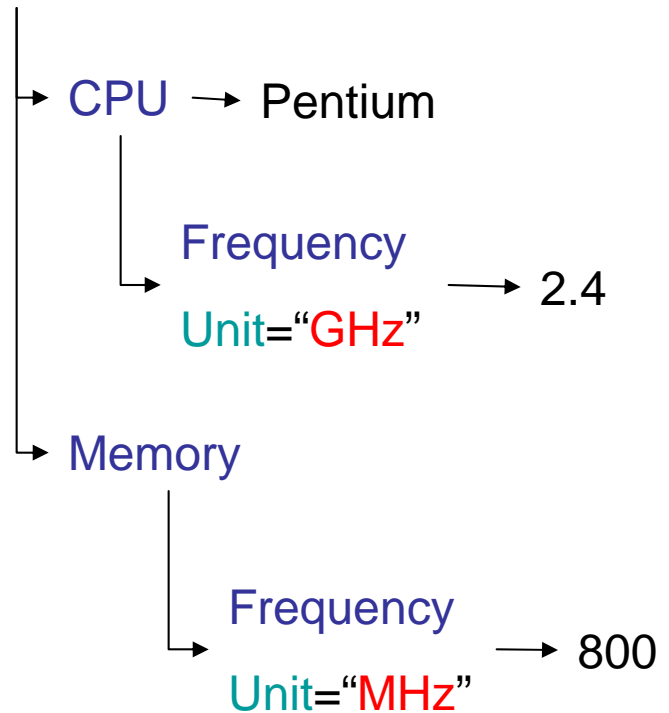
```
  <Memory>DDR-3
```

```
    <Frequency Unit="MHz">800</Frequency>
  </Memory>
```

```
  ...
```

```
</Computer>
```

Computer



Building tree-like structure in a computer language (e.g. Python)

```
root=createElement(...parameters...)
```

```
child1=root.createElement(...parameters...)
```

```
child1_of_child1=child1.createElement(...parameters...)
```

```
child2=root.createElement(...parameters...)
```

```
child1_of_child2=child2.createElement(...parameters...)
```

Replacing XML with Python syntax:

```
import CompuCellSetup
```

```
from XMLUtils import ElementCC3D
```

```
cc3d=ElementCC3D("CompuCell3D")
```

```
potts=cc3d.ElementCC3D("Potts")
```

```
potts.ElementCC3D("Dimensions",{"x":100,"y":100,"z":1})
```

```
potts.ElementCC3D("Anneal",{},10)
```

```
potts.ElementCC3D("Steps",{},1000)
```

```
potts.ElementCC3D("Temperature",{},10)
```

```
potts.ElementCC3D("NeighborOrder",{},2)
```

```
<CompuCell3D>
```

```
<Potts>
```

```
<Dimensions x="100" y="100" z="1"/>
```

```
<Anneal>10</Anneal>
```

```
<Steps>10000</Steps>
```

```
<Temperature>10</Temperature>
```

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

```
</Potts>
```

```
</CompuCell3D>
```

Notice , by using Python we have even saved few lines

Rules:

- To open XML document, create parent ElementCC3D:

```
cc3d=ElementCC3D("CompuCell3D")
```

- For nesting XML elements inside another XML element use the following:

```
potts=cc3d.ElementCC3D("Potts")
```

- If the element has attribute use Python dictionary syntax to list the attributes:

```
potts.ElementCC3D("Dimensions",{"x":100,"y":100,"z":1})
```

- If the XML element has value but no attributes use the following:

```
potts.ElementCC3D("NeighborOrder",{},2)
```

- If the XML element has both value and attributes combine two previous examples

```
potts.ElementCC3D("NeighborOrder",{"LatticeType":"Hexagonal"},2)*
```

*for illustration purposes only

Python-based simulation – template script

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

import CompuCellSetup

sim,simthread = CompuCellSetup.getCoreSimulationObjects()

configureSimulation(sim)

CompuCellSetup.initializeSimulationObjects(sim,simthread)

from PySteppables import SteppableRegistry
steppableRegistry=SteppableRegistry()

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

But you need to implement **configureSimulation** function:

Python

```
def configureSimulation(sim):  
    import CompuCellSetup  
    from XMLUtils import ElementCC3D  
    cc3d=ElementCC3D("CompuCell3D")  
    potts=cc3d.ElementCC3D("Potts")  
    potts.ElementCC3D("Dimensions",{"x":100,"y":100,"z":1})  
    potts.ElementCC3D("Steps", {},1000)  
    potts.ElementCC3D("Temperature", {},10)  
    potts.ElementCC3D("NeighborOrder", {},2)  
    cellType=cc3d.ElementCC3D("Plugin", {"Name":"CellType"})  
    cellType.ElementCC3D("CellType", {"TypeName":"Medium", "TypeId":"0"})  
    cellType.ElementCC3D("CellType", {"TypeName":"Condensing", "TypeId":"1"})  
    cellType.ElementCC3D("CellType", {"TypeName":"NonCondensing", "TypeId":"2"})  
    volume=cc3d.ElementCC3D("Plugin", {"Name":"Volume"})  
    volume.ElementCC3D("TargetVolume", {},25)  
    volume.ElementCC3D("LambdaVolume", {},2.0)
```

Continued...

```
contact=cc3d.ElementCC3D("Plugin",{"Name":"Contact"})
contact.ElementCC3D("Energy", {"Type1":"Medium", "Type2":"Medium"},0)
contact.ElementCC3D("Energy", {"Type1":"NonCondensing", "Type2":"NonCondensing"},16)
contact.ElementCC3D("Energy", {"Type1":"Condensing", "Type2":"Condensing"},2)
contact.ElementCC3D("Energy",{"Type1":"NonCondensing", "Type2":"Condensing"},11)
contact.ElementCC3D("Energy", {"Type1":"NonCondensing", "Type2":"Medium"},16)
contact.ElementCC3D("Energy", {"Type1":"Condensing", "Type2":"Medium"},16)
blobInitializer=cc3d.ElementCC3D("Steppable",{"Type":"BlobInitializer"})
blobInitializer.ElementCC3D("Gap",{},0) blobInitializer.ElementCC3D("Width",{},5)
blobInitializer.ElementCC3D("CellSortInit",{},"yes")
blobInitializer.ElementCC3D("Radius",{},40)
# next line is very important and very easy to forget about. It registers XML description and points
# CC3D to the right XML file (or XML tree data structure in this case)
```

CompuCellSetup.setSimulationXMLDescription(cc3d)

Full example:

Demos/PythonOnlySimulationsExamples/cellsort-2D-player-new-syntax.py

Example: Scaling contact energies – advantage of using Python to configure entire simulation

energyScale=10

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

```
    global energyScale
```

```
    .
```

```
    .
```

```
    contact=cc3d.ElementCC3D("Plugin",{"Name":"Contact"})
```

```
    contact.ElementCC3D("Energy", {"Type1":"Medium", "Type2":"Medium"},0)
```

```
    contact.ElementCC3D("Energy", {"Type1":"NonCondensing", "Type2":"NonCondensing"},1.6*energyscale)
```

```
    contact.ElementCC3D("Energy", {"Type1":"Condensing", "Type2":"Condensing"},0.2*energyscale)
```

```
    contact.ElementCC3D("Energy", {"Type1":"NonCondensing", "Type2":"Condensing"},1.1*energyscale)
```

```
    contact.ElementCC3D("Energy", {"Type1":"NonCondensing", "Type2":"Medium"},1.6*energyscale)
```

```
    contact.ElementCC3D("Energy", {"Type1":"Condensing", "Type2":"Medium"},1.6*energyscale)
```

It would be a bit awkward (but not impossible) to have same functionality in CC3DML...

CompuCell Player 101

The background features a 3D visualization of a cell simulation, possibly a microorganism, rendered in a light green color. This visualization is overlaid on a semi-transparent green interface that includes various control elements. At the top, there are labels for 'Configure' and 'Help'. Below these, there are radio buttons for '3D', 'XY', and 'YZ' views, with 'YZ' currently selected. Next to the 'YZ' view are two numerical input fields, both containing the value '20'. To the right, there is a 'Plot Type' dropdown menu. At the bottom of the interface, there are numerical values: '0.00', '0.125', '0.250', and '0.375', which likely represent time or simulation steps. The overall aesthetic is clean and technical, typical of scientific software.

CompuCell Player – The Basics

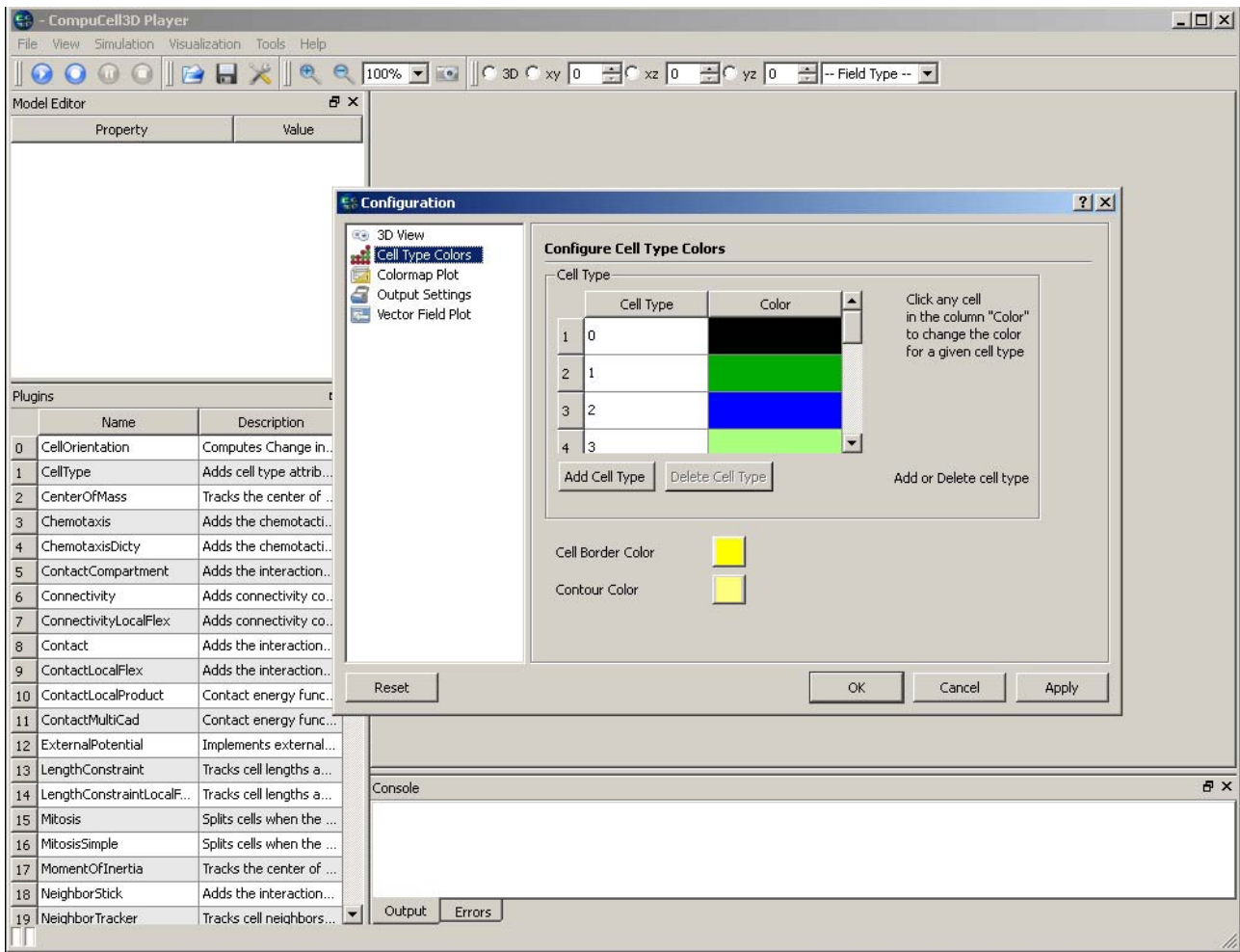
- **Caution:** CompuCellPlayer is still under development so some options may not work properly however, for the most part this is fully functional ComUCell3D front end and much better than its predecessor
- For the most part , CompuCell Player is fairly intuitive to use. It is quite important to get familiar with configure options to the get most out of the Player
- CompuCell Player is a **graphical front-end to CompuCell3D** computational part. It is written in Python using PyQt4 and VTK. It also uses C++ wrapped code for performance critical processing.
- Because of PyQt4 , it provides native look and feel on Linux, OSX and Windows. This means that dialog and windows will look pretty much the same as other dialogs in your operating system. Traditionally, Qt toolkits look the ugliest on OSX.
- **CompuCell Player provides basic visualization capabilities for CompuCell3D** simulations. It automatically detects types of plots for given simulation.
- **Saves users the hassle of writing visualization code.**
- Helps debugging simulations
- Next version of Player will allow users to write their own visualization pipeline using VTK standards

Capabilities of CompuCellPlayer– Why You Should Use the Player.

