Explanation of DLA simulation in python

(Shilpi Bhargava)

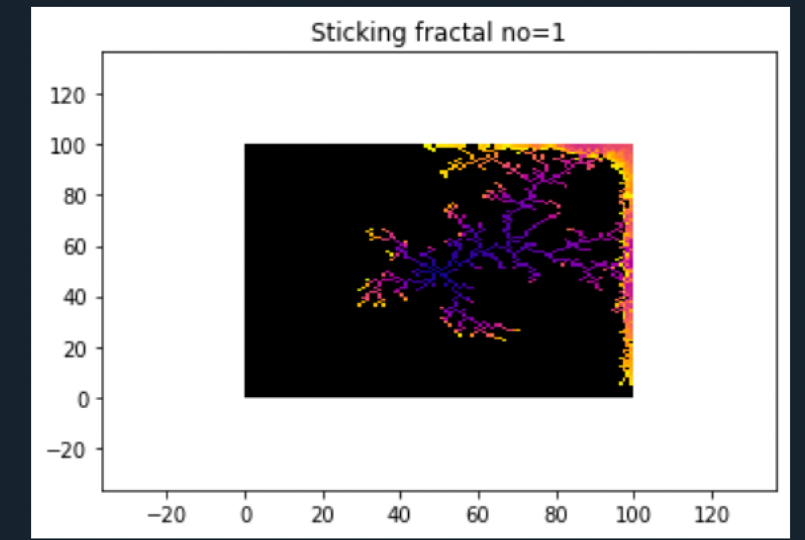
Objective: To simulate 40,000 particles with a seed particle at center of 500\*500 matrix:

* Every particle to originate at the boundary of matrix.
* Simulation to run for a given value of stickiness.

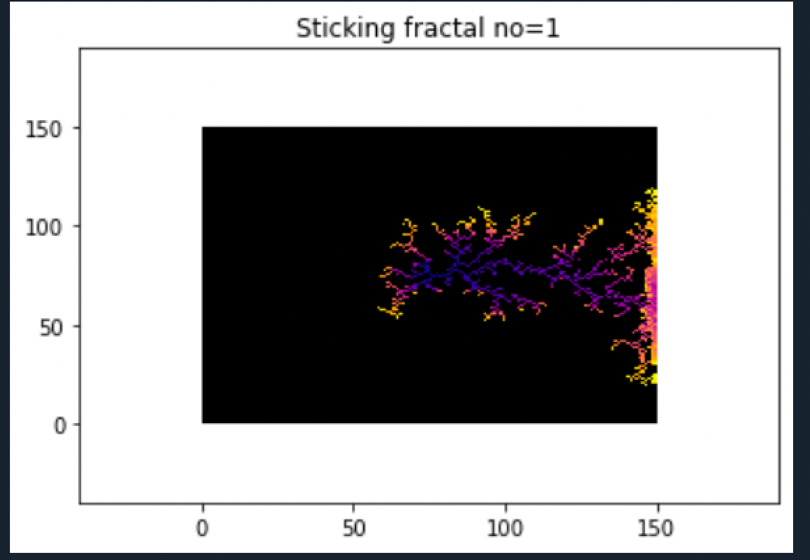
**Approach Brute- Force:**

First tried a brute-force approach, which is detailed as below but it took 2 days of run-time. Also, with this it was not fitting on 500 size matrix but a higher matrix is needed.

