COMPUTE CANCER

All about models and implementation techniques

Tag: cellular automaton

Some tips&tricks for 3D tumor visualization

29 APRIL 2016

JPOLESZCZUK

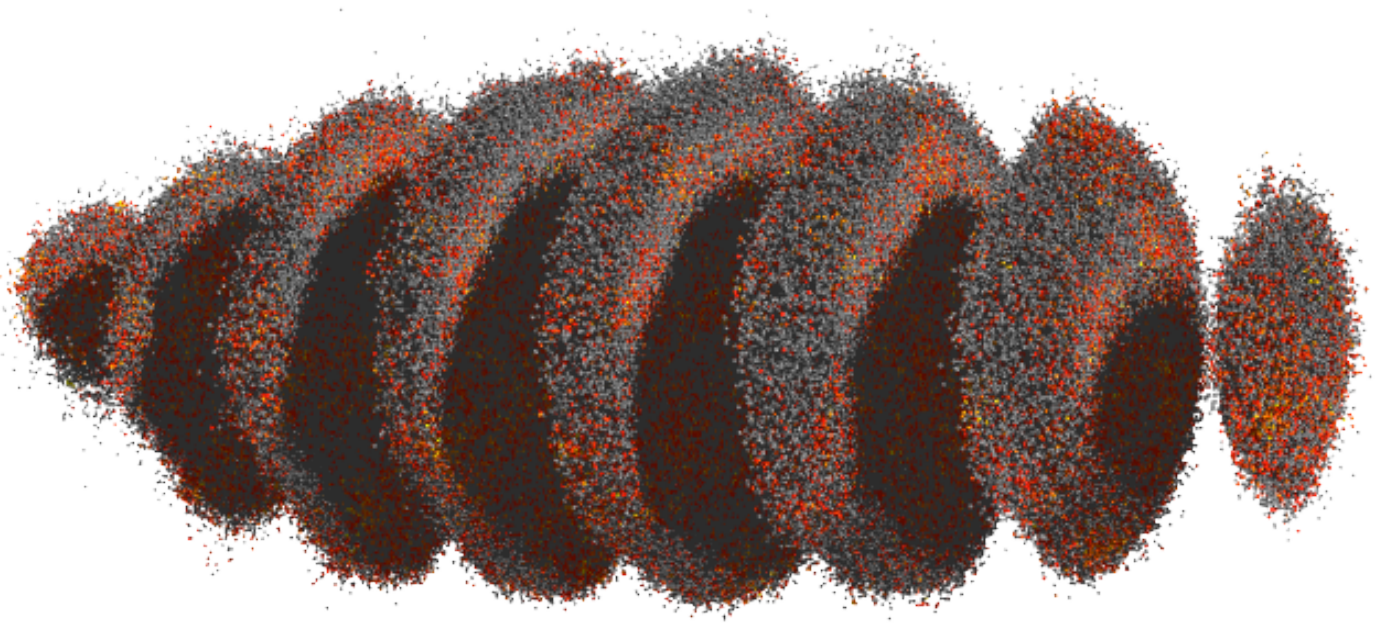
1 COMMENT

There are several reasons why we rarely see 3D agent based models of tumor growth, with I think the two most important ones being: 1) 3D simulations are way more computationally expensive than 2D simulations; 2) it's harder to visualize results of 3D simulations to present all necessary information. The other frequently used argument is that there is no need for 3D, because the crucial phenomena can be observed as well in 2D. However, when one considers e.g. stochastic models there might be a huge difference in the system behavior between 2D and 3D case (see very nice post Why is that a 2D random walk is recurrent, while a 3D random walk is transient? (https://www.quora.com/Why-is-it-that-a-2D-random-walk-is-recurrent-while-a-3D-random-walk-is-transient)).

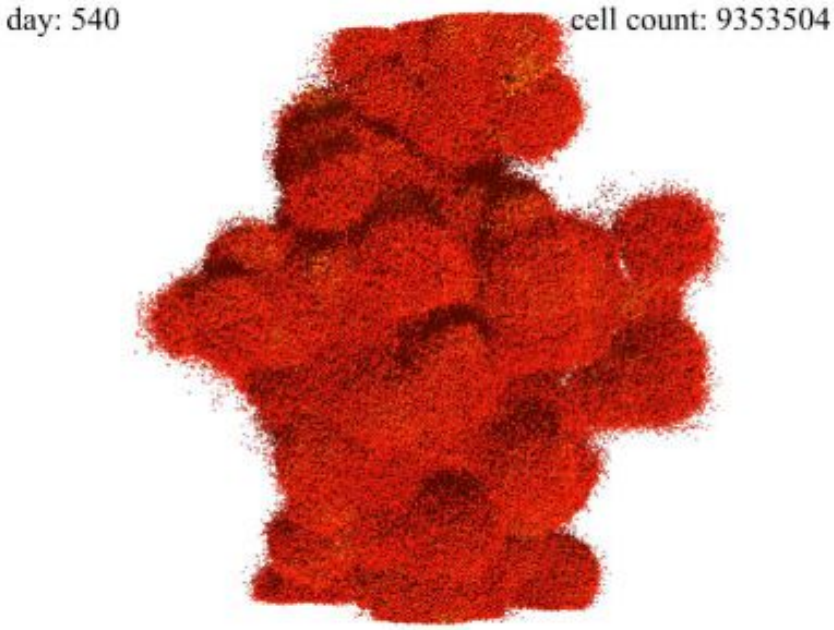
Possible improvements in the computational speed are highly dependent on the given model setting and typically require sophisticated techniques. Problem with nice visualizations, however, just requires proper tools. Today, I will show my attempts to make a good-looking 3D tumor visualizations, which I have taken over the last several years.

Several years ago when working with @heikman (https://twitter.com/theheikman) on the project about the impact of cellular senescence I was using The Persistence of Vision Raytracer (POV-Ray) (http://www.povray.org) in order to obtain high quality 3D tumor visualization. After simulating the tumor up to a given cell number I was parsing the information about cells locations and their basic

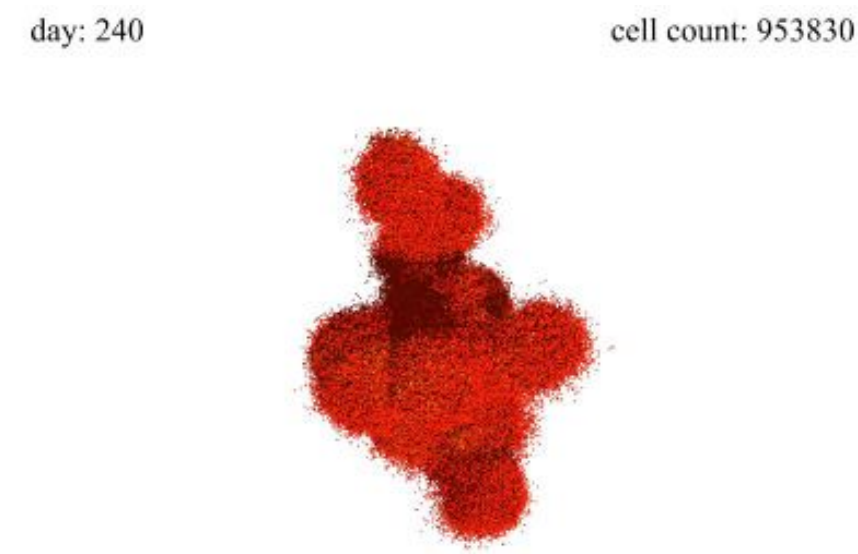
properties into a POV-Ray specific format. In my approach, each cell was separate actor in the scene with possible different surface/texture/color properties. As it can be expected renderings with raytracing software are very good (see Fig.1 below), but the whole procedure to prepare input files to POV-Ray, to set-up properly camera and light was really time-consuming. In addition, rendering takes a long time, so there is no chance for “live” rendering during the simulation, which would allow to observe simulation behavior.



(https://computecancer.wordpress.com/2016/04/29/some-tipstricks-for-3d-tumor-visualization/senescence1_1_slice_final/)



(https://computecancer.wordpress.com/2016/04/29/some-tipstricks-for-3d-tumor-visualization/no_senescence_p15_10_17/)



(https://computecancer.wordpress.com/2016/04/29/some-tipstricks-for-3d-tumor-visualization/no_senescence_p15_10_7/)

Figure 1. Exemplary 3D tumor renderings obtained using POV-Ray software (http://www.povray.org). Bottom-left picture shows tumor consisted of 9.4 million of cells.

I really wanted to have a tool that would allow me to seamlessly simulate and visualize 3D tumor at the same time. I quickly realized that rendering each cell as a separate object is not a good approach, when you simulate more than 1 million cells. Hence, I started looking around for an algorithm that would allow to extract tumor surface from the cloud of cells. I quickly found well known Marching Cubes algorithm (https://en.wikipedia.org/wiki/Marching_cubes) that does the trick very well. I've implemented the algorithm and visualized resulting surface using Visualization Library (http://visualizationlibrary.org/)in C++. I was quite happy with the results (see Fig. 2 below), however, at some point code became so complicated and so model specific that I assumed that it doesn't have any future.