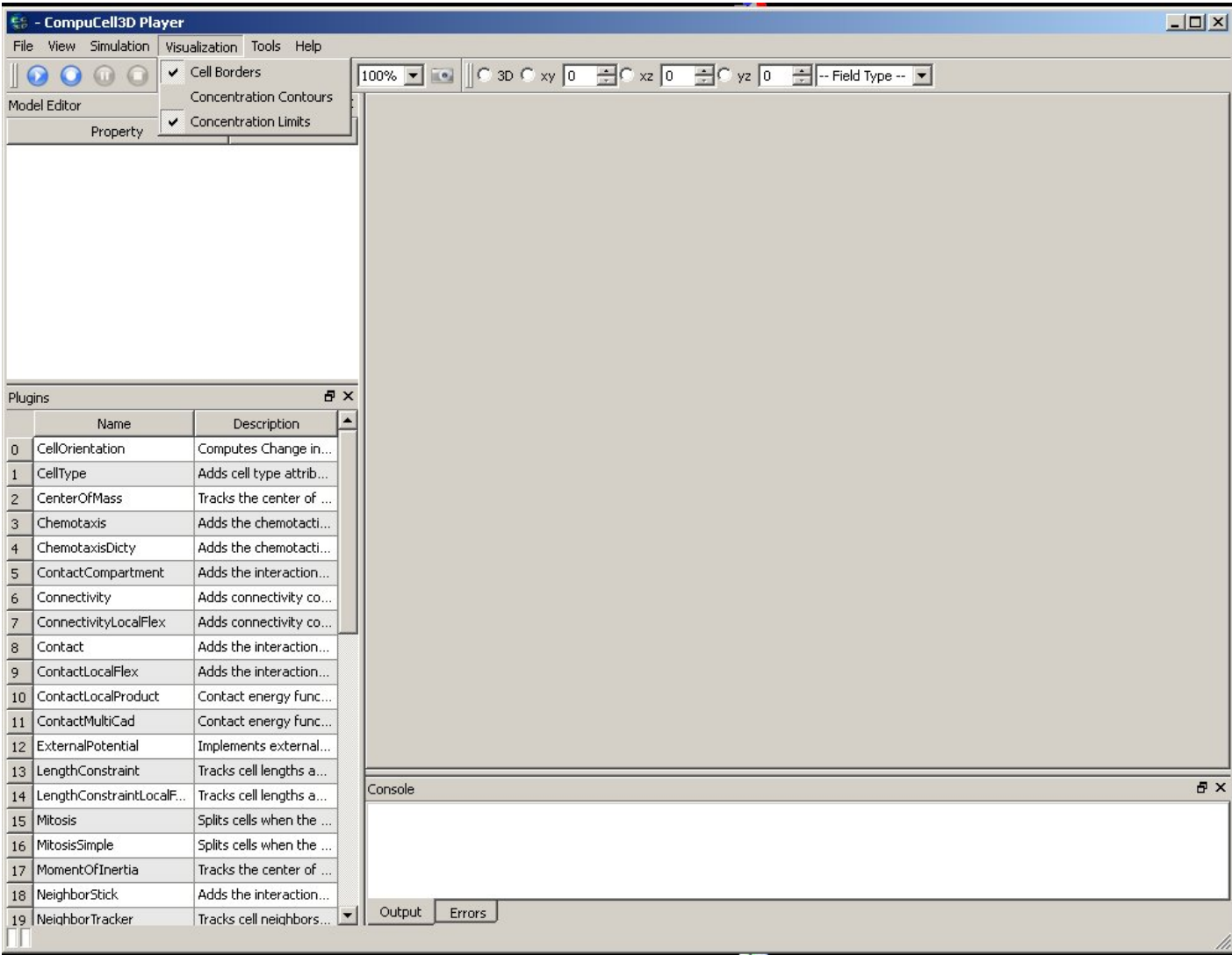
- Provides wide range of visualization - cell field plots, concentration plots, vector field plots in both 2- and 3-D.
- Allows to store multiple lattice views in a single run. For example users can store multiple projections of the cell lattice, concentration fields, various 3D views *etc...* in a single run.
- Can be run in GUI and silent mode (*i.e.* without displaying GUI but still saving screenshots)
- Is ready to be used on clusters that do not have X-server installed. This feature is essential for doing “production runs” of your simulations.
- Concentration fields and vector fields initialized from Python level can easily be displayed in the Player. Yes, you can control the Player visualizations from Python level.
- Configurable from XML level for those users who prefer typing to clicking

Configuring the Player

Most of Player's configuration options are accessible through **Tools->Configuration...** and **Visualization** menus.

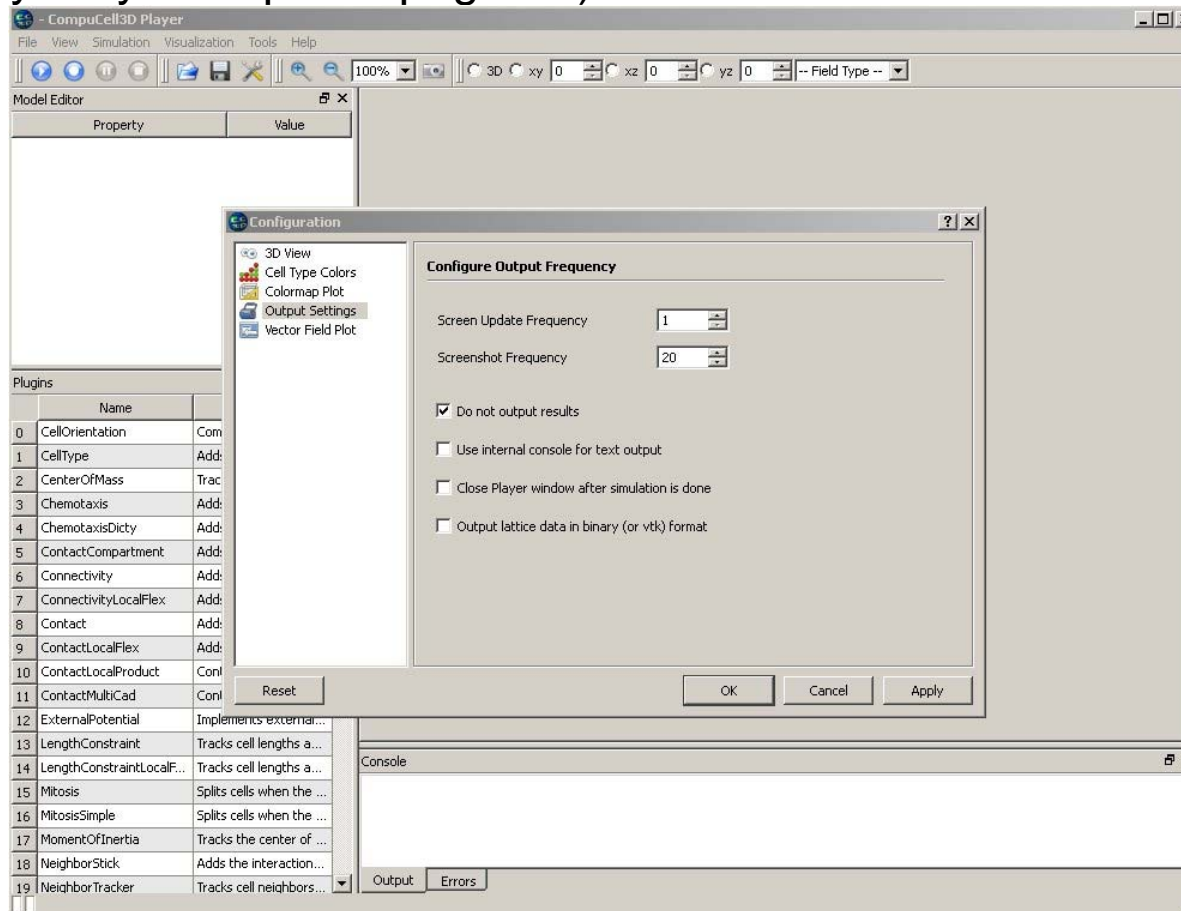


Visualization Menu allows you to choose whether in 2D cell borders should be displayed or not (in 3D borders are not drawn at all). You can also select to draw isocontour lines for the concentration plots and turn on and off displaying of the information about minimum and maximum concentration.



Screen update frequency is a parameter that defines how often (in units of MCS) Player screen should be updated. Note, if you choose to update screen too often (say every MCS) you will notice simulation speed degradation because it does take some time to draw on the screen. You may also choose not to output any files by checking “**Do not output results**” check-box. Additionally you have the option to output simulation data in the VTK format for later replay.

Screenshot frequency determines how often screenshots of the lattice views will be taken (currently Player outputs *.png files).



Screenshots

Screenshots are taken every “Screenshot Frequency” MCS

By default Player will store screenshots of the currently displayed lattice view.

In addition to this users can choose to store additional screenshots at the same time. Simply switch to different lattice view, click camera button. Those additional screenshots will be taken irrespectively of what Player currently displays.

Once you selected additional screenshots it is convenient to save screenshot description file (it is written automatically by the Player, user just provide file name). Next time you decide to run CompuCell3D you may just use command

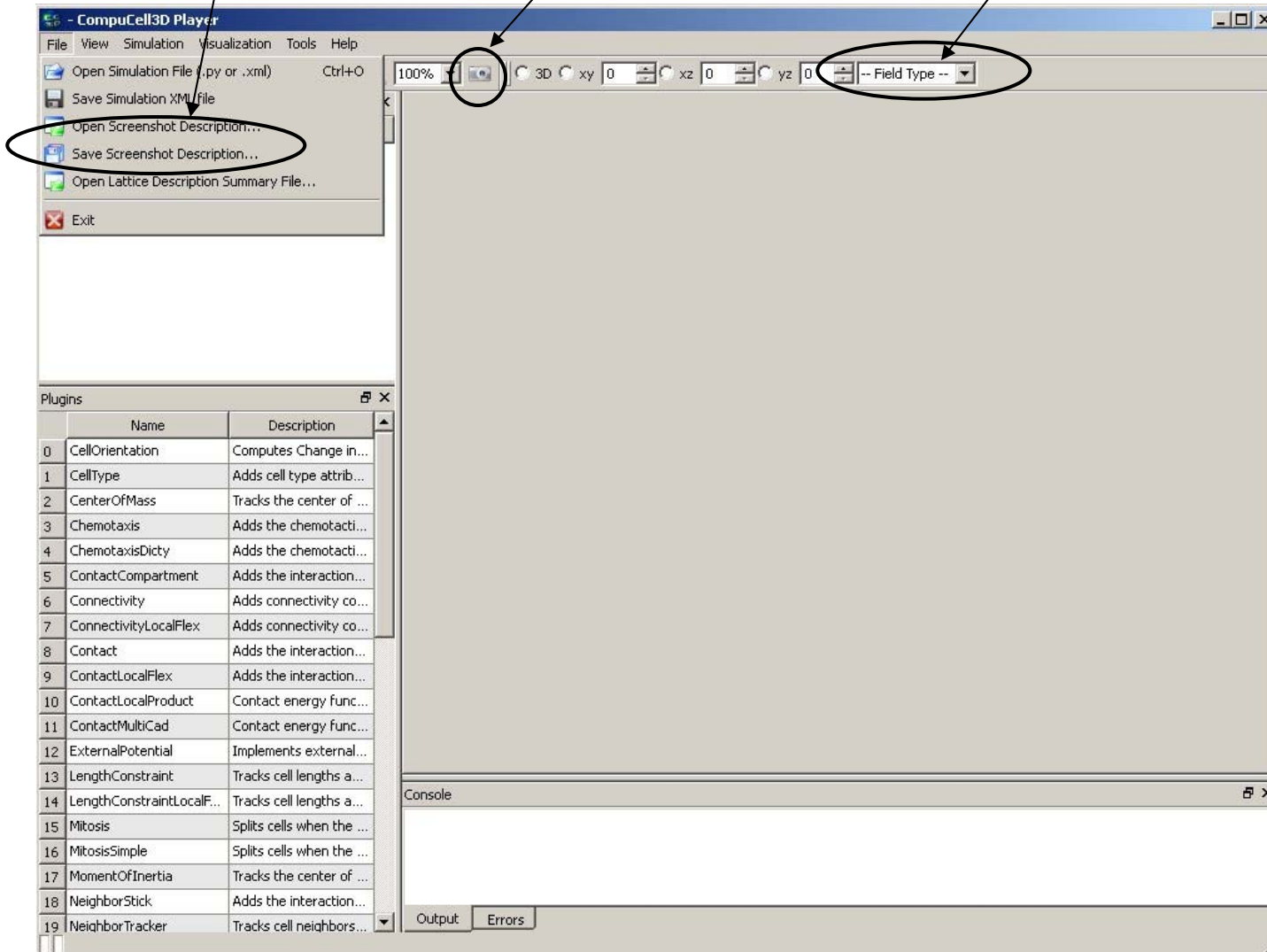
```
compuCell3d.sh -s screenshotDescriptionFile_cellsort.txt -i cellsort_2D.xml
```

This will run simulation where stored screenshots will be taken

When you picked lattice views, you may save screenshot description file for later reuse

Click camera button on select lattice views

Notice, you may change plot types as well



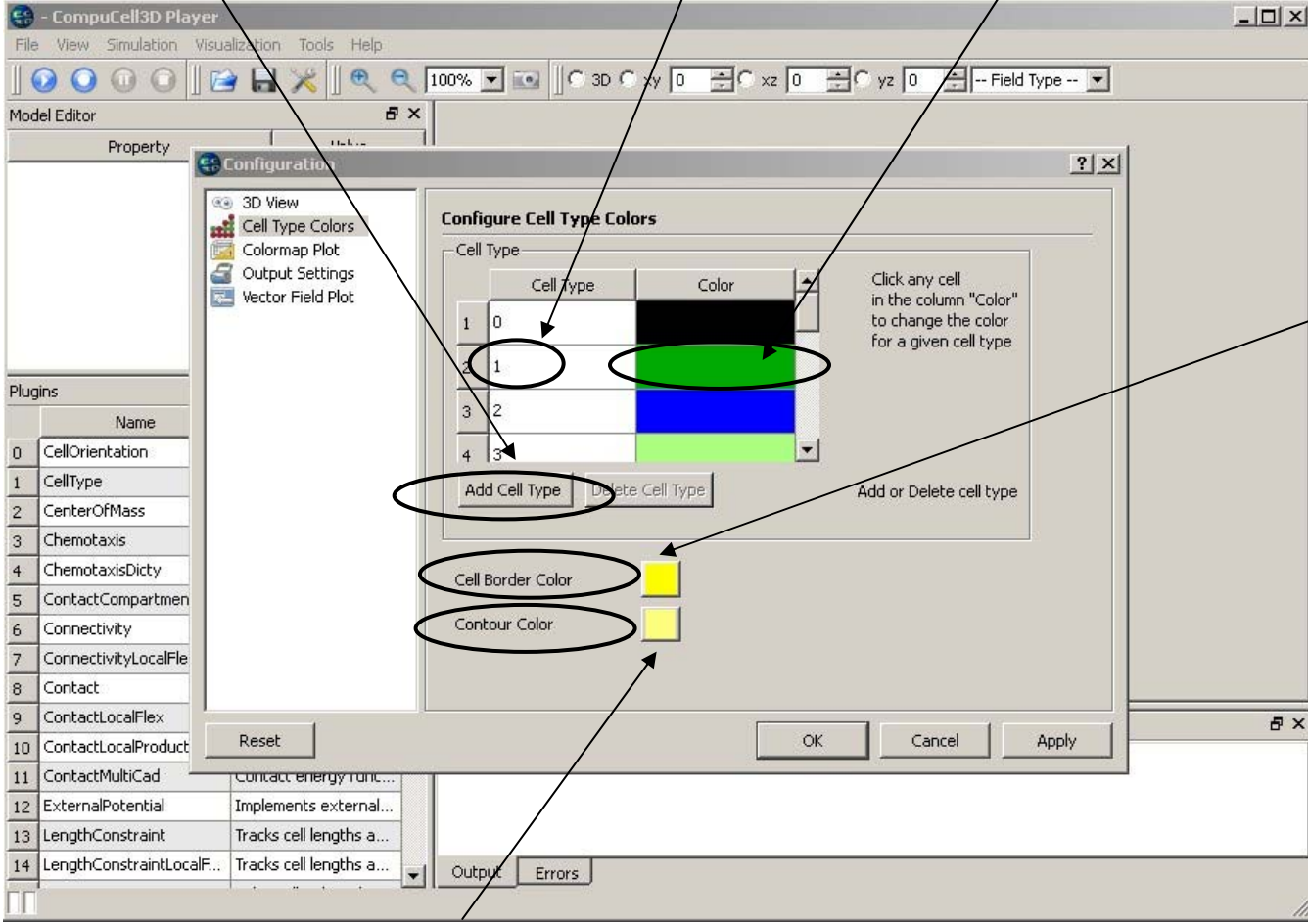
Configure->cell type colors...

