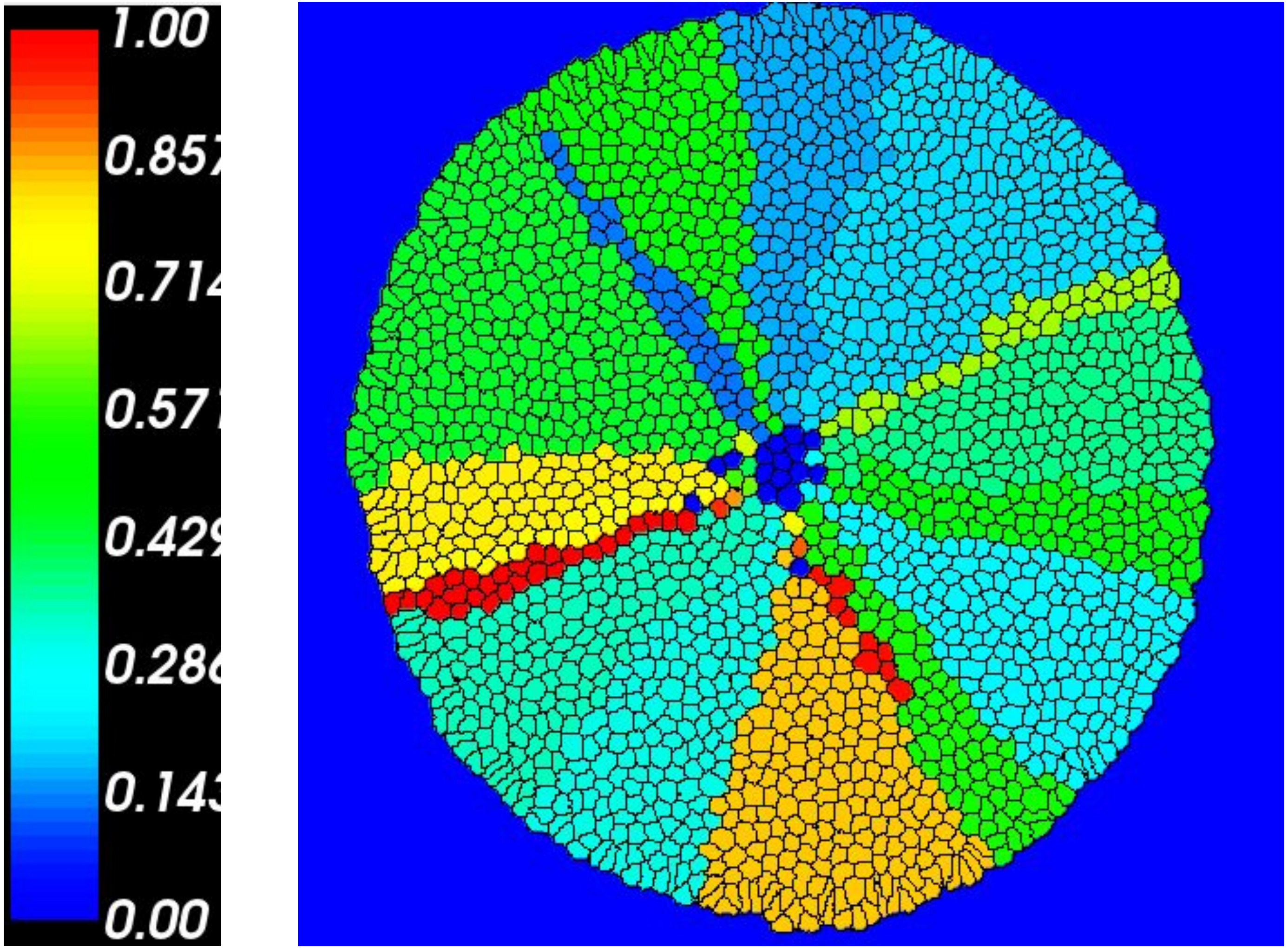
Cell Lineage Maps


1.00
0.857
0.714
0.571
0.429
0.286
0.143
0.00

Cell Lineages

1. In tumor formation it becomes important to keep track of cell lineages
2. In this case we explore lineage patterns and size of lineage clusters by exploring cellular behaviours of:
    a. Growth:
    b. Division
    c. Death