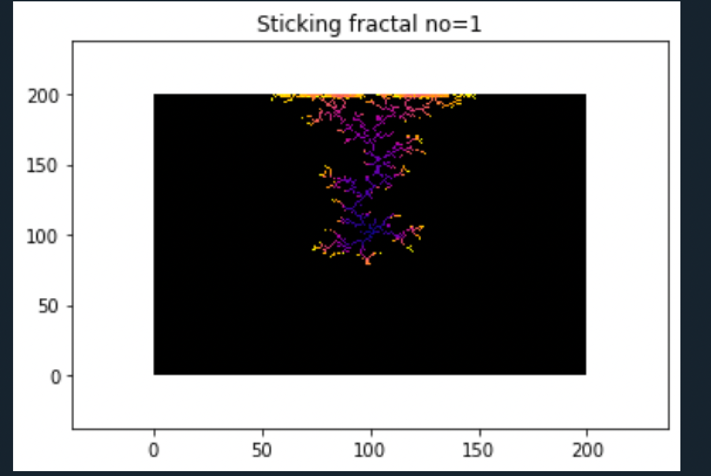
With a smaller matrix of 100X100, tried simulating 1000 particles



With same no. of particles and matrix size 150 the time taken was 1.5 min more.

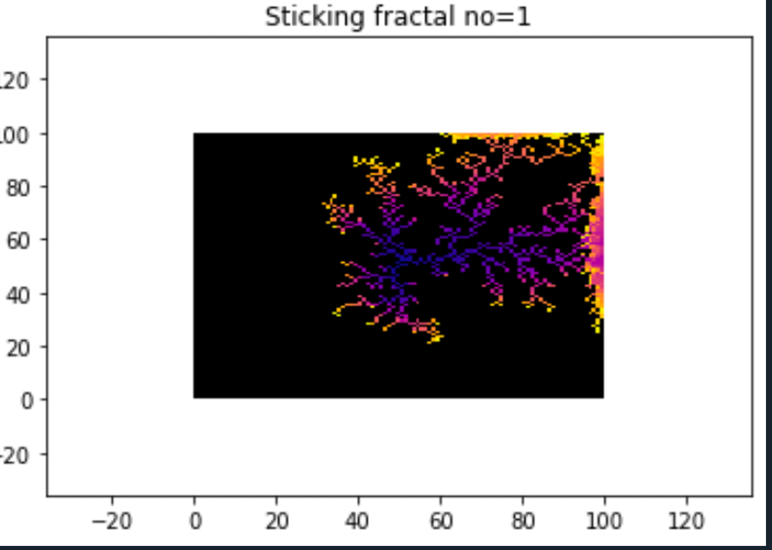


Further with 250 matrix and 1000 particles, time taken was 40 min.

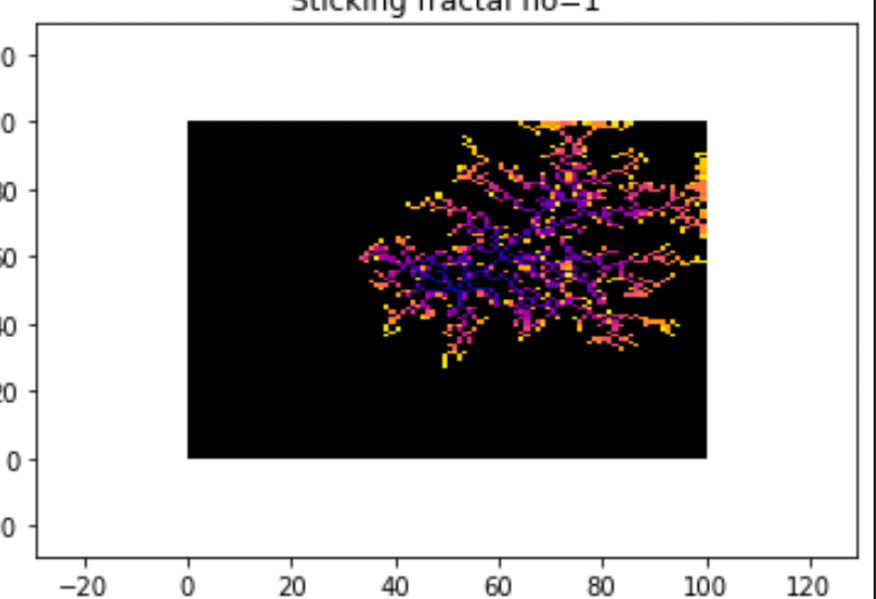


Also at matrix size = 100, and 1000 particles, below results for different stickiness:

Stickiness 0.5



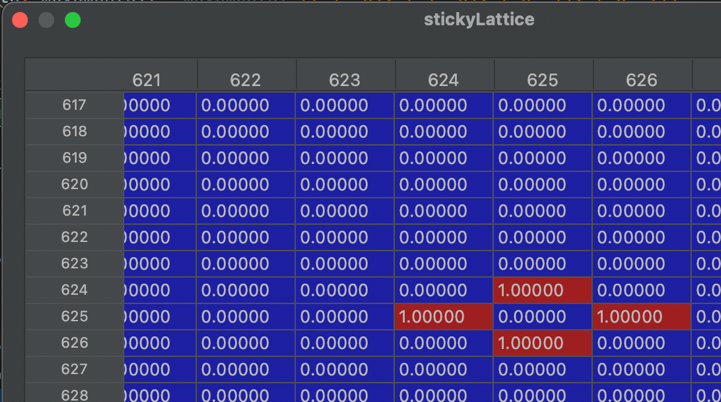
Stickiness 0.0001 (but it took 40 min to run)



--------------------------------------------------X-------------------------------------------------

Brute-Force Explanation :

**Initialize function** -- > The function takes input as length of matrix/lattice and the possible points where particles can come and get attached to an initial center point (movements). This function will create a m\*m matrix, center point is made equal to 1. It will then find cells nearby the center where new particles can come and stick. Right now its along 4 edges of initial cell (625,625) if length of matrix = 1250. The above function basically returns two matrices: Lattice and Stick lattice, Each look like this below.

After this some variables are initialized, and we enter a while loop to iterate through all 40,000 particles. Here’s what happens in while loop:

The function **pick\_starting\_position()** is called and it randomly chooses a boundary point.

Moving =1

We enter another while loop till moving = 1:

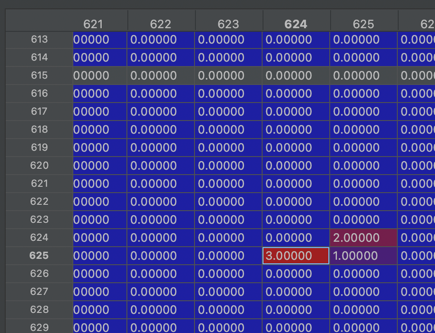
For particle no.1 the **generate\_path()** is called, it takes arguments as steps(1000) and starting point returned from above function. A particle will take 1000 steps in 4 directions randomly.It will return path, a 1000 by 2 matrix of different coordinated the particle will travel.

If in the above path we find any coordinate which matches with stickyLattice with cell value as 1, and also if prob of sticking for that particle is less than user provided threshold then **particle\_collision()** function is called;

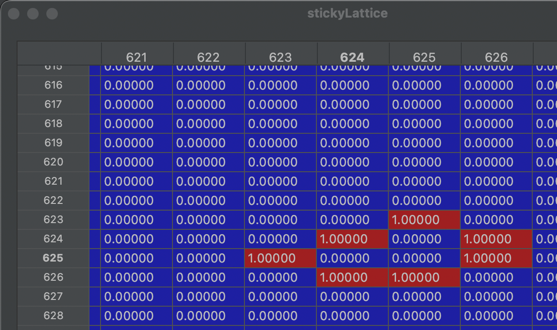
Graphical user interface, application

Description automatically generated

In this function the current state of lattice and stick lattice is passed along with for current particle the path and the x,y coordinates where the path of particle coincides with sticky sites in stickyLattice. Inside this function, the lattice matrix is updated at the cell where particle is collided, cell value equals particle number as shown below.



After this the stickyLattice is updated from image in left to image in right. Based on the lattice obtained above the neighboring cells are marked as next probable sticking points. The particle collision function returns the new lattice and stickyLattice.

