Header:

Author: Crissy Hendrickson
Title: Diffusion Limited Aggregation
Group Members: Andy Kim, Peter Kim, Karl Fenzl

Analysis of the Problem:

Diffusion limited aggregation occurs when there is a stationary particle and another particle which is released from far away and moves randomly until it reaches and attaches to the first particle. If multiple particles are released from a distance and move randomly until they reach the cluster of particles, a pattern will develop which involves several antennas branching out from the center. Using Python, a stochastic model can be developed of the process by having a stationary dot at the center of a square lattice and then having another dot preform a random walk from the boundary until it reaches the stationary dot. Repeating the process for several walking dots will then produce a cluster with the previously described pattern. If at any point a walking dot comes in contact with the boundary, it will be reflected back into the lattice. Additionally, using animation from matplotlib a movie can be made which shows how the cluster grows with each additional dot that is released from the boundary and time lapsed images of the growing cluster can be created. The final cluster can be expected to have several antennas branching out from the center. If the maximum horizontal extent of the cluster is plotted as a function of time, it can be expected that the line will be relatively linear with steps between the data points. Furthermore, if the distance each walking dot traveled to attach to the cluster was plotted as a function of time it can be expected that there will be a negative coloration between the distance traveled and time.

Model Design:

Computational models are simplified representations of phenomena, systems, or processes. They are used to help us study and understand different aspects of life and can lead to predictions involving responses in un-tested scenarios. Often because of certain assumptions and simplifications, they are much easier to work with than the systems they represent and can point to new insight and comprehension. To translate a problem into a computational model there are five main steps: analysis of a problem, model development and formulation, model implementation, model verification, and model interpretation. The first step, analysis of a problem, involves determining what the question and unknowns are and what the answer would look like. The second step, model development and formulation, includes gathering relevant data, making assumptions, determining variable relationships and units, and writing down specific equations or rules. Model implementation involves writing a program in a computer language to solve the problem while model verification involves testing special cases whose answers are known to confirm the solution is accurate and the program does what it is supposed to do. Finally, model interpretation includes answering the main question sometimes with figures or tables and discussing more about the solution: if it was expected, what it means and how it could change if the assumptions are relaxed.

In the model for the diffusion limited aggregation, several assumptions were made before the model was implemented. The model was assumed to be dynamic since it depends on time and stochastic because the solution involves an element of chance and may be different even

under the same conditions. It is also assumed to be a spatially extended model because the solution is given by a field that is different at different points in space. Because time changes in specific well separated steps as each new walking point is released, time is considered to be discrete. Furthermore, space is considered to be discrete because a lattice is used to specify it. The size of the lattice was also chosen to be 100x100 and the number of dots in the cluster was chosen to be 1000 so diffusion limited aggregation could be better observed. To help speed up the code, if a walking dot did not attach to the cluster after 1000 steps it was thrown out.

In the model development, several functions were used in Python to help find the solution. Their purpose and relationship to each other is described in the following section.

Model Solution:

To find the solution, the code at the end of the report was run in python. First, the bounds, the location of the original stationary dot, and the number of dots to be included in the cluster were defined. Then a function was written called random_walk that used the random number generator in numpy to output the result of one step in a random walk. The previously described function also reflected any dots that came into contact with the boundary by multiplying the random step by negative one. Next a function was written called point_check, which checked to see if the walking dot was next to the cluster and returned the dot's coordinates and whether it had attached to the cluster or not. Specifically, point_check used a for loop to compare the position of the walking dot, to the elements in the arrays which had the locations of the dots already in the cluster. Additionally, a function called start_point_gen was created which used the random number generator in numpy to return a random point on the boundary of the lattice.