3. We can then ask the question of what differences in patterns and lineage size can arise due to differences in describing the properties above

Initializing Cell Properties

Initializing the first mother cell and her properties. We can do this in the start function of any class:

```
def start(self):
    ###defining center of the lattice (xpos,ypos) to create the mother cell
    xpos=(self.dim.x)/2.
    ypos=(self.dim.y)/2.
    ###self.newCell creates a newCell
    stem_cell=self.newCell(self.STEM)
    ###self.cellField places the cell in the lattice
    self.cellField[xpos-self.cell_size:xpos+self.cell_size-1,ypos-self.cell_size:ypos+self.cell_size-1,0]=stem_cell
    ###set the cell's targetVolume and lambdaVolume
    stem_cell.targetVolume=self.targetCell ###These values can be defined in tha main python file as well
    stem_cell.lambdaVolume=self.lambdacell
    ###creating a dictionary attribute for tracking lineage cell ids
    stem_cell.dict['list'] = [stem_cell.id]
    #creating a dictionary attribute for tracking division time
    stem_cell.dict['list_time']=0
    #creating a dictionary attribute for tracking number of divisions
    stem_cell.dict['num_div']=0
    #creating a dictionary attribute for tracking color
    stem_cell.dict['color']=0
```

Assigning Growth Properties to Cells

- We can allow cells to either grow uniformly at each mcs:

```
for cell in self.cellListByType(self.STEM):
    cell.targetVolume+=0.5
```

- Or apply a simple version of ‘contact inhibited growth’ in which only cells on the surface grow:

```
for cell in self.cellListByType(self.STEM):
    for neighbor in self.getCellNeighbors(cell): ###To find cells only in contact with medium, loop over
        if(not neighbor.neighborAddress):    ### cell neighbor array
            cell.targetVolume+=1.0
```

Cell Division and Updating Daughter Cell's Properties

- Cells will divide once they twice of it's target volume:
    - ```
      for cell in self.cellListByType(self.STEM):
          if cell.volume > 2.0*self.targetCell:
              cells_to_divide.append(cell)
      ```
- Cells can divide in any random orientation:
    - ```
      for cell in cells_to_divide:
          self.divideCellRandomOrientation(cell)
      ```
- 'UpdateAttributes' is a built-in-function which will allow the cells which divided to update their properties:
    - ```
      def updateAttributes(self):
          self.parentCell.targetVolume = self.targetcell  #Reassigning the parent Cell's target Volume
          self.cloneParent2Child() #This will copy all the parent cell's properties to the child cell
          self.childCell.dict['list'].append(self.childCell.id) #Add the child cell id to it's lineage map
          self.childCell.dict['list_time']=self.mcs #Also keep track of the cell division time
          self.childCell.dict['num_div']=self.parentCell.dict['num_div']+1 #This is the num of divisions
          self.childCell.dict['color']=self.parentCell.dict['color']
      ```
      ```

Defining Cell Death

- We can also define cell death by defining a certain cell radius and setting the target Volume of the cells inside to 0
  - ```
    for cell in self.cellListByType(self.STEM):
            dist = sqrt((xcom - cell.xCOM) ** 2 + (ycom - cell.yCOM) ** 2)
            if(dist<50):
                    cells_to_kill.append(cell.id)

    for cellid in cells_to_kill:
            cell = self.attemptFetchingCellById(cellid)
                    cell.targetVolume=0
    ```
    ```

Criteria for Selecting Cells for Display-1/4

- We can pick cells whose center of mass lie beyond a certain radius:
  - ```
    def cells_beyond_radius(self,min_radius):
        xcom=self.dim.x/2. ##defining the x center of lattice to calculate dist from
        ycom=self.dim.y/2. ##defining the y center of lattice to calculate dist from
        cell_list=[]    ##cell_list will contain all the cells picked beyond the radii
        for cell in self.cellListByType(self.STEM):
            dist=sqrt((xcom-cell.xCOM)**2+(ycom-cell.yCOM)**2) ##calculating dist from lattice center
            if(dist>=min_radius): ##only pick cells which exceed the min radius
                if (cell.id not in cell_list and cell.id != 1):##avoid id=1 since it created all daughter cells
                    cell_list.append(cell.id) ##Append all the cells chosen to the list
    return cell_list ##This function will return the list of our chosen cells
    ```
    ```