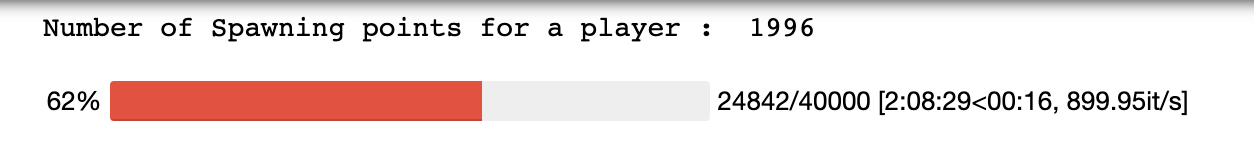
 

After this **lattice\_radius\_check()** is called, this function taken in the current lattice and updates the killradius (radius\*2). It first checks how far are all particles from the center of matrix and records the radius of farthest particles. If its greater than current set value of radius =15, then the radius and killradius gets updated.

After this the value of particle number and its lattice radius is updated in latticeRadiusData list.

Suppose in the path of 1000 steps no point is found matching the sticky sites, then a function **kill\_check()** is called. It checks the individual points in path are how far from center, if even a single point is less than killradius of 30, that particle path is not killed rather the starting point is set as the end of path.

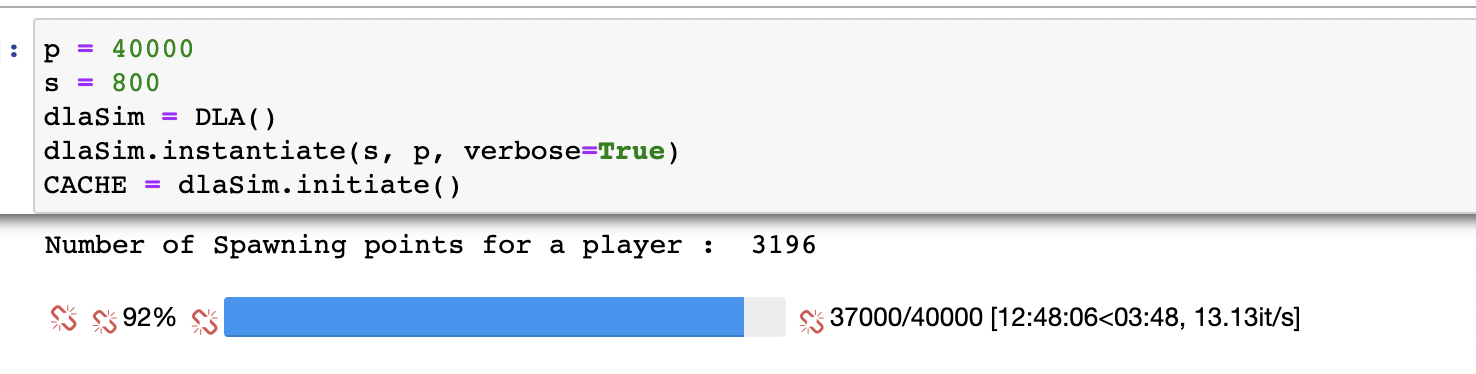
Started testing with stickiness =1, and 40,000 particles and 500 size matrix, but the program terminated at 24,842 particles because no boundary point was left. Probably with a very low stickiness it might run fine.



The above can be explained for a small sample, if particles are way more than matrix size, with above algo all boundary points gets blocked and no new can originate.



Re-run again with matrix of 800.Though with a bigger matrix from one particle to next progress is very slow. But the kernel died at 37k particles as shown below taking more than 12 hours.