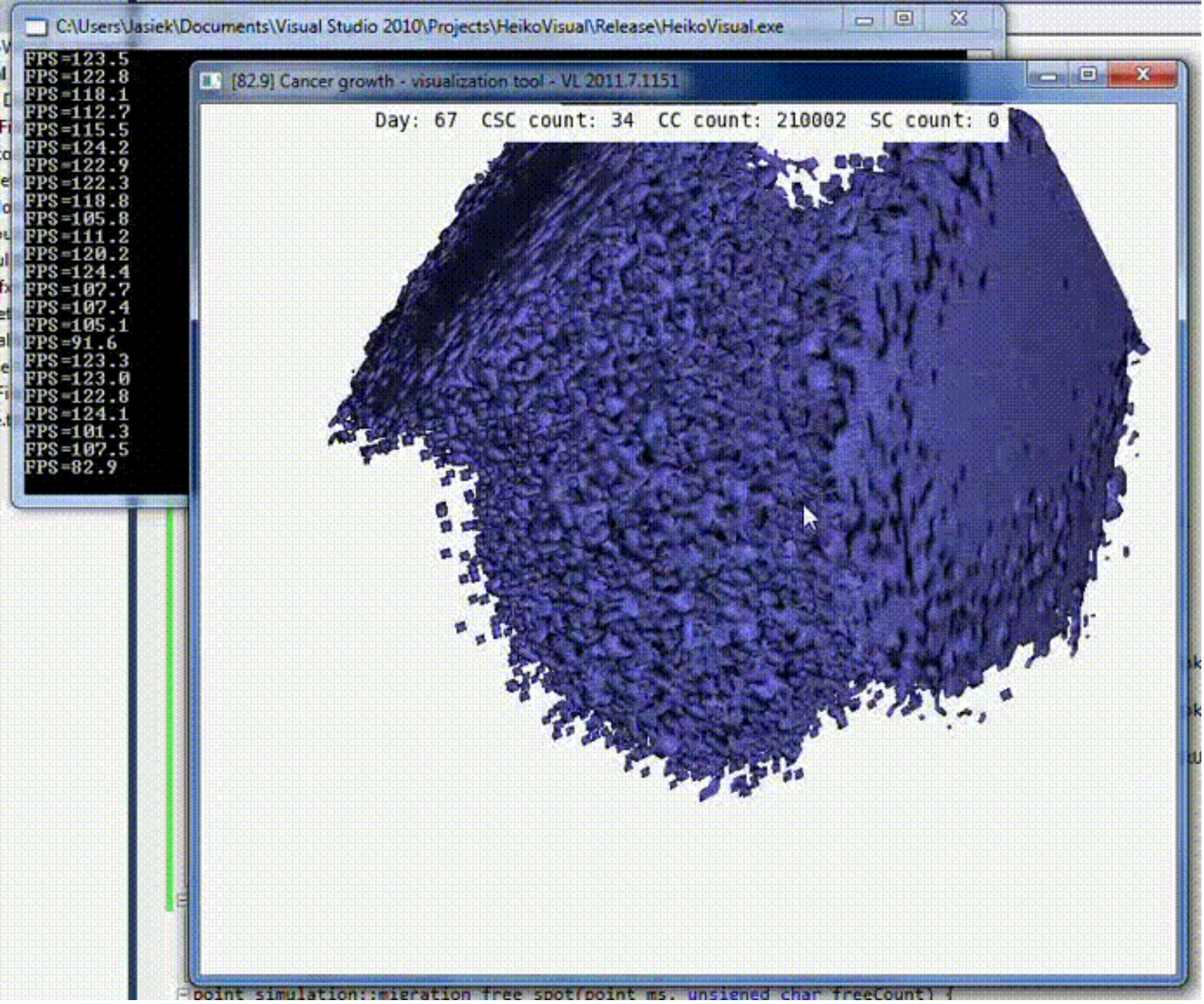


Figure 2. "Live" 3D tumor visualization during the simulation. Code developed in C++ using Visualization Library. (http://visualizationlibrary.org/)

Quite recently I started using VTK library (http://www.vtk.org) from within C++ (you can also use it form within Python (http://www.bu.edu/tech/support/research/training-consulting/online-tutorials/vtk/)) and I quickly realized that it might be the ultimate tool for 3D visualization. Now I will show how easy it is to produce great visualizations that can be used in "live" renderings during 3D simulations using VTK library.

In the following example I will use glyph mapper that basically takes my cloud of points (cells), creates cubes around them, and extracts the surface. First, let us include all necessary headers and define necessary variables.

```
1  #include "vtkCubeSource.h"
2  #include "vtkPolyDataMapper.h"
3  #include "vtkRenderer.h"
4  #include "vtkRenderWindow.h"
```

```
#include "vtkRenderWindowInteractor.h"
#include "vtkInteractorStyleTrackballCamera.h"
#include "vtkPolyData.h"
#include "vtkGlyph3DMapper.h"

vtkSmartPointer<vtkRenderer> renderer;
vtkSmartPointer<vtkRenderWindow> renderWindow;
vtkSmartPointer<vtkRenderWindowInteractor> renderWindowInteractor;
vtkSmartPointer<vtkPoints> points;
```

Now we can write the function that will initialize the renderer and rendering window together with its interactor object.

```
void initializeRendering() {
    renderer = vtkSmartPointer<vtkRenderer>::New(); //creating new re
    renderer->SetBackground(1., 1., 1.); //setting background color t

    renderWindow = vtkSmartPointer<vtkRenderWindow>::New(); //creati
    renderWindow->SetSize(1000, 1000); //setting size of the window
    renderWindow->AddRenderer(renderer); //adding previously created

    renderWindowInteractor = vtkSmartPointer<vtkRenderWindowInteracto
    renderWindowInteractor->SetRenderWindow(renderWindow); //adding p

    points = vtkSmartPointer<vtkPoints>::New();          //vector with
}
```

I assume that we have our cell objects are stored in STL vector structure and that each cell contains information about its (x,y,z) position in 3D space. Having that we can write a really short function that adds our cells to the scene.

```
void addCells(vector<Cell>::iterator begin, vector<Cell>::iterator en
    vector<Cell>::iterator cell = begin;
    for (cell; cell<end; cell++)
points->InsertNextPoint(cell->x,cell->y,cell->z);
}
```

Finally, we can write the rendering function that is invoked in the main part of the code.

```
void visualizeLesion(vector<Cell>::iterator begin, vector<Cell>::iter

    initializeRendering();

    addCells(begin, end);

    // Combine into a polydata
    vtkSmartPointer<vtkPolyData> polydata = vtkSmartPointer<vtkPolyDa
    polydata->SetPoints(points);

    vtkSmartPointer<vtkCubeSource> cubeSource = vtkSmartPointer<vtkCu

    vtkSmartPointer<vtkGlyph3D> glyph3D = vtkSmartPointer<vtkGlyph3D>
    glyph3D->SetSourceConnection(cubeSource->GetOutputPort());
```

```
44      glyph3D->ScalingOff();
45      glyph3D->SetInputData(polydata);
46      glyph3D->Update();
47
48      // Create a mapper and actor
49      vtkSmartPointer<vtkPolyDataMapper> mapper = vtkSmartPointer<vtkPo
50      mapper->SetInputConnection(glyph3D->GetOutputPort());
51
52      vtkSmartPointer<vtkActor> actor = vtkSmartPointer<vtkActor>::New
53      actor->SetMapper(mapper);
54
55      // Add the actor to the scene
56      renderer->AddActor(actor);
57
58      // Render
59      renderWindow->Render();
60
61      vtkSmartPointer<vtkInteractorStyleTrackballCamera> style = vtkSma
62
63      renderWindowInteractor->SetInteractorStyle( style );
64
65      renderWindowInteractor->Start();
66  }
```

Snapshot of resulting rendering window is presented in Fig. 3 – and yes, I agree, it's not very impressive. However, we can easily tweak it and make it significantly better.

Figure 3. Exemplary 3D tumor visualized using 3D glyph mapper in VTK.

First of all to have better a better perspective we can slice the tumor in half and take resulting parts slightly apart. This can be achieved by modification of the addCells function (note that in my setting center of the tumor is at (x,y,z) = (250,250,250)).

```
30  void addCells(vector<Cell>::iterator begin, vector<Cell>::iterator en
31      vector<Cell>::iterator cell = begin;
32      for (cell; cell<end; cell++) {          if (cell->z < 250) {
33          } else {
34              points->InsertNextPoint(cell->x,cell->y,cell->z+50);
35          }
36      }
37  }
```

Fig. 4 shows the snapshot of the rendering window after slicing the tumor – much better, but there is still place for easy improvement. Let's add some color!

Figure 4. Exemplary 3D tumor visualized using 3D glyph mapper in VTK.

I assume that each cell has also information about the number of free spots in its direct neighborhood. We will color the tumor based on that information.