Criteria for Selecting Cells for Display-2/4

- Similar to the last function we can define cells which lie at a particular radius by looping through all angle values:
  - ```
    def cells_at_radius(self,min_radius,max_radius):
        xcom = self.dim.x / 2.  ##defining the x center of lattice to calculate dist from
        ycom = self.dim.y / 2.  ##defining the y center of lattice to calculate dist from
        r_arr = np.linspace(min_radius, max_radius, 80) ## we define range of radii between min and max values
        theta_arr = np.linspace(0, 360, 500) ##for each radius, we parse through all the angle values
        cell_list = [] ##cell_list will contain all the cells picked at the radii
        for theta in theta_arr:##loop to parse over angles
            for radius in r_arr:##for each angle, loop over radii
                xdist = xcom + radius * cos(theta * pi / 180.) ##x coordinates from center are calculated
                ydist = ycom + radius * sin(theta * pi / 180.) ##y coordinates from center are calculated
                xcoor = int(xdist) ##converted to integer values to find them on the lattice
                ycoor = int(ydist) ##converted to integer values to find them on the lattice
                pt = CompuCell.Point3D() ##defines a lattice vector in CC3D
                pt.x = xcoor ##specifying the x coordinates of the vector
                pt.y = ycoor ##specifying the y coordinates of the vector
                pt.z = 0 ##in 2d setting the z coordinates to 0
                cell = self.cellField.get(pt) ##this is to access the cell occupying the lattice point
                if (cell and cell.type != self.MEDIUM): ##this is to make sure we don't choose a medium cell
                    if (cell.id not in cell_list and cell.id != 1): ##also avoid picking up cell.id=1
                        cell_list.append(cell.id) ##Append all the cells chosen to the list
    return cell_list
    ```
    ```

Criteria for Selecting Cells for Display-3/4

- We can pick cells which divided within a particular time interval too:
  -
```
def cells_within_time(self,time_interval):
    cell_list=[] ## cell_list will contain all the cells which divided within the time interval
    for cell in self.cellListByType(self.STEM):
        if(time_interval-200<cell.dict['list_time']<time_interval+200): ##defining cells which divided within 200 mcs
                                                                        ##of the time interval
            if (cell.id not in cell_list and cell.id != 1): ## avoid cellid=1 since it created all daughter cells
                cell_list.append(cell.id) ##Append all the cells chosen to the list
return cell_list ##This function will return the list of our chosen cells
```

Criteria for Selecting Cells for Display-4/4

- We can pick cells which divided after a particular time interval too:
  - ```
    def cells_within_time(self,time_interval):
        cell_list=[] ## cell_list will contain all the cells which divided within the time interval
        for cell in self.cellListByType(self.STEM):
            if (cell.dict['list_time'] > time_interval ): ##defining cells which divided beyond a certain time
                if (cell.id not in cell_list and cell.id != 1):##avoid cellid=1 since it created all daughter cells
                    cell_list.append(cell.id) ##Append all the cells chosen to the list

    return cell_list ##This function will return the list of our chosen cells
    ```
    ```

Make Sure Chosen Cells Unique!

