PHYS 3142 Spring 2020

Computational Methods in Physics

Final Project

LIU, Ziao 20494087

**Project Title: Diffusion-limited aggregation（DLA）model**

**Contains:**

**I. Introduction** p1

**II. Method and Algorithm** p1~3

**III. Experiment Procedure** p3~6

**IV. Data Analysis and Problem solution** p6~8

**V. Conclusion** p8~9

**I. Introduction:**

In this project, I used Python3 and C++ to compose programs with applying Monte Carlo method to simulate the particle-random-walk in a 2D space to form the diffusion-limited aggregation (DLA) model. The basic idea of DLA model is that a reference particle is set at origin point and spawn a random-walking particle away from the origin randomly, there is a probability that the random-walking particle will attach to the recent cluster if the cluster is at its adjacent sites.

**II. Method and Algorithm:**

In the zip pack, there are 28 files. “\*.cpp” is the C++ source file. “\*.h” is the C++ header file. “\*.py” is the Python script file. “\*.csv” is the Comma-Sperated-Values file.

**final\_DLA\_Simulator.py:**

This file is for create DLA pattern.

In the file, several functions are defined to spawn a new particle, to check the adjacent site with binary data return, to check if a particle is to be kill, to generate the random walk for the particle and to check whether the particle will attach to the cluster.

For the running script part, first, the lists of particles’ coordinates are created. Then, for each particle, record the direction it can move and the status of the particle, and make it walk in the lattice randomly. If the particle is in the check range, do checks. If the particle attaches to the cluster, add its coordinates to the list and update the max R.

**final\_DLA\_Density.py:**

This file is for calculate the relation between density and the radius, and the fractal dimension.

In the file, most functions are the same as what in final\_DLA\_Simulator.py.

A class is defined for threads. In the class, DLA\_s(), which is the main part in final\_DLA\_Simulator.py, is called to get the coordinates of the cluster. Then the ana() is applied to count the particles in specific radius range. As the ana() and its parameters are common, the thread lock should be activated before it called, and when finishing calling, the thread lock should release to the system.

Here, in the running script part, 6 threads are created to finish multiple tasks simultaneously. With over 22 clusters are analyzed, calculate the density and figure out the fractal dimension.

**read\_ana.py:**

The file is used to read the density v.s. radius data file in csv type. And then, compute the relation.

**read\_cluster.py:**

The file is used to read the coordinates of the clusters and draw the fractal pattern.

**ll\_Cr.h:**

The header file is to define the link-list of n\_Cr type nodes.

Its member variables are: head node, tail node, the length of the link-list.

**n\_Cr.h:**

The header file is to define the node of C[r]. As it is the basic element in the double direction link-list, it contains the previous node, the next node and its own data, average radius and number of the particles in the radius range.

**DLA\_func.h:**

This file is the C++ edition of final\_DLA\_Density.py. It contains those functions as what in final\_DLA\_Density.py.

**main.cpp:**

This is the main program of the C++ source code.

In this file, basic const index are defined as global variable for convenience.

The global functions are fw(), to write the C[r] and r into csv files, cluster\_out(), to write the coordinates of the clusters into csv file, and tfunc(), to call the functions in DAL\_func.h.

In the main function, two link-list are created to deal with the conditions that pnn=1 and pnn=0.3.

**III. Experiment Procedure:**

1) Compute the coordinates for a cluster:

To begin with, by referring to the description of the project, I use an algorithm to compose the Python program. The pseudocode is:

*Cluster()*

*P[] are list to store position of particles in cluster*

*Add (0,0) to P //seed*

*For each particle in N, do*

*(x,y) = rand()*

*while the particle did not stick on cluster:*

*Random change (x,y)*

*If (x,y) touch P and (x,y) stick on P*

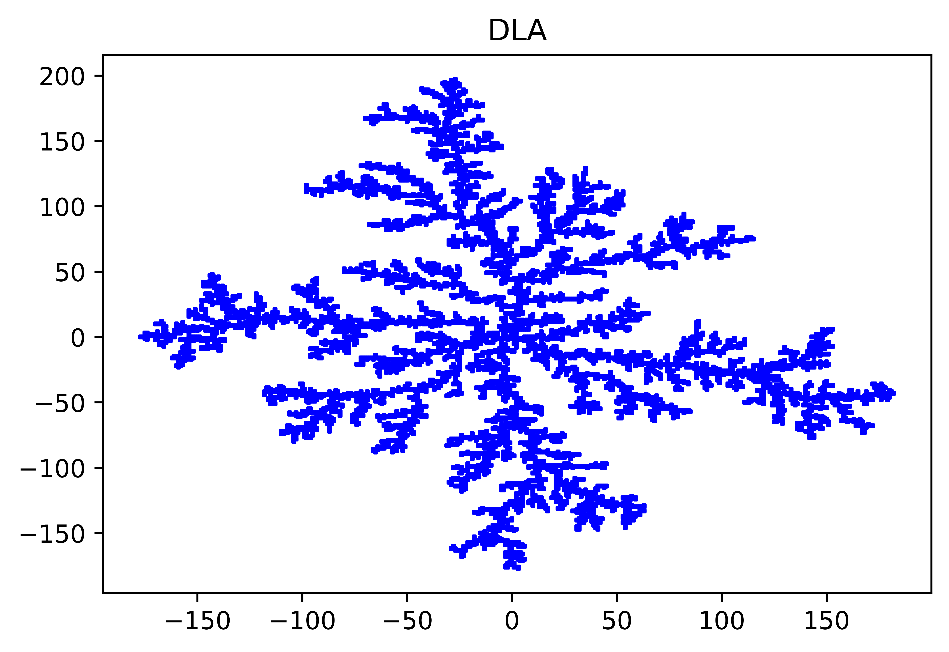
*Add (x,y) to P*

*Else if (x,y) is going to be killed*

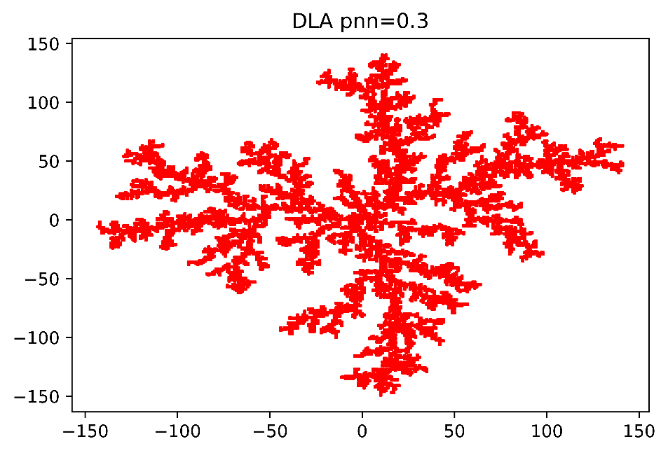
*(x,y)=rand()*

*Return P*

With this algorithm, I wrote final\_DLA\_Simulator.py. The main idea is that, spawn a particle randomly, move it randomly, check sticking randomly, and apply Monte Carlo method at where I use word “randomly”.



*final\_1.png pnn=1*



*final\_2.png pnn=0.3*

Above are two graph obtained by this Python program. By comparing them, it is obvious that when pnn=1, the occupied space of the pattern is larger. Then I can guess that with a smaller pnn value, the occupied space of the pattern will be larger.

For each pattern, the program ran about 8 hours. Especially, when over 5000 particles are add to the cluster, for the new particle, average processing time is around 4s.

2) Compute the relation between C[r] and r:

With the coordinates obtained from the previous part, by counting the number of the particles in each radius range, and normalize them with total number of particles and the volume between the spheres of the Euclidian space.

The pseudocode is:

*Density v.s. radius():*

*n=24 // more than 20 clusters*

*C is a list with all elements are 0*

*R is a list with all elements are 0*

*For i in n, do*

*P = Clusters()*

*If (P.x,P.y) in range(r-0.5, r+0.5)*

*C[r]=C[r]+1*

*For i in C.length, do*

*C[i] = ln(C[i]/(24\*N\*2\*PI\*i))*

*R[i] = ln(i)*

*Linearfit(R,C)*

With this algorithm, I composed final\_DLA\_Density.py. Then, I find a great problem, such that as the Clusters() is required around 8 hours to run for one time, there are over 20 clusters for both pnn=1 and pnn=0.3, which means that it needs 320 hours, i.e. over 13 days, to compute the relation. The funny fact is that I need to deal with this project before I submit the Assignment 11.