First, we need to add additional headers and variables.

```
#include "vtkCubeSource.h"
#include "vtkPolyDataMapper.h"
#include "vtkRenderer.h"
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkInteractorStyleTrackballCamera.h"
#include "vtkPolyData.h"
#include "vtkGlyph3DMapper.h"
#include "vtkPointData.h"
#include "vtkFloatArray.h"
#include "vtkLookupTable.h"

vtkSmartPointer<vtkRenderer> renderer;
vtkSmartPointer<vtkRenderWindow> renderWindow;
vtkSmartPointer<vtkRenderWindowInteractor> renderWindowInteractor;

vtkSmartPointer<vtkPoints> points;

vtkSmartPointer<vtkFloatArray> scalar;
```

Then we need to modify the function that initializes rendering

```
void initializeRendering() {
    renderer = vtkSmartPointer<vtkRenderer>::New(); //creating new re
    renderer->SetBackground(1., 1., 1.); //setting background color t

    renderWindow = vtkSmartPointer<vtkRenderWindow>::New(); //creati
    renderWindow->SetSize(1000, 1000); //setting size of the window
    renderWindow->AddRenderer(renderer); //adding previously created

    renderWindowInteractor = vtkSmartPointer<vtkRenderWindowInteracto
    renderWindowInteractor->SetRenderWindow(renderWindow); //adding p

    points = vtkSmartPointer<vtkPoints>::New();          //vector with
    scalar = vtkSmartPointer<vtkFloatArray>::New(); //vector with sca
    scalar->SetNumberOfComponents(1); //one value per point
}
```

and addCells function

```
void addCells(vector<Cell>::iterator begin, vector<Cell>::iterator en
    vector<Cell>::iterator cell = begin;
    for (cell; cell<end; cell++) {          if (cell->z < 250) {
        } else {
            points->InsertNextPoint(cell->x,cell->y,cell->z+50);
        }

        scalar->InsertNextValue((double)cell->nSpots/26.);
    }
}
```

Finally, we can modify the visualization function.

```
30  void visualizeLesion(vector<Cell>::iterator begin, vector<Cell>::iter
31
32      initializeRendering();
33
34      addCells(begin, end);
35
36      //creating new color table
37      vtkSmartPointer<vtkLookupTable> colorLookupTable = vtkSmartPointe
38      colorLookupTable->SetNumberOfColors(256);
39      colorLookupTable->SetHueRange( 0.667, 0.0);
40      colorLookupTable->Build();
41
42      // Combine into a polydata
43      vtkSmartPointer<vtkPolyData> polydata = vtkSmartPointer<vtkPolyDa
44      polydata->SetPoints(points);
45      polydata->GetPointData()->SetScalars(scalar);
46
47      vtkSmartPointer<vtkCubeSource> cubeSource = vtkSmartPointer<vtkCu
48
49      vtkSmartPointer<vtkGlyph3D> glyph3D = vtkSmartPointer<vtkGlyph3D>
50      glyph3D->SetColorModeToColorByScalar();
51      glyph3D->SetSourceConnection(cubeSource->GetOutputPort());
52      glyph3D->ScalingOff();
53      glyph3D->SetInputData(polydata);
54      glyph3D->Update();
55
56      // Create a mapper and actor
57      vtkSmartPointer<vtkPolyDataMapper> mapper = vtkSmartPointer<vtkPo
58      mapper->SetInputConnection(glyph3D->GetOutputPort());
59      mapper->SetLookupTable(colorLookupTable);
60
61      vtkSmartPointer<vtkActor> actor = vtkSmartPointer<vtkActor>::New
62      actor->SetMapper(mapper);
63
64      // Add the actor to the scene
65      renderer->AddActor(actor);
66
67      // Render
68      renderWindow->Render();
69
70      vtkSmartPointer<vtkInteractorStyleTrackballCamera> style = vtkSma
71
72      renderWindowInteractor->SetInteractorStyle( style );
73
74      renderWindowInteractor->Start();
75  }
```

Snapshot of sliced and colored tumor is presented in Fig. 5 – fast, good-looking and easy.

Figure 5. Exemplary 3D tumor visualized using 3D glyph mapper in VTK. Color represents the number of free spots in the direct neighborhood of each cell (jet colormap).

Avoiding boundary effects – dynamically expanding lattice for ABM

20 NOVEMBER 2015

JPOLESZCZUK

LEAVE A COMMENT

In a typical setting, agent-based model (ABM) simulations take place on a domain of fixed size, e.g. square lattice of size 500 x 500. Depending on the model, this can introduce so-called boundary effects, i.e. model predictions that are in part caused by the limited amount of space. Some time ago me and @theheikman published a paper in which we investigated the impact of evolution of cancer stem cells on the rate of tumor growth (link to the paper here (http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1004025)). Because of the ABM formulation, we had to implement domain that can freely expand on demand. Otherwise, the tumor evolution would stop once the cells fill all the space. Of course, we could set the initial domain size to be so big that the space wouldn't be filled in the simulated timeframe. However, it is hard to know a priori what domain size will be sufficient. In this post I will show how to implement in C++ an ABM lattice that dynamically expands if one of the cells touches its boundary. We will be working on pointers and memory allocating/freeing routines.

Our lattice will be a boolean array in which value of true will indicate that the spot is occupied by the cell. First, we need to create lattice of initial size NxN.

```
1  lattice=new bool *[N];
2  for (int i=0; i&lt;N; i++) {
3  lattice[i] = new bool [N] ();
4  fill_n(lattice[i], N, false);
5  }
```

An element of the lattice can be easily accessed using double indexing, i.e. lattice[i][j] = true.

Now we need to have a procedure that will expand our lattice, by fixed amount of rows (gN) from top, bottom and fixed amount of columns (also gN) to right and left.

```
 1  void expandLattice() {
 2  bool **aux;
 3  aux=new bool *[N+2*gN];
 4
 5  for (int i=0; i&lt;gN; i++) { //adding empty columns to the left
 6  aux[i] = new bool [N+2*gN] ();
 7  fill_n(aux[i], N+2*gN, false);
 8  }
 9
10  for (int i=N+gN; i&lt;N+2*gN; i++) { //adding empty columns to the ri
```

```
aux[i] = new bool [N+2*gN] ();
fill_n(aux[i], N+2*gN, false);
}

//copying the interior columns
for (int i=gN; i&lt;N+gN; i++) {
bool *aux2;
aux2 = new bool [N+2*gN] ();
fill_n(aux2, N+2*gN, false); //filling with false values
memcpy(aux2+gN, lattice[i-gN],N*sizeof(bool)); //copying existing val
free(lattice[i-gN]);
aux[i] = aux2;
}

free(lattice);
lattice=aux;
N += 2*gN;
}
```

Very important part of the above procedure is invoking free() function in order to deallocate the memory previously occupied by the lattice.

And that is about it: if an event of touching the boundary is detected, we just invoke the expandLattice() function. We also need to remember to free the allocated memory at the very end of the simulation, by putting

```
for (int i=0; i&lt;N; i++)
free(lattice[i]);

free(lattice);
```

at the end of main() function.

In CAexpandP (https://computecancer.files.wordpress.com/2015/11/caexpandp.doc) file you can find the code for the cancer stem cell driven model of tumor growth considered in the previous posts that uses dynamically expanding lattice (change extension from .doc to .cpp). Plot below nicely shows how the domain size grows together with the tumor when using that code with initial N = 100 and gN = 100 (plot shows the average of 20 simulations).

Cancer Stem Cell CA in Python

20 AUGUST 201520 AUGUST 2015

CHANDLER GATENBEE

1 COMMENT

Python doesn't seem to be the first programming language people go to when developing cellular automata models. However, given that Python is an object-orientated language that is easy to read and write, it might actually be ideal for such models, especially if you prefer to think from the perspective of the agent (if you'd rather model using matrices you can do that too by using Python's NumPy package). As we'll see, it can also be pretty fast. Below I'll cover one way to implement the Cancer Stem cell CA described in the Cancer Stem Cell CA in Matlab (https://computecancer.wordpress.com/2015/05/31/cancer-stem-cell-driven-tumor-growth-model-in-less-than-70-lines-of-code/) post.

GETTING PYTHON

If you don't have Python, the easiest way to get it and nearly all of its scientific packages is to download the Anaconda distribution from https://store.continuum.io/cshop/anaconda/ (https://store.continuum.io/cshop/anaconda/). This installation and all modules will be kept in it's own folder, so it won't interfere with the Python that comes bundled in your OS. It also means that uninstalling is just a matter of dropping the folder in the trash. You may also want to download an IDE, a list of which can be found on the Anaconda website (http://docs.continuum.io/anaconda/ide_integration (http://docs.continuum.io/anaconda/ide_integration) ;my personal favorite is PyCharm).

As far as versions are concerned, the "go-to" version is Python 2.7. However, 2.7 is no longer being developed so you will eventually have to migrate to Python 3. For a long time many avoided this upgrade because several of the most important packages hadn't been ported over, but that has since changed and most of the critical packages like Numpy, SciPy, Matplotlib, Pandas, etc… are now available in Python 3. If you choose to go with Python 2, it's important to note that when dividing you need to use a float in the denominator, as division by an integer returns a rounded down integer. This isn't the case with Python 3, however.

OVERVIEW

Before we begin coding, I'd like to provide an overview of how the model will be implemented. Instead of using a matrix to track cells, we'll create a dictionary of cells. The keys to this dictionary will be the cell's position, while the value will be the cell itself. Each cell we create will know the positions of it's neighbors, and the cell will use the dictionary of cells to look up its neighbors (a dictionary in Python is really just a hash table, so this look-up is fairly fast). A nice advantage of using a dictionary is that we can easily add and remove entries from it, giving us the ability to iterate only across cells, not empty positions in a matrix. This effectively allows the domain to grow and shrink according to how many cells are present. During each round of this model, we'll shuffle the values in this dictionary and iterate through them, having each cell execute it's actions.

One final note. I've taken screenshots of the code to make it easier to read, but the raw code can be found in this .doc file: CSC_CA_in_python (https://computecancer.files.wordpress.com/2015/08/csc_ca_in_python.doc). It's not possible to

upload .py files, so if you download the code, just change the extension from .doc to .py and then run the CA. Or, just copy and paste into your own file and run it.

