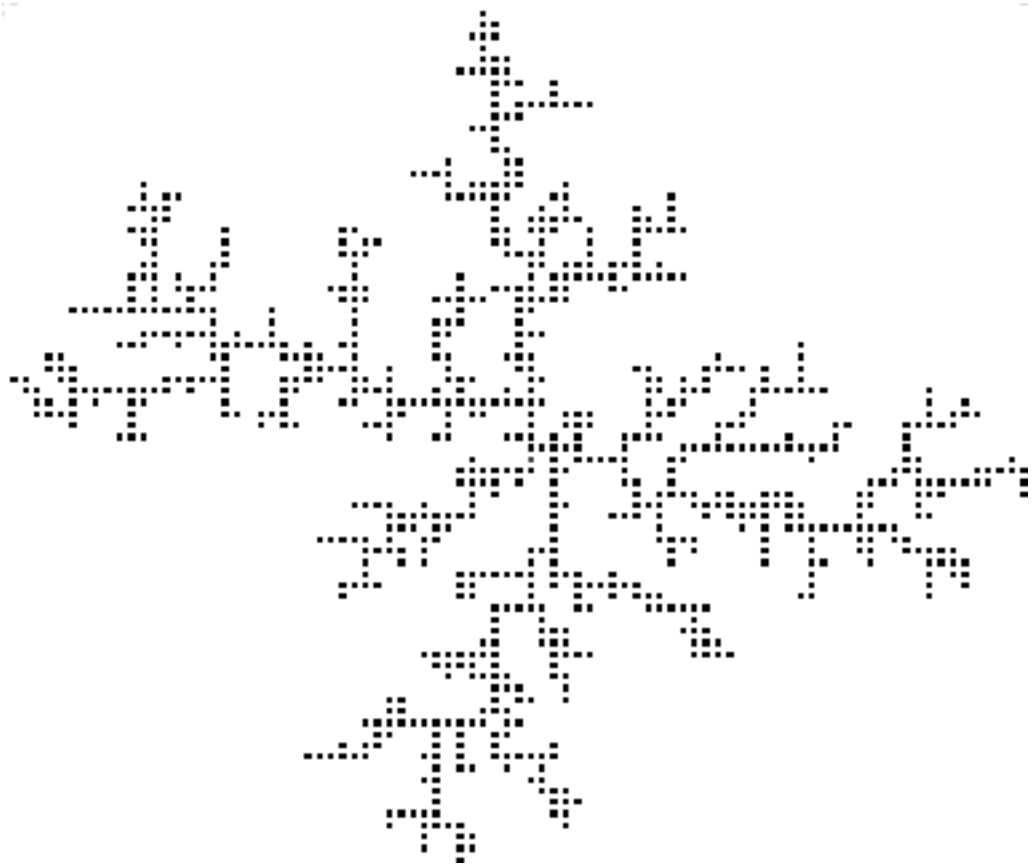

Diffusion Limited Aggregation

Souzan Hammadi

December, 2021

Contents

1 Introduction

Diffusion limited aggregation (DLA) is a growth mechanism where particles move randomly and attach to a structure as soon as they collide with it. They then grow into branched structures, much like dendrites. The difference here is that dendritic growth is somewhat deterministic i.e. the particles in the system follow a certain path due to a driving force, while DLA is occurring due to the random motion of particles.

This concept was first introduced in 1981 by Witten and Sandler [1] and it is their work that has been the instruction manual for this coding project. The code has been implemented in the python programming language. The following sections will explore this model and how it can be used.

2 Code manual

A model of a DLA process is built in the Python programming language. The python packages used are numpy and matplotlib. In short, a grid of a certain size is built. An initial seed is added to the center of the grid and a walker is randomly added at a certain distance from the center. This walker randomly moves randomly in the grid, until it either reaches the seed and attaches to it or touches the boundaries in which it leaves the

system. Another walker is then introduced, and this continues until an irregular dendritic shape starts taking form.