- However, using our criteria may not guarantee unique mothers. To cast out the ones picked which might have simply have mothers within our list, we can use the following function:
  - ```
    def unique_parents(self,cell_list):
        ele_to_delete=[]
        for cellid in cell_list: ##parsing through cells chosen according to criteria
            for cellid2 in cell_list: ##for each cell, parsing through the list again
                cell = self.attemptFetchingCellById(cellid2) ##this will return the cell object at a particular id
                if (cellid != cellid2 and cellid in cell.dict['list']): ##this checks any parent lineage id is present in 'list'
                    if ((cellid2 > cellid) and (cellid2 not in ele_to_delete)): ##only need to delete higher ids
                        ele_to_delete.append(cellid2) ## deleted cells append to the list called ele_to_delete

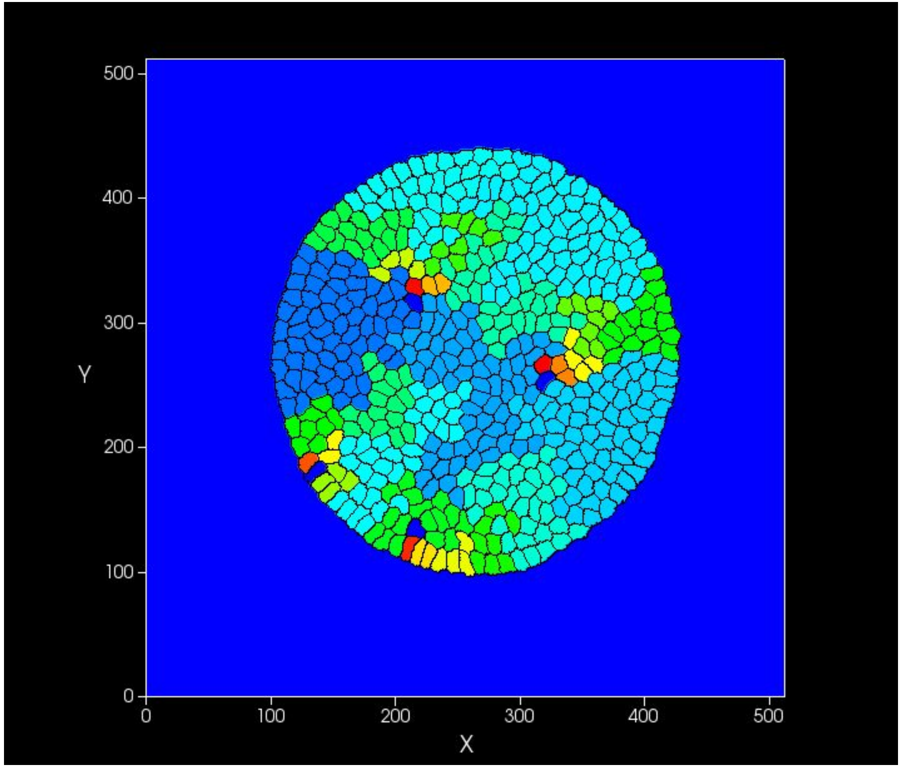
        temp_l = cell_list[:]  ##copy list by reference to avoid deleting from main list
        for cellif in ele_to_delete:
            temp_l.remove(cellif )##delete specific id by using 'remove'

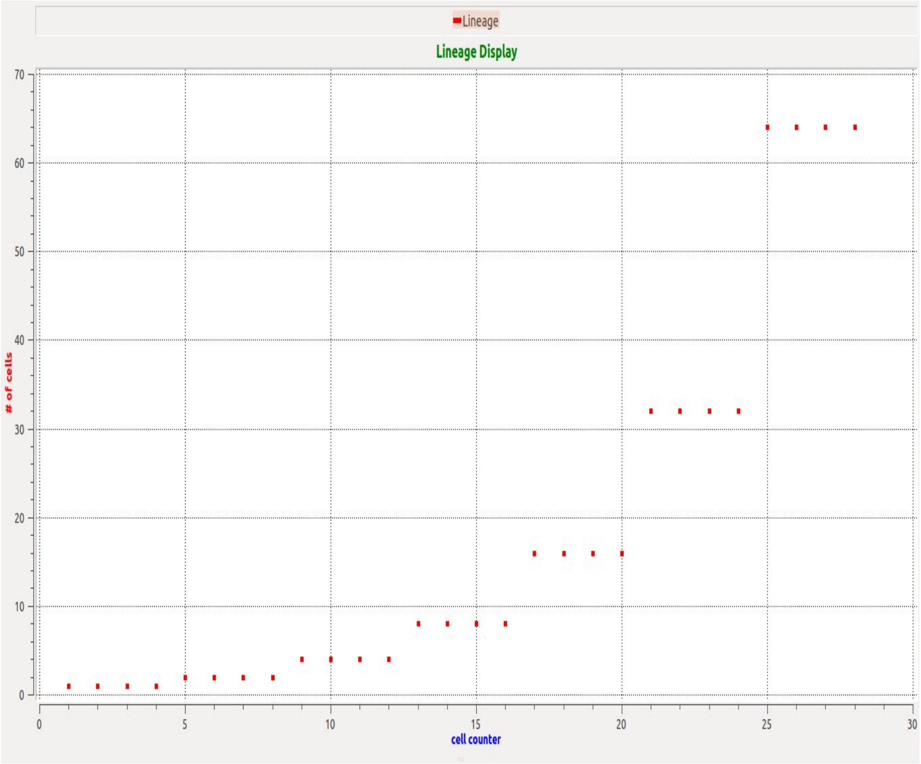
    return temp_l
    ```
    ```

Coloring Cells -1 / 2

- Now with unique cell mothers, we find all the progenitors
  - ```
    self.temp_l.sort() ##we can sort the list so lower id cells are colored in shades of blue, high red
    ll = len(self.temp_l)
    counter = 1
    num_cell_list=[]
    for id in self.temp_l:
        cell_main = self.attemptFetchingCellById(id)
        counter_c = 0
        group = []
        group.append(id)
        counter_c += 1
        for cell in self.cellListByType(self.STEM):
            if (id != cell.id and id in cell.dict['list']):
                if (cell.dict['num_div'] >= cell_main.dict['num_div']): ##Make sure we only select all daughter cells
                    group.append(cell.id)
                    counter_c += 1
        for cell2id in group:
            cell2 = self.attemptFetchingCellById(cell2id)
            cell2.dict['color']=counter / float(ll) ## This is where we update the color of each lineage.
            self.cellidField2[cell2]=cell2.dict['color']