GETTING THINGS READY

Let's start coding by importing the packages we will be using.The first is NumPy, Python's package for scientific computing (http://www.numpy.org (http://www.numpy.org)), a package you'll almost always need. Second, we'll import the package we'll use to visualize our results, matplotlib (http://matplotlib.org (http://matplotlib.org)). Next, we'll import Numba, a 'just in time' compiler than can dramatically speed up sections of code in which numerical calculations are called for (http://numba.pydata.org (http://numba.pydata.org)). Finally, importing the random module will allow us to randomly select cells.

```
import numpy as np #numerical python
import matplotlib.pyplot as plt #Package used for plotting
from numba import jit, int64 # just in time compiler to speed up CA
import random
```

(https://computecancer.files.wordpress.com/2015/08/imports2.png)

We'll begin writing up our CA by defining a few methods that we'll use to construct the world of our CA. First, lets construct that dictionary which will contain the list of positions in a central position's Moore neighborhood. To do this, we can use both list comprehension and dictionary comprehension. Basically, these approaches allow us to build lists or dictionaries without creating an empty list and messy for loop to populate that list/dictionary. Not only is list comprehension more convenient and cleaner, but it is also faster than the alternative.

```
def build_neighbor_pos_dictionary(n_row, n_col):
    list_of_all_pos_in_ca = [(r, c) for r in np.arange(0, n_row) for c in np.arange(0, n_col)]

    dict_of_neighbors_pos_lists = {pos: build_neighbor_pos_list(pos, n_row, n_col) for pos in list_of_all_pos_in_ca}

    return dict_of_neighbors_pos_lists



def build_neighbor_pos_list(pos, n_row, n_col):
    """
    Use list comprehension to create a list of all positions in the cell's Moore neighborhood.
    Valid positions are those that are within the confines of the domain (n_row, n_col)
    and not the same as the cell's current position.
    """
    # Unpack the tuple containing the cell's position
    r, c = pos

    l = [(r+i, c+j)
         for i in [-1, 0, 1]
         for j in [-1, 0, 1]
         if 0 <= r + i < n_row
         if 0 <= c + j < n_col
         if not (j == 0 and i == 0)]

    return l
```

(https://computecancer.files.wordpress.com/2015/08/builders.png)

SPEEDUP TRICKS

Here we'll setup a few methods to speed up our CA. We'll frequently request a random binomial number and shuffle cells, so we'll call those methods once, here at the top, so they don't have to be reevaluated each time we call them, something that would slow down our model. Next, we'll create a few methods to speed up the generation of random binomial numbers (something we'll do frequently) by creating methods to call *binomial* using numba's just in time compiler (jit). To do this, all we have to do is create a wrapper method, and then add the @jit decorator. This small modification provides a dramatic speed-up; without it the CA and plotting takes 16 min, but adding the simple @jit decorator reduces the time it takes down to around 5 minutes! Who says Python is always slow🙂

(https://computecancer.files.wordpress.com/2015/08/speedups.png)

CANCER CELL CLASS

(https://computecancer.files.wordpress.com/2015/08/cancer_cell_class.png)

OK, now that everything is set up, we'll start creating our Cancer Cell class, which defines all of the cancer cell's attributes and behaviors. First, we define the mandatory *__init__* method, which defines the attributes each newly created cancer cell will have. In this case, each new cancer cell will be assigned a position (*pos*), the number of divisions it has remaining (*divisions_remaining*), and the list containing the positions in its neighborhood (*neighbor_pos_list)*. This list is assigned simply by using the cell's position to look up that list in the *dictionary_of_neighbor_pos_lists* that we created earlier.

Now that we've defined our cell's attributes, we can start defining what exactly our cells will do. Because this is a fairly simple CA, we'll just define all of the behaviors in one method (sort of…), which we'll call *act*. First we'll write up the bit of code for cell division. Recall that a cancer cell can only divide if that the cell is lucky and has an empty space in its neighborhood. In this case, if *divide_q* is 1, the cell is lucky and has a chance to divide. If the cell is indeed lucky, it then uses the *locate_empty_neighbor_position* method to find all of the free spaces in its neighborhood (the second requirement for division). Jumping to that *locate_empty_neighbor_position* method, the first thing we'll do is create a list of positions that are NOT in the cell dictionary, that is, spaces which are currently unoccupied. If there are actually empty positions, a randomly selected empty position in the neighborhood is returned. Moving back to the *act* method, a new Cancer Cell is created, assigned that random position, made to look up its neighbor list in the *dictionary_of_neighbor_pos_lists*, and then added to the cell dictionary. Finally, the dividing cell decreases the number of divisions that it can undergo by one.

After a cell has tried to divide, it will determine if it needs to die. Recall that a cell will die for one of two reasons: 1) it dies because it has exceed its maximum proliferative potential, that is, *divisions_remaining* <= 0; or 2) it dies spontaneously, that is, if *die_q* = 1 . If either of these conditions are true, the cell and it's position are deleted from the cell dictionary, thus removing it from the CA and freeing up space.

CANCER STEM CELL CLASS

```python
class CancerStemCell(CancerCell):

    def __init__(self, pos, dictionary_of_neighbor_pos_lists):

        super(CancerStemCell, self).__init__(pos, dictionary_of_neighbor_pos_lists)
        self.PLOT_ID = 2

    def act(self, agent_dictionary, dictionary_of_neighbor_pos_lists):

        divide = divide_q()
        if divide == 1:

            empty_pos = self.locate_empty_neighbor_position(agent_dictionary)
            if empty_pos is not None:

                symmetric_division = divide_symmetrically_q()
                if symmetric_division == 1:
                    daughter_cell = CancerStemCell(empty_pos, dictionary_of_neighbor_pos_lists)

                else:
                    daughter_cell = CancerCell(empty_pos, dictionary_of_neighbor_pos_lists)

                agent_dictionary[empty_pos] = daughter_cell
```

(https://computecancer.files.wordpress.com/2015/08/cancer_stem_cell_class.png)

Now that our Cancer Cell class is created, we can start defining the behaviors of Cancer Stem Cells. We could simply clog up our code by adding a bunch of if/else statements to our Cancer Cell's *act* method, but instead we'll create a Cancer Stem Cell class, which will be a subclass of the Cancer Cell class. Classes in Python have inheritance, which means that our new Cancer Stem Cell class has all of attributes and methods of the Cancer Cell class, and so we don't need to redefine all attributes or re-write the *locate_empty_neighbor_position* method. The only attribute we'll redefine is the *PLOT_ID*, which we'll use when visualizing the results of our CA. Next, we'll redefine the *act* method. The process of division is the same as before, except that the stem cell can either divide symmetrically or asymmetrically. Which type of division occurs is determined by calling the *divide_symmetrically_q* method. If *divide_symmetrically* = 1, a new Cancer Stem Cell is created and added to the cell dictionary; if *divide_symmetrically* = 0, a normal Cancer Cell is created and added to the dictionary instead. Now we have our Cancer Cells and Cancer Stem Cells setup and ready for action!

RUN THE MODEL

(https://computecancer.files.wordpress.com/2015/08/run1.png)

```python
if __name__ == "__main__":
    N_ROW = 2000
    N_COL = 2000
    MAX_REPS  = 6*20*34
    DICTIONARY_OF_NEIGHBOR_POS_LISTS = build_neighbor_pos_dictionary(N_ROW, N_COL)

    center_r = int(round(N_ROW/2.0))
    center_c = int(round(N_COL/2.0))
    center_pos = (center_r, center_c)

    initial_cancer_stem_cell = CancerStemCell(center_pos, DICTIONARY_OF_NEIGHBOR_POS_LISTS)
    cell_dictionary = {center_pos:initial_cancer_stem_cell}

    for rep in range(MAX_REPS):
        #### USE BELOW LINE FOR PYTHON 2 ####
        cell_list = cell_dictionary.values()

        #### USE BELOW LINE FOR PYTHON 3 ####
        # cell_list = list(cell_dictionary.values())

        shuffle(cell_list)
        for cell in cell_list:
            cell.act(cell_dictionary, DICTIONARY_OF_NEIGHBOR_POS_LISTS)
```

(https://computecancer.files.wordpress.com/2015/08/run2_7.png)

The last thing we have to do is setup the initial conditions of our CA, which we'll do under the *if __name__ == "__main__"* line. First, we'll define the constants at the top. Next, we'll create the initial Cancer Stem Cell, which will be positioned in the middle of world. We do this by initializing this first stem cell, using the position in the center of the world and *DICTIONARY_OF_NEIGHBOR_POS_LISTS*, and then add it to the cell dictionary. Next, we'll copy the cells in the cell dictionary to a list (line 189 for Python 2, or line 192 for Python 3). While copying the values to a list isn't ideal, it does provide two advantages. First, we can shuffle this list, allowing us to move through our cells randomly. Second, this list is what allows us to change the cell dictionary on the fly, as the length of a dictionary cannot be changed while iterating through it. Finally, we loop through our list of randomly ordered cells, having each one conduct their actions by calling their *act* method, a process that is repeated until maximum number of reps has been met.