To enter new cell type click **“Add Cell Type”** button

Enter cell type number here

Click here to change color for cell type 1

Click here to change cell border color

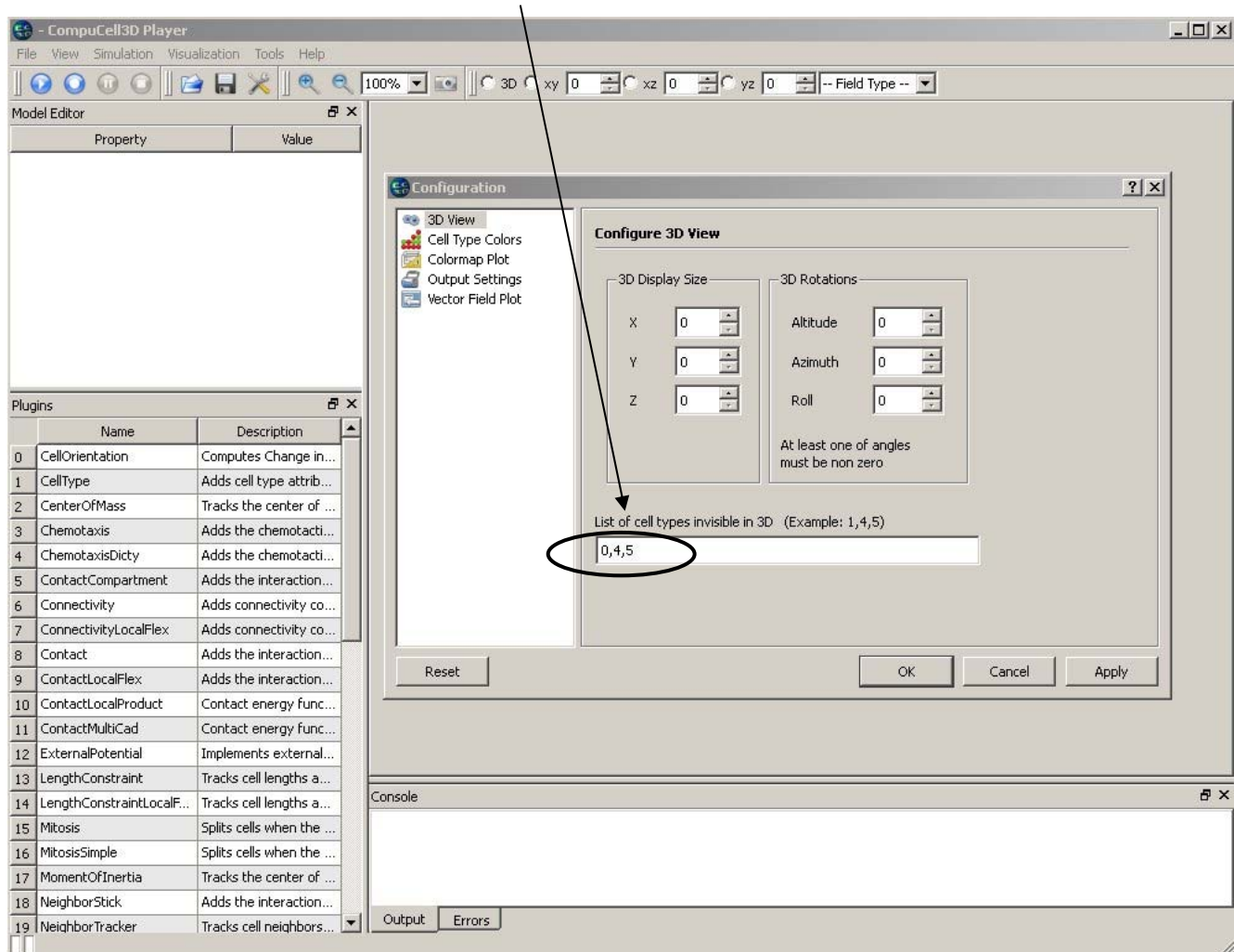


Click here to change isocontour color

Configure->Cell types invisible in 3D...

Sometimes when you open up the simulation and switch to 3D view you may find that your simulation looks like solid a parallelepiped. This might be due to a box made out of frozen cells that hides inside other cells. In this case you need to make the box invisible.

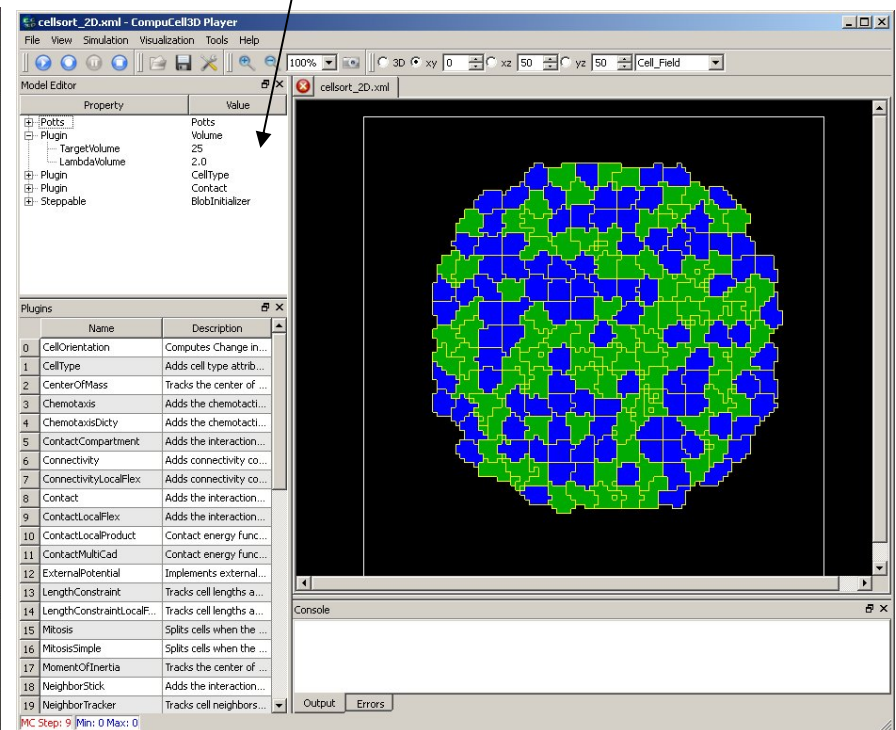
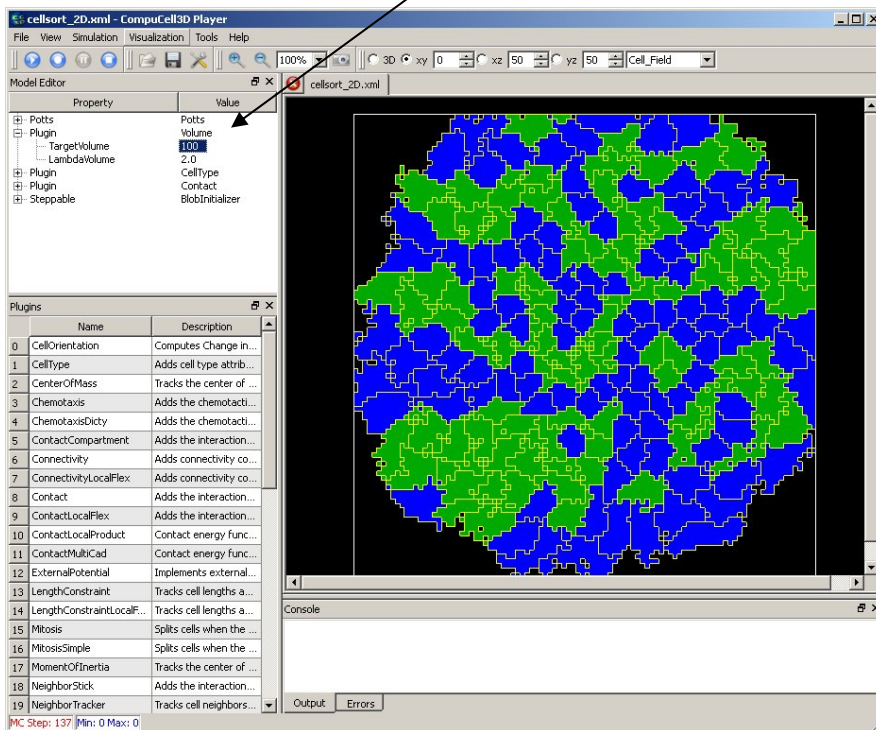
Type cell type number that you want to be invisible in 3D in this box. Notice, by default Player will not display Medium (type 0). Here we also make types 4 and 5 invisible



Steering the simulation

CompuCell3D Player will allow you to change most of the parameters of the XML file while the simulation is running.

Use steering panel to change simulation parameters. Make sure you pause simulation before doing this



Target volume = 100

Screenshot was taken before simulation had time to equilibrate

Target volume = 25

Using different kind of lattices with CompuCell3D

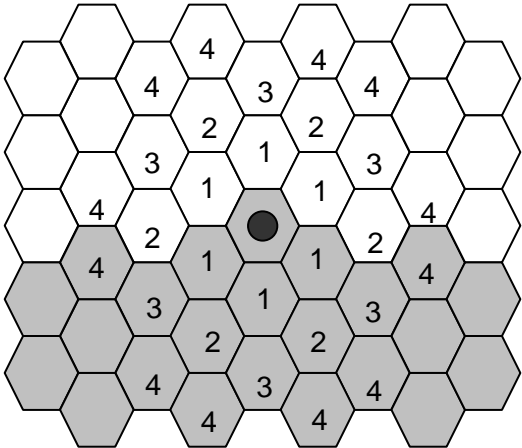
- Current version of CompuCell3D allows users to run simulations on square and hexagonal lattices.
- Other regular geometries (e.g. triangular) can be implemented fairly easily
- Some plugins work on square lattice only - e.g. local connectivity plugin
- Switching to hexagonal lattice requires only one line of code in the Potts section

<LatticeType>Hexagonal</LatticeType>

- Model parameters may need to be adjusted when going from one type lattice to another. This is clearly an inconvenience but we will try to provide a solution in the future
- Different lattices have varying degrees of lattice anisotropy. In many cases using lower anisotropy lattice is desired (e.g. foam coarsening simulation on hexagonal lattice). It is also important to check results of your simulation on different kind of lattices to make sure you don't have any lattice-specific effects.
- CompuCell3D makes such comparisons particularly easy

Nearest neighbors in 2D and their Euclidian distances from the central pixel

		4	3	4	
	4	2	1	2	4
	3	1	●	1	3
	4	2	1	2	4
		4	3	4	



	2D Square Lattice		2D Hexagonal Lattice	
Neighbor Order	Number of Neighbors	Euclidian Distance	Number of Neighbors	Euclidian Distance
1	4	1	6	$\sqrt{2}/\sqrt{3}$
2	4	$\sqrt{2}$	6	$\sqrt{6}/\sqrt{3}$
3	4	2	6	$\sqrt{8}/\sqrt{3}$
4	8	$\sqrt{5}$	12	$\sqrt{14}/\sqrt{3}$

SquareLattice:

Square in 2D

Cube in 3D

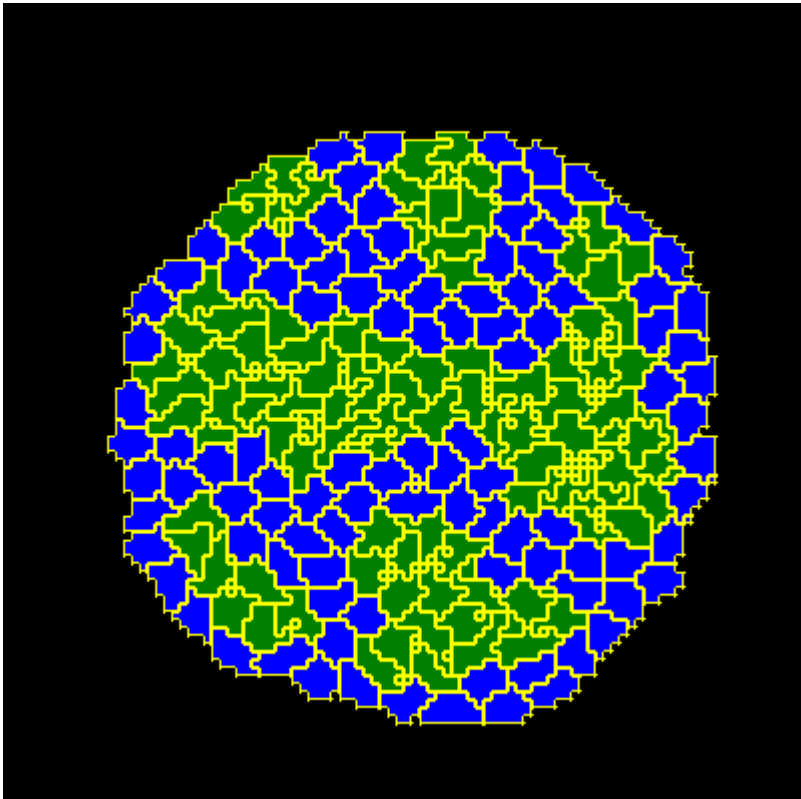
Hexagonal lattice:

Hexagon in 2D

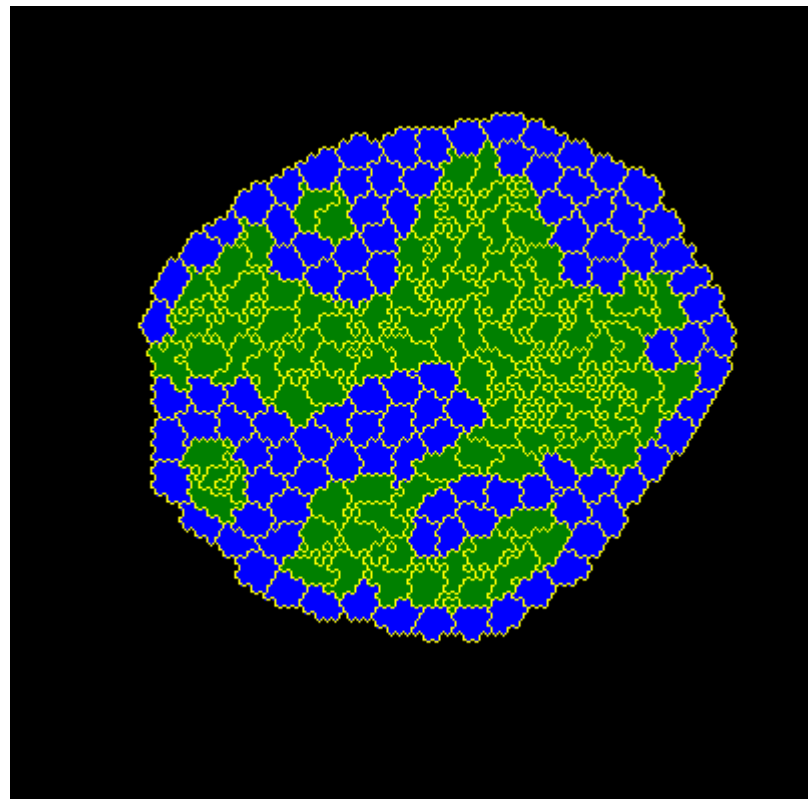
Rhombic dodecahedron in 3D

Cell-sorting simulation on square and hexagonal lattices

The simulation parameters were kept the same for the two runs



1000 MCS



1000 MCS

Cell Attributes

CompuCell3D cells have a default set of attributes:

Volume, surface, center of mass position, cell id etc...