2.1 Build grid

The grid is built using numpy.meshgrid, 1D arrays representing the coordinates of a grid are used as input. This return three coordinate matrices; one with x values, one with y values and one with the values of the combined coordinate points. The x and y matrices are used to plot the system, while the value matrice is used as the lattice on which the random walk is taking place.

```
4   def set_grid(size,step):
5       x = np.arange(0,size+step,step)
6       y = np.arange(0,size+step,step)
7       xx, yy = np.meshgrid(x, y)
8       matrix_points = np.zeros((np.shape(yy)[0],np.shape(xx)[0]))
9       return xx, yy, matrix_points
```

An outer circle with the radius of half the box size is defined. This is the outer boundary of the system. An inner circle of a used defined radius is set, it is at the interface of the inner circle that the walkers are introduced. By giving these gridpoints certain values (now set as 0.2 and 0.5 for a nice colormap contrast), one can easily keep track of when the walker has approached the limits.

```
11  def set_outer_circle(matrix_points):
12      circle_limit = np.shape(matrix_points)[0]/2
13
14      for i in range(np.shape(matrix_points)[0]):
15          for j in range(np.shape(matrix_points)[1]):
16
17              if np.sqrt((i-circle_limit)**2+(j-circle_limit)**2) >= circle_limit:
18                  matrix_points[i][j] = 0.5
19
20      return matrix_points
21
22  def set_inner_circle(matrix_points):
23
24      circle_limit = np.shape(matrix_points)[0]/2.2
25
26      center = np.shape(matrix_points)[0]/2 - circle_limit
27
28      for i in range(np.shape(matrix_points)[0]):
29          for j in range(np.shape(matrix_points)[1]):
30
31              if np.sqrt((i-circle_limit-center)**2+(j-circle_limit-center)**2) >= circle_limit and matrix_points[i][j] != 0.5:
32                  matrix_points[i][j] = 0.2
33
34      return matrix_points
```

2.2 Add seed

A seed is added randomly close to the central of the box. How close it is to the central is determined by a limit the that user sets themselves. This is so that the nucleation point of our structure is randomized.

```
def add_initial_seed(matrix_points,size,step):
    #sets the initial seed, puts it at random somwhere close to the central
    random_point = [np.random.choice(np.shape(matrix_points)[0],1, replace=False)[0],
                    np.random.choice(np.shape(matrix_points)[0],1, replace=False)[0]]
    x_seed = xx[random_point[0]][random_point[1]]
    y_seed = yy[random_point[0]][random_point[1]]

    seed_limit_1 = (size/2)-(step*2)
    seed_limit_2 = (size/2)+(step*2)

    if seed_limit_2>x_seed>seed_limit_1 and seed_limit_2>y_seed>seed_limit_1:
        matrix_points[random_point[0]][random_point[1]] = 0.8
    else:
        while (x_seed>seed_limit_2 or x_seed<seed_limit_1) or (y_seed>seed_limit_2 or y_seed<seed_limit_1):
            random_point = [np.random.choice(np.shape(matrix_points)[0],1, replace=False)[0],
                            np.random.choice(np.shape(matrix_points)[0],1, replace=False)[0]]
            x_seed = xx[random_point[0]][random_point[1]]
            y_seed = yy[random_point[0]][random_point[1]]
        matrix_points[random_point[0]][random_point[1]] = 0.8
    return matrix_points, random_point
```

2.3 Random walk

The process can be described in five simple steps:

- A walker is introduced at the inner circle

```
def release_walker(matrix_points): #releases walker from the inner sphere vertix
    release = False

    while release == False:
        random_point = [np.random.choice(np.shape(matrix_points)[0],1, replace=False)[0],
                        np.random.choice(np.shape(matrix_points)[0],1, replace=False)[0]]
        try:
            grannar = neighbours(matrix_points, random_point)
            occurency = int(np.count_nonzero(np.array(grannar) == 0.2))
            if 0.0 in grannar and (occurency == 2 or occurency == 1):
                release = True
        except:
            continue
    return random_point
```

- There are four alternative steps, either moving up one step, moving down one step, moving to the left or to the right. The step is picked randomly.

```
def random_step(random_point):
    #there are four movements the walker can take
    scenarios = ["step_forward_x", "step_backward_x",
    "step_up_y", "step_down_y"]
    which_scenario = np.random.choice(np.shape(scenarios)[0])
    if scenarios[which_scenario] == "step_forward_x":
        random_point[0] +=1
    if scenarios[which_scenario] == "step_backward_x":
        random_point[0] -=1
    if scenarios[which_scenario] == "step_up_y":
        random_point[1] +=1
    if scenarios[which_scenario] == "step_down_y":
        random_point[1] -=1
    return random_point
```

- If the new position in the neighbouring sites are empty, then the walker is allowed to move. If there is a neighbour, then it settles

- If the walker settles, a new one is released. Otherwise it continues until it reaches a neighbour and settles there.