        counter += 1
        num_cell_list.append((id,counter_c))
    ```
    ```

Coloring Cells -2 / 2

- We can also collect the lineage information to plot cluster sizes:
    - ```
      num_cell_list.sort(key=lambda tup: tup[1])
      dis_c=1
      for (id,len_grp) in num_cell_list:
          self.pW.addDataPoint('Lineage',dis_c,len_grp)
          dis_c+=1
      ```
      ```

Examples: Uniform Growth Without Death - 1/4

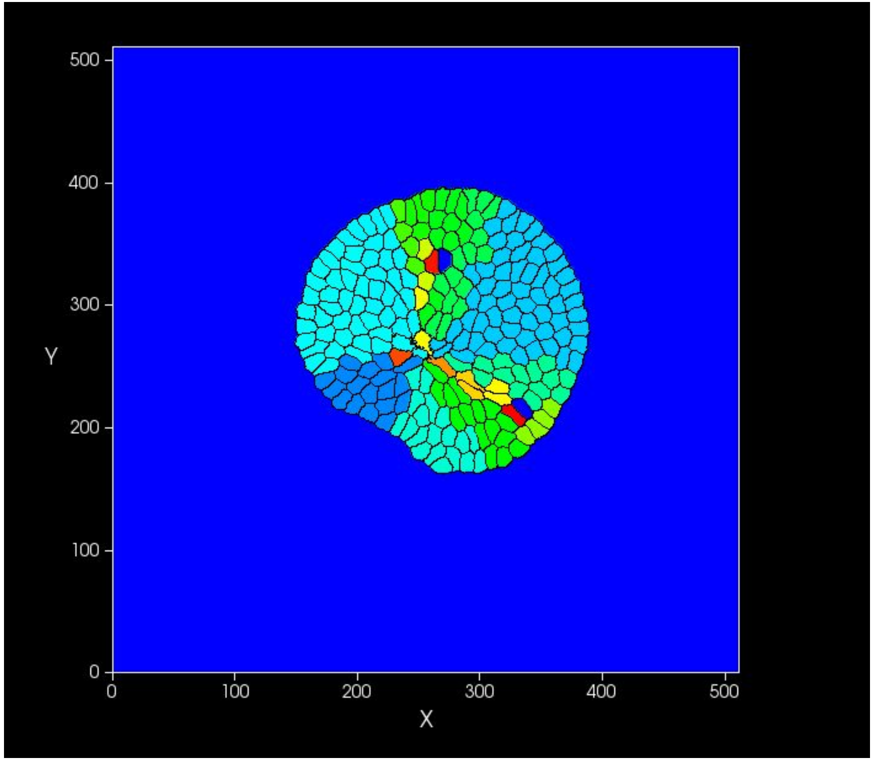




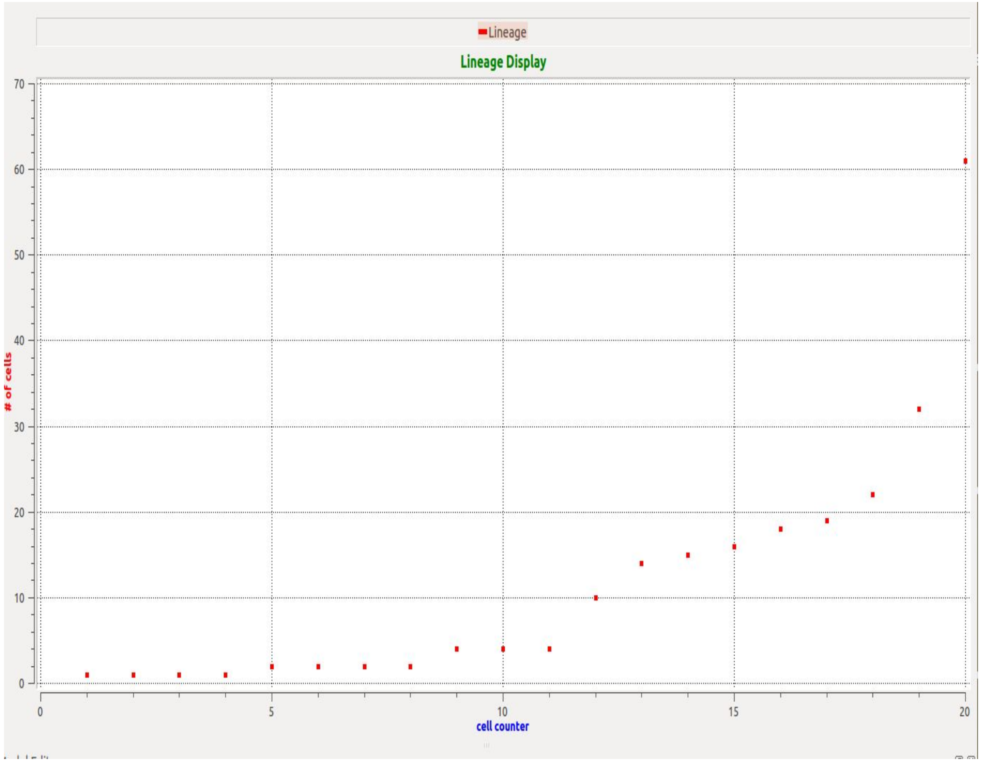
Uniform Growth after 2000 mcs. Criteria picks cells which divided after 500 mcs

