PHYS 3142 - Final Project: Diffusion-Limited Aggregation

Wemp S. Pacheco Rodriguez

May 28, 2022

Abstract

In this report, we study the model of diffusion-limited aggregation proposed by Witten and Sander in [1] which was later developed by Meakin in [2]. We will first present examples of the cluster growth with two different sticking probabilities $P_{nn} = 1$ and $P_{nn} = 0.3$. Then, we will study the density function $C(r)$ focusing on computation of the linear relationship $\ln(C(r)) \sim -\alpha \ln(r)$. Finally, we will use the result obtained for the parameter α to calculate the fractal dimension for both sticking probabilities $P_{nn} = 1$ and $P_{nn} = 0.3$.

1 Background

The diffusion-limited aggregation theory was first proposed by Witten and Sander in 1981 in [1] motivated from previous studies of aggregates created by quench condensation of some metal vapors in [3]. The DLA theory seeks to explain the growth of a cluster of particles in a medium where diffusion constitutes the means of transport and it has been used widely to model a vast range of growth processes such as random dendritic growth and dielectric breakdown.

In contrast to the Eden growth model which simply adds particles to the lattice at random in sites adjacent to the cluster, in diffusion-limited aggregation new particles are added at a given distance from the seed particle and then proceed to undergo random walks due to Brownian motion until the collide with the cluster. This pattern produces clusters that are more sparse when compared to the compact cluster produced by the Eden model and was based on the observation that the studied metal aggregates in [3] present patterns of aggregation that resemble random walks and percolating cluster.

The original proposed algorithm assumed that particles stick to the cluster immediately after reaching a position adjacent to the cluster in the lattice as this is what is expected in the metal aggregates but variations of the algorithm use a sticking probability P_{nn} in order to decide when the particle sticks to the cluster.

Most of our calculations in this report will be focused on the comparison of the produced cluster between the sticking probabilities $P_{nn} = 1$ and $P_{nn} = 0.3$. In particular we will be comparing the calculated fractal dimension D obtained from the density-density correlation function $C(r)$ for each of these two values of the sticking probability.

2 The Diffusion-Limited Aggregation Model

Our algorithm for the two-dimensional diffusion-limited aggregation model starts by setting a seed particle in the middle of the lattice. Then, we will aggregate particles into the cluster 1-by-1. For each particle, we start the particle off at random cell at distance $R_{max} + 5$ from the seed and then let it undergo a random walk by jumping to one of its 4 nearest neighbor cells with equal probability. If the particle reaches a position adjacent to the cluster we let it become glued to the cluster with probability P_{nn} or continue its random walk otherwise. If at any point the particle leaves the circle centered at the seed particle with radius $3R_{max} + 10$ we will kill that particle and repeat the process (the +10 adjustment is to account for the initial state $R_{max} = 0$). Note that for the purpose of optimization we only check neighbor cell when the distance from the particle to the seed reaches $R_{max} + 2$ and when we check for filled neighbors we also compute the possible cells for the next jump of the particle.

2.1 Simulation Results

For the two-dimensional cluster model with $N = 10,000$ particles, we obtain the following two clusters for sticking probabilities $P_{nn} = 1$ and $P_{nn} = 0.3$ respectively.