Additional attributes are added during runtime:

List of cells neighbors, polarization vector etc...

To keep parameters up-to-date users need to declare appropriate plugins in the CC3DML configuration file.

For example, to make sure surface of cell is up-to-date users need to make sure that **SurfaceTracker** plugin is registered:

Include :

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

or use Surface plugin which will implicitly call SurfaceTracker

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

```
  <LambdaSurface>0.0</LambdaSurface>
```

```
  <TargetSurface>25.0</TargetSurface>
```

```
</Plugin>
```

But here surface tracking costs you extra calculation of surface energy term:

$$E = \dots + \lambda (s - S_T)^2 + \dots$$

More Flexible Specification of Surface and Volume Constraints

```
<Plugin Name="VolumeFlex">  
  <VolumeEnergyParameters CellType="Amoeba" TargetVolume="150" LambdaVolume="10"/>  
  <VolumeEnergyParameters CellType="Bacteria" TargetVolume="10" LambdaVolume="50"/>  
</Plugin>
```

You may specify different volume and surface constraints for different cell types. This can be done entirely at the XML level.

$$E = \lambda^V_{\tau} (v_{\tau} - V_{\tau})^2$$

Type dependent quantities

$$E = \lambda^S_{\tau} (s_{\tau} - S_{\tau})^2$$


```
<Plugin Name="SurfaceFlex">  
  <SurfaceEnergyParameters CellType="Amoeba" TargetSurface="60" LambdaSurface="10"/>  
  <SurfaceEnergyParameters CellType="Bacteria" TargetSurface="12" LambdaSurface="20"/>  
</Plugin>
```

Even More Flexible Specification of Surface and Volume Constraints

<Plugin Name="VolumeLocalFlex"/>

$$E = \lambda^V_{\sigma} (v_{\sigma} - V_{\sigma})^2$$

<Plugin Name="SurfaceLocalFlex"/>

$$E = \lambda^S_{\sigma} (s_{\sigma} - S_{\sigma})^2$$


Notice that all the parameters are local to a cell. Each cell might have different target volume (target surface) and different λ volume (surface). You will need to use Python to initialize or manipulate those parameters while simulation is running. There is currently no way to do it from XML level. I am not sure it would be practical either.

Tracking Cell Neighbors

Sometimes in your simulation you need to have access to a current list of cell neighbors. CompuCell3D makes this task easy:

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

Inserting this statement in the plugins section of the XML will ensure that at any given time the list of cell neighbors will be accessible to the user. You can access such a list either using C++ or Python. In addition to storing neighbor list, a common surface area of a cell with its neighbors is stored.

Tracking Center of Mass of Each Cell

Including

<Plugin Name="CenterOfMass"/>

statement in your XML code (remember to put it in the correct place) will enable cell centroid tracking:

$$x_{CM}^C = \sum_{i-pixel} x_i \quad y_{CM} = \sum_{i-pixel} y_i \quad z_{CM}^C = \sum_{i-pixel} z_i$$

To get a center of mass of cell you will need to divide centroids by the cell volume:

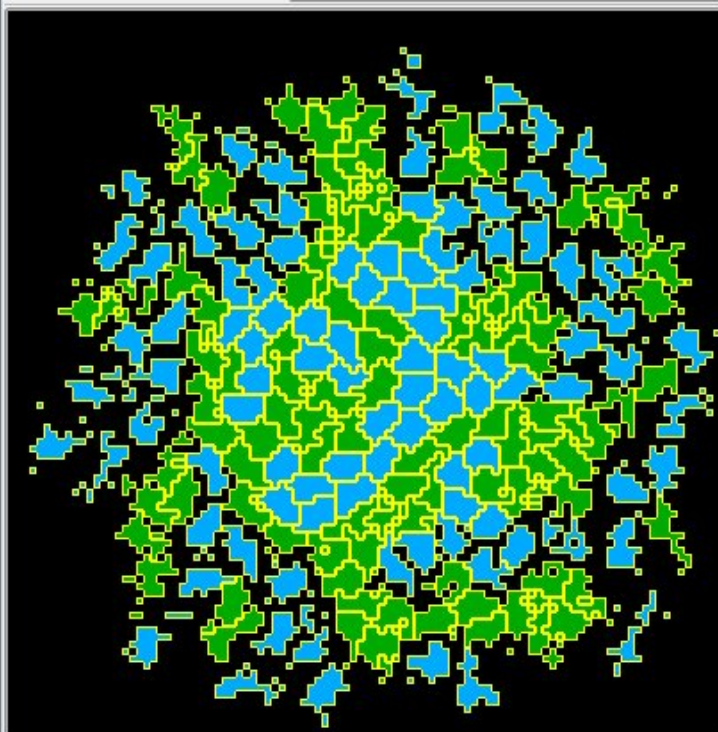
$$x_{CM} = \frac{x_{CM}^C}{V} \quad y_{CM} = \frac{y_{CM}^C}{V} \quad z_{CM} = \frac{z_{CM}^C}{V}$$

Practical way of guessing contact energy hierarchy

Basic facts:

- Cells that have high contact energies between themselves, when they come together they **increase** overall energy of the system. **Such cells tend to stay away from each other.**
- Cells that have low contact energies between themselves, when they come together they **decrease** overall energy of the system. **Such cells tend to cluster together.**
- Those two rules are helpful when determining contact energy hierarchy. Simply cells of one type like to be surrounded by those cells with which the contact energy is the lowest.
- And vice versa, if you want to make two cells not to touch each other, make sure that contact energy between them is high.

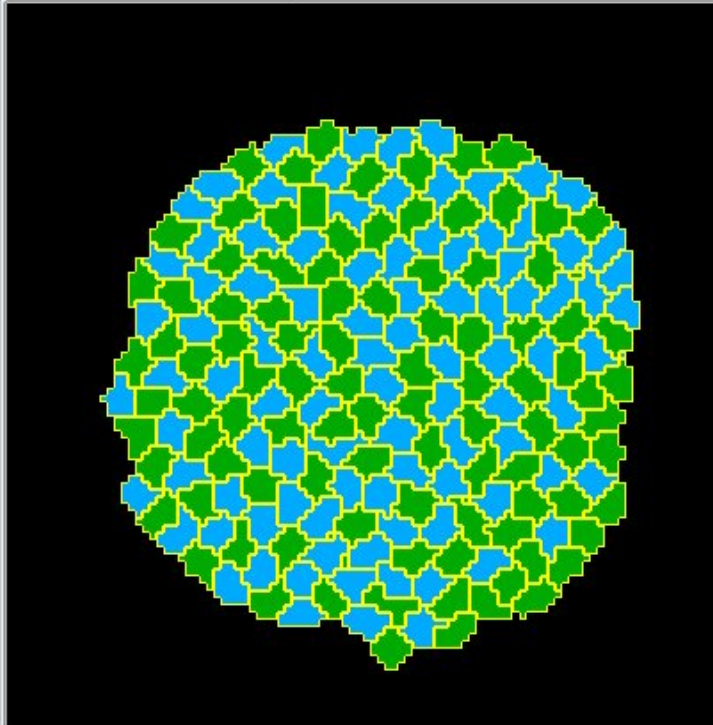
Examples of different contact energy hierarchies



Cell sorting simulation where cells of both type like to be surrounded by medium. That is contact energy between **Condensing** and **Medium** as well as between **NonCondensing** and **Medium** is very low

$$J_{CM} = J_{NM} < J_{NN} < J_{CC} < J_{NC}$$

Examples of different contact energy hierarchies



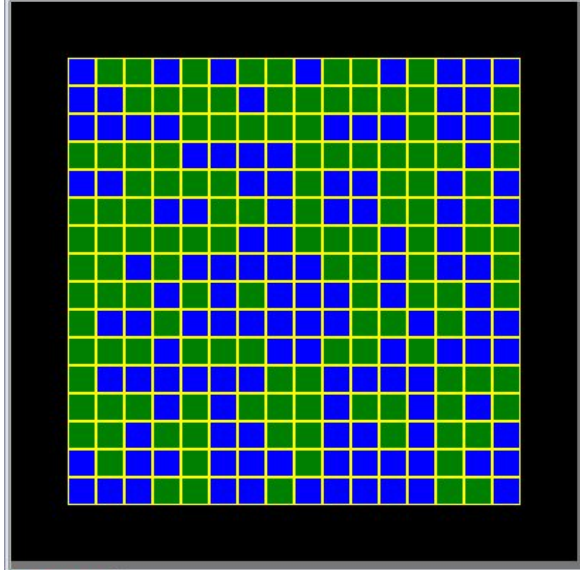
Cell sorting simulation where cells of both type do not like to be surrounded by medium and cells of homotypic cells do not like each other

$$J_{NC} \ll J_{NN} = J_{CC} < J_{CM} = J_{NM}$$

XML initializers - UniformInitializer

You may initialize simple geometries of cell clusters directly from XML

```
<Steppable Type="UniformInitializer">  
  <Region>  
    <BoxMin x="10" y="10" z="0"/>  
    <BoxMax x="90" y="90" z="1"/>  
  
    <Types>Condensing,NonCondensing</Types>  
  
    <Gap>0</Gap>  
    <Width>5</Width>  
  </Region>  
</Steppable>
```



Specify box size and position

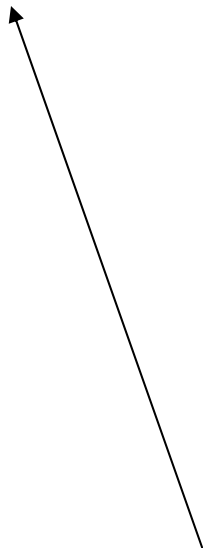
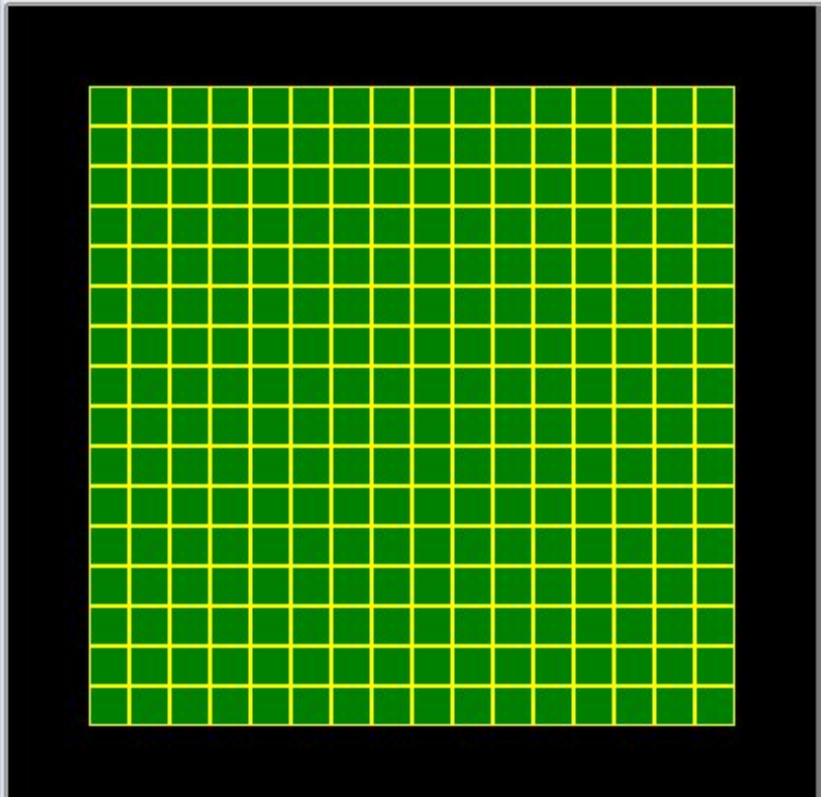
Specify cell types – here the box will be filled with cells whose types are randomly chosen (either 1 or 2)

Choose cell size and space between cells

```
<Steppable Type="UniformInitializer">
  <Region>
    <BoxMin x="10" y="10" z="0"/>
    <BoxMax x="90" y="90" z="1"/>

    <Types>Condensing</Types>

    <Gap>0</Gap>
    <Width>5</Width>
  </Region>
</Steppable>
```

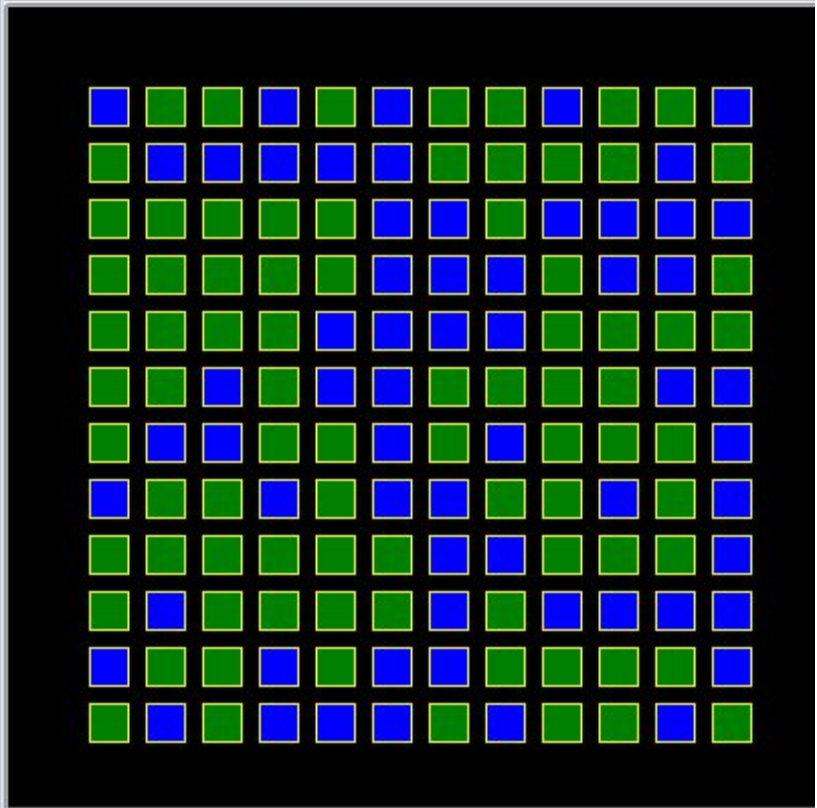


Notice, we have only specified one type (Condensing) thus all the cells are of the same type

```
<Steppable Type="UniformInitializer">
  <Region>
    <BoxMin x="10" y="10" z="0"/>
    <BoxMax x="90" y="90" z="1"/>

    <Types>Condensing,NonCondensing</Types>

    <Gap>2</Gap>
    <Width>5</Width>
  </Region>
</Steppable>
```