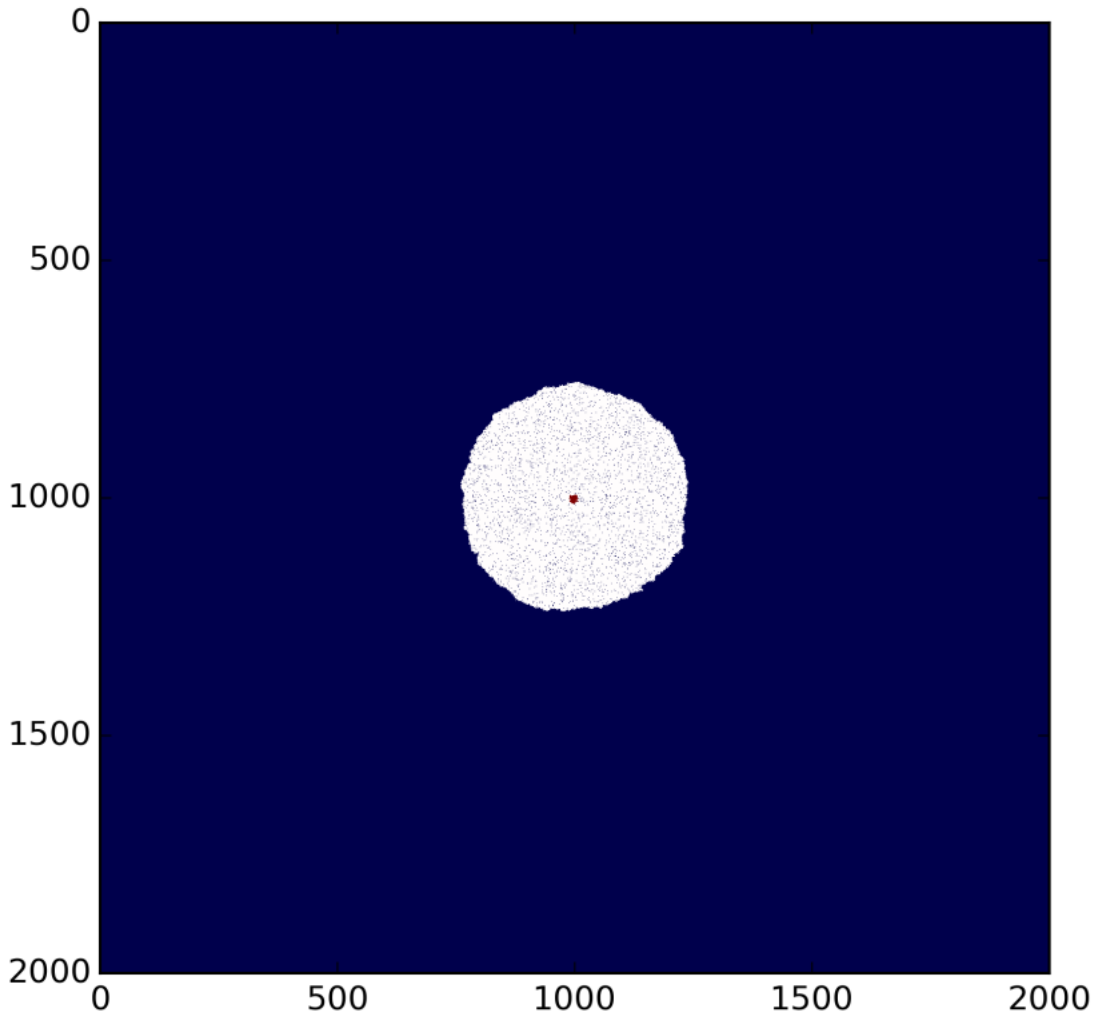
VISUALIZE RESULTS

```python
visualization_matrix = np.zeros((N_ROW, N_COL))
for cell in cell_dictionary.values():
    visualization_matrix[cell.pos] = cell.PLOT_ID

plt.imshow(visualization_matrix, interpolation='none', cmap='seismic', vmin=0, vmax=2)
plt.savefig("./CSC_CA.png", dpi=300)
```

(https://computecancer.files.wordpress.com/2015/08/visualization.png)

After we've completed our simulation, we would like to visualize our results, so we create a *visualization_matrix*, which is just matrix of zeros. Because we don't care about the order of our dictionary, and won't be modifying it's length, we just iterate through the cells in our cell dictionary. Each cell then adds it's PLOT_ID number to the matrix, so that stem cells will be one color, and normal cells another color. Now we use matplotlib's *imshow* and *show* methods to visualize the results:



(https://computecancer.files.wordpress.com/2015/08/csc_ca1.png)

CONCLUSION

Using Python has its pros and cons. The pros are that it's easy to read and write, reducing development time and making it easier to share code. The con is that, in its native form, Python is not the fastest. However, as we've seen, there are tools to increase performance. Numba is only one of those tools, but there are others, with the most popular probably being Numba Pro, CUDA, Cython, and PyPy. There are also ways to take advantage of multithreading and mulitprocessing to speed up your models even more. So, if its possible to use these tools, you can have both fast development and execution speed! I hope you found this post helpful, and that maybe you'll even consider using Python for your next project.

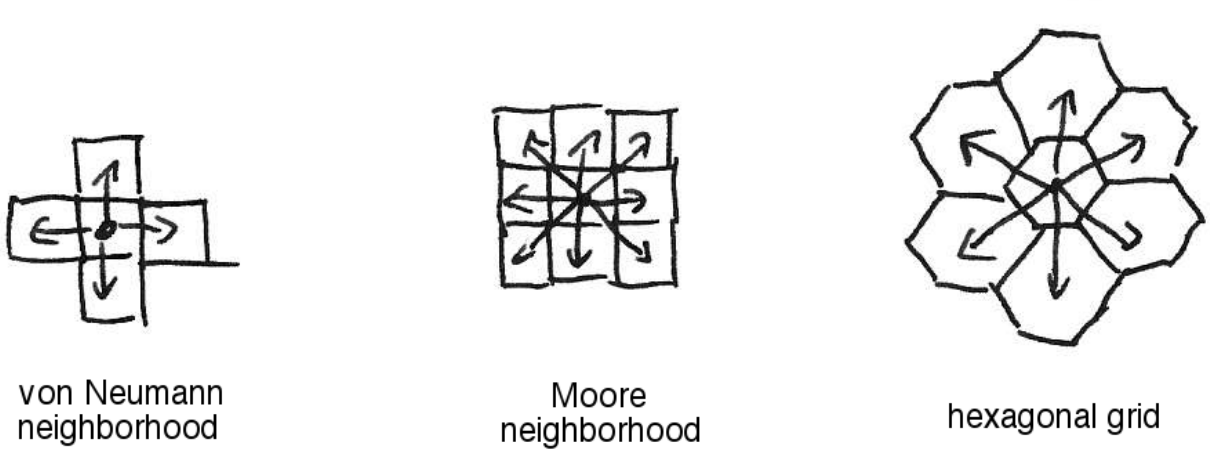
Quick implementation of hexagonal lattice for ABM

7 AUGUST 2015

JPOLESZCZUK

LEAVE A COMMENT

In the agent based modeling we are typically more interested in the rules governing the cell fate rather than the basic setting of the lattice (if we don't look at the off lattice model). However, the particular setting of the computational domain can have an effect on the model dynamics. In 2D we typically consider the following lattices and neighborhoods:



(https://computecancer.files.wordpress.com/2015/08/picture1.png)

with the rectangular grid with Moore neighborhood being probably most frequently utilized. With von Neumann neighborhood we have the fewest number of neighbors and cells are spatially saturated earlier. The problem with Moore neighborhood is that the distance to all of the sites is not the same. Hexagonal grid has good properties (same distance, 6 neighbors), but is less frequently utilized, because implementation is more involved. In todays post I will show that essentially there is no difference in implementation between all of those lattices.

We will use the codes for basic ABM model posted before as a template (MATLAB version here (https://computecancer.wordpress.com/2015/05/31/cancer-stem-cell-driven-tumor-growth-model-in-less-than-70-lines-of-code/)and C++ version here (https://computecancer.wordpress.com/2015/06/06/cancer-stem-cell-driven-tumor-growth-model-in-c/)). In both cases we considered rectangular lattice with Moore neighborhood. If we set migration probability to zero, set large value of proliferation capacity and set the spontaneous death rate to zero, i.e. we simulate essentially only division events, we will see in visualizations of simulated tumors what kind of neighborhood we assumed:

(https://computecancer.files.wordpress.com/2015/08/square.png)

Let us see how the visualization looks like for different types of neighborhoods/lattices.

In both implementations we had a separate array defining the neighborhood of the cell. Thus, in order to modify the code to von Neumann neighborhood we need to change only two consecutive lines in the MATLAB code in which we define the neighborhood and the permutations table:

```
aux = int32([-N -1 1 N])'; %indices to heighborhood
Pms = perms(uint8(1:4))'; %permutations
```

In C++ implementation we need to modify the definition of the neighborhood:

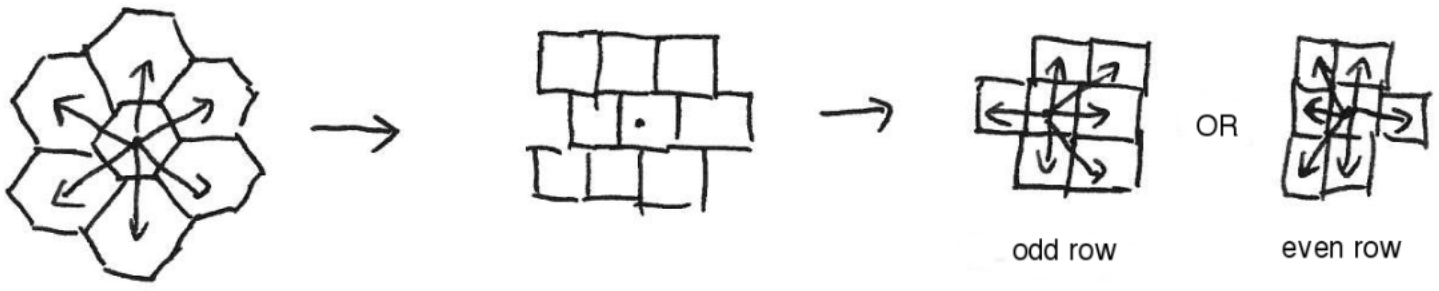
```
static const int indcNeigh[] = {-N, -1, 1, N};//neighborhood
```

and the *for* loop in the *returnEmptyPlace* function:

```
for(int j=0;j<4;j++) {//searching through neighborhood
        if (!lattice[indx+indcNeigh[j]]) {
            neigh[nF] = indx+indcNeigh[j];
            nF++;
        }
}
```

As it could be expected, switching to von Neumann neighborhood made the simulated tumor diamond shaped.