Then I decide to try other ways, because of the low efficiency in computing the clusters one by one. Thus I use multiple threads to process several clusters simultaneously and boost efficiency. The pseudocode of the multiple threads method is:

*MT(C,R):*

*P=Clusters()*

*Lock the thread*

*If (P.x,P.y) in range(r-0.5, r+0.5)*

*C[r]=C[r]+1*

*For i in C.length, do*

*C[i] = ln(C[i]/(24\*N\*2\*PI\*i))*

*R[i] = ln(i)*

*Unlock the thread*

*Density with MT():*

*n=24*

*C is a list with all elements are 0*

*R is a list with all elements are 0*

*while i < n, do*

*t1 = thread(MT(C,R))*

*start t1*

*t2 = thread(MT(C,R))*

*start t2*

*……*

*t6 = thread(MT(C,R))*

*start t6*

*if t1~t6 all ends*

*i=i+6*

*continue loop*

*linear fit(R,C)*

I ran this code for over 10 hours and I found that for the first 6 threads, the quickest one had just finished creating around 6000 particles. Such a low efficiency make me rewrite the code in C++.

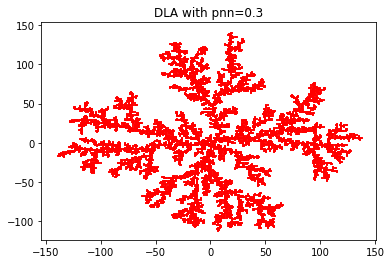
3) Monte Carlo with C++

As above, I just rewrite the code in C++ style with Microsoft Visual Studio 2019. Due to the character of C++, I change some code like applying link-list and passing pointer as parameter.

Initially, I decided use C++ multi threads as it is the real multi threads. As what I expected, the program occupy half of my CPU, which is Intel® i7-7700HQ. The efficiency of code was boosted. However, after finishing 16 cluster with pnn=1, the program hang.

Then with searching some tips, I change the program into single thread and using Mersenne Twister method to generate random number for Monte Carlo. Even with only one thread, the efficiency of the code in C++ is greater than that in Python, such that creating a cluster needs only around 20 minutes.

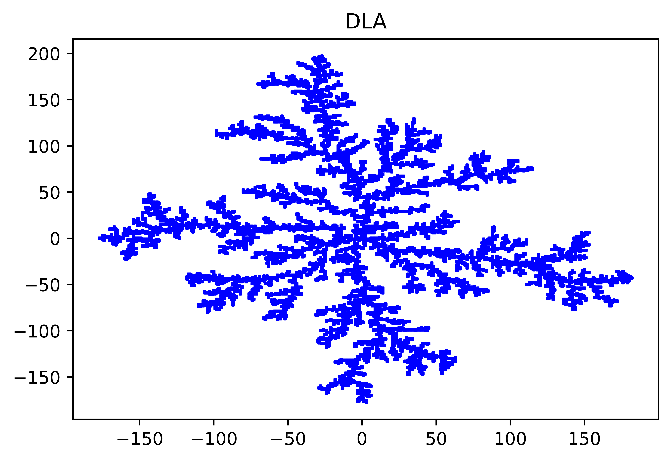
By test the correctness of the code, using python plot the fractal pattern obtained by C++.

 *final\_3.png*

It is lake what I got in Python, so it is correct.