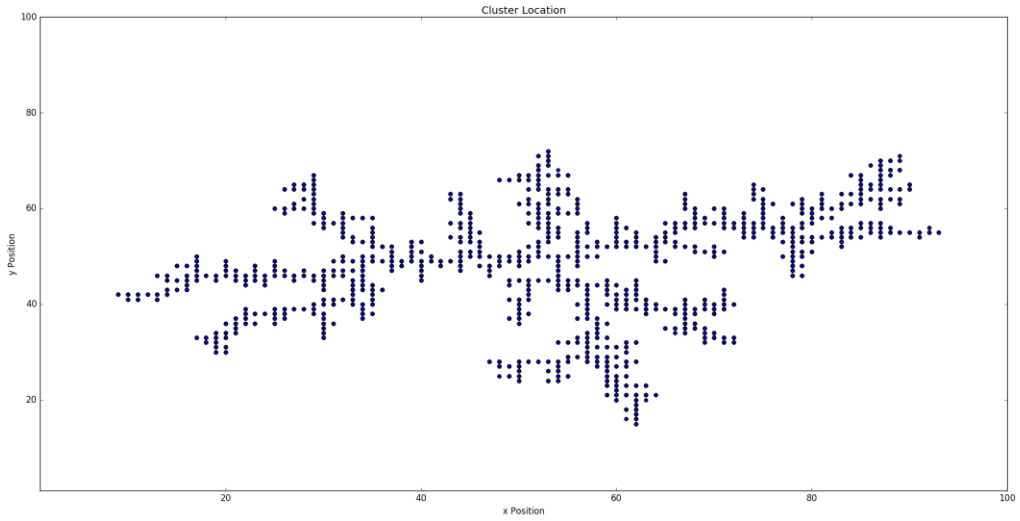
Finally, the random_walk, point_check, and start_point_gen functions were all put together into another function called dla to simulate diffusion limited aggregation. In the dla function, a while loop kept track of the number of dots in the cluster and a for loop kept track of the movement of a single walking dot. For each new dot created, the start_point_gen function gave its initial location. Then using the random_walk function, a single random step was taken. The new location of the dot was then compared with the location of the dots in the cluster using the point_check function to see if the dot attached to the cluster or not. If the dot had not attached to the cluster, another random step would be taken and the new location would then be compared again with the location of the dots in the cluster. The process would be repeated until the dot attached to the cluster or 1000 steps were taken in which case the dot would be thrown out. When the dot did attach to the cluster, its location would be added to the arrays which had the location of the dots already in the cluster.

Once the cluster had the desired number of dots, a graph could then be made using plt.scatter and the arrays which had the location of the dots. The graph was then examined to see if the dots created the pattern that was expected. Additionally, the maximum horizontal extent of the cluster was plotted as a function of time and analyzed to see if it behaved as expected. On the graphs, time point refers to the number of dots that have attached to the cluster. The maximum horizontal extent was found inside the dla function with an array that kept track of the horizontal extent of the points in the cluster and another array which recorded the maximum extent in the previous array. A movie that showed the cluster growing as the dots were added to it was also made using animation from matplotlib and time lapsed images of the growing cluster were created. To verify the solution, the distance each walking dot traveled to attach to the cluster was

plotted as a function of time using an array generated inside the dla function that contained the distances. The graph was then also examined to see if it behaved as expected.

Results, Verification, and Conclusion:

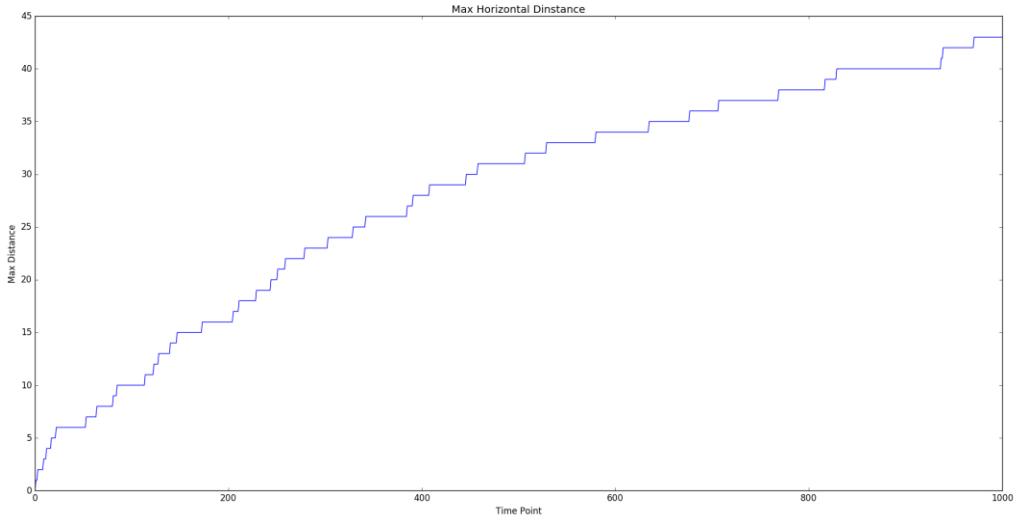
The following graph shows the location of the dots in the cluster.