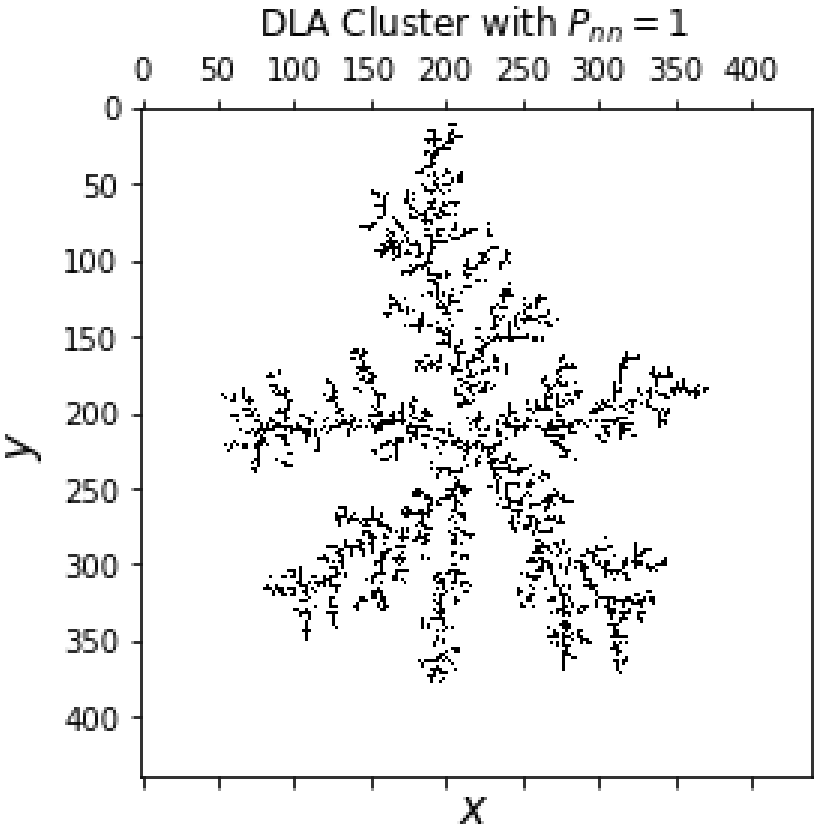

Figure 1: DLA simulation with $P_{nn} = 1$

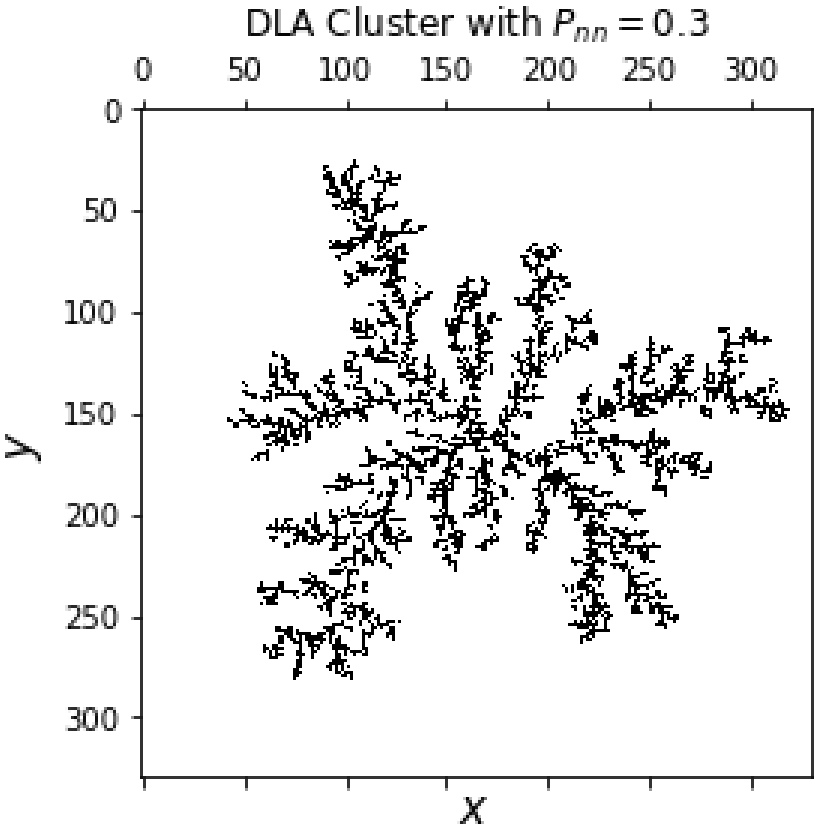

Figure 2: DLA simulation with $P_{nn} = 0.3$

3 The Density Function $C(r)$

We define the density function $C(r)$ as the number of particles at distance r from the reference particle for all possible reference particles in the cluster, this value is normalized by the number of particles N and the area on the lattice of positions corresponding to radius r, that is, normalized over the possible directions.

