DIFFUSION LIMITED AGGREGATION

Rectangular Attractor

Paul Zebarth

PHYS – 4611: Final Project

Introduction

Diffusion limited aggregation (DLA) is a process defined by, randomly walking particles aggregating. In principle, this process is applicable in every field concerned with the diffusion of particles, e.g. biology, chemistry, physics, etc. Random walks are a Monte Carlo method. Meaning it is an algorithm which relies on random sampling to obtain results. Random walks are used in DLA to model the Brownian motion of particles in a medium. Brownian motion applies to gas, liquid, or solid particles in a liquid or gas colloid. By using a Brownian motion models, by way of Monte Carlo random walk algorithms, the DLA model has become a way to model physical processes of diffusion.

Computational Approach

Python is an object-oriented programing (OOP) language. OOP allows the creation of object instances, and can act on its attributes with methods to reveal data it contains. This allows fast and easy programming of objects which can be broken into useful data. Using methods on objects simplifies analysis and can carry out complicated calculations easier than functional programming. Python is also a high-level language. This means Python is an interpreted language compared conversely to a compiled language. It must be run through an interpreter program before the code is executed on the processor. This allows Python to be programmed using easily defined objects and libraries with supporting functions and methods, but compromises its computational speed. To code a physical modeling program, it would be simplest to use a high-level OOP language.

Method and Procedure

DLA can have multiple different visualizations. To begin, you need to define the seed particle, this is what randomly walking particles will aggregate to. This seed particle can take various geometries. The seed can be at the center of an image, there can be a line of seed particles or there can be a border of seed particles. These are called attractors because they are the points which attract particles, and attractors can have various geometries. The possibilities of DLA as stated before can depend on geometry of the attractor, in addition, DLA can take place in two-dimensions and three-dimensions. The possibilities of DLA are infinite and apply to many areas of science. The basic process for DLA is described by (Fig. 1).

1. Define a seed particle
2. Start a randomly walking particle
3. Does the randomly walking particle hit another particle?
   - If no: start at Step 2
   - If yes: continue to Step 4
4. Add particle to coordinates of stuck particles
5. Plot coordinates of stuck particles

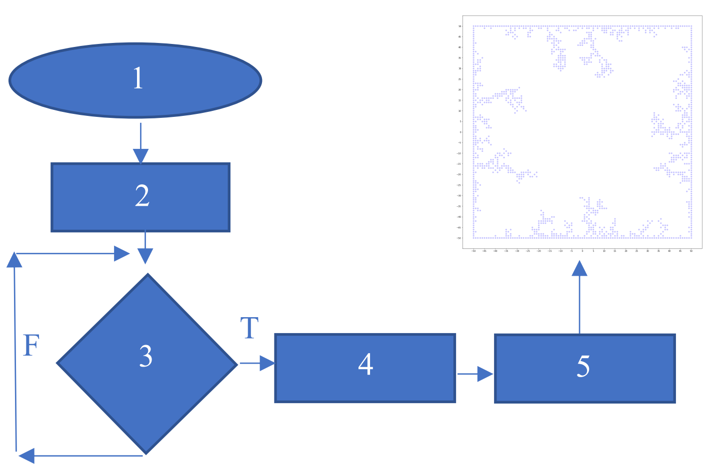


Figure 1: Programming Flowchart

For my project I focused on the rectangular attractor, which would be defined by seed particles along the border of a rectangle. I followed the sequence of a DLA process (Fig. 1) and added in the specifics for the rectangular attractor (Appendix A). I defined a function called “rect_att_plot” which takes “length” as the singular parameter. It will produce a border of seed particles defined by the user’s input of “length” (Appendix A, line 26-35). The “length” parameter defines the length from the origin to the border. The function then releases particles one at a time from the origin (Appendix A, line 41). The particle can move in the x direction, or y direction by one unit. It can also move in the x and y direction diagonally which demonstrates the true process of Brownian motion (Appendix A, line 20). When one particle’s random walk reaches another particle, it is then added to the coordinates of stuck particles in “particle_stick” (Appendix A, line 37-57). Then the next particle will be released. When the particles reach the origin, the function outputs a plot (Appendix A, line 58-66).

The number of particles is set at 10,000 within the defined function “rect_att_plot” because the loops terminates when the aggregated particles reach the origin, so a high number will not affect performance (Appendix A, line 17). This ensures the particles aggregate to the origin, if they do not reach the origin when the particles have run out, the plot will be produced regardless. A while loop would ideally be in place of the first for loop (Appendix A, line 37). However, checking the two-dimensional array “particle_stick” for the value [[0,0]] at the center of the plot needs an iterative process in the way I’ve defined this problem. The “step_number” variable is set at 30,000 within the loop to accommodate a large “length” defined by the user. This allows the particles to have enough steps to reach the border defined by “length” and aggregate to produce the plot with particles terminating at the origin.

The next function I created was “rect_att_basic_animation” (Appendix B). This produces a plot for each step a particle takes. This function is lengthy and repetitive. This is because I allowed the user to control six parameters (Appendix B, line 74-87). This function follows much of the same process of the “rect_att_plot” function. The caveat being that at each step, a plot is produced. While the program runs to completion, the outputted plots update for each step which creates a basic animation.

The last function I created was “rect_att_animation”. This function uses the principles of “rect_att_plot” and “rect_att_basic_animation” (Appendix C). The function takes one parameter defined by the user, which is “length” (Appendix C, line 204-205). At each step a particle takes, a plot is saved to a path directory as a .png file (Appendix C, line 259). Then once the particles have aggregated at the origin, I used the cv2 and glob libraries to read the files and compile a .mp4 animation from all the .png files saved in the path directory (Appendix C, line 282-296).

Analysis and Results

“rect_att_plot”

The plots produced by the function “rect_att_plot” are of high quality, conversely, the computational time is lengthy for large values of “length” defined by the user. I recorded computational times for various values of “length” defined by the user in an effort to determine how computational time was affected. The results are tabulated in Table 1 and shown in Fig. 2.

Table 1: Comparison of Parameter "length" and Computational Time

Length	Computational Time (s)
5	0.2365
10	0.61275
30	56.3
40	272
50	979
100	36051

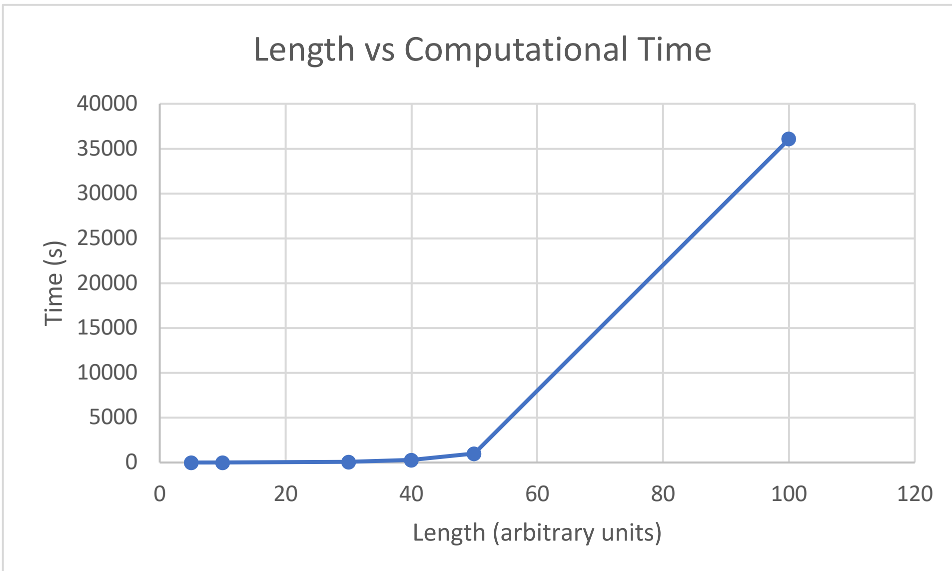


Figure 2: Time Increase due to Parameter "length" Increase

Fig. 2 shows a computational time dependence which is exponential with "length". Further data points beyond the last would be greater than 10 hours to collect or be useful to the user of the program. However, the plots created by this program are of high quality and demonstrate the ordered behavior that we would expect from DLA (Appendix D). By allowing the particle to travel laterally and diagonally, I produced images which resemble other research in DLA (Bourke, Paul. "Constrained diffusion-limited aggregation in 3 dimensions." *Computers & Graphics* 30.4 (2006): 646-649.). This function can be run for any value of parameter. However, best quality images considering computation time, would be defined by a "length" of 50 with a computation time of roughly 16 minutes.

"rect_att_basic_animation"

This function works well as a demonstrator of the DLA process for "lengths" less than 10 and "particles" less than 2 (Appendix B, line 74-87). By using a large or small value for "step_number" we can see how a particle would stick or not stick based on its value (Appendix B, line 74-87). This function is taxing on your computers CPU and RAM if it's used for large "lengths" greater than 5 and "particles" greater than 2, as there are more steps and therefore frames to the animation. The more frames there are the more your computer has to store. This should be used as a visual understanding of the DLA process. However, it will work for any of the parameters the user can define no matter how large or small.

"rect_att_animation"

I then improved the previous function "rect_att_basic_animation". As a result of the files being saved directly to the path directory, they are stored at several frames a second. The process of converting the .png files to an animation takes less than a second. This is a very convenient process to make an animation. However, for "lengths" greater than 10 and "particles" more than 2, it is not efficient. For a "length" of 10 it took over 3000 frames to compile an animation. This function will work for any user defined parameter but efficiency drops over a "length" of 5.

Conclusion

Producing a rectangular attractor with the DLA process was difficult. There are easier methods to accomplish the same output, albeit with much lower quality. By allowing the particle to move in every direction laterally and diagonally, I have shown high quality DLA defined qualitatively by the branching in the aggregate particles which resemble trees. Using two different animation techniques, I was able to show the process of DLA for a value of "length" less than 5 efficiently. More research into making more efficient animations would be needed to increase the "length" that would be useful to the user and less taxing on the computer.

Appendices

Appendix A: Rectangular Attractor Plotting Function

```python
def rect_att_plot(length):

    """
    Parameters
    ----------
    length       : integer
                   length from origin to border of square, or the half length of the full square.
    """

    particles = 10000
    step_number = 30000
    dimension = 2
    step_options = [-1, 0, 1]
    origin = np.zeros((1, dimension))
    boundary = [-length,length]
    stick_coords = np.zeros((1, dimension))
    stick_path = np.zeros((1, dimension))

    particle_stick = []
    for i in np.arange(boundary[0],boundary[1]+1):

        particle_stick.append([boundary[1],i])
        particle_stick.append([boundary[0],i])
        particle_stick.append([i,boundary[1]])
        particle_stick.append([i,boundary[0]])

    particle_stick = np.array(particle_stick)
    particle_stick = np.unique(particle_stick, axis=0)

    for j in np.arange(1,particles+1):

        step_shape = [step_number, dimension]
        steps = np.random.choice(a=step_options, size=step_shape)
        path = np.concatenate([origin,steps]).cumsum(0)

        for k in np.arange(0,len(path)):
            stuck = False
            for l in np.arange(0,len(particle_stick)):
                if not stuck:
                    if path[k,0] == particle_stick[l,0] and path[k,1] == particle_stick[l,1:]:
                        stick_path = np.concatenate([origin,steps[:k-1]]).cumsum(0)
                        stick_coords = stick_path[-1:]
                        stuck = True
                    else:
                        continue
            if stuck:
                if boundary[0] < stick_coords[:,0] < boundary[1]:
                    if boundary[0] < stick_coords[:,1] < boundary[1]:
                        particle_stick = np.concatenate([particle_stick,stick_coords])
                        break
        if stick_coords[:,0] == 0 and stick_coords[:,1] == 0:
            break


    plt.scatter(particle_stick[:,0], particle_stick[:,1], alpha=1, color='blue');
    plt.rcParams["figure.figsize"] = (20,20)
    plt.xticks(np.arange(boundary[0],boundary[1]+1,5))
    plt.yticks(np.arange(boundary[0],boundary[1]+1,5))
    plt.show()

# rect_att_plot(5)
```

Appendix B: Rectangular Attractor Basic Animation

```python
def rect_att_basic_animation(particles, step_number, length, show_path=True,\
                                      with_dots=True, pause=0.0001):

    """
    Parameters
    ----------
    particles    : integer
                   number of particles
    step_number  : integer
                   number of steps (if no output steps aren't high enough)
    length       : integer
                   length from origin to border of square, or the half length of the full square.
    show_path    : boolean (optional)
                   If True, random walk path is shown, True by default
    with_dots    : boolean (optional)
                   If True, dots are random-walking particles, True by default.
    pause        : float (optional)
                   Pausing time between 2 steps, 0.1 secondes by default.
    """

    dimension = 2
    step_options = [-1, 0, 1]
    origin = np.zeros((1, dimension))
    boundary = [-length,length]
    stick_coords = np.zeros((1, dimension))
    stick_path = np.zeros((1, dimension))

    particle_stick = []
    for i in np.arange(boundary[0],boundary[1]+1):
        particle_stick.append([boundary[1],i])
        particle_stick.append([boundary[0],i])
        particle_stick.append([i,boundary[1]])
        particle_stick.append([i,boundary[0]])

    particle_stick = np.array(particle_stick)
    particle_stick = np.unique(particle_stick, axis=0)

    plt.rcParams["figure.figsize"] = (20,20)
    plt.xticks(np.arange(boundary[0],boundary[1]+1,5))
    plt.xlim(boundary[0],boundary[1])
    plt.yticks(np.arange(boundary[0],boundary[1]+1,5))
    plt.ylim(boundary[0],boundary[1])

    plt.scatter(particle_stick[:,0], particle_stick[:,1], alpha=1, color='blue')

    for j in np.arange(1,particles+1):

        step_shape = [step_number, dimension]
        steps = np.random.choice(a=step_options, size=step_shape)
        path = np.concatenate([origin,steps]).cumsum(0)

        for k in np.arange(0,len(path)):
            stuck = False
            for l in np.arange(0,len(particle_stick)):
                if not stuck:
                    if path[k,0] == particle_stick[l,0] and path[k,1] == particle_stick[l,1:]:
                        for m in range(k):
                            stick_path = np.concatenate([origin,steps[:m]]).cumsum(0)
                            if show_path == True and with_dots == False:
                                plt.xticks(np.arange(boundary[0],boundary[1]+1,5))
                                plt.xlim(boundary[0],boundary[1])
                                plt.yticks(np.arange(boundary[0],boundary[1]+1,5))
                                plt.ylim(boundary[0],boundary[1])
                                plt.plot(stick_path[:,0],stick_path[:,1], alpha=0.4, color='red')
                                plt.scatter(particle_stick[:,0], particle_stick[:,1], alpha=0.5,\
                                            color='blue')
```

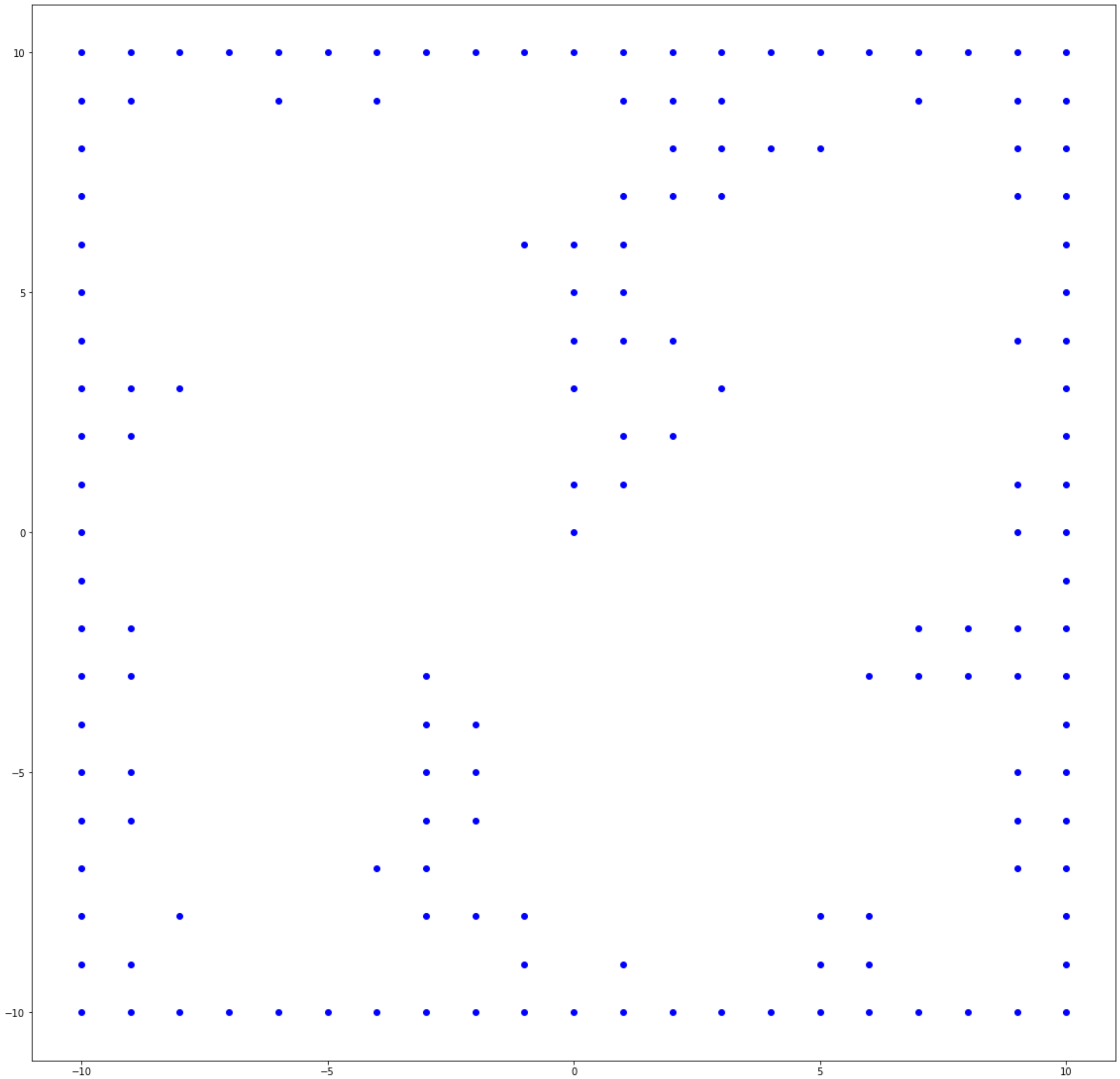
```
        for k in np.arange(0,len(path)):
            stuck = False
            for l in np.arange(0,len(particle_stick)):
                if not stuck:
                    if path[k,0] == particle_stick[l,0] and path[k,1] == particle_stick[l,1:]:
                        for m in range(k):
                            stick_path = np.concatenate([origin,steps[:m]]).cumsum(0)
                            if show_path == True and with_dots == False:
                                plt.xticks(np.arange(boundary[0],boundary[1]+1,5))
                                plt.xlim(boundary[0],boundary[1])
                                plt.yticks(np.arange(boundary[0],boundary[1]+1,5))
                                plt.ylim(boundary[0],boundary[1])
                                plt.plot(stick_path[:,0],stick_path[:,1], alpha=0.4, color='red')
                                plt.scatter(particle_stick[:,0], particle_stick[:,1], alpha=0.5,\
                                            color='blue')
                                plt.pause(pause)
                                plt.show()
                            if with_dots == True and show_path == False:
                                plt.xticks(np.arange(boundary[0],boundary[1]+1,5))
                                plt.xlim(boundary[0],boundary[1])
                                plt.yticks(np.arange(boundary[0],boundary[1]+1,5))
                                plt.ylim(boundary[0],boundary[1])
                                plt.scatter(particle_stick[:,0], particle_stick[:,1], alpha=0.5,\
                                            color='blue')
                                plt.scatter(stick_path[m,0],stick_path[m,1], alpha=0.5, color='blue')
                                plt.pause(pause)
                            if show_path == True and with_dots == True:
                                plt.xticks(np.arange(boundary[0],boundary[1]+1,5))
                                plt.xlim(boundary[0],boundary[1])
                                plt.yticks(np.arange(boundary[0],boundary[1]+1,5))
                                plt.ylim(boundary[0],boundary[1])
                                plt.plot(stick_path[:,0],stick_path[:,1], alpha=0.4, color='red')
                                plt.scatter(particle_stick[:,0], particle_stick[:,1], alpha=0.5,\
                                            color='blue')
                                plt.scatter(stick_path[m,0],stick_path[m,1], alpha=0.5, color='blue')
                                plt.pause(pause)
                                plt.show()
                            else:
                                plt.scatter(particle_stick[:,0], particle_stick[:,1], alpha=0.5,\
                                            color='blue')

                        stick_coords = stick_path[-1:]
                        stuck = True

            if stuck:
                if boundary[0] < stick_coords[:,0] < boundary[1]:
                    if boundary[0] < stick_coords[:,1] < boundary[1]:
                        particle_stick = np.concatenate([particle_stick,stick_coords])
                        break

        if stick_coords[:,0] == 0 and stick_coords[:,1] == 0:
            break

        if show_path == True:
            plt.xticks(np.arange(boundary[0],boundary[1]+1,5))
            plt.xlim(boundary[0],boundary[1])
            plt.yticks(np.arange(boundary[0],boundary[1]+1,5))
            plt.ylim(boundary[0],boundary[1])
            plt.plot(stick_path[:,0],stick_path[:,1], alpha=0.4, color='red')
            plt.scatter(particle_stick[:,0], particle_stick[:,1], alpha=1, color='blue')

        if with_dots == True:
            plt.xticks(np.arange(boundary[0],boundary[1]+1,5))
            plt.xlim(boundary[0],boundary[1])
            plt.yticks(np.arange(boundary[0],boundary[1]+1,5))
            plt.ylim(boundary[0],boundary[1])
            plt.scatter(particle_stick[:,0], particle_stick[:,1], alpha=1, color='blue')
            plt.show()

# rect_att_basic_animation(1,1000, 5)
```

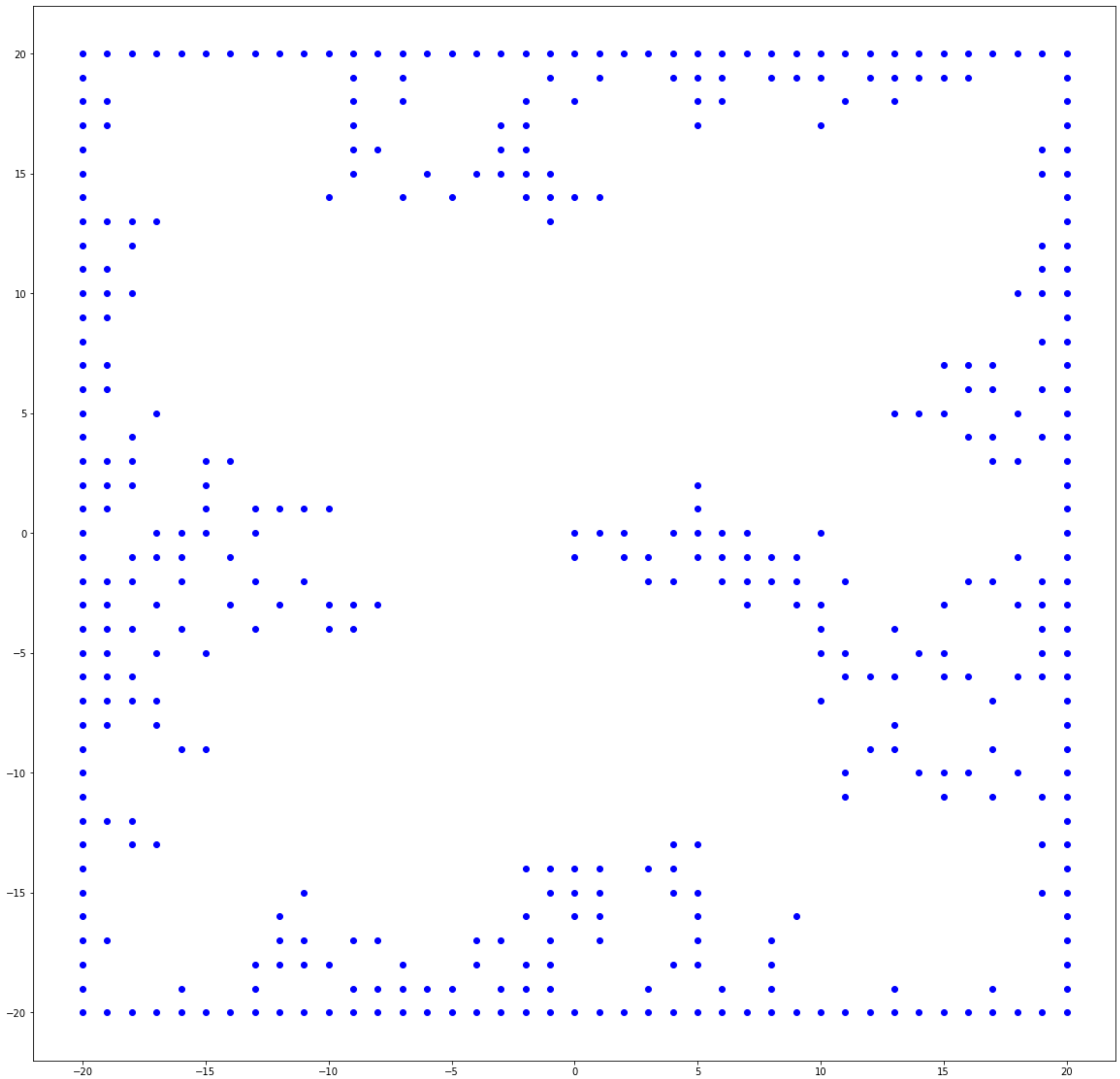
Appendix C: Rectangular Attractor Animation From .PNG Files

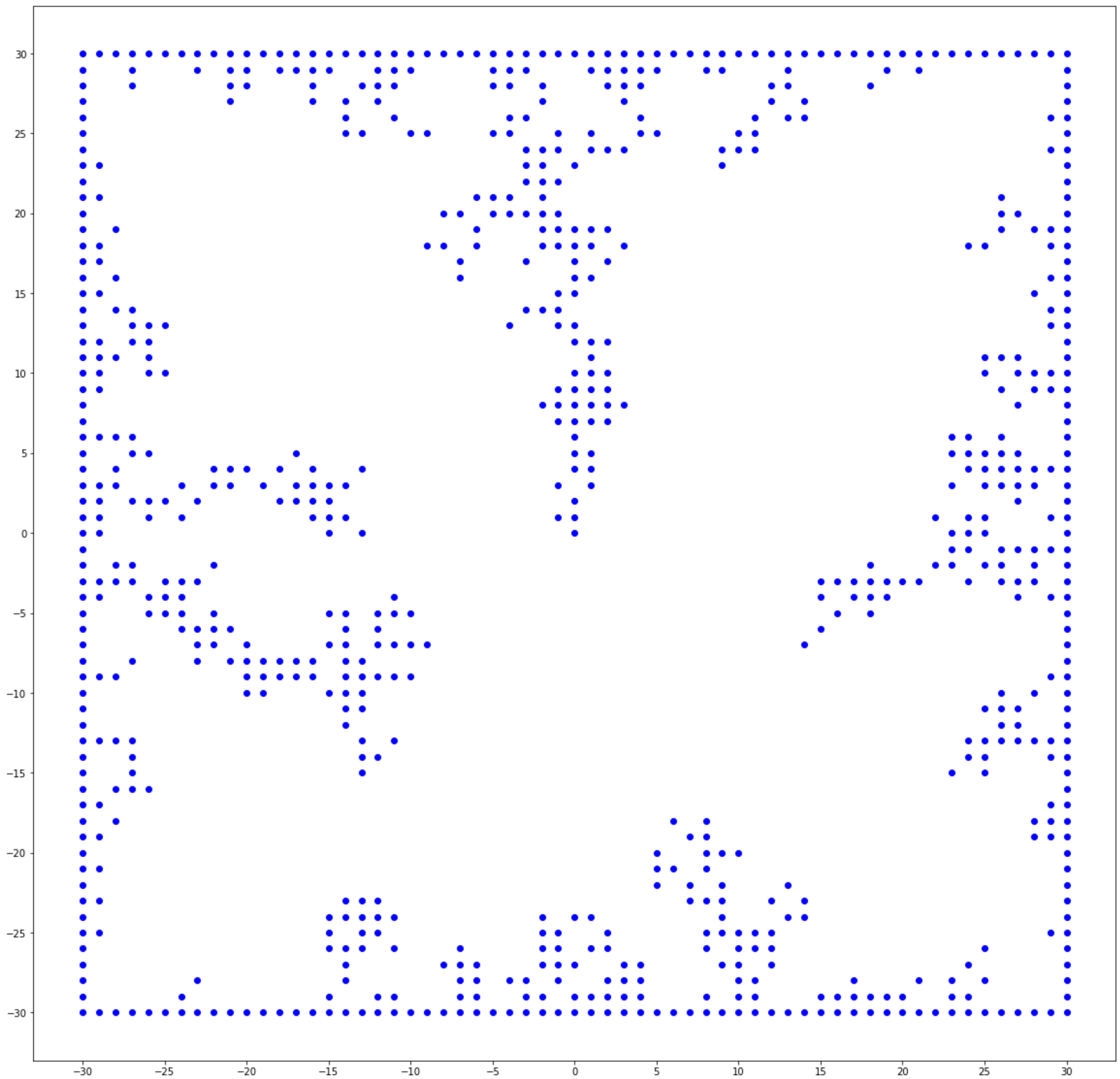
```python
def rect_att_animation(length):

    """
    Instructions
    ----------
    Place this function in a file inside a folder titled Animation on your desktop and set the working
    directory of the spyder console to the folder Animation on your desktop as well. This function
    read and writes data to that folder. You will have to chane the instance class path_dir to your
    own path directory for this progran to work as intended.

    Parameters
    ----------
    length       : integer
                   length from origin to border of square, or the half length of the full square.
    """

    particles = 10000
    step_number = 30000
    pause = 0.0001
    dimension = 2
    step_options = [-1, 0, 1]
    origin = np.zeros((1, dimension))
    boundary = [-length,length]
    stick_coords = np.zeros((1, dimension))
    stick_path = np.zeros((1, dimension))
    file_name = "{:09d}_movie.png"
    file_number_adjustment = 1000

    particle_stick = []
    for i in np.arange(boundary[0],boundary[1]+1):
        particle_stick.append([boundary[1],i])
        particle_stick.append([boundary[0],i])
        particle_stick.append([i,boundary[1]])
        particle_stick.append([i,boundary[0]])

    particle_stick = np.array(particle_stick)
    particle_stick = np.unique(particle_stick, axis=0)

    plt.rcParams["figure.figsize"] = (20,20)
    plt.xticks(np.arange(boundary[0],boundary[1]+1,5))
    plt.xlim(boundary[0],boundary[1])
    plt.yticks(np.arange(boundary[0],boundary[1]+1,5))
    plt.ylim(boundary[0],boundary[1])
    plt.scatter(particle_stick[:,0], particle_stick[:,1], alpha=1, color='blue')

    count = 0

    for j in np.arange(1,particles+1):

        step_shape = [step_number, dimension]
        steps = np.random.choice(a=step_options, size=step_shape)
        path = np.concatenate([origin,steps]).cumsum(0)

        for k in np.arange(0,len(path)):
            stuck = False
            for l in np.arange(0,len(particle_stick)):
                if not stuck:
                    if path[k,0] == particle_stick[l,0] and path[k,1] == particle_stick[l,1:]:
                        for m in range(k):
                            stick_path = np.concatenate([origin,steps[:m]]).cumsum(0)
                            plt.xticks(np.arange(boundary[0],boundary[1]+1,5))
                            plt.xlim(boundary[0],boundary[1])
                            plt.yticks(np.arange(boundary[0],boundary[1]+1,5))
                            plt.ylim(boundary[0],boundary[1])
                            plt.scatter(particle_stick[