Let us now consider hexagonal lattice. The very first observation that we need to make is that hexagonal lattice is essentially an rectangular lattice with shifted odd (or even) rows and two definitions of neighborhood (one for cells in odd rows, second for cells in even rows), see picture below



(https://computecancer.files.wordpress.com/2015/08/concept.png)

Thus, in the modified implementation we again only need to change the definition of the neighborhood and add the condition for choosing the proper one. In MATLAB it can be achieved by introducing two definitions of neighborhoods

```
auxR = int32([-N -1 1 N-1 N N+1])'; %indices to heighborhood
auxL = int32([-N -1 1 -N-1 N -N+1])'; %indices to heighborhood

Pms = perms(uint8(1:6))'; %permutations
```

and modify the lines in the main loop in which we create the neighborhood for all viable cells (create variable *S*)

```
odd = mod(cells,2) ~= 0; %selecting cells in the odd rows
S = zeros(6,length(cells));
if any(odd) %creating neighborhood for the cells in the odd rows
    SR = bsxfun(@plus,cells(odd),auxR(Pms(:,randi(nP,1,sum(odd)))));
    S(:,odd) = SR;
end
even = ~odd;
if any(even) %creating neighborhood for the cells in the even rows
    SL = bsxfun(@plus,cells(even),auxL(Pms(:,randi(nP,1,sum(even)))));
    S(:,even) = SL;
end
```

In C++ the changes are even easier. We again introduce two neighborhoods:

```
static const int indcNeighR[] = {-N,-1,1,N-1,N,N+1};//neighborhood
static const int indcNeighL[] = {-N,-1,1,-N-1,N,-N+1};//neighborhood
```

and modify the *returnEmptyPlace* function

```
for(int j=0;j<6;j++) {//searching through neighborhood
        if (indx % 2) {//if odd row
            if (!lattice[indx+indcNeighR[j]]) {
                neigh[nF] = indx+indcNeighR[j];
                nF++;
            }
        } else {//if even row
            if (!lattice[indx+indcNeighL[j]]) {
                neigh[nF] = indx+indcNeighL[j];
                nF++;
            }
        }
}
```

We might also want to change the code for visualization. In the previous settings it was enough to represent each cell as a pixel in the image. In case of hexagonal lattice in order to be correct we need to somehow shift the pixels in odd (or even rows). The easiest way to achieve that is to represent each cell as 4 pixels and adjust position according to the row number:

(https://computecancer.files.wordpress.com/2015/08/drawing.png)

The modified visualization code in MATLAB is the following

```
function visualizeHex( N, cells, cellsIsStem, cellsPmax, pmax )

    %select cells in the odd rows
    odd = mod(cells,2) ~= 0;

    M = ones(2*N,2*N,3); %matrix for image, we expand it
```

```
    %disperse cell by one spot
    i = mod(double(cells)-1,N)+1; %row, REMEMBER THAT CELLS ARE UINTs
    j = ceil(double(cells)/N); %column
    cells = (i*2-1)+(2*j-1)*2*N;

    %add cells to the top right in odd rows
    cells = [cells reshape(bsxfun(@plus,cells(odd),[-1; 2*N-1; 2*N]),
    cellsIsStem = [cellsIsStem reshape(repmat(cellsIsStem(odd),3,1),1
    cellsPmax = [cellsPmax reshape(repmat(cellsPmax(odd),3,1),1,[])];

    %add cells to top left in even rows
    even = [~odd false(1,3*sum(odd))];
    cells = [cells reshape(bsxfun(@plus,cells(even),[-1; -2*N-1; -2*N
    cellsIsStem = [cellsIsStem reshape(repmat(cellsIsStem(even),3,1),
    cellsPmax = [cellsPmax reshape(repmat(cellsPmax(even),3,1),1,[])]

    %plotting
    color = hot(3*pmax);
    M(cells(~cellsIsStem)) = color(cellsPmax(~cellsIsStem)+1,1);
    M(cells(~cellsIsStem)+4*N*N) = color(cellsPmax(~cellsIsStem)+1,2)
    M(cells(~cellsIsStem)+8*N*N) = color(cellsPmax(~cellsIsStem)+1,3)

    CSCs = cells(cellsIsStem);
    M(CSCs) = color(2*pmax,1);
    M(CSCs+4*N*N) = color(2*pmax,2);
    M(CSCs+8*N*N) = color(2*pmax,3);

    figure(1)
    clf
    imshow(M);

end
```

Finally, simulating the tumor with exactly the same parameters settings on the hexagonal lattice results in way more circular tumor.

(https://computecancer.files.wordpress.com/2015/08/hexagonal.png)

Quick parallel implementation of local sensitivity analysis procedure for agent-based tumor growth model

26 JUNE 201526 JUNE 2015

JPOLESZCZUK

LEAVE A COMMENT

In the last couple of posts I've shown how to implement agent-based model of cancer stem cell driven tumor growth, both in MATLAB (https://computecancer.wordpress.com/2015/05/31/cancer-stem-cell-driven-tumor-growth-model-in-less-than-70-lines-of-code/)and C++ (https://computecancer.wordpress.com/2015/06/06/cancer-stem-cell-driven-tumor-growth-model-in-c/). Having the implementations we can go one step further and perform some analysis of the tumor growth characteristics predicted by the model. We will start with performing local sensitivity analysis, i.e. we will try to answer the question how the perturbation of parameter values impacts the growth dynamics. Typically it is being done by perturbing one of the parameters by a fixed amount and analyzing the response of the model to that change. In our case the response will be the percentage change in average tumor size after 3 months of growth. Sounds fairly simple, but…

We have 5 different parameters in the model: proliferation capacity, probability of division, probability of spontaneous death, probability of symmetric division, and probability of migration. Moreover, let us assume that we would like to investigate 3 different values of parameter perturbation magnitude (5%, 10% and 20%). In order to be able to analyze the change in the average size we need to have its decent estimator, i.e. we need sufficient number of simulated stochastic trajectories – let us assume that 100 simulations is enough to have a good estimation of "true" average. So in order to perform the sensitivity analysis we will need to perform 100 + 3*5*100 = 1600 simulations (remember about the growth for nominal parameters values). Even if single simulation takes typically 30 seconds, then we will wait more than 13 hours to obtain the result using single CPU – that is a lot!

After looking at the above numbers we can make a straightforward decision right now – we will use C++ instead of MATLAB, because the model implementation in C++ is a several times faster. However 1) we will need to write a lot of a code in order to perform sensitivity analysis, 2) using multiple CPU is not that straightforward as in MATLAB. Is there a better way to proceed?

Few weeks ago I've shown here (https://computecancer.wordpress.com/2015/06/13/wrapping-c-code-of-agent-based-tumor-growth-model-into-matlab/) how to wrap your C++ in order to use it from within MATLAB as a function without loosing the C++ performance. Why not to use it to make our lives easier by making sensitivity code short and harness easily the power of multiple CPUs?

We will start the coding (in MATLAB) with setting all simulation parameters

```
1  nSim = 100; %number of simulations to perform for a given set of paran
2  tmax = 30*3; %number of days to simulate
3  N = 1000; %size of the simulation domain
4
5  %nominal parameter values [rhomax, pdiv, alpha, ps, pmig]
6  nominal = [10, 1/24, 0.05, 0.3,10/24];
7
8  perturb = [0.05 0.1 0.2]; %perturbation magnitudes, percent of initial
```

Now we just need to construct a loops that will iterate through all possible perturbations and simulations. If we don't use Parallel Toolbox, i.e. don't use multiple CPUs, it doesn't really matter how we will do that – performance will be similar. Otherwise, implementation is not that straightforward even though from MATLABs documentation it seems that it is enough to change *for* to *parfor*. The most important thing is how will we divide the work between the CPUs. The simples idea is to spread the considered perturbations values among the CPUs – that will allow to use 15 CPUs in our setting. However, I've got machine with 24 CPUs, so that would be a waste of resources – bad idea. The other idea would be to use *parfor* loop to spread all 100 simulations for a given perturbation value on all 24 CPUs and go through all perturbation values in a simple loop – now we are using all available CPUs. But are we doing that efficiently? No. The thing is that CPUs need to be synchronized before proceeding to the next iteration of the loop going through perturbation values. So some of the CPUs will be idle while waiting for other ones to finish with *parfor* loop. In order to make the code even more efficient we will just use one *parfor* loop and throw all 1600 simulation on 24 CPUs at the same time. Let us first prepare the output variable.