The computation of the normalized density function $C(r)$ is done using the Monte-Carlo method by first computing for each reference point v the value of the sum over all cells in the lattice at distance r from the reference particle v:

$$C_v(r) = \frac{1}{V(r)} \sum_{x \in \{\text{cells at distance } r\}} \rho(v)\rho(x)$$

where the volume $V(r)$ is given by the number of cell in the lattice at distance r from v and for each position v in the lattice we defined $\rho(v) = 1$ if the cell v holds a particle and $\rho(v) = 0$ otherwise. Then, the normalized density function $C(r)$ is obtained by computing

$$C(r) = \frac{1}{N} \sum_{v \in \text{lattice}} C_v(r)$$

so that the resulting density-density correlation function $C(r)$ has been normalized over both directions and particles.

3.1 Relationship between $\ln(C(r))$ and $\ln(r)$

We note that a linear relationship between $\ln(C(r))$ and $\ln(r)$ is observed only for sufficiently small $r \ll R_{max}$ as evident from the following plots obtained from the average of 20 simulations.

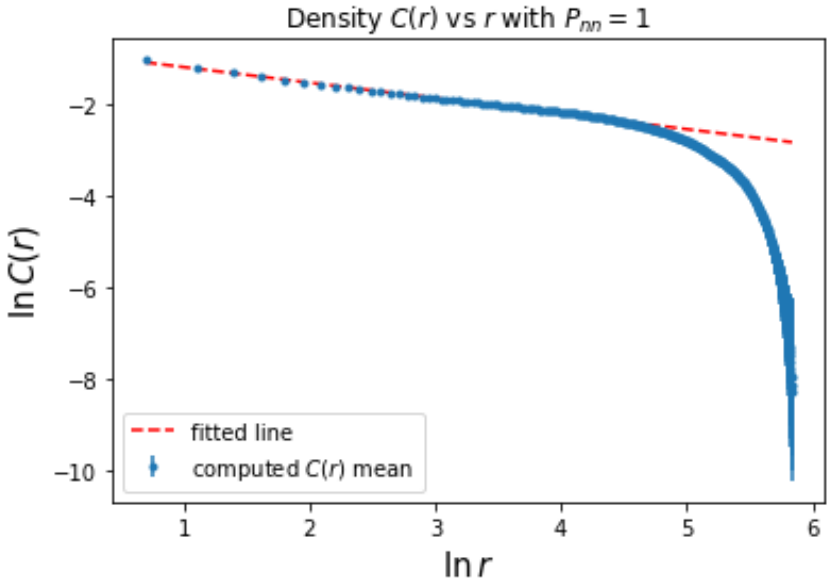

Figure 3: Average $\ln(C(r))$ vs $\ln(r)$ for $P_{nn} = 1$

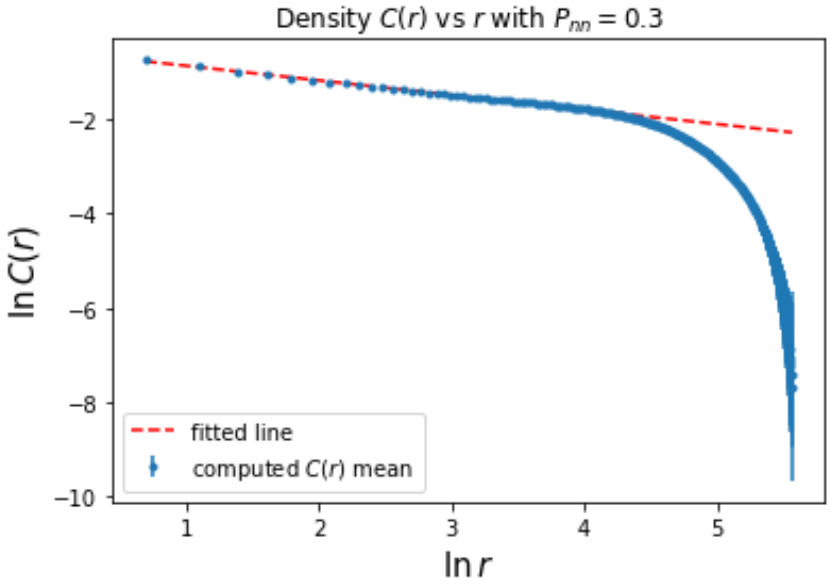

Figure 4: Average $\ln(C(r))$ vs $\ln(r)$ for $P_{nn} = 0.3$

For this reason we limit our analysis to $2 \leq r \leq e^{3.5} \approxeq 33$. Now, in order to estimate the relation between $\ln(C(r))$ and $\ln(r)$, we run 20 simulations for each sticking probability, note that we use parallel programming in numba to thread the simulations, and for each simulation we do a linear regression on the pair $(\ln(r), \ln(C(r)))$. By averaging the resulting slope over all the 20 simulations for each sticking probability $P_n n$, we obtain the relationships

$$\ln(C(r)) \sim (-0.339 \pm 0.024)\ln(r) \quad \text{for } P_{nn} = 1$$
$$\ln(C(r)) \sim (-0.309 \pm 0.017)\ln(r) \quad \text{for } P_{nn} = 0.3$$

We observe that, for the case $P_{nn} = 1$, this estimate coincides with the one obtained in [1] where they show $C(r) \sim r^{-0.343\pm0.004}$. The average $\ln(C(r))$ obtained from our simulations and their corresponding standard deviations can be observed in the following log-log plots:

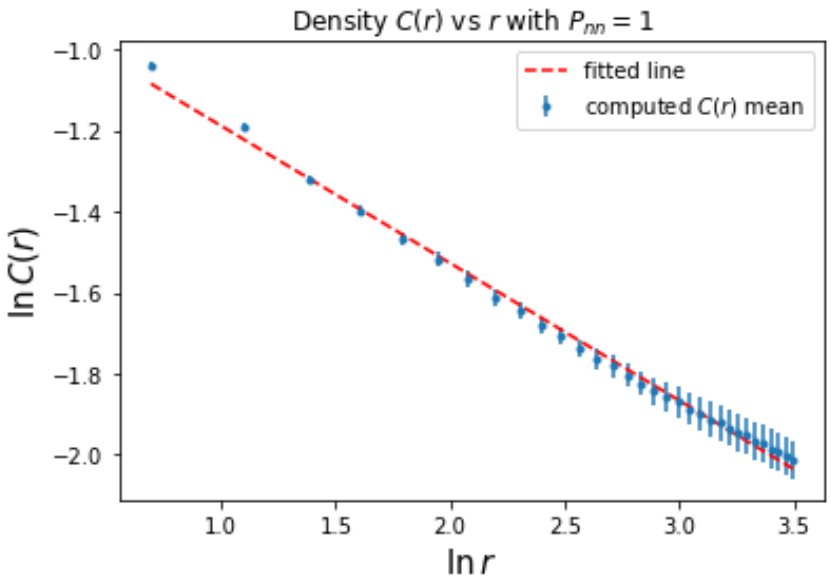

Figure 5: Linear relation between $\ln(C(r))$ and $\ln(r)$ for $P_{nn} = 1$

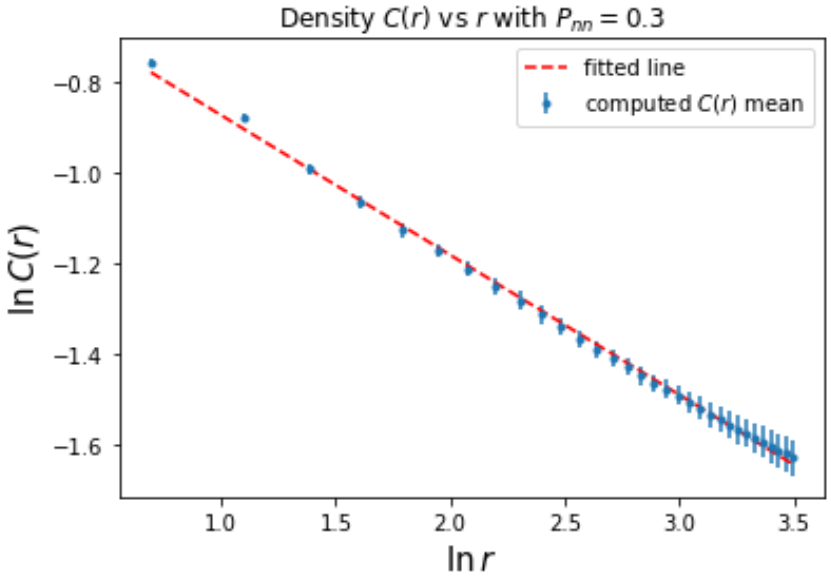

Figure 6: Linear relation between $\ln(C(r))$ and $\ln(r)$ for $P_{nn} = 0.3$

4 The Fractal Dimension

In [1] it was shown that the density function $C(r)$ follows the relation

$$C(r) \sim r^{-\alpha}$$

with $\alpha = d - D$ where d and D are the Euclidean and fractal dimension respectively. From our results in the previous section section we know that for $P_{nn} = 1$ and $P_{nn} = 0.3$

$$\begin{aligned} C(r) &\sim r^{-0.339} \quad \text{for } P_{nn} = 1 \\ C(r) &\sim r^{-0.309} \quad \text{for } P_{nn} = 0.3 \end{aligned}$$

Thus, we obtain the values for the parameter $\alpha_1 \approxeq 0.339$ and $\alpha_{0.3} \approxeq 0.309$ respectively. Therefore, the fractal dimension in each case is given by

$$\begin{aligned} D = d - \alpha_1 = 2 - 0.339 = 1.661 \quad \text{for } P_{nn} = 1 \\ D = d - \alpha_{0.3} = 2 - 0.309 = 1.691 \quad \text{for } P_{nn} = 1 \end{aligned}$$

5 Discussion

From the simulations shown in figures 1 and 2 it appears that the density-density correlation function should change to an observable degree according to the sticking coefficient P_{nn}. Moreover, from our results in the previous two sections we indeed obtain evidence that the fractal dimension of the aggregate increases with a lower sticking coefficient. It would be interesting to explore the nature of the relation in more details, and study whether the relation also holds in some form for the multiple-seed DLA model variation which uses a set of initial seeds instead of an unique seed particle.

References

[1] Thomas A Witten Jr and Leonard M Sander. Diffusion-limited aggregation, a kinetic critical phenomenon. *Physical review letters*, 47(19):1400, 1981.

[2] Paul Meakin, Jens Feder, Torstein Jo, et al. Radially biased diffusion-limited aggregation. *Physical Review A*, 43(4):1952, 1991.

[3] Stephen R Forrest and Thomas A Witten Jr. Long-range correlations in smoke-particle aggregates. *Journal of Physics A: Mathematical and General*, 12(5):L109, 1979.