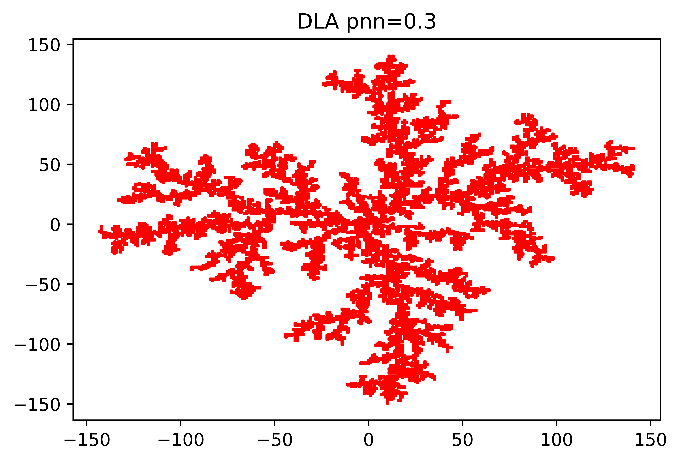
**IV. Data Analysis and Problem solution:**

1) Pattern with N=10000 and pnn=1.0



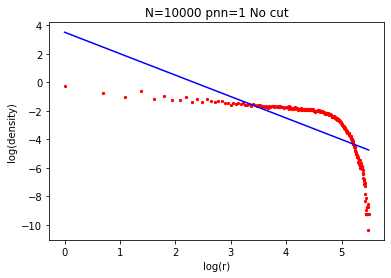
According to the graph, the range of x-axis is approximately [-180,180] and the range of y-axis is about[-180, 200].

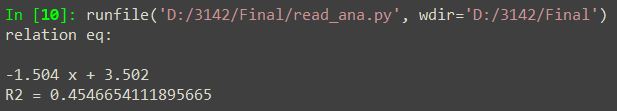
2) Pattern with N=10000 and pnn=0.3



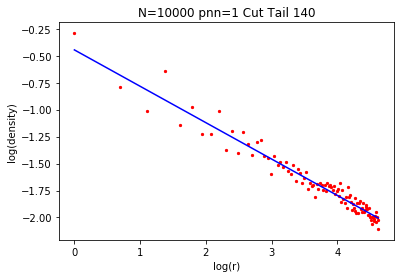
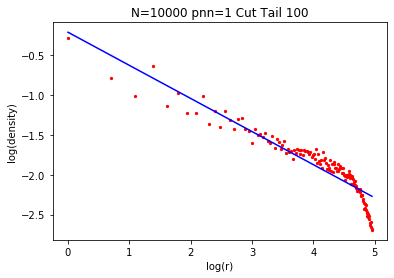
According to the graph, the range of x-axis is approximately [-150, 150] and the range of y-axis is about [-150, 150]. It is obvious that this pattern is smaller in range than the previous one.

3) here total 21 clusters are counted. Relation between density and radius. When N=10000 and pnn=1, the graphs and linear fits are: (data in output1-21.csv)

 *final4\_1.png*

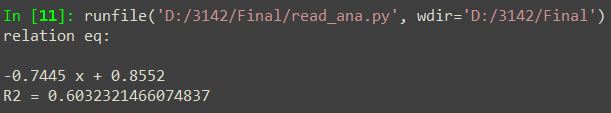
 *final4\_1o.jpg*

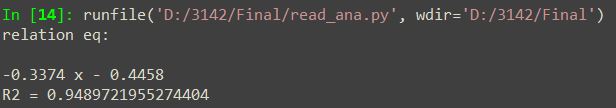
The error of the fit equation is too large, because there is a dramatic drop when ln(r)>5. So I cut the tail.



*final4\_2.png final4\_3.png*

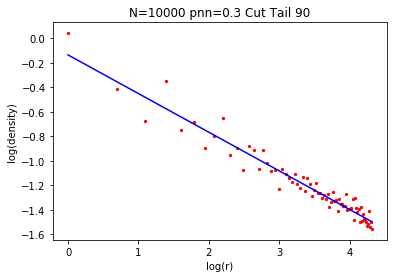
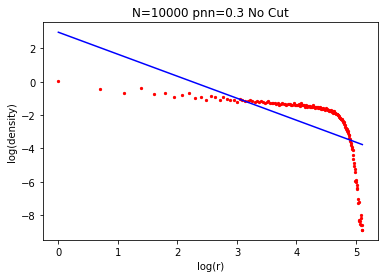
With the linear fit function:





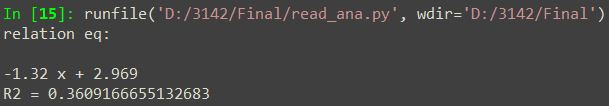
With cutting the tail, the accuracy of the relation equation gains to an acceptable value.