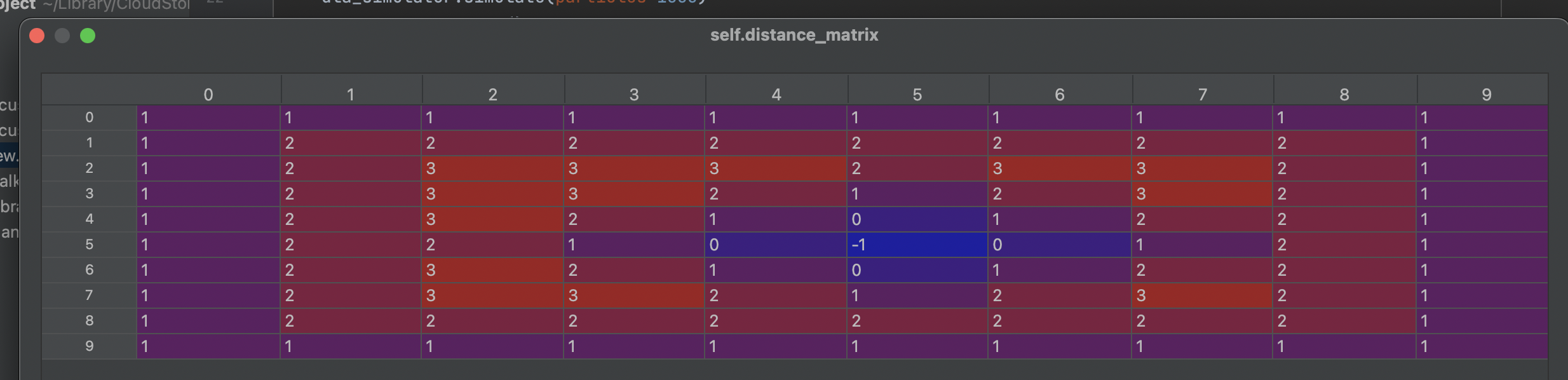
--------------------------------------------------------XXXXXXXXX--------------------------------------------------------

Second, I tried few modifications to brute force as below:

1. Pre-calculate Distance matrix is precalculated with each cell having value of how many steps a particle can take so to remain in bounding area. Brute force initializes dist matrix as all zero. This step lets dist. Matrix to update just when a particle collides and not in every loop
2. In the above particles are aggregating at boundary if they don’t find space to move.
3. In brute-force the proximity chosen is in all random directions sometimes out of bounds of matrix.

Here are the details of code:

**Initialization** : Object of class DLASimulator created, and distance matrix is initialized as below, with the max. possible steps a particle in each cell can take and value -1 for pre-occupied cell. init\_distance\_matrix() is called :

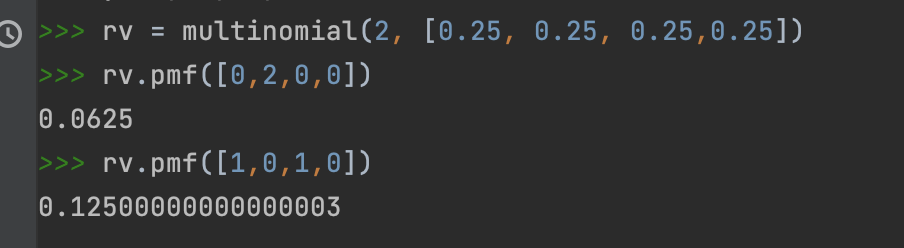


Also a probability list is created by calling generate\_list\_of\_position\_probability\_dicts(). This function makes use of multiprocessing capability of python. It will process function calc\_position\_probability\_list\_symmetric() in three 3 different process/ different cores of computer and return result. (3 -> is the max. no. of possible steps in above matrix).