Lineage Cluster Distribution for the Map

Examples: Uniform Growth With Death - 2/4

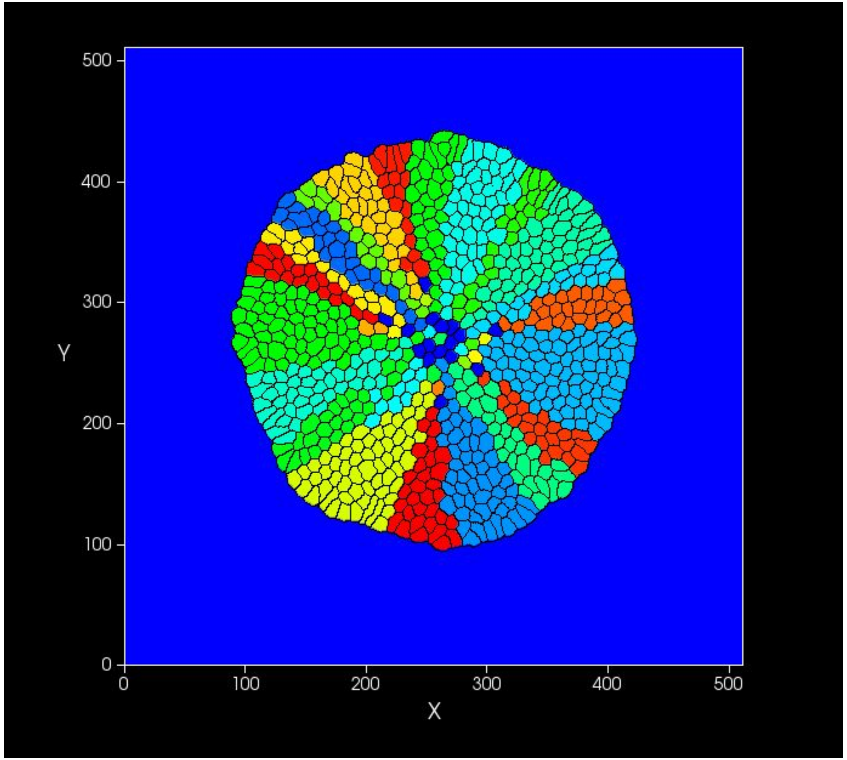


Uniform Growth after 2000 mcs with death.
Criteria picks cells which divided after 500 mcs