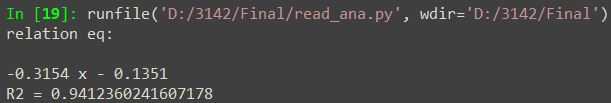
For N=10000 and pnn=0.3, the linear fit graphs are: (data in output2-21.csv)



*final5\_1.png final5\_2.png*

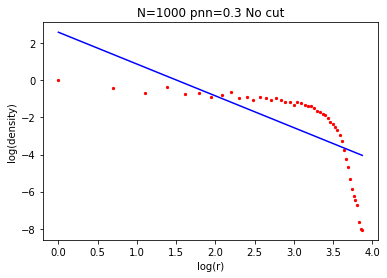
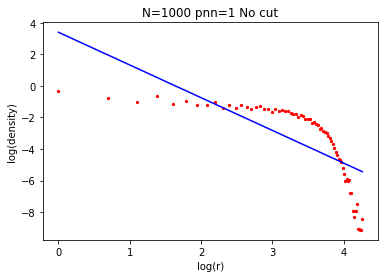
With the linear fit function:

 *final5\_1o.jpg*

 *fianl5\_2o.jpg*

The same phenomenon occurred and cut the tail to attain the more correct line.

What if N is not 10000? I change it to 1000 and the graphs are: (data in output1-21k.csv and output2-21k.csv)



*final6\_1.png final6\_2.png*

There is also tail at the end of each graph. Data in this part is the main cause that influence the error of linear fitting. When the particles random walk, the probability for a particle to stick on the region far from the origin is much lower because there are not enough particles to fulfill the region. This phenomenon is not related to the total number of particles, in other words, if there were infinite particle join the fractal, the linear fit would be perfect.

4) Due to the reason mentioned above, I choose more accurate result to calculate fractal dimension.

For N=10000, pnn=1, 21 clusters,

For N=10000, pnn=0.3, 21 clusters,

We can find that with pnn decreasing, d increases. And fractal dimension is not related to the number of particles.

**V. Conclusion:**

In this project, I used two program language to solve the DLA model with Monte Carlo method. When spawning the cluster with pnn=1, it needs more time to do so than the cluster with pnn=0.3. Then we can predict that in a range that pnn is not too small, with pnn decreasing, the time to spawn the cluster with Monte Carlo method will decline, as the reason is the other fact obtained in the project that with a smaller pnn, the occupied space of cluster is smaller. And this is due to with smaller pnn, it requires more crashes between free particle and cluster, which will offer a larger probability that the particle will move closer to the origin.

These facts are true in 2-D space. We can also raise the prediction that these facts are true in n-D space, which can also proved by applying Monte Carlo, the only difference between 2-D and n-D are that, for each particle, there are n coordinates, it can jump to 2n adjacent sites.

With these facts, I guess, for an extreme small pnn value, in n-D space, the final fractal pattern will be like an n-D sphere, but not a perfect sphere. As shown in the graphs with pnn=1 and pnn=0.3, the pattern shrinkages.

Back to the program. The reason I use two languages is that when applying Monte Carlo method, it required a large quantity of random number generating and comparison. If using Python, there is a system setting called Global Interpreter Lock (GIL), which is to lock thread globally. For every 100 ticks, GIL will release and change the thread and lock again. Therefore, even multiple threads are used, only one thread will be run during GIL is working. As Monte Carlo is a CPU-bound method, the multi-processing might work better than multi-thread.

Moreover, Python itself is an interpreted language, which contains objects with uncertain data type until the code is compiled. So the interpreter needs extra time to interpret the code and maintain the variables. C++ is a compiled language, which requires programmer to specify the data type. Meanwhile, Python is protected by Shell in system, but C++ run more directly, the reason is that Python will collect trash automatically while C++ needs manual trash collection. Generally, Python will run slower than C++. As I also want to apply multi-thread in C++, but there are some unknown bugs forcing me choose single-thread, to solve the problem requires C++ programming skills, but my time is limited so that I cannot write very complex program.

Therefore, when process physics phenomenon simulation, we may choose high efficiency tool to shorten the time.