A Source Code

```
import numpy as np
import matplotlib.pyplot as plt
import numba

# Initial Parameters
P_nn = 1 # Sticking probability at 1st neighbor
P_sn = 0 # Sticking probability at 2nd neighbor
R_max = 1 # Current maximum distance from seed
dR_ch = 2 # We start checking neighbors at R_max + dR_ch from seed
dR_new = 5 # We start particles at R_max + dR_new from seed
R_lim = 3 # We kill particles if further than R_lim*R_max from
                                        seed
N = 10_000 # Number of particles
size = 501 # Initial size of lattice
center = size//2 # Center Coordinate

@numba.jit(nopython=True)
def linreg(x, f, n):
    x_sum, f_sum, x2_sum, xf_sum = 0,0,0,0
    for i in range(n):
        x_sum += x[i]
        f_sum += f[i]
        x2_sum += x[i] ** 2
        xf_sum += x[i] * f[i]
    slope = (n*xf_sum - x_sum*f_sum)/(n*x2_sum - x_sum**2)
    intercept = (f_sum - slope*x_sum)/n
    return slope, intercept

@numba.jit(nopython=True)
def start_pos(R_max, dR_new):
    angle = 2*np.pi*np.random.rand() # Generate a random angle in
                                        [0,2*pi)
    R_str = R_max + dR_new # Current radius at which we span
                                        particle
    dx = round(R_str*np.cos(angle)) # x diff from center
    dy = round(R_str*np.sin(angle)) # y diff from center
    return np.asarray([dx,dy], dtype=np.float64)

@numba.jit(nopython=True)
def simulate(Lattice, size, P_nn, P_sn, dR_ch, dR_str, R_lim, N):
    # Nearest neighbors list
    to_check = [
        np.asarray([1,0],dtype=np.int64), #Up
        np.asarray([0,1],dtype=np.int64), #Right
        np.asarray([-1,0],dtype=np.int64), #Down
        np.asarray([0,-1],dtype=np.int64) #Left
    ]

    # Insert seed particle
    center = size//2 # Center of lattice = [center, center]
    c_pos = np.asarray([center,center],dtype=np.float64)
    Lattice[center][center] = True
    R_max = 0 # There is only the seed in the cluster
    R_adj = 10 # So that the first particle can move
    N -= 1

    # Insert other particles
    while(N > 0):
```

```
        # Initialize Particle
        dv = start_pos(R_max,dR_new)
        pos = center+dv
        pos = pos.astype(np.int64)
        d = np.sqrt((pos[0]-c_pos[0])**2 + (pos[1]-c_pos[1])**2)

        # Begin Random Path
        Fixed = False
        n_list = to_check # List of nearest neighbors for next
                                                    jump
        k = 4 # number of empty nearest neighbors
        while(not Fixed):
            # Make the particle jump one position
            direction = np.random.randint(k)
            pos += n_list[direction]
            # Recalculate distance
            d = np.sqrt((pos[0]-c_pos[0])**2 + (pos[1]-c_pos[1])**
                                                        2)

            # Check the current position
            if d > R_lim*R_max + R_adj: # Kill this particle, try
                                                        again
                dv = start_pos(R_max,dR_new)
                pos = center+dv
                pos = pos.astype(np.int64)
                d = np.sqrt((pos[0]-c_pos[0])**2 + (pos[1]-c_pos[1
                                                        ])**2)
            # Check if it stick to a nearest neighboor
            elif d <= R_max + dR_ch:
                n_list, k = [], 0 # Store which neighbors are
                                                        empty for
                                                        next jump
                for dv in to_check:
                    v = pos + dv
                    v = v.astype(np.int64)
                    # We have a neighbor at this cell
                    if Lattice[v[0]][v[1]]:
                        p = np.random.rand()
                        # The particle sticks with probability
                                                            P_nn
                        if p < P_nn:
                            pos = pos.astype(np.int64)
                            Lattice[pos[0]][pos[1]] = True
                            Fixed = True
                            if d > R_max:
                                R_max = round(d)
                            N -= 1
                            break
                    # Store which neigbors are empty for next jum
                    else:
                        n_list.append(dv)
                        k += 1
    return Lattice, R_max

@numba.jit(nopython=True)
def distance_mask(size):
    D_Latt = np.full((size,size), 0, dtype=np.uint64)
    c_pos = np.asarray([size//2,size//2], dtype=np.float64)
    for x in range(size):
        for y in range(size):
```

```
            pos = np.asarray([x,y], dtype=np.float64)
            D_Latt[y][x] = round(np.sqrt((pos[0]-c_pos[0])**2 + (
                                                    pos[1]-c_pos[1])
                                                    **2))
    return D_Latt

@numba.jit(nopython=True)
def density(Lattice, R_max, size, N):
    # Init the density function C(r) array
    C_len = 2*R_max+1 # Any two points have distance at most 2*
                                            R_max
    C = np.zeros(C_len, dtype=np.float64)
    # For all reference points
    for cx in range(size):
        for cy in range(size):
            if Lattice[cy][cx]:
                # Init number of particles at distance r
                C_ref = np.zeros(C_len, dtype=np.float64)
                # Init estimate of the area of the disk (r-0.5,r+0
                                                        .5)
                T = np.zeros(C_len, dtype=np.float64)

                for x in range(size):
                    for y in range(size):
                        r = round(np.sqrt((x-cx)**2 + (y-cy)**2))
                        if r < C_len:
                            T[r] +=1
                        if Lattice[y][x]:
                            C_ref[r] += 1

                C += C_ref/T # Normalize by area of the disk (r-0.
                                                    5,r+0.5)

    for r in range(1,C_len):
        if C[r] == 0: # Check the endpoint
            C_len = r;
            break
    C /= N # Normalize by number of particles
    return C[2:], C_len-2 # We only take r > 1 for the log-log
                                            plot

@numba.jit(nopython=True,parallel=True)
def density_avg(trials,size, P_nn, P_sn, dR_ch, dR_new, R_lim, N):
    # Initialize Average
    max_d = int(np.ceil(np.sqrt(2*size**2)))
    C_arr = np.zeros((trials,max_d), dtype=np.float64)
    C_len_arr = np.zeros(trials, dtype=np.int64)
    for trial in numba.prange(trials):
        Lattice = np.full((size,size), False, dtype=numba.b1)
        Lattice, R_max = simulate(Lattice, size, P_nn, P_sn, dR_ch
                                                , dR_new, R_lim, N)
        C, C_len = density(Lattice, R_max, size, N)
        C_len_arr[trial] += C_len
        for r in range(C_len):
            C_arr[trial,r] += C[r]

    # Get the log
    C_max = np.min(C_len_arr) # We limit ourselves to the non-
                                            zeros
    C_arr = C_arr[:,:C_max]
```

```
    C_arr = np.log(C_arr)
    r_arr = np.log(np.linspace(2,C_max+1,C_max))

    # Do Linear regression on the log-log
    limit = 32 # We choose to limit the relation to r <= e^{3.5} =
                                            33
    slope = np.zeros(trials, dtype=np.float64)
    intercept = np.zeros(trials, dtype=np.float64)
    for trial in range(trials):
        slope[trial], intercept[trial] = linreg(r_arr[:limit],
                                                C_arr[trial][:limit]
                                                , limit)

    # Calculate avg ln(C(r))
    C_avg = np.zeros(C_max,dtype=np.float64)
    C_std = np.zeros(C_max,dtype=np.float64)