- If the new positions is out of the limit, i.e. within the inner and outer circle then, the process starts over with releasing a new walker.

```
for i in range(N):
    random_point = release_walker(matrix_points)
    try:
        while matrix_points[random_point[0]][random_point[1]] == 0.0:
            grannar = neighbours(matrix_points, random_point)

            if 1.0 in grannar or 0.8 in grannar: #checks if there is a neighbour == 1 or the seed == 0.8
                #now we reached our neighbour, lets settle down
                matrix_points[random_point[0]][random_point[1]] = 1.0
                particles += 1

            else:
                random_point = random_step(random_point)
                #outer circle limit
                if matrix_points[random_point[0]][random_point[1]] == 2.0:
                    #we're now out of bounds
                    raise Exception("out!")

    except Exception:
        continue
```

This process can be described by the following flowchart.

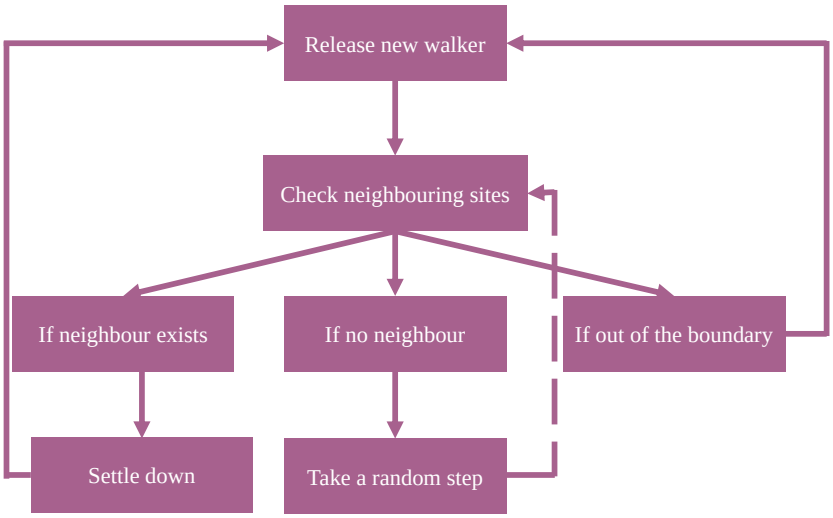

Figure 1: Flow chart of the diffusion limited aggregation algorithm.

3 Calculate fractal dimension

The fractal dimension can be calculated according to the equation given in reference [2],

$$D = \frac{log(N(\delta))}{log(1/\delta)} \tag{1}$$

, where δ is the size of the subgrids and $N(\delta)$ is the number of subgrids that are covered by the fractal. By plotting these values for different subgrid sizes, one can then then take the slope to be the fractal dimension.

```
105 def fractal_dimension(k, matrix_points):
106     div = []
107 
108     for i in range(2,k+1):
109         div.append(2**i)
110     #the pixels are in the array matrix_points
111 
112     cover = np.zeros(np.shape(div)[0])
113     inverse_delta = np.zeros(np.shape(div)[0])
114 
115     def view_as_blocks(arr, BSZ):
116     # reference: https://stackoverflow.com/questions/44782476/split-a-numpy-array-both-horizontally-and-vertically
117         # arr is input array, BSZ is block-size
118         m,n = arr.shape
119         M,N = BSZ
120         return arr.reshape(m//M, M, n//N, N).swapaxes(1,2).reshape(-1,M,N)
121 
122     for i in range(np.shape(div)[0]):
123         inverse_delta[i] += 1/div[i]
124         boxes = view_as_blocks(matrix_points,(int(div[i]),int(div[i])))
125 
126         for j in range(np.shape(boxes)[0]): #loop over all boxes
127             occupied = np.count_nonzero(boxes[j] == 1.0) #is the pixels in the grid occupied?
128             if int(occupied) > 0: #if box is occupied count it as one
129                 cover[i] += 1
130 
131     D = (np.log(cover)[1]-np.log(cover)[len(cover)-1])/(np.log(inverse_delta)[1]-np.log(inverse_delta)[len(inverse_delta)-1])
```

4 Results

Diffusion limited growth was simulated using the algorithm described in the previous section. Here, one such growth is given in Figure 2.