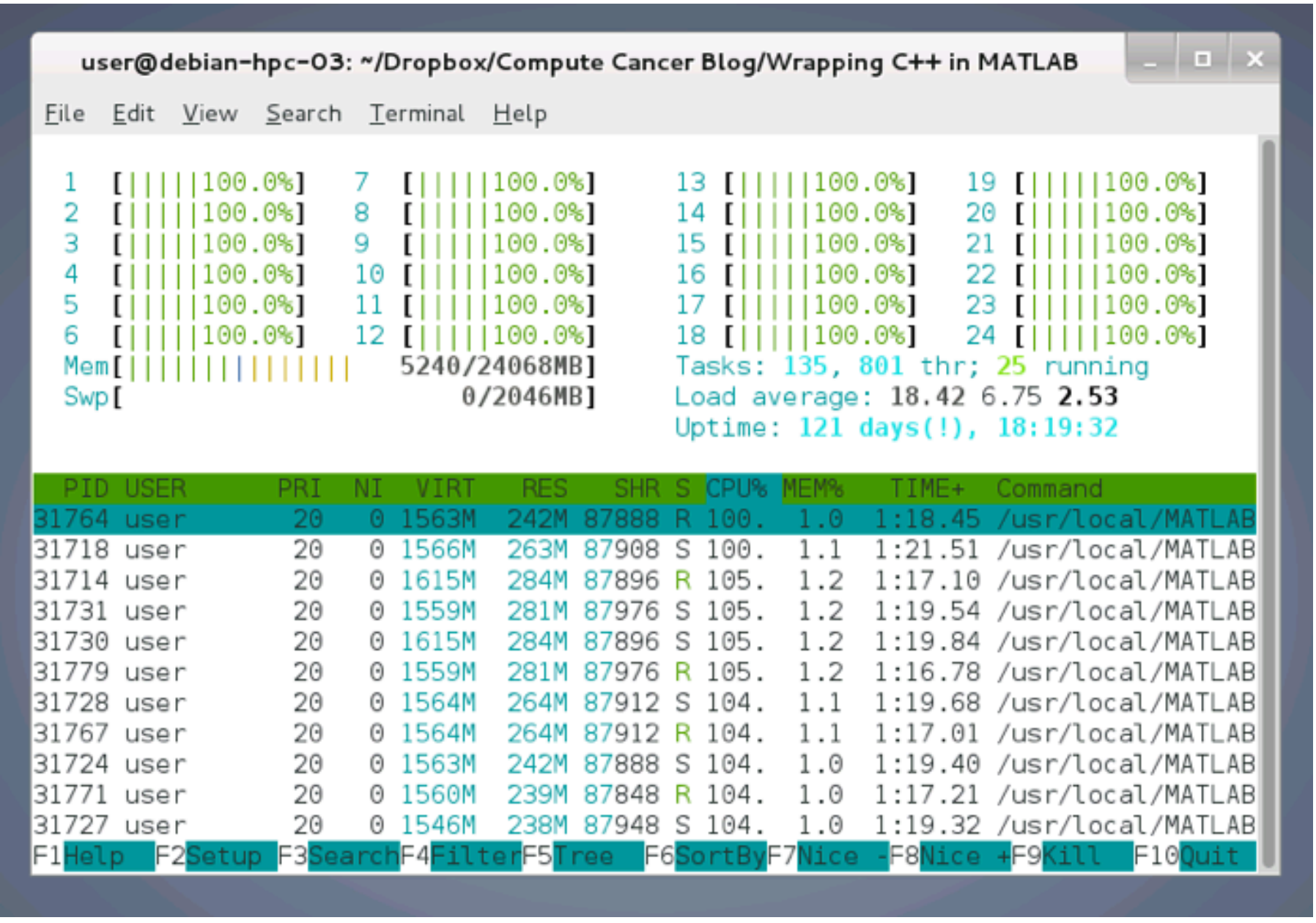
```
HTCG = zeros(1,nSim + length(nominal)*length(perturb)*nSim);
```

Before writing the final piece of the code we need to solve one more issue. Namely, in the C++ implementation we used srand(time(NULL)) to initiate the seed for random number generator. It is perfectly fine when we use single CPU – each simulation will take some time and we don't need to worry about uniqueness of the seed. The problem is when we want to use multiple CPUs – all initial parallel simulations will start with exactly the same seed. One way to solve that is to pass the current loop iteration number (*i*) to C++ and use srand(time(NULL)+i) – that is what I have done. After solving that issue we can write the final piece of the code.

```
parfor i = 1:length(HTCG)
    %%PREPARING PARAMETERS VALUES
    params = nominal; %setting parameters to nominal values
    if i>nSim %simulation is for perturbed parameters
        %translating linear index to considered parameter and perturb
        j = ceil((i-nSim)/(nSim*length(perturb)));
        k = ceil((mod((i-nSim)-1,nSim*length(perturb))+1)/nSim);
        %updating parameters
        params(j) = params(j)*(1+perturb(k));
        if k == 1 %if proliferation capacity parameter we need to rou
           params(j) = round(params(j));
        end
    end

    %%RUNNING SIMULATION AND SAVING OUTPUT
    [~, cells] = CA(params,[tmax*24,N,i]);
    HTCG(i) = length(cells)/3;
    clear mex %important! without that the internal
              %variables in CA functions won't be cleared and next si
              %won't begin with single initial cell
end
```

Then in the command line we start the parallel pool with 24 workers (CPUs), by typing parpool(24) command, and run the code. Screenshot below shows nicely how all of the 24 CPUs are being used – no resource wasted!



(https://computecancer.files.wordpress.com/2015/06/cpus-occupancy.png)

We can then add few additional lines of the code to plot the results.

```
nom = mean(HTCG(1:100)); %average for nominal parameters value
%calculating averages for perturbed sets values
av = squeeze(mean(reshape(HTCG(101:end),nSim,length(perturb),length(r

%plotting results of sensitivity analysis
set(0,'DefaultAxesFontSize',18)
figure(1)
clf
bar(perturb*100,(av-nom)/nom*100)
legend({'\rho_{max}', 'p_{div}', '\alpha', 'p_s', 'p_{mig}'})
xlabel('% perturbation')
ylabel('% change')
```

And “voila!” – the resulting figure shows that the perturbation in the proliferation capacity has the highest impact on the tumor growth dynamics.

(https://computecancer.files.wordpress.com/2015/06/sensresults.png)

Cancer stem cell driven tumor growth model in C++

6 JUNE 2015

JPOLESZCZUK

12 COMMENTS

Because of the feedback that I've received after publishing first three posts, I've decided to change a tool of interest from MATLAB to C++. Today, I'll show how to quickly implement a cancer stem cell driven tumor growth model in C++. It is almost the same model as implemented in my previous post (https://computecancer.wordpress.com/2015/05/31/cancer-stem-cell-driven-tumor-growth-model-in-less-than-70-lines-of-code/) (I'll explain why it is "almost" the same at the end of this post).

Quick guide to the model: A cell, either cancer stem cell (CSC) or non-stem cancer cell (CC), occupies a single grid point on a two-dimensional square lattice. CSCs have unlimited proliferation potential and at each division they produce either another CSC (symmetric division) or a CC (asymmetric division). CCs are eroding their proliferation potential at each division and die when its value reaches 0. Moreover, at each proliferation attempt, CCs may undergo spontaneous death and then be removed from the system.

First we start with including the necessary headers and setting the namespace. Apart from the standard functions and datatypes, we will use the vector datatype to store the cells and a shuffling function from algorithm library.

```
1  #include <vector>
2  #include <algorithm>
3  #include <stdlib.h>
4
5  using namespace std;
```

Now we can define a cell. It will be defined by three variables: index to the place on the lattice (integer), remaining proliferation capacity (char), and boolean variable defining stemness.

```
1  struct cell {//defining cell
2      int place;
3      char p;
4      bool is_stem;
5  };
```

Next step is to define the lattice size, the lattice itself, vector that will contain all viable cells, and auxiliary variable defining the cells neighborhood.

```
1  static const int N = 2000; //lattice size
2  bool lattice[N*N] = {false}; //empty lattice
3  vector<cell> cells; //vector containing all cells present in the syste
4
5  static const int indcNeigh[] = {-N-1, -N, -N+1, -1, 1, N-1, N, N+1};//
```

The parameters of the model are defined as follows.

```
1  char pmax=10; //proliferation capacity
2  double pDiv=1./24.; //division probability
3  double alpha=0.05; //spontaneous death probability
4  double ps=0.05; //probability of symmetric division
5  double pmig=10./24.; //probability of migration
```

Having made all of those initial definitions we can finally start coding the main program. Let us start with writing the function that will initialize the whole simulation, i.e. fill the lattice boundary and put the initial stem cell.

```
 1  void initialize() {
 2      for (int i=0; i<N; i++) {lattice[i]=true;};//filling left
 3      for (int i=0; i<N*N; i=i+N) {lattice[i]=true;};//filling top
 4      for (int i=N-1; i<N*N; i=i+N) {lattice[i]=true;};//filling botton
 5      for (int i=N*(N-1); i<N*N; i++) {lattice[i]=true;};//filling righ
 6
 7      lattice[N/2*N+N/2] = true; //initial cell in the middle
 8      cell initialCell = {N/2*N+N/2,pmax,true};//initial cell definitio
 9      cells.push_back(initialCell);
10  }
```

As in the previous post (https://computecancer.wordpress.com/2015/05/31/cancer-stem-cell-driven-tumor-growth-model-in-less-than-70-lines-of-code/), we set all the boundary values of *lattice* to true in order to make sure that we will not address any site out of the lattice. Typically one would use an *if* statement when addressing the lattice site to make sure that index to a site is within the domain. Here, because we have set the boundaries to *true* without adding those sites to *cells* vector, we don't need to do that – boundary will be treated as an occupied site, but not as a viable cell.