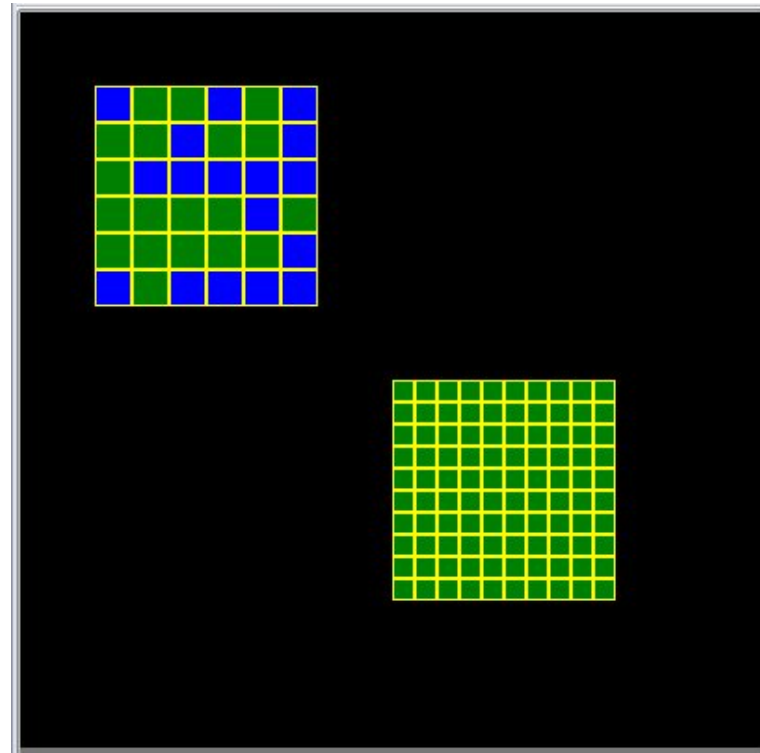


Introducing a gap between cells

```

<Steppable Type="UniformInitializer">
  <Region>
    <BoxMin x="10" y="10" z="0"/>
    <BoxMax x="40" y="40" z="1"/>
    <Gap>0</Gap>
    <Width>5</Width>
    <Types>Condensing,NonCondensing</Types>
  </Region>
  <Region>
    <BoxMin x="50" y="50" z="0"/>
    <BoxMax x="80" y="80" z="1"/>
    <Gap>0</Gap>
    <Width>3</Width>
    <Types>Condensing</Types>
  </Region>
</Steppable>

```

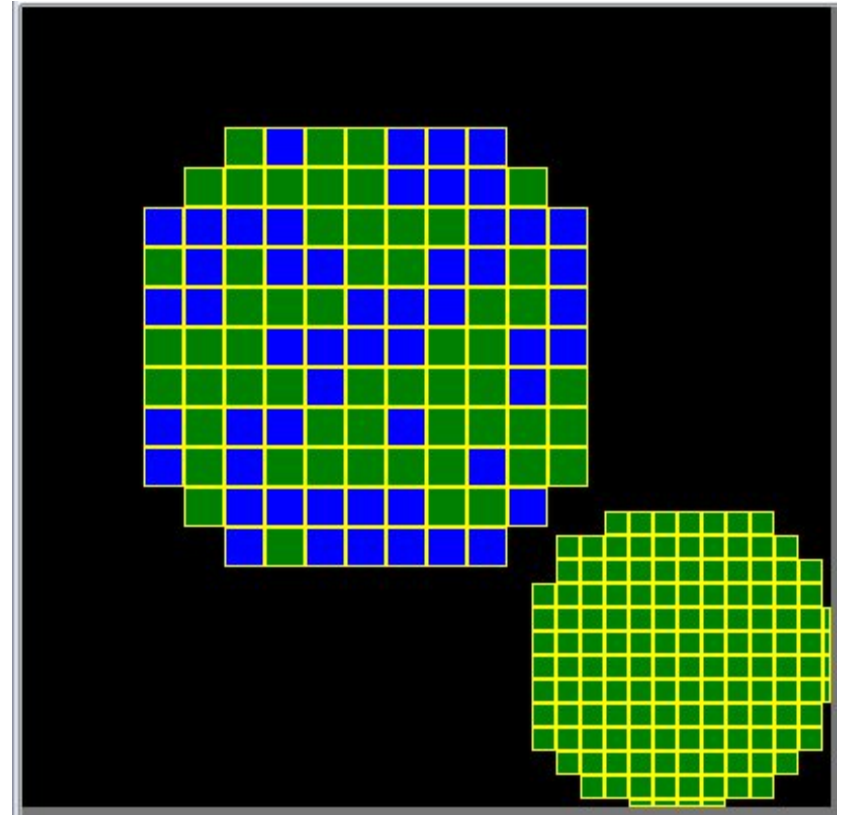


Notice, we have defined two regions with different cell sizes and different types

XML initializers - BlobInitializer

```
<Steppable Type="BlobInitializer">
  <Region>
    <Radius>30</Radius>
    <Center x="40" y="40" z="0"/>
    <Gap>0</Gap>
    <Width>5</Width>
    <Types>Condensing,NonCondensing</Types>
  </Region>

  <Region>
    <Radius>20</Radius>
    <Center x="80" y="80" z="0"/>
    <Gap>0</Gap>
    <Width>3</Width>
    <Types>Condensing</Types>
  </Region>
</Steppable>
```



Defining two regions with different cell sizes and different types for BlobInitializer is very similar to the same task with UniformInitializer. There are some new XML tags which differ the two initializers.

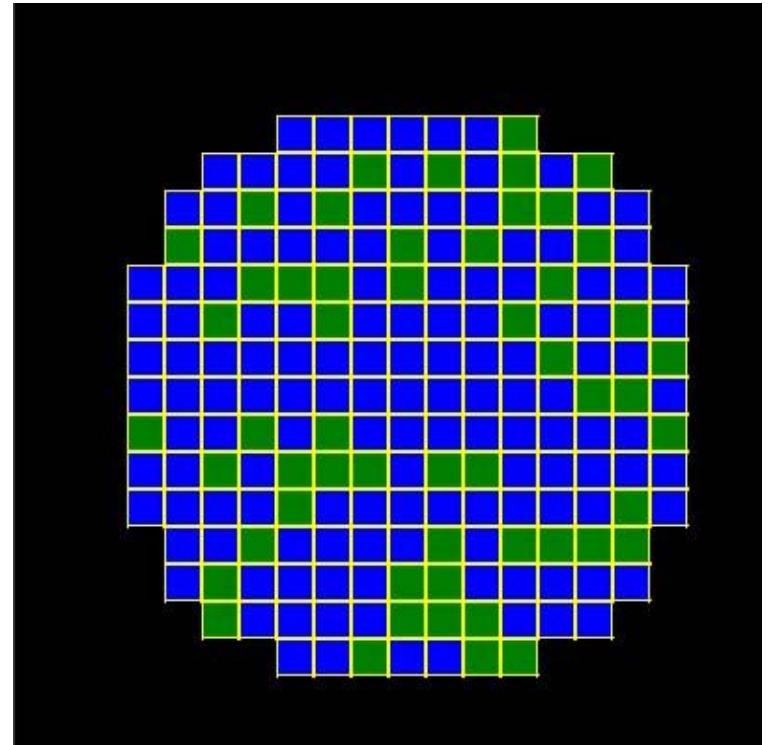
Population control using initializers

When using BlobInitializer or UniformInitializer you may list same type many times:
<Types>Condensing,NonCondensing,NonCondensing,NonCondensing</Types>

The number of cells of a given type will be proportional to the number of times a given type is listed inside the <Types> tag.

In the above example the 3/4 of cells will be NonCondensing and 1/4 will be Condensing

```
<Steppable Type="BlobInitializer">
  <Region>
    <Radius>40</Radius>
    <Center x="50" y="50" z="0"/>
    <Gap>0</Gap>
    <Width>5</Width>
    <Types>
      Condensing,
      NonCondensing,
      NonCondensing,
      NonCondensing
    </Types>
  </Region>
</Steppable>
```



Using PIFInitializer

Use PIFInitializer to create sophisticated initial conditions. PIF file allows you to **compose cells from single pixels or from larger rectangular blocks**

The syntax of the PIF file is given below:

Cell_id Cell_type x_low x_high y_low y_high z_low z_high

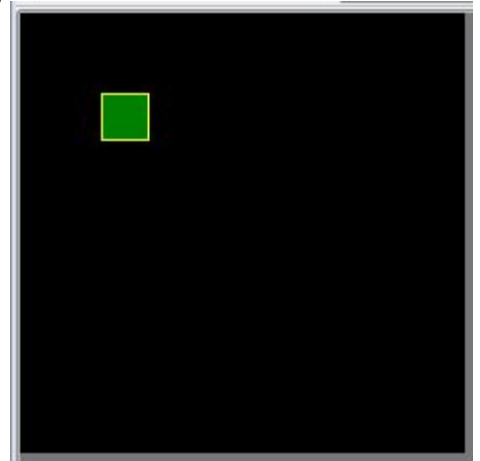
Example (file: amoebae_2D_workshop.pif):

0 amoeba 10 15 10 15 0 0

This will create rectangular cell with x-coordinates ranging from 10 to 15 (inclusive), y coordinates ranging from 10 to 15 (inclusive) and z coordinates ranging from 0 to 0 inclusive.

```
<Steppable Type="PIFInitializer">  
  <PIFName>amoebae_2D_workshop.pif</PIFName>  
</Steppable>
```

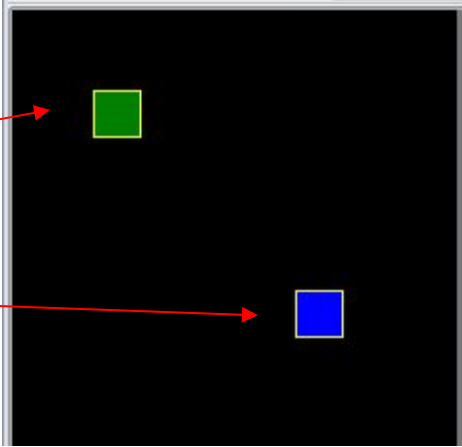
0,0



Let's add another cell:

Example (file: amoebae_2D_workshop.pif):

```
0 Amoeba 10 15 10 15 0 0
1 Bacteria 35 40 35 40 0 0
```

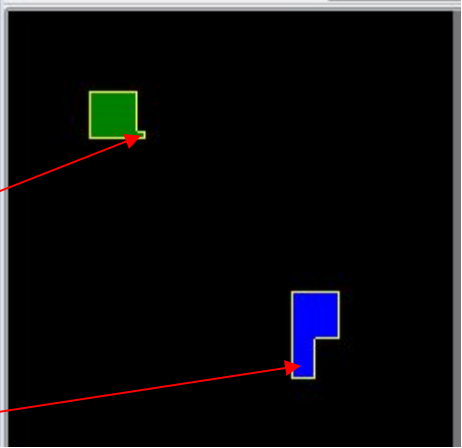


Notice that new cell has different cell_id (1) and different type (Bacterium)

Let's add pixels and blocks to the two cells from previous example:

Example (file: amoebae_2D_workshop.pif):

```
0 Amoeba 10 15 10 15 0 0
1 Bacteria 35 40 35 40 0 0
0 Amoeba 16 16 15 15 0 0
1 Bacteria 35 37 41 45 0 0
```

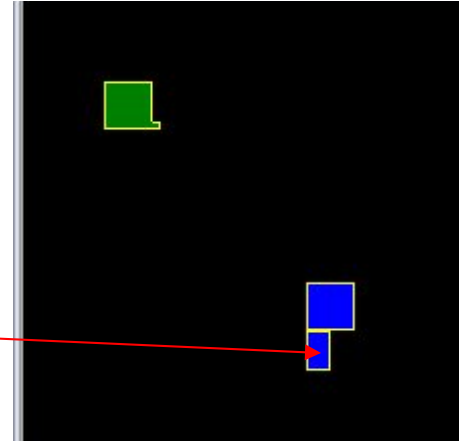


To add pixels, start new pif line with existing cell_id (0 or 1 here) and specify pixels.

This is what happens when you do not reuse cell_id

Example (file: amoebae_2D_workshop.pif):

```
0 Amoeba 10 15 10 15 0 0
1 Bacteria 35 40 35 40 0 0
0 Amoeba 16 16 15 15 0 0
2 Bacteria 35 37 41 45 0 0
```



Introducing new cell_id (2) creates new cell.

PIF files allow users to specify arbitrarily complex cell shapes and cell arrangements. The drawback is, that typing PIF file is quite tedious task and , not recommended. Typically PIF files are created using scripts.

In the future release of CompuCell3D users will be able to draw on the screen cells or regions filled with cells using GUI tools. Such graphical initialization tools will greatly simplify the process of setting up new simulations. This project has high priority on our TO DO list.

PIFDumper - yet another way to create initial condition

PIFDumper is typically used to output cell lattice every predefined number of MCS. It is useful because, you may start with rectangular cells, “round them up” by running CompuCell3D , output cell lattice using PIF dumper and reload newly created PIF file using PIFInitializer.

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

Above syntax tells CompuCell3D to store cell lattice as a PIF file every 100 MCS.

The files will be named *amoebae.100.pif* , *amoebae.200.pif* etc...

To reload file , say *amoebae.100.pif* use already familiar syntax:

```
<Steppable Type="PIFInitializer">  
  <PIFName>amoebae.100.pif</PIFName>  
</Steppable>
```

PIFTracer and other PIF Generators

PIFTracer (works only on OSX) allows users to “paint” cells using experimental pictures as a template. Currently it does not support PIF format for compartmental cells but we developed short Python script that provides temporary fix.

We are working on another tool that can generate PIF file based on colors of the underlying gif image. That is the color of the cells in the image determines cell type.