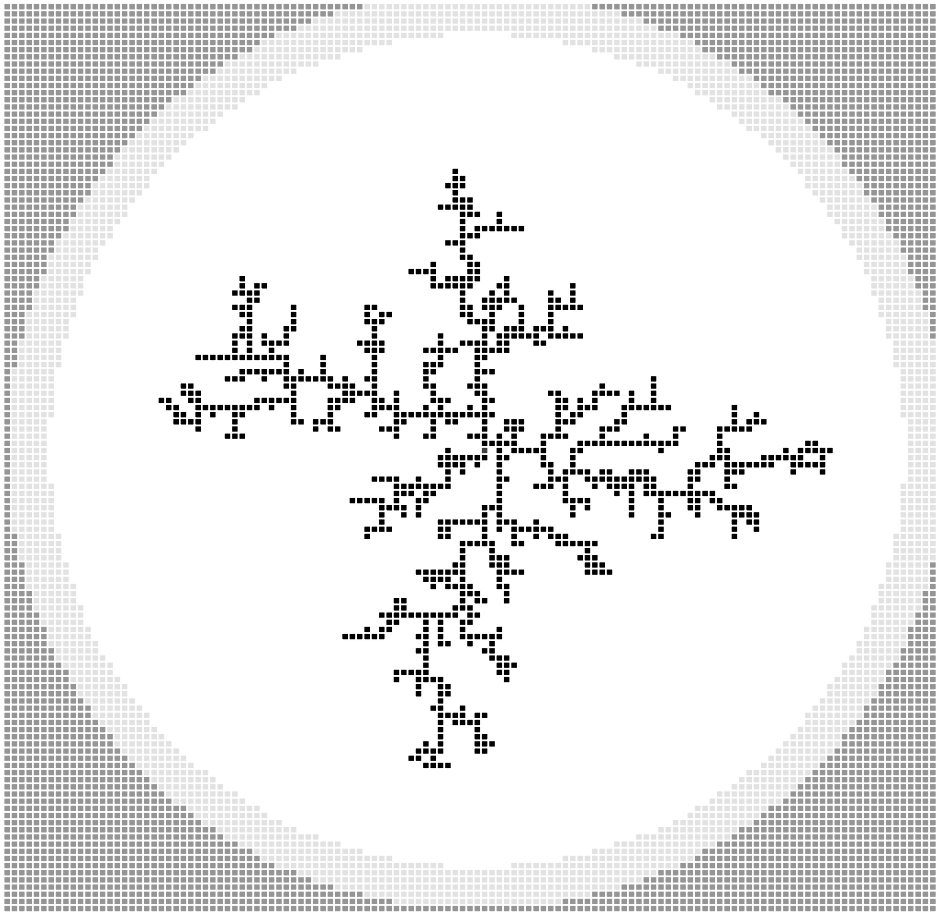

Figure 2: DLA simulation with 911 particles showing the outer and inner circle is light grey.

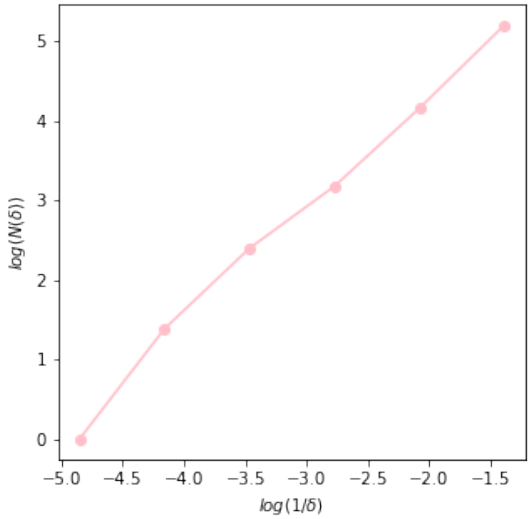

Figure 3: The logarithm of the number of covered subgrids $N(\delta)$ vs. the logarithm of the inverse subgrid size δ.

The first point is excluded, as that one is for the subgrid size of NxN which des not represent the fractal dimension. A rough estimate of the slope is calculated by taing the second and last point. This gives a fractal dimension with the exact value of 1.5. The code runs again with the same settings generating another fractal, see Figure 4.