As shown by the graph above, the dots formed a cluster with several antennas branching out from the center which was expected. The pattern exists because the particles are more likely to attach to the ends of the antennas than other spots on the cluster. The longer the antenna, the greater the probability that a walking dot will attach there. Since the dots attach in the expected locations and there are not multiple clusters, the solution is verified.

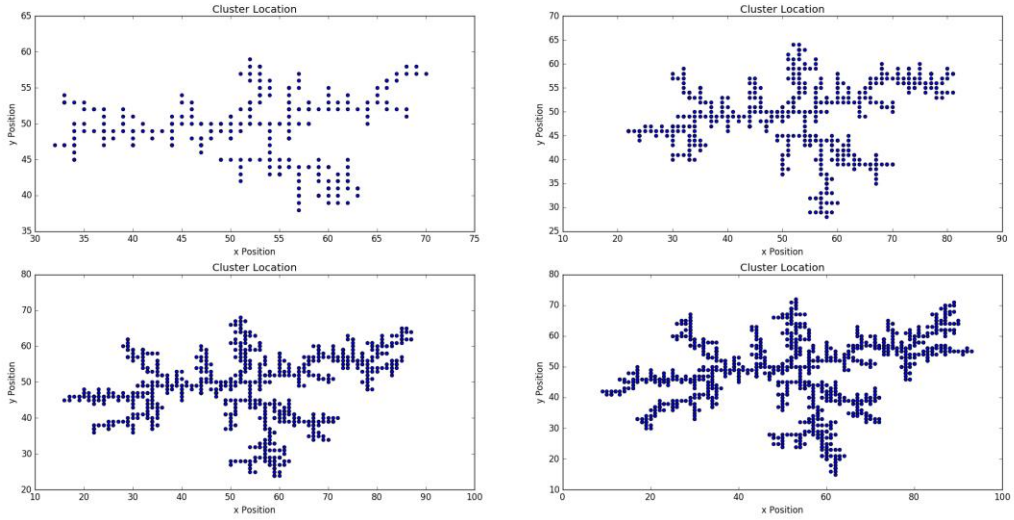
Next time to improve the model, it could be optimized so it runs faster. Currently, the code takes about three hours to run. However, if the walking dots were initially released randomly from the perimeter of a circle with a radius 20 steps away from the center, the time it takes to run would decreases. Each time the cluster grows, the radius of the circle could then be increased to account for the larger size.

A graph was also made that showed the maximum horizontal extent of the cluster as a function of time.



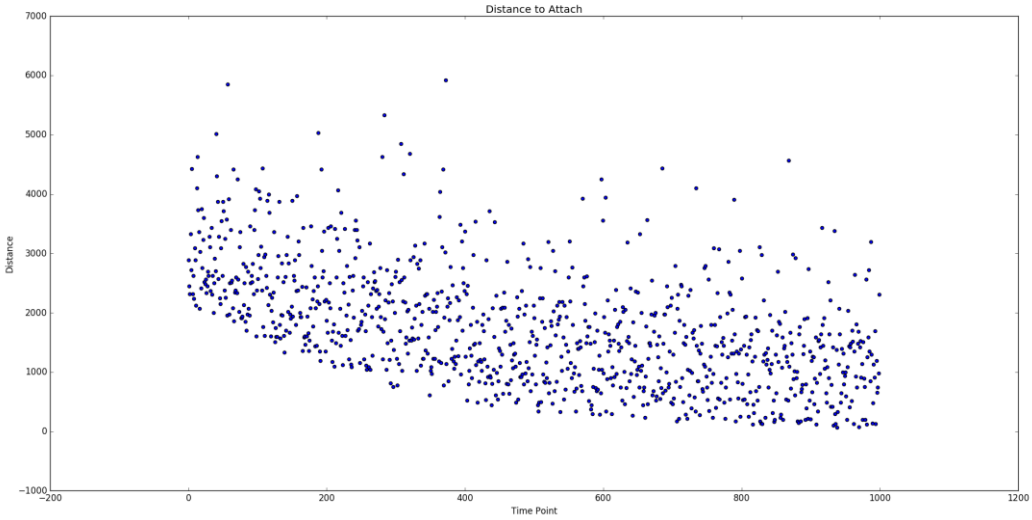
As shown by the graph above the size of the cluster grows approximately linearly with time with steps between the data points. Here, time point refers to the number of dots that have attached to the cluster. The trend was expected because as more dots are added to the cluster, its horizontal extent can be expected to increase as the cluster grows.