Chemotaxis

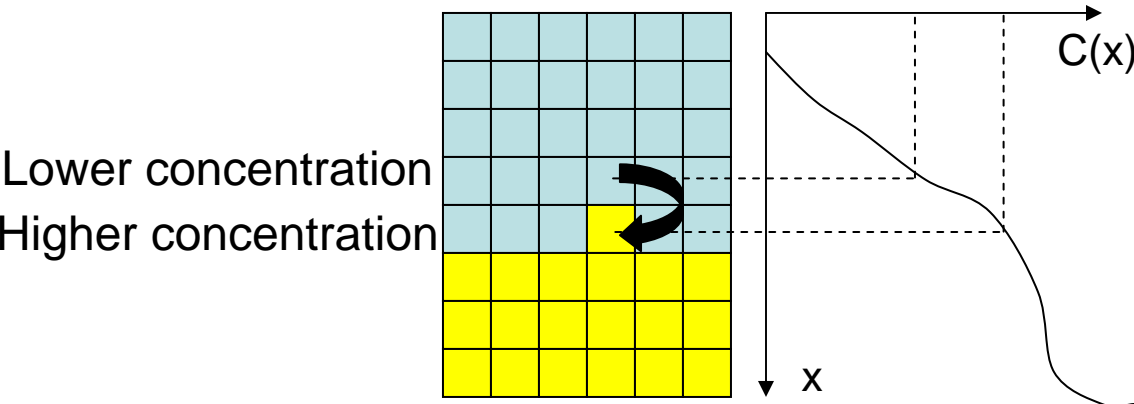
Basic facts

- Chemotaxis is defined as cell motion induced by a presence (gradient) of a chemical.
- In GGH formalism chemotaxis is implemented as a spin copy bias which depends on chemical gradient.
- Chemotaxis was first introduced to GGH formalism by Paulien Hogeweg from University of Utrecht, Netherlands
- In CompuCell3D Chemotaxis plugin provides wide range of options to support different modes of chemotaxis.
- Chemotaxis plugin requires the presence of at least one concentration field. The fields can be inserted into CompuCell3D simulation by means PDE solvers or can be created, initialized and managed explicitly from the Python level

Chemotaxis Term – Most Basic Form

$$\Delta E_{chem} = -\lambda(c(x_{destination}) - c(x_{source}))$$

If concentration at the spin-copy destination pixel ($c(x_{destination})$) is higher than concentration at the spin-copy source ($c(x_{source})$) AND λ is positive then ΔE is negative and such spin copy will be accepted. The cell chemotacts up the concentration gradient



Chemorepulsion can be obtained by making λ negative

Alternative Formulas For Chemotaxis Energy

$$\Delta E_{chem} = -\lambda \left(\frac{c(x_{destination})}{a + c(x_{destination})} - \frac{c(x_{source})}{a + c(x_{source})} \right)$$

$$\Delta E_{chem} = -\lambda \left(\frac{c(x_{destination})}{a \cdot c(x_{destination}) + 1} - \frac{c(x_{source})}{a \cdot c(x_{source}) + 1} \right)$$

Chemotaxis - XML Examples

$$\Delta E_{chem} = -\lambda(c(x_{destination}) - c(x_{source}))$$


```
<Plugin Name="Chemotaxis">  
  <ChemicalField Source="FlexibleDiffusionSolverFE" Name="FGF">  
    <ChemotaxisByType Type="Amoeba" Lambda="300"/>  
    <ChemotaxisByType Type="Bacteria" Lambda="200"/>  
  </ChemicalField>  
  <ChemicalField Source="FlexibleDiffusionSolverFE" Name="FGF4">  
    <ChemotaxisByType Type="Amoeba" Lambda="-300"/>  
  </ChemicalField>  
</Plugin>
```

Notice , that different cell types may have different chemotactic properties. For more than 1 chemical fields the change of chemotaxis energy expression is given below:

$$\Delta E_{chem} = \sum_{i-field} -\lambda_i(c_i(x_{destination}) - c_i(x_{source}))$$

Chemotaxis - XML Examples continued

$$\Delta E_{chem} = -\lambda(c(x_{destination}) - c(x_{source}))$$

```
<Plugin Name="Chemotaxis">  
  <ChemicalField Source="FlexibleDiffusionSolverFE" Name="FGF">  
    <ChemotaxisByType Type="Amoeba" Lambda="300"/>  
    <ChemotaxisByType Type="Bacteria" Lambda="200"/>  
  </ChemicalField>  
  <ChemicalField Source="FlexibleDiffusionSolverFE" Name="FGF4">  
    <ChemotaxisByType Type="Amoeba" Lambda="-300" SaturationCoef="2.0"/>  
  </ChemicalField>  
</Plugin>
```

$$\Delta E_{chem} = -\lambda \left(\frac{c(x_{destination})}{a + c(x_{destination})} - \frac{c(x_{source})}{a + c(x_{source})} \right)$$

Chemotaxis - XML Examples continued

$$\Delta E_{chem} = -\lambda(c(x_{destination}) - c(x_{source}))$$

```

<Plugin Name="Chemotaxis">
  <ChemicalField Source="FlexibleDiffusionSolverFE" Name="FGF">
    <ChemotaxisByType Type="Amoeba" Lambda="300"/>
    <ChemotaxisByType Type="Bacteria" Lambda="200"/>
  </ChemicalField>
  <ChemicalField Source="FlexibleDiffusionSolverFE" Name="FGF4">
    <ChemotaxisByType Type="Amoeba" Lambda="-300" SaturationLinearCoef="2.0"/>
  </ChemicalField>
</Plugin>

```

$$\Delta E_{chem} = -\lambda \left(\frac{c(x_{destination})}{a \cdot c(x_{destination}) + 1} - \frac{c(x_{source})}{a \cdot c(x_{source}) + 1} \right)$$



PDE Solvers

- CompuCell3D has built-in diffusion , reaction diffusion and advection diffusion PDE solvers. Those are, probably most frequently used solver in GGH modeling.
- CompuCell3D uses explicit (**unstable but fast**) method to solve the PDE. Constantly changing boundary conditions practically rule out more robust, but slow implicit solvers.
- Because of instability users should make sure that their PDE parameters do not produce wrong results (which could manifest themselves as “rough” concentration profiles, “insane” concentration values, NaN’s - Not A Number etc...). Future release of CompuCell3D will provide tools to detect potential PDE instabilities.
- Additional solvers can be implemented directly in C++ or using BioLogo. BioLogo is especially attractive because it takes as an input human readable PDE description and generates fast C++ code.
- Typically a concentration from the PDE solver is read by other CompuCell3D modules to adjust cell properties. Currently the best way of dealing with this is through Python interface.

Flexible Diffusion Solver

```
<Steppable Type="FlexibleDiffusionSolverFE">
  <DiffusionField>
    <DiffusionData>
      <FieldName>FGF</FieldName>
      <DiffusionConstant>0.010</DiffusionConstant>
      <DecayConstant>0.000</DecayConstant>
      <ConcentrationFileName>diffusion_2D.pulse.txt</ConcentrationFileName>
    </DiffusionData>
  </DiffusionField>
</Steppable>
```

Define diffusion field

Define diffusion parameters

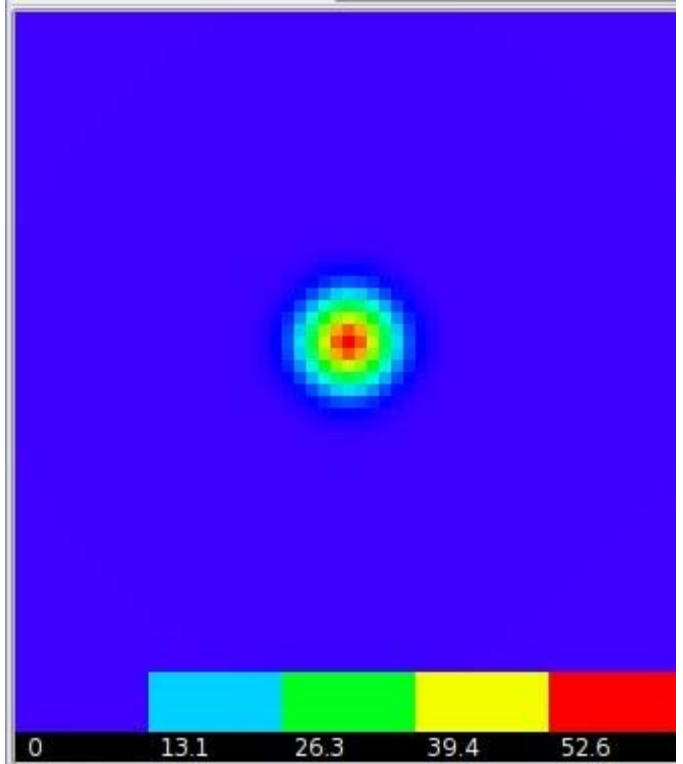
Read-in initial condition

Initial Condition File Format:

x y z concentration

Example:

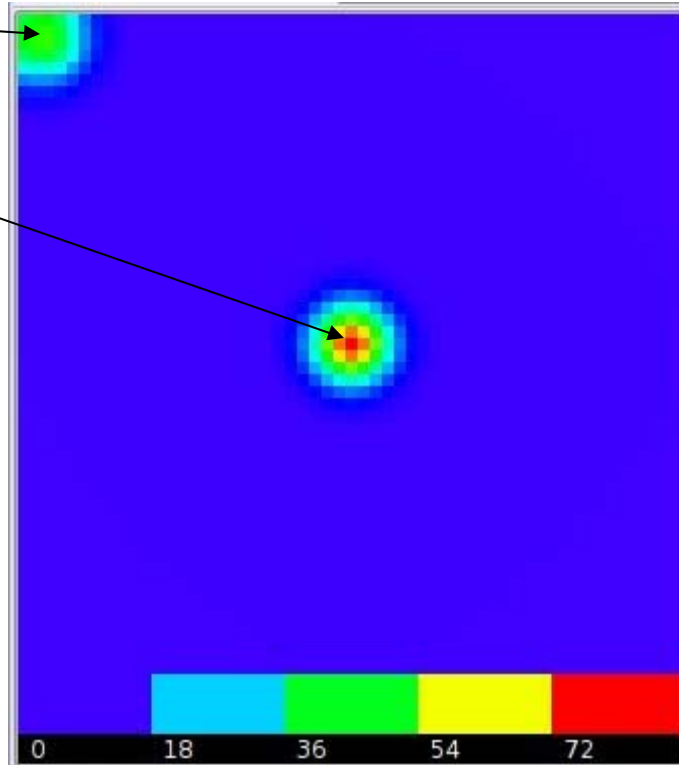
```
27 27 0 2000.0
45 45 0 0.0 ...
```



Two-pulse initial condition

Initial condition (`diffusion_2D.pulse.txt`):

5 5 0 1000.0
27 27 0 2000.0



```
<Steppable Type="FlexibleDiffusionSolverFE">
```

```
<DiffusionField>
```

```
<DiffusionData>
```

```
<FieldName>FGF</FieldName>
```

```
<DiffusionConstant>0.010</DiffusionConstant>
```

```
<DecayConstant>0.000</DecayConstant>
```

```
<DoNotDiffuseTo>Medium</DoNotDiffuseTo>
```

```
<ConcentrationFileName>diffusion_2D.pulse.txt</ConcentrationFileName>
```

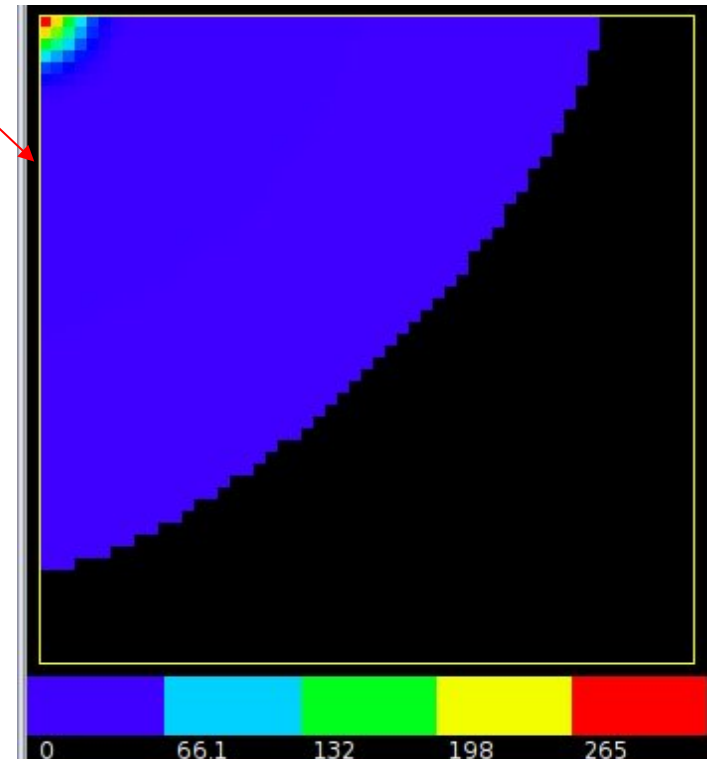
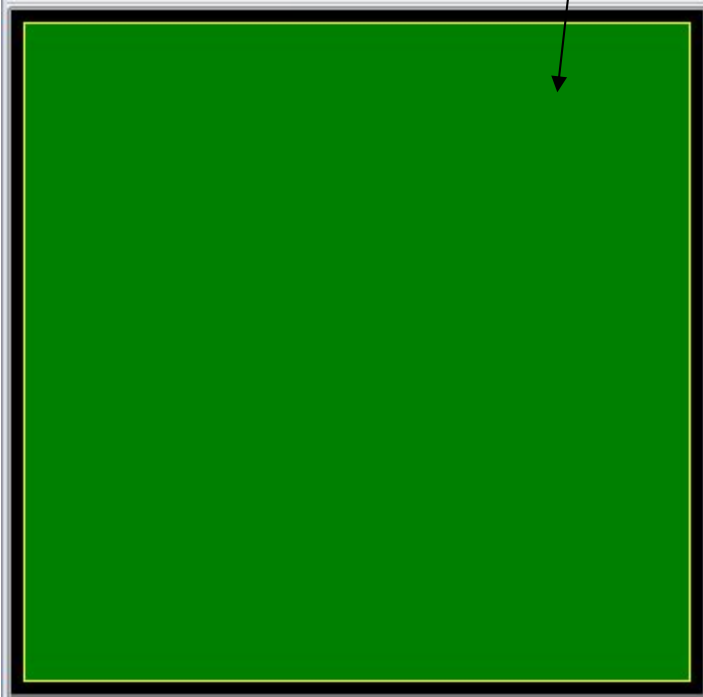
```
</DiffusionData>
```

```
</DiffusionField>
```

```
</Steppable>
```