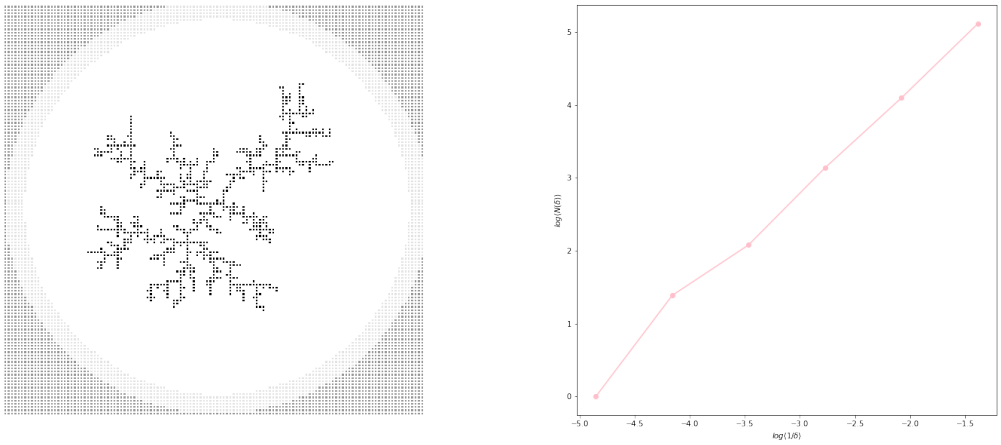

Figure 4: Another simulation using the same settings.

Using the second and last points give the dimension of 1.48. However by looking at the points in the graph, a more representative slope would be between point two and the last and this gives a value of 1.51.

5 Outlook

It took approximately 5 minutes to run the simulation that produced the fractal in Figure 2. The code is quite hard-coded and could be made more efficient for faster results and larger systems.

One could also improve the code further by properly fitting the datapoints of the box counting and thus obtaining a more exact value for the slope.

6 References

[1] Witten Jr, T. A., Sander, L. M. (1981). Diffusion-limited aggregation, a kinetic critical phenomenon. Physical review letters, 47(19), 1400.

[2] So, G. B., So, H. R., Jin, G. G. (2017). Enhancement of the box-counting algorithm for fractal dimension estimation. Pattern Recognition Letters, 98, 53-58.

7 Code

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

def set_grid(size,step):
    x = np.arange(0,size+step,step)
    y = np.arange(0,size+step,step)
    xx, yy = np.meshgrid(x, y)
    matrix_points = np.zeros((np.shape(yy)[0],np.shape(xx)[0]))
    return xx, yy, matrix_points

def set_outer_circle(matrix_points):
    circle_limit = np.shape(matrix_points)[0]/2

    for i in range(np.shape(matrix_points)[0]):
        for j in range(np.shape(matrix_points)[1]):

            if np.sqrt((i-circle_limit)**2+(j-circle_limit)**2) >= circle_limit:
                matrix_points[i][j] = 0.5

    return matrix_points

def set_inner_circle(matrix_points):

    circle_limit = np.shape(matrix_points)[0]/2.2

    center = np.shape(matrix_points)[0]/2 - circle_limit

    for i in range(np.shape(matrix_points)[0]):
        for j in range(np.shape(matrix_points)[1]):

            if np.sqrt((i-circle_limit-center)**2+(j-circle_limit-center)**2) >= circle_limit and matrix_points[i][j] != 0.5:
                matrix_points[i][j] = 0.2

    return matrix_points

def add_initial_seed(matrix_points,size,step):
    #sets the initial seed, puts it at random somwhere close to the central
    random_point = [np.random.choice(np.shape(matrix_points)[0],1, replace=False)[0],
                    np.random.choice(np.shape(matrix_points)[0],1, replace=False)[0]]
    x_seed = xx[random_point[0]][random_point[1]]
    y_seed = yy[random_point[0]][random_point[1]]

    seed_limit_1 = (size/2)-(step*2)
    seed_limit_2 = (size/2)+(step*2)

    if seed_limit_2>x_seed>seed_limit_1 and seed_limit_2>y_seed>seed_limit_1:
        matrix_points[random_point[0]][random_point[1]] = 0.8
    else:
        while (x_seed>seed_limit_2 or x_seed<seed_limit_1) or (y_seed>seed_limit_2 or y_seed<seed_limit_1):
            random_point = [np.random.choice(np.shape(matrix_points)[0],1, replace=False)[0],
                            np.random.choice(np.shape(matrix_points)[0],1, replace=False)[0]]
            x_seed = xx[random_point[0]][random_point[1]]
            y_seed = yy[random_point[0]][random_point[1]]
        matrix_points[random_point[0]][random_point[1]] = 0.8
    return matrix_points, random_point

def release_walker(matrix_points): #releases walker from the inner sphere vertix
    release = False