The second auxiliary function that we will use returns the index to the randomly selected empty space around a given spot.

```
 1  int returnEmptyPlace(int indx) {
 2      int neigh[8], nF = 0;
 3      for(int j=0;j<8;j++) {//searching through neighborhood
 4          if (!lattice[indx+indcNeigh[j]]) {//if free spot
 5              neigh[nF] = indx+indcNeigh[j]; //save the index
 6              nF++; //increase the number of found free spots
 7          }
 8      }
 9      if(nF) {//selecting free spot at random
10          return neigh[rand() % nF];
11      } else {//no free spot
12          return 0;
```

```
    }
}
```

If there is no free spot the function returns 0.
Finally we can code the part in which all the magic happens – main simulation procedure. It is hard to dissect the whole procedures into parts, so I did my best to explain everything in the comments.

```
void simulate(int nSteps) {

    vector<cell> cellsTmp;
    int newSite;
    cell currCell, newCell;

    for (int i=0; i<nSteps; i++) {
       random_shuffle(cells.begin(), cells.end()); //shuffling cells
       while (!cells.empty()) {
           currCell=cells.back(); //pick the cell
           cells.pop_back();
           newSite = returnEmptyPlace(currCell.place);

           if (newSite) {//if there is a new spot
               newCell = currCell;
               newCell.place = newSite;
               if ((double)rand()/(double)RAND_MAX < pDiv) {
                   if (currCell.is_stem) {
                       lattice[newSite]=true;
                       cellsTmp.push_back(currCell);
                       if ((double)rand()/(double)RAND_MAX > ps) {//a
                           newCell.is_stem = false;
                       }
                       cellsTmp.push_back(newCell);
                   } else if (currCell.p > 0 && (double)rand()/(doubl
                       currCell.p--;
                       newCell.p--;
                       lattice[newSite] = true;
                       cellsTmp.push_back(currCell);
                       cellsTmp.push_back(newCell);
                   } else {
                       lattice[currCell.place] = false;
                   }
               } else if ((double)rand()/(double)RAND_MAX < pmig) {
                   lattice[currCell.place] = false;
                   lattice[newSite] = true;
                   cellsTmp.push_back(newCell);
               } else {//doing nothing
                 cellsTmp.push_back(currCell);
               }
           } else {//no free spot
               cellsTmp.push_back(currCell);
           }
       }
       cells.swap(cellsTmp);
    }
```

```
47 |  }
```

Now we wrap everything in the main function, compile and run.

```
1 | int main() {
2 |     srand(time(NULL)); //initialize random number generator
3 |     initialize(); //initialize CA
4 |     simulate(24*30*6);
5 |     return 0;
6 | }
```

Everything in about 100 lines of the code.
What about the speed of the code? It took about 40 seconds to simulate the tumor presented below, which is consisted of about 460,000 cells (on 2000×2000 lattice).

(https://computecancer.files.wordpress.com/2015/06/screen-shot-2015-06-06-at-12-24-53-am.png)

Why is it "almost" the same model as the one presented in previous post? The answer is in details. In the above C++ implementation lattice is updated after every single cell movement/death, i.e. because of the random shuffling a new free spot can be created for a cell that at the beginning of the iteration was quiescent (without a free spot), so it can successfully proliferate in the same iteration. In the MATLAB implementation that kind of behavior was avoided.

Cancer stem cell driven tumor growth model in less than 70 lines of code

31 MAY 20151 JUNE 2015

JPOLESZCZUK

9 COMMENTS

Today I've decided to show how to efficiently code a cancer stem cell driven tumor growth model in less then 70 lines of code in MATLAB.

Quick guide to the model: A cell, either cancer stem cell (CSC) or non-stem cancer cell (CC), occupies a single grid point on a two-dimensional square lattice. CSCs have unlimited proliferation potential and at each division they produce either another CSC (symmetric division) or a CC (asymmetric division). CCs are eroding their proliferation potential at each division and die when its value reaches 0. Moreover, at each proliferation attempt, CCs may undergo spontaneous death and then be removed from the system.

We start with defining initial settings of a domain and simulation timespan.

```
N = 1000; %square domain dimension
nSteps = 6*30*24; %number of simulation steps
```

Next we define the values of all parameters used in the simulation.

```
pprol = 1/24; %probability of proliferation
pmig = 10/24; %probability of migrating
pdeath = 1/100; %probability of death
pmax = 10; %initial proliferation capacity of CC
ps = 3/10; %probability of symmetric division
```

Now we can intialize domain and place initial CSC in its center. Our computational domain is represented by N x N Boolean matrix *L* (a *true* value indicates the lattice point is occupied). An additional integer vector *cells* is maintained to store indices for all viable cells present in the system (to avoid costly search for cells on the lattice).

```
L = false(N,N);
L([1:N 1:N:N*N N:N:N*N N*(N-1):N*N]) = true; %boundary
L(N*round(N/2)+round(N/2)) = true;
cells = int32(N*round(N/2)+round(N/2));
cellsIsStem = true;
cellsPmax = uint8(pmax);
```

To reduce the amount of used memory we utilized int32 and unit8 types as they occupy less memory than double (double is default type in MATLAB). We also set all the boundary values of *L* to true in order to make sure that we will not address any site out of the lattice. Typically one would use an *if* statement when addressing the lattice site to make sure that index to a site is within the domain. Here, because we have set the boundaries to *true* without adding those sites to *cells* vector, we don't need to do that – boundary will be treated as an occupied site, but not as a viable cell.

We can now define auxiliary variables that will be utilized further.

```
aux = int32([-N-1 -N -N+1 -1 1 N-1 N N+1])'; %indices to heighborhood
Pms = perms(uint8(1:8))'; %permutations
nP = size(Pms,2); %number of permutations
```

Variable *aux* simply defines cell's neighborhood and *Pms* is a variable in which we store all possible permutations of variable *aux*.

Now we can start the main loop of the program and randomly shuffle cells at its beginning.

```
for i = 1:nSteps

sh = randperm(length(cells));
cells = cells(sh);
cellsIsStem = cellsIsStem(sh);
cellsPmax = cellsPmax(sh);
```

Now few lines in which we first create indices to neighborhoods of all of the cells already in random order (variable *S*), then we flag by 0 all of the spots that are occupied and finally select only indices to those cells that have at least one free spot in the neighborhood (variable *indxF*).

```
S = bsxfun(@plus,cells,aux(Pms(:,randi(nP,1,length(cells)))));
S(L(S)) = 0; %setting occupied spots to 0
indxF = find(any(S)); %selecting cells with at least one free spot
nC = length(indxF); %number of cells with free spot
```

Now we can decide about the faith of the cells that can perform action (are not completely surrounded by other cells).

```
P = rand(1,nC)<pprol; %proliferation
Ps = P & rand(1,nC)<ps & cellsIsStem(indxF); %symmetric division
De = P & (cellsPmax(indxF) == 0); %proliferation capacity exhaution
D = P & (rand(1,nC)<pdeath) & ~cellsIsStem(indxF); %death at prolifera
M = ~P & (rand(1,nC)<pmig); %go when no grow
```

In the additional loop we perform update of the system.

```
del = D | De; %cells to delete
act = find((P | M) & ~del); %indices to the cells that will perform a
for ii = act %only for those that will do anything
    ngh = S(:,indxF(ii)); %cells neighborhood
    ngh(ngh==0) = []; %erasing places that were previously occupied
    indO = find(~L(ngh),1,'first'); %selecting free spot
    if ~isempty(indO) %if there is still a free spot
        L(ngh(indO)) = true; %updating occupancy
        if P(ii) %proliferation
            cells = [cells uint32(ngh(indO))]; %adding new cell
            if Ps(ii) %symmetric division
               cellsIsStem = [cellsIsStem true];
               cellsPmax = [cellsPmax cellsPmax(indxF(ii))];
            else
               cellsIsStem = [cellsIsStem false];
               cellsPmax = [cellsPmax cellsPmax(indxF(ii))-1];
               if ~cellsIsStem(indxF(ii))
                cellsPmax(indxF(ii)) = cellsPmax(indxF(ii))-1;
               end
            end
        else %migration
            L(cells(indxF(ii))) = false; %freeing spot
            cells(indxF(ii)) = uint32(ngh(indO));
        end
    end
end
```

At the end we need to erase cells that died.

```
    if ~isempty(del) %updating death
        L(cells(indxF(del))) = false;
        cells(indxF(del)) = [];
        cellsIsStem(indxF(del)) = [];
        cellsPmax(indxF(del)) = [];
    end
end
```

This is it. Whole CA is coded and we are ready to run simulations!

In order to visualize simulations results we can implement short function.

```
function visualize( N, cells, cellsIsStem, cellsPmax, pmax )

    M = ones(N,N,3); %matrix for image
    color = hot(3*pmax); %colormap
    %drawing CCs
    M(cells(~cellsIsStem)) = color(cellsPmax(~cellsIsStem)+1,1);
    M(cells(~cellsIsStem)+N*N) = color(cellsPmax(~cellsIsStem)+1,2);
    M(cells(~cellsIsStem)+2*N*N) = color(cellsPmax(~cellsIsStem)+1,3)
    %drawing CSCs, we want to draw them as 3x3 points
    aux = int32([-N-1 -N -N+1 -1 1 N-1 N N+1])';
    CSCs = cells(cellsIsStem);
    plusSurr = bsxfun(@plus,CSCs,aux);
    M(plusSurr) = color(2*pmax,1);
    M(plusSurr+N*N) = color(2*pmax,2);
    M(plusSurr+2*N*N) = color(2*pmax,3);

    figure(1)
    imshow(M);
end
```

Below is the visualization of the tumor simulated using the above code. It is consisted of 369,504 cells in total (8563 CSCs). The whole simulation took about 12 minutes on my laptop, what taking into account the lattice size and number of cells is a reasonable time.

(https://computecancer.files.wordpress.com/2015/05/higherps1.png)