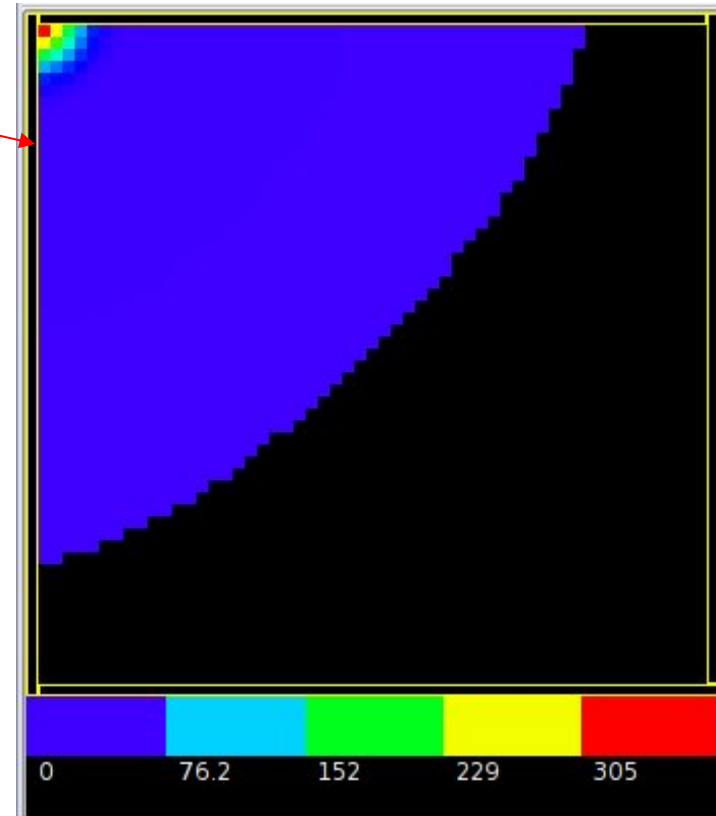
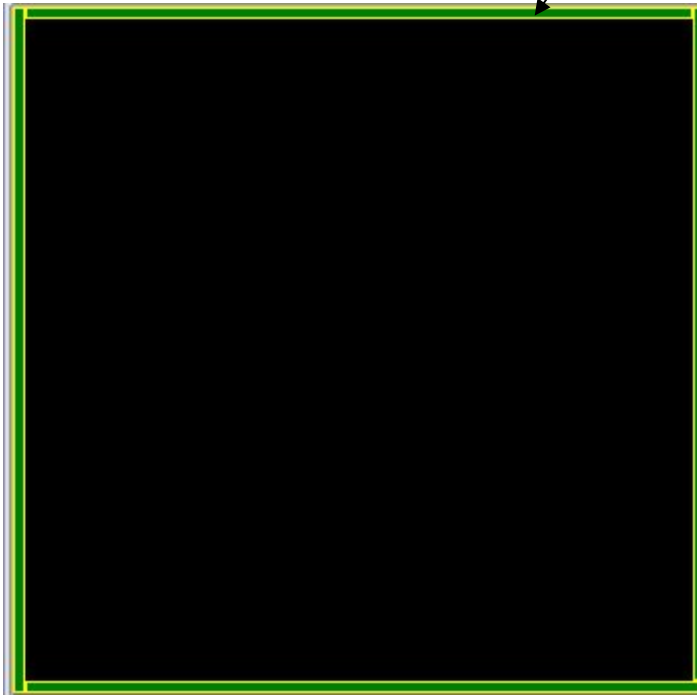
You may specify diffusion regions

FGF will diffuse inside big cell and will not go to Medium



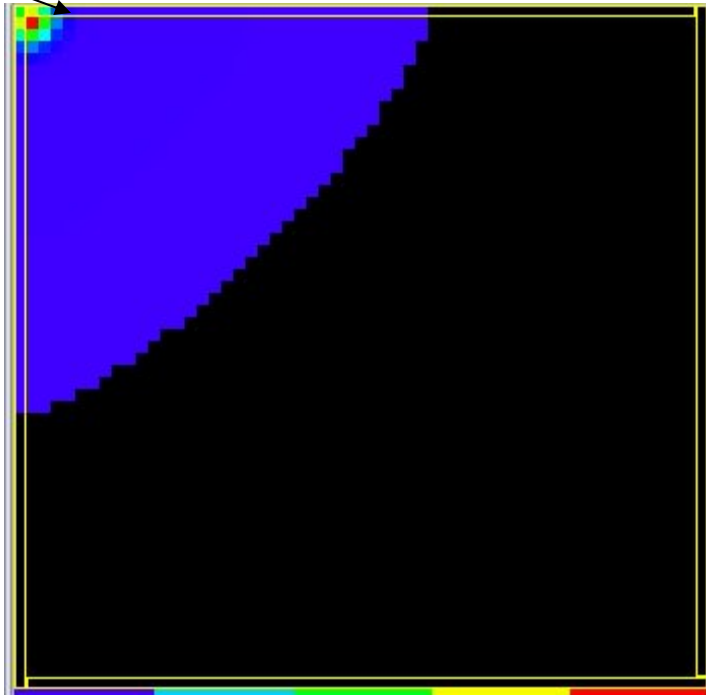
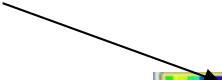

```
<Steppable Type="FlexibleDiffusionSolverFE">
  <DiffusionField>
    <DiffusionData>
      <FieldName>FGF</FieldName>
      <DiffusionConstant>0.010</DiffusionConstant>
      <DecayConstant>0.000</DecayConstant>
      <DoNotDiffuseTo>Wall</DoNotDiffuseTo>
      <ConcentrationFileName>diffusion_2D_wall.pulse.txt</ConcentrationFileName>
    </DiffusionData>
  </DiffusionField>
</Steppable>
```

FGF will not diffuse to the Wall



```
<Steppable Type="FlexibleDiffusionSolverFE">
  <DiffusionField>
    <DiffusionData>
      <FieldName>FGF</FieldName>
      <DiffusionConstant>0.010</DiffusionConstant>
      <DecayConstant>0.000</DecayConstant>
      <!--DoNotDiffuseTo>Wall</DoNotDiffuseTo-->
      <ConcentrationFileName>diffusion_2D_wall.pulse.txt</ConcentrationFileName>
    </DiffusionData>
  </DiffusionField>
</Steppable>
```

Now FGF diffuses everywhere



PDE Solver Caller Plugin

By default PDE solver is called once per MCS. You may call it more often, say 3 times per MCS by including PDESolverCaller plugin:

```
<Plugin Name="PDESolverCaller">  
  <CallPDE PDESolverName="FlexibleDiffusionSolverFE" ExtraTimesPerMC="2"/>  
</Plugin>
```

Notice, that you may include multiple **CallPDE** tags to call different PDESolvers with different frequencies.

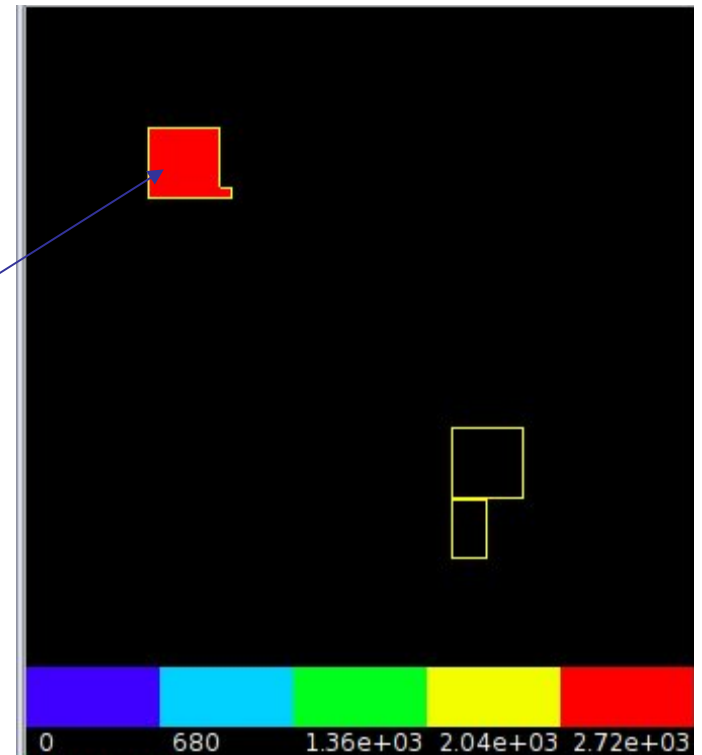
You typically use this plugin to avoid numerical instabilities when working with large diffusion constants

Secretion

CompuCell3D offers several modes for including secretion in your simulations. Let's look at concrete examples:

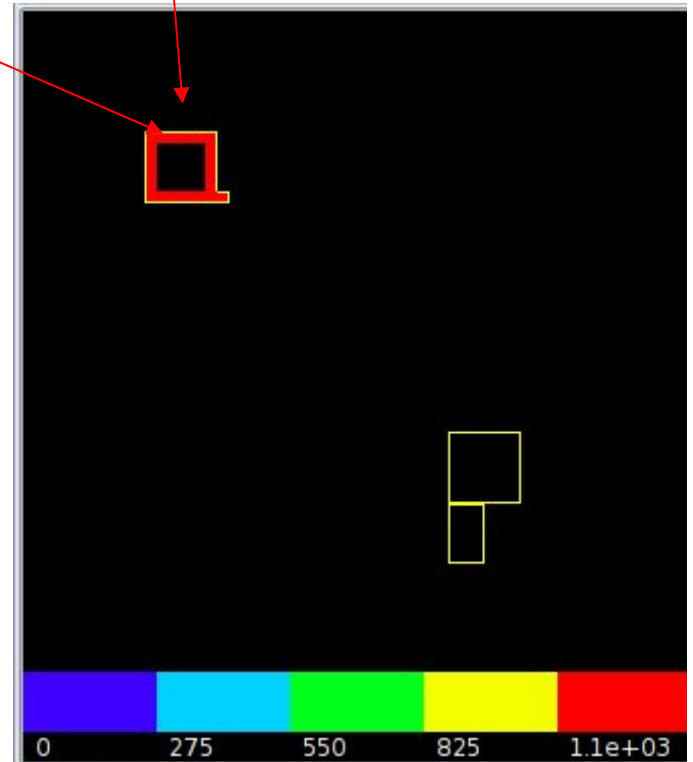
```
<Steppable Type="FlexibleDiffusionSolverFE">
  <DiffusionField>
    <DiffusionData>
      <FieldName>FGF</FieldName>
      <DiffusionConstant>0.000</DiffusionConstant>
      <DecayConstant>0.000</DecayConstant>
    </DiffusionData>
    <SecretionData>
      <Secretion Type="Amoeba">20</Secretion>
    </SecretionData>
  </DiffusionField>
</Steppable>
```

We turned diffusion off and have cells of type Amoeba secrete FGF. Secretion takes place at every pixel belonging to Amoeba cells. At each MCS we increase the value of the concentration at those pixels by 20 units.



```
<Steppable Type="FlexibleDiffusionSolverFE">
  <DiffusionField>
    <DiffusionData>
      <FieldName>FGF</FieldName>
      <DiffusionConstant>0.000</DiffusionConstant>
      <DecayConstant>0.000</DecayConstant>
    </DiffusionData>
    <SecretionData>
      <SecretionOnContact Type="Amoeba" SecreteOnContactWith="Medium">20.1</SecretionOnContact>
    </SecretionData>
  </DiffusionField>
</Steppable>
```

Secretion will take place in those pixels belonging to Amoeba cells that have contact with Medium

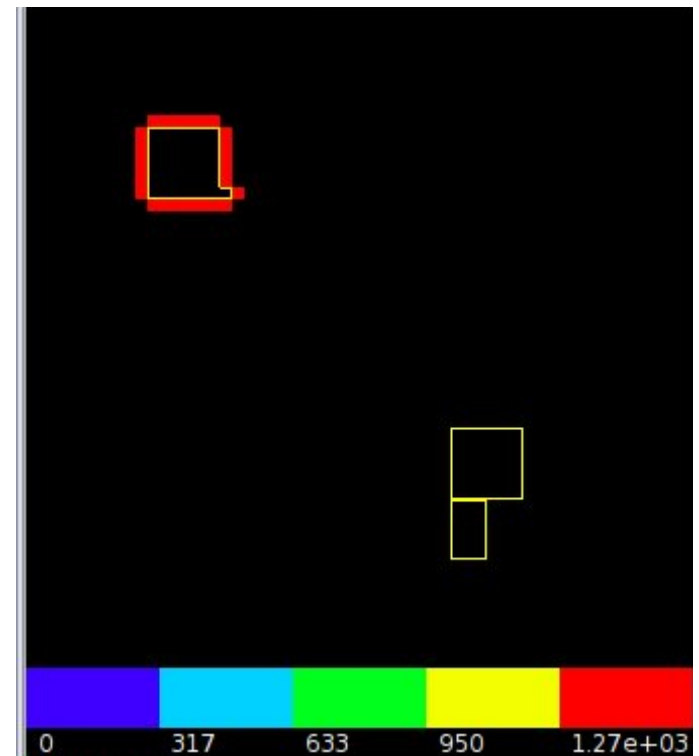


```

<Steppable Type="FlexibleDiffusionSolverFE">
  <DiffusionField>
    <DiffusionData>
      <FieldName>FGF</FieldName>
      <DiffusionConstant>0.000</DiffusionConstant>
      <DecayConstant>0.000</DecayConstant>
    </DiffusionData>
    <SecretionData>
      <SecretionOnContact Type="Medium" SecreteOnContactWith="Amoeba">20.1</SecretionOnContact>
    </SecretionData>
  </DiffusionField>
</Steppable>

```

Secretion will take place in those pixels belonging to Medium cells that have contact with Amoeba