    while release == False:
        random_point = [np.random.choice(np.shape(matrix_points)[0],1, replace=False)[0],
                        np.random.choice(np.shape(matrix_points)[0],1, replace=False)[0]]
        try:
            grannar = neighbours(matrix_points, random_point)
            occurency = int(np.count_nonzero(np.array(grannar) == 0.2))
            if 0.0 in grannar and (occurency == 2 or occurency == 1):
                release = True
        except:
            continue
    return random_point

def random_step(random_point):
    #there are four movements the walker can take
    scenarios = ["step_forward_x", "step_backward_x",
    "step_up_y", "step_down_y"]
    which_scenario = np.random.choice(np.shape(scenarios)[0])
    if scenarios[which_scenario] == "step_forward_x":
        random_point[0] +=1
    if scenarios[which_scenario] == "step_backward_x":
        random_point[0] -=1
    if scenarios[which_scenario] == "step_up_y":
        random_point[1] +=1
    if scenarios[which_scenario] == "step_down_y":
        random_point[1] -=1
    return random_point

def neighbours(matrix_points, random_point):
    a = int(random_point[0]-1)
    b = int(random_point[0]+1)
    c = int(random_point[1]-1)
    d = int(random_point[1]+1)
```

```
        out = int(np.shape(matrix_points[0])[0])-1
        if a < 0 or a > out or b < 0 or b > out or c < 0 or c > out or d < 0 or d > out:
            raise Exception("outside")

        right = matrix_points[random_point[0]][random_point[1]+1]
        left = matrix_points[random_point[0]][random_point[1]-1]
        bottom = matrix_points[random_point[0]+1][random_point[1]]
        top = matrix_points[random_point[0]-1][random_point[1]]

        grannar = [right, left, top, bottom]

        return grannar

    def fractal_dimension(k, matrix_points):
        div = []

        for i in range(2,k+1):
            div.append(2**i)
        #the pixels are in the array matrix_points

        cover = np.zeros(np.shape(div)[0])
        inverse_delta = np.zeros(np.shape(div)[0])

        def view_as_blocks(arr, BSZ):
        # reference: https://stackoverflow.com/questions/44782476/split-a-numpy-array-both-horizontally-and-vertically
            # arr is input array, BSZ is block-size
            m,n = arr.shape
            M,N = BSZ
            return arr.reshape(m//M, M, n//N, N).swapaxes(1,2).reshape(-1,M,N)

        for i in range(np.shape(div)[0]):
            inverse_delta[i] += 1/div[i]
            boxes = view_as_blocks(matrix_points,(int(div[i]),int(div[i])))

            for j in range(np.shape(boxes)[0]): #loop over all boxes
                occupied = np.count_nonzero(boxes[j] == 1.0) #is the pixels in the grid occupied?
                if int(occupied) > 0: #if box is occupied count it as one
                    cover[i] += 1

        D = (np.log(cover)[1]-np.log(cover)[len(cover)-1])/(np.log(inverse_delta)[1]-np.log(inverse_delta)[len(inverse_delta)-1])

        print("The dimension of the fractal is:", D)



    #-------DETERMINES THE SIZE OF OUR GRID SYSTEM-----------------------------
    k = 7
    #--------------------------------------------------------------------------

    N = 2**k
    step = 1/N
    size = 1 - step

    xx, yy, matrix_points = set_grid(size, step)

    matrix_points, random_point = add_initial_seed(matrix_points,size,step)

    matrix_points = set_outer_circle(matrix_points)

    matrix_points = set_inner_circle(matrix_points)

    N = 95000

    particles = 0

    for i in range(N):
        random_point = release_walker(matrix_points)
        try:
            while matrix_points[random_point[0]][random_point[1]] == 0.0:
                grannar = neighbours(matrix_points, random_point)

                if 1.0 in grannar or 0.8 in grannar: #checks if there is a neighbour == 1 or the seed == 0.8
                    #now we reached our neighbour, lets settle down
                    matrix_points[random_point[0]][random_point[1]] = 1.0
                    particles += 1

                else:
                    random_point = random_step(random_point)
                    #outer circle limit
                    if matrix_points[random_point[0]][random_point[1]] == 2.0:
                        #we're now out of bounds
                        raise Exception("out!")

        except Exception:
            continue

    plt.pcolormesh(xx, yy, matrix_points,cmap='Greys', edgecolors="white")
    plt.axis("off")
    plt.rcParams["figure.figsize"] = (10,10)

    print(particles)

    plt.show()

    fractal_dimension(k, matrix_points)
```