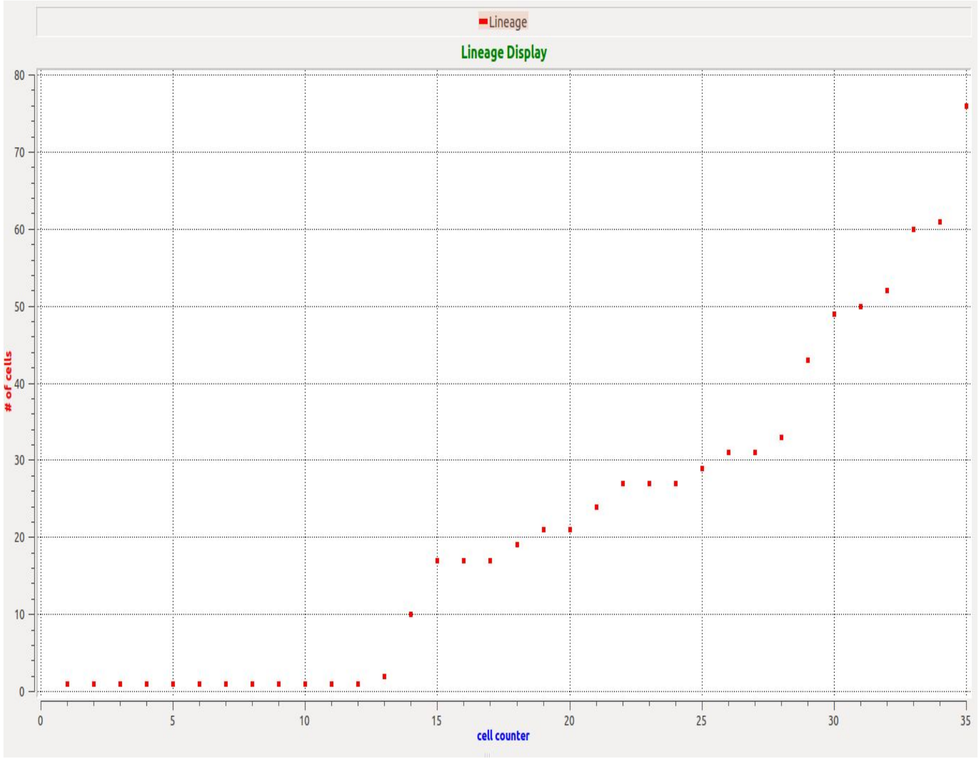


Lineage Cluster Distribution for the Map

Examples: Uniform Growth With Death - 3/4

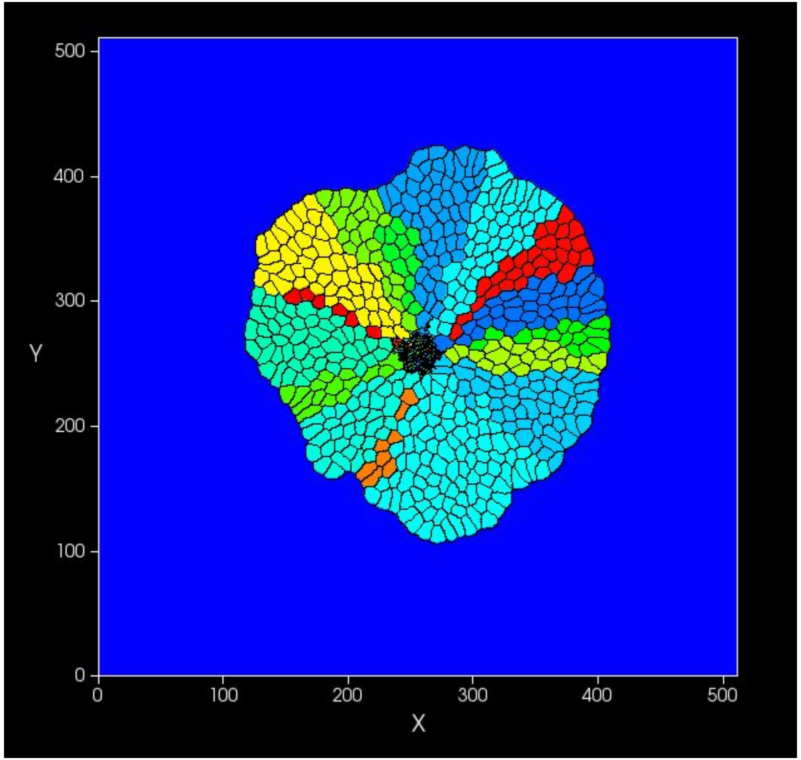


Inhibited Growth after 2000 mcs .Criteria picks cells which divided after 500 mcs