    for r in range(C_max):
        C_avg[r] = np.mean(C_arr[:,r])
        C_std[r] = np.std(C_arr[:,r])

    return slope.mean(), slope.std(), intercept.mean(), C_avg,
                                            C_std, r_arr

#%% Section 1
"""
Run First Simulation with P_nn = 1
"""
# Set up the lattice and run simulation
np.random.seed() # Initialize a seed
LatticeA = np.full((size,size), False, dtype=bool) # We use a
                                        boolean array for the
                                        lattice
print("Starting simulation with P_nn = 1...")
LatticeA, clip_d = simulate(LatticeA, size, P_nn, P_sn, dR_ch,
                                        dR_new, R_lim, N)
print("\tFinished simulation.")
# Clip the lattice to display the pattern
clip_d = np.ceil(clip_d + 10)
lower, upper = int(center - clip_d), int(center + clip_d + 1)
LatticeA = LatticeA[lower:upper, lower:upper]

# Plot Generation
plt.matshow(LatticeA, cmap=plt.cm.binary)
plt.title("DLA Cluster with $P_{nn} = 1$")
plt.xlabel("$x$", fontsize=15)
plt.ylabel("$y$", fontsize=15)
plt.savefig("SimulationA")

#%% Section 2
"""
Run Second Simulation with P_nn = 0.3
"""
P_nn = 0.3

# Set up the lattice and run simulation
np.random.seed() # Initialize a seed
LatticeB = np.full((size,size), False, dtype=bool) # We use a
                                        boolean array for the
                                        lattice
print("Starting simulation with P_nn = 0.3...")
```

```
LatticeB, clip_d = simulate(LatticeB, size, P_nn, P_sn, dR_ch,
                                    dR_new, R_lim, N)
print("\tFinished simulation.")

# Clip the lattice to display the pattern
clip_d = np.ceil(clip_d + 10)
lower, upper = int(center - clip_d), int(center + clip_d + 1)
LatticeB = LatticeB[lower:upper, lower:upper]

# Plot Generation
plt.figure()
plt.matshow(LatticeB, cmap=plt.cm.binary)
plt.title("DLA Cluster with $P_{nn} = 0.3$")
plt.xlabel("$x$", fontsize=15)
plt.ylabel("$y$", fontsize=15)
plt.savefig("SimulationB")

#%% Section 3
"""
Calculations of Density function
"""
trials = 20 # Number of clusters to use for calculation
print("Starting density simulations...")
for P_nn in [1,0.3]:
    print("\tStarting simulations for P_nn = {0}...".format(P_nn))

    # Calculate the average array
    slope, slope_std, intercept, C_log, C_std, r_log = density_avg
                                        (trials,size, P_nn, P_sn
                                        , dR_ch, dR_new, R_lim,
                                        N)
    limit = 32

    print("\tThe slope is:", slope, "+-", slope_std)

    # Generate plot bound by limit = 32
    plt.figure()
    plt.errorbar(r_log[:limit], C_log[:limit], yerr=C_std[:limit],
                                         fmt='.', label='
                                        computed $C(r)$ mean')
    plt.plot(r_log[:limit], intercept + slope*r_log[:limit], 'r--'
                                        , label='fitted line')
    if P_nn == 1:
        plt.title("Density $C(r)$ vs $r$ with $P_{nn} = 1$")
    else:
        plt.title("Density $C(r)$ vs $r$ with $P_{nn} = 0.3$")
    plt.xlabel("$\ln r$", fontsize=15)
    plt.ylabel("$\ln C(r)$", fontsize=15)
    plt.legend()
    name = "Density1" if P_nn == 1 else "Density3"
    plt.savefig(name)

    # Generate entire plot
    plt.figure()
    plt.errorbar(r_log, C_log, yerr=C_std, fmt='.', label='
                                        computed $C(r)$ mean')
    plt.plot(r_log, intercept + slope*r_log, 'r--', label='fitted
                                        line')
    if P_nn == 1:
        plt.title("Density $C(r)$ vs $r$ with $P_{nn} = 1$")
```

```
    else:
        plt.title("Density $C(r)$ vs $r$ with $P_{nn} = 0.3$")
    plt.xlabel("$\ln r$", fontsize=15)
    plt.ylabel("$\ln C(r)$", fontsize=15)
    plt.legend()
    name = "RawDensity1" if P_nn == 1 else "RawDensity3"
    plt.savefig(name)
```