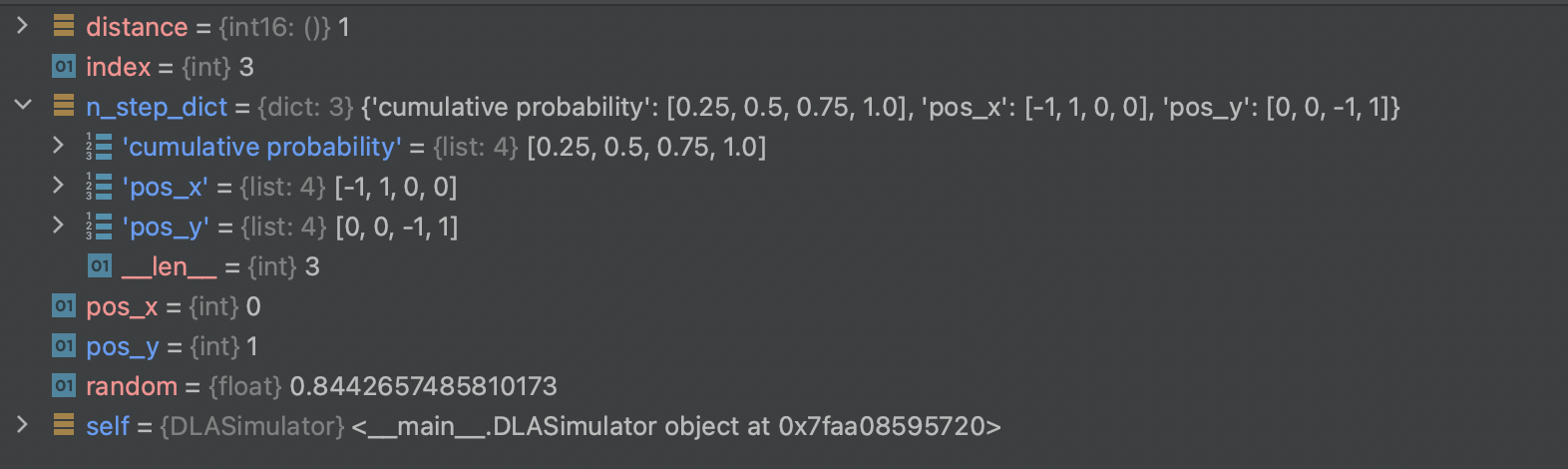
The cumulative probability for each possible step is calculated by first generating a pmf using multinomial distribution (# of trials, [prob. Of 4 possible outcomes]).

e.g for 2 steps in 2-D random walk. 9 possible outcomes are there, due to symmetry its just 3 🡪

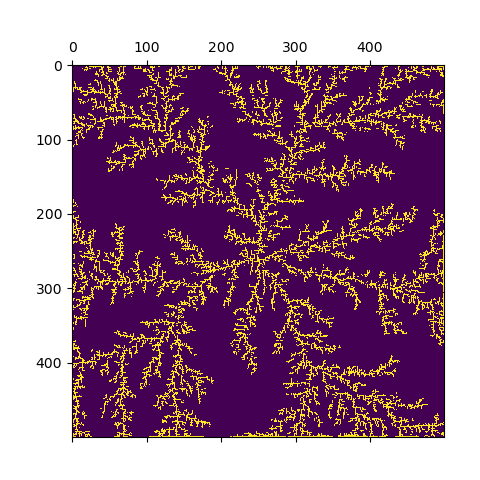
(0,0),(-2,0),(-1,-1) for these 3 the prob. For these 3 below are the probability calculated :



**Iterate through each particle:** A border point is chosen for generating particle, a random next step is generated from **get\_random\_step()** based on value of max steps that border cell can take. A point is chosen from the distribution of multinomial function based on value of max steps. E.g. random = 0.84, this probability corresponds to index 3. Hence, (0,1) is chosen.



The above process is repeated till we encounter a cell with distance zero (All potential sticky sites will have value zero -> a particle there can’t take any more steps) and that cell value is changed from zero to -1. The output with 40,000 particles is as below:



In the above implementation particles are not allowed to leave the matrix.

Although the above approach is quite fast but it’s not able to incorporate the stickiness feature, because each cell has pre-calculated steps and probability defined for each. Stickiness logic was working fine with brute-force approach.