Additionally, time lapsed images of the growing cluster were created.



The top left image was taken after 250 dots were attached to the cluster while the top right image was taken after 500 dots were attached to the cluster. The bottom left image was taken after 750 dots were attached to the cluster and finally the bottom right image was taken after 1000 dots were attached to the cluster. As shown by the four graphs, when more dots are added the cluster grows so the antennas become bigger which was expected.

To verify the solution, the distance each walking dot traveled to attach to the cluster was plotted as a function of time.



As shown by the graph above there is a negative coloration between the distance each walking dot travels and the number of dots in the cluster. The behavior is expected because as the cluster grows, the antennas get closer to the boundary where the walking dots are released from so there is not as large of a distance between them. Therefore, when there are more dots in the cluster, the walking dots do not have to travel as far to attach to the cluster. Since the plot matches what was expected, the solution is verified.

Code:

```
# -*- coding: utf-8 -*-
"""
Created on Sun Apr  9 21:32:55 2017

@author: Crissy
"""

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation
import time

bounds = 20 #bounds of the lattice

original_point = (bounds / 2, bounds / 2)
#original stationary dot

numpoints= 100 #number of total dots in cluster

latt_points_x = np.zeros(numpoints + 1)
```

```
# will keep track of the x location of the dots in the cluster
latt_points_y = np.zeros(numpoints + 1)
# will keep track of the y location of the dots in the cluster

time_coords = np.zeros(numpoints + 1)
#will record the horizontal extent of the dots in the cluster
final_coords = np.zeros(numpoints + 1)
#will record the maximum values from time_coords

radius=np.zeros(numpoints+1)
#will record the distance each walking dot travels to attach to
the cluster

latt_points_x[0] = original_point[0]
latt_points_y[0] = original_point[1]

#possible neighbor points
nx = [-1, 0, 1, 0]
ny = [0, 1, 0, -1]

#random walk that tells if the particle moves up, down, left, or
right
#and produces an output of the new particle location
# also checks to see if boundry and reflects if it will pass
boundry

def random_walk(point_coords, bounds):
    x = point_coords[0]
    y = point_coords[1]

    rand_step_x = (2 * (np.random.rand() < .5)) - 1
    rand_step_y = (2 * (np.random.rand() < .5)) - 1

    #reflects if points pass boundries
    if x + rand_step_x > bounds or x + rand_step_x < 0:
        rand_step_x = rand_step_x * -1
    if y + rand_step_y > bounds or y + rand_step_y < 0:
        rand_step_y = rand_step_y * -1

    x = x + rand_step_x
    y = y + rand_step_y

    return (x, y)
```

```
#checks to see if point attaches
#and returns a point if it is supposed to be attached
#returns coordinates and whether it is near cluster

def point_check(point_coords):
    pt_attach = False

    for i in range(len(nx)):
        x = point_coords[0] + nx[i]
        y = point_coords[1] + ny[i]

        for n in range(len(latt_points_x)):
            if x == latt_points_x[n] and y == latt_points_y[n]\

            and x != 0 and y != 0:
                pt_attach = True
                break

        if pt_attach == True:
            break

    return pt_attach, point_coords

#produces random coordinate of point on the borders of lattice

def start_point_gen(lattice_bounds):

    initial_pt = np.random.rand()

    if initial_pt < .25:
        initial_x = np.random.randint(0,lattice_bounds)
        initial_y = 0
    elif initial_pt < .5 and initial_pt >= .25:
        initial_y = np.random.randint(0,lattice_bounds)
        initial_x = 0
    elif initial_pt < .75 and initial_pt >= .5:
        initial_x = np.random.randint(0,lattice_bounds)
        initial_y = lattice_bounds
    elif initial_pt >= .75:
        initial_y = np.random.randint(0,lattice_bounds)
        initial_x = lattice_bounds

    start_point = (initial_x, initial_y)
```

```
    return start_point

#%%
# for loop = for individual point movement
# while loop = for number of total cluster points

def dla(total_cluster_pts, lattice_bounds):
    start_time = time.time()

    cluster_pts = 0
    pt_counter = 1 # used to add elements to lattice array

    while cluster_pts < total_cluster_pts:
        new_pt = start_point_gen(lattice_bounds)
        xint=new_pt[0]
        yint=new_pt[1]
        #number of times random particle moves before being
        terminated
        #= assumption
        for num_moves in range(1000):
            new_pt = random_walk(new_pt, lattice_bounds)
            flag, coords = point_check(new_pt)

            if flag == True:
                latt_points_x[pt_counter] = coords[0]
                latt_points_y[pt_counter] = coords[1]


radius[pt_counter]=np.abs((latt_points_x[pt_counter]-xint))**2\
                 + (np.abs(latt_points_y[pt_counter]-yint)**2)


                time_coords[pt_counter] = np.abs((coords[0]\
                 - original_point[0]))

                final_coords[pt_counter] = max(time_coords)

                pt_counter = pt_counter + 1
                cluster_pts = cluster_pts + 1
                plt.scatter(latt_points_x, latt_points_y)
                plt.xlim(1,bounds)
                plt.ylim(1,bounds)
                plt.title('DLA')
                print(str(num_moves)+'==number of moves    ',\
```

```
                str((cluster_pts/numpoints)*100)+'% complete')
                break

            if cluster_pts == total_cluster_pts:
               elapsed_time = time.time() - start_time
               return elapsed_time


dla(numpoints, bounds)

plt.scatter(latt_points_x, latt_points_y)
plt.title('Cluster Location')
plt.xlabel('x Position')
plt.ylabel('y Position')
plt.figure()
plt.plot(np.arange(0,numpoints+1,1),final_coords)
plt.title('Max Horizontal Dinstance')
plt.xlabel('Time Point')
#time point refers to the number of dots that have attached to
the cluster
plt.ylabel('Max Distance')
plt.figure()

#Time lapse
t1=numpoints/4
t2=numpoints/2
t3=numpoints*(3/4)
t4=numpoints
plt.subplot(2,2,1)
plt.scatter(latt_points_x[0:t1], latt_points_y[0:t1])
plt.title('Cluster Location')
plt.xlabel('x Position')
plt.ylabel('y Position')
plt.subplot(2,2,2)
plt.scatter(latt_points_x[0:t2], latt_points_y[0:t2])
plt.title('Cluster Location')
plt.xlabel('x Position')
plt.ylabel('y Position')
plt.subplot(2,2,3)
plt.scatter(latt_points_x[0:t3], latt_points_y[0:t3])
plt.title('Cluster Location')
plt.xlabel('x Position')
plt.ylabel('y Position')
plt.subplot(2,2,4)
```

```
plt.scatter(latt_points_x[0:t4], latt_points_y[0:t4])
plt.title('Cluster Location')
plt.xlabel('x Position')
plt.ylabel('y Position')

#verification-- Distance walking dots travel vs. number of
cluster points
plt.figure()
plt.scatter(np.arange(0,numpoints,1),radius[1:numpoints+1])
plt.title('Distance to Attach')
plt.xlabel('Time Point')
plt.ylabel('Distance')

#movie
from matplotlib import animation

fig=plt.figure()
ax=plt.axes(xlim=(0, bounds), xlabel='x Position', ylim=(0,
bounds),\
            ylabel='y Position', title='Cluster Location')

scat = ax.scatter([], [], s=60)

def init():
    scat.set_offsets([])
    return scat,

def animate(i):
    data = np.hstack((latt_points_x[:i,np.newaxis],\
                      latt_points_y[:i, np.newaxis]))
    scat.set_offsets(data)
    return scat,

anim = animation.FuncAnimation(fig, animate, init_func=init, \
                               frames=len(latt_points_x)+1,
                               interval=200, blit=False,
                               repeat=False)
```