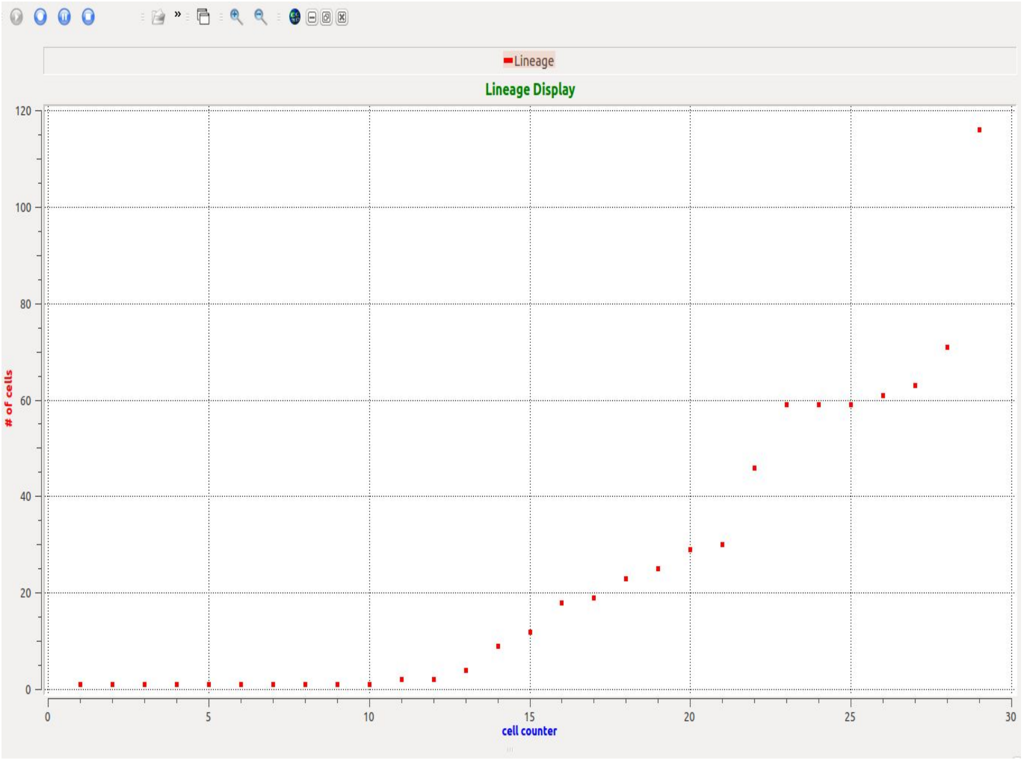


Lineage Cluster Distribution for the Map

Examples: Uniform Growth With Death - 4/4



Inhibited Growth after 2000 mcs .Criteria picks cells which divided after 500 mcs



Lineage Cluster Distribution for the Map