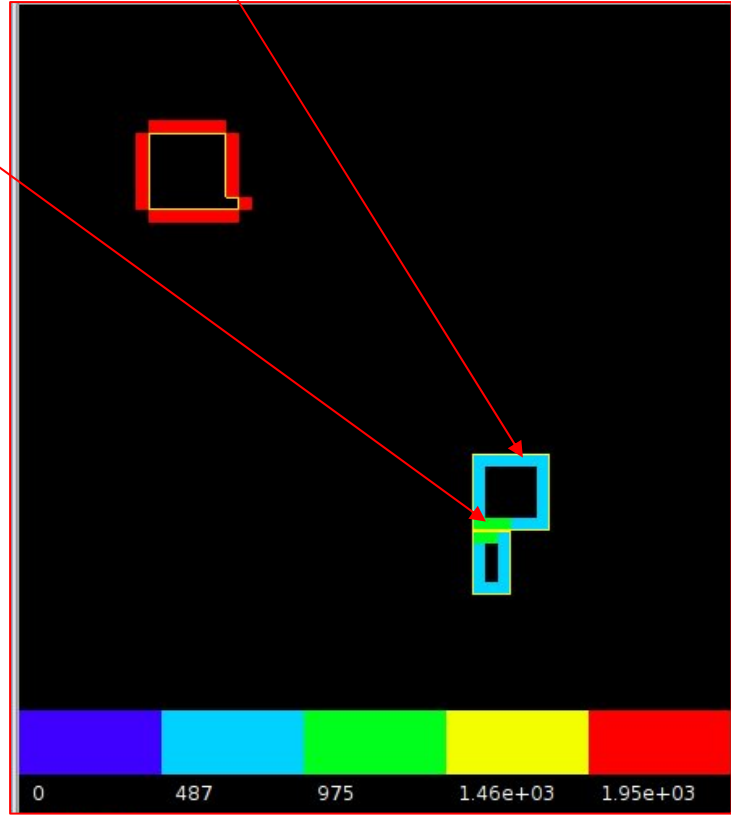


```

<Steppable Type="FlexibleDiffusionSolverFE">
  <DiffusionField>
    <DiffusionData>
      <FieldName>FGF</FieldName>
      <DiffusionConstant>0.000</DiffusionConstant>
      <DecayConstant>0.000</DecayConstant>
    </DiffusionData>
    <SecretionData>
      <SecretionOnContact Type="Medium" SecreteOnContactWith="Amoeba">20.1</SecretionOnContact>
      <SecretionOnContact Type="Bacteria" SecreteOnContactWith="Bacteria">10.1</SecretionOnContact>
      <SecretionOnContact Type="Bacteria" SecreteOnContactWith="Medium">5.1</SecretionOnContact>
    </SecretionData>
  </DiffusionField>
</Steppable>

```

1. Secretion will take place in those pixels belonging to Medium cells that have contact with Amoeba.
2. There will be secretion in pixels of Bacteria cells that have contact with medium.
3. Secretion will also take place in those pixels of bacteria cells that have contact with other bacteria cells



Reaction-Diffusion set of PDE's

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

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

⋮

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

Solving general set of above PDE's can be tricky because functions 'f' can have arbitrary form. There are two ways to deal with this problem:

1. For each set of PDE's write new PDE solver. This is not a bad idea if you can do it "on the fly". If you can write a code that automatically generates and compiles PDE solver you will see no performance degradation
2. Use fast math expression parser that will interpret mathematical expressions during run time

CompuCell3D 3.4.1 uses the second solution. The reason was that it was the simplest to implement and also one does not have to bother about compilers installed on users machines. However such PDE solver will not be as fast as the compiled one

Let's consider a simple example

$$\frac{\partial F}{\partial t} = 0.01\nabla^2 F + F - F^3/3 + 0.3 - H$$

$$\frac{\partial H}{\partial t} = 0.01\nabla^2 H + 0.08F - 0.064H + 0.056$$

```
<Steppable Type="ReactionDiffusionSolverFE">
```

```
<DiffusionField>
```

```
<DiffusionData>
```

```
<FieldName>F</FieldName>
```

```
<DiffusionConstant>0.01</DiffusionConstant>
```

```
<ConcentrationFileName>Demos/diffusion/FN.pulse.txt</ConcentrationFileName>
```

```
<AdditionalTerm>F-F*F*F/3+0.3-H</AdditionalTerm>
```

```
</DiffusionData>
```

```
</DiffusionField>
```

```
<DiffusionField>
```

```
<DiffusionData>
```

```
<FieldName>H</FieldName>
```

```
<DiffusionConstant>0.01</DiffusionConstant>
```

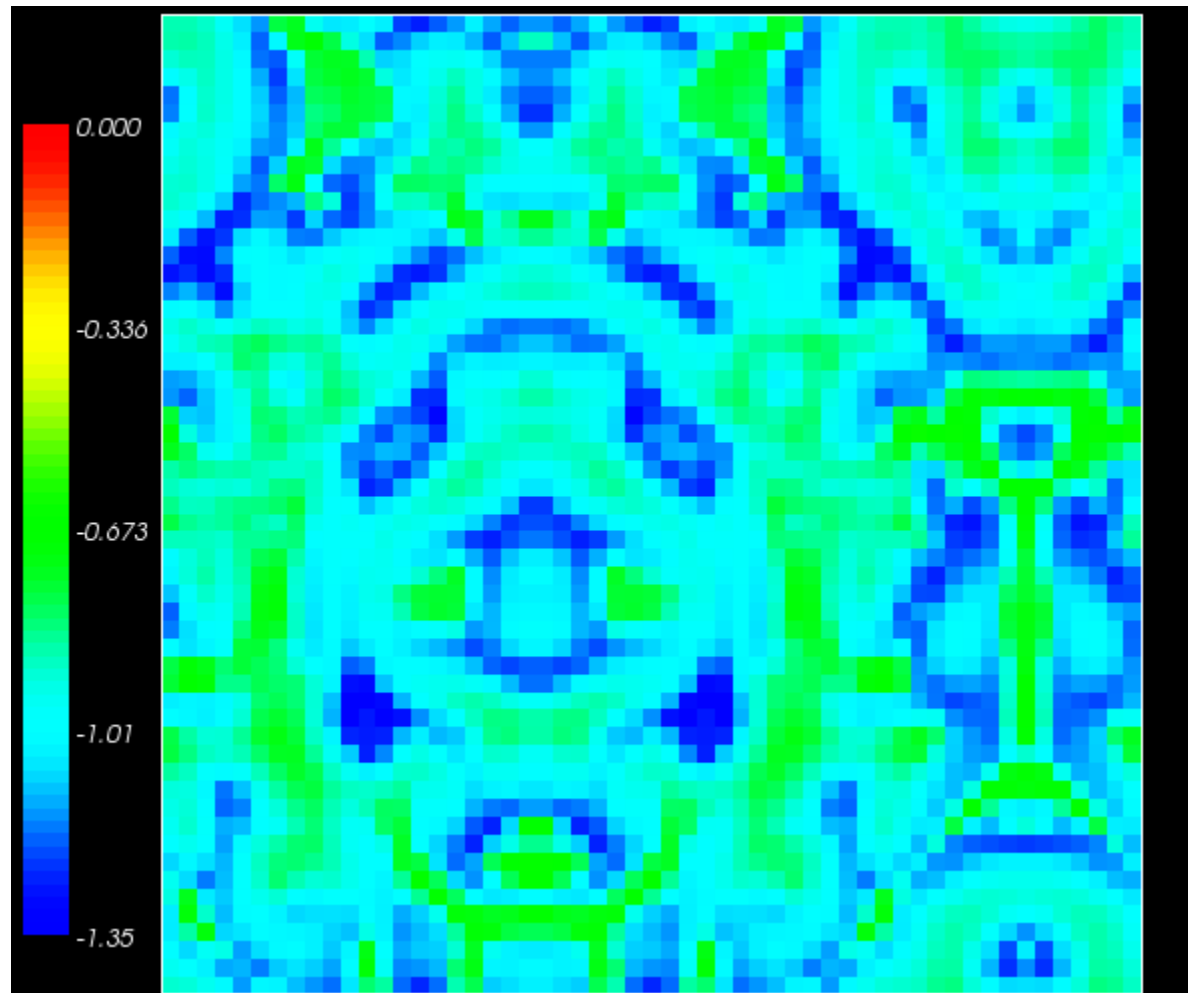
```
<AdditionalTerm>0.08*F-0.064*H+0.056</AdditionalTerm>
```

```
</DiffusionData>
```

```
</DiffusionField>
```

```
</Steppable>
```

Functions of F and H are coded using quite naturally looking syntax. The output of the above simulation with periodic boundary conditions may look like



It is quite interesting that the slowdown due to interpreting user defined functions is very small.

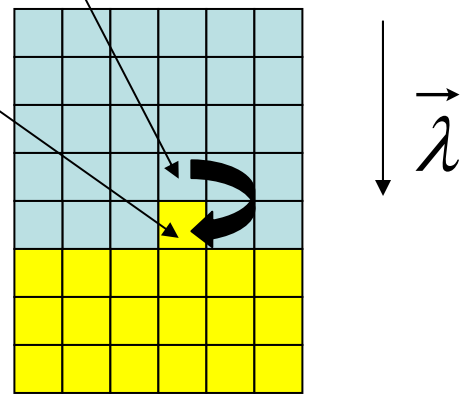
Imposing Directed Motion of Cells

One can impose artificial spin flip bias that would have an effect of moving cell in the direction OPPOSITE to Lambda vector specified below. The magnitude of the lambda vector determines the “amount” of spin copy bias.

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

$$\Delta E_{external_potential} = -\vec{\lambda} \cdot (\vec{x}_{destination} - \vec{x}_{source})$$

ΔE will be negative (favoring spin copy) →



Connectivity Plugin

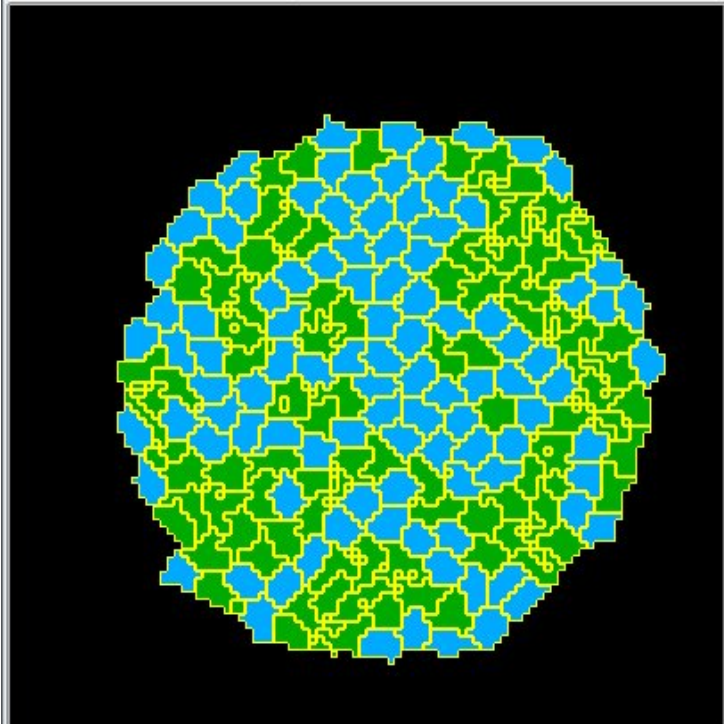
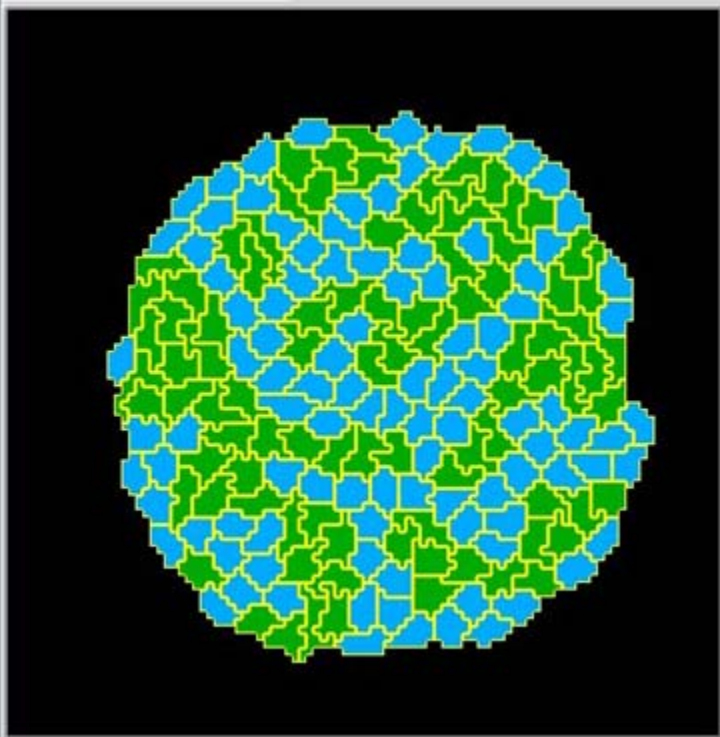
Connectivity plugin ensures that **2D cells** are not fragmented and are simply connected. It decreases probability of certain spin flips which can break connectedness of a cell. Users can specify energy penalty that will be incurred if the spin copy is to break connectedness of the cell.:

Syntax:

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

Note: this plugin will not work properly with hexagonal lattice

Cell sorting simulation with and without connectivity plugin



Length Constraint Plugin

Length constraint plugin is used to force cells to keep preferred length along cell's longest axis (we assume that cells have elliptical shape):

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

```
  <LengthEnergyParameters TargetLength="15" LambdaLength="2.0"/>
```

```
</Plugin>
```

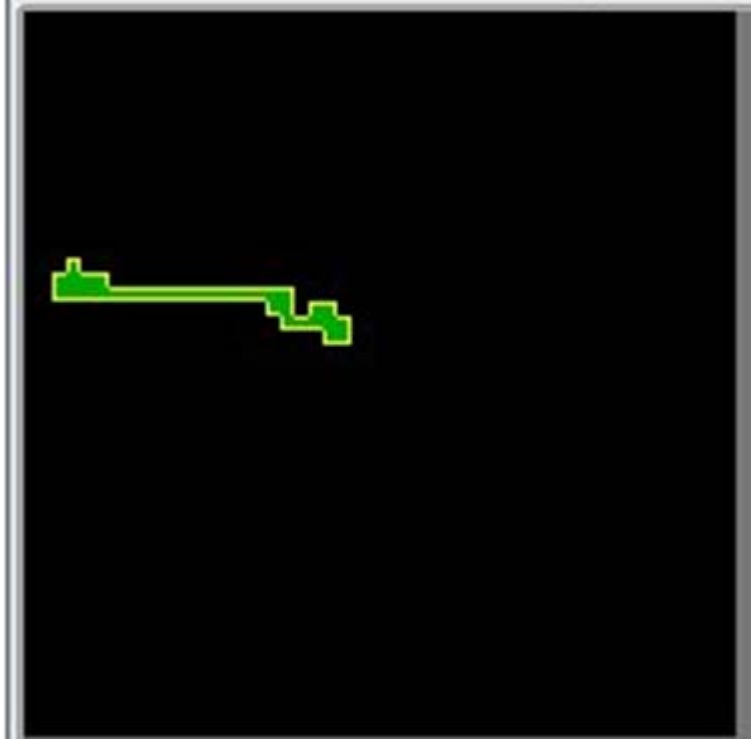
The LambdaLength and TargetLength play similar role to LambdaVolume and TargetVolume from Volume Plugin.

IMPORTANT: Length Constraint Plugin has to be used together with connectivity plugin or else cells might become fragmented. The applicability of the LengthConstraint and Connectivity Plugins is limited to 2D simulations.

For more information see

“Cell elongation is key to in silico replication of in vitro vasculogenesis and subsequent remodeling” by **Roeland M.H. Merks** *et al* *Developmental Biology* 289 (2006) 44– 54

Length constraint plugin at work



Note: this plugin will not work properly with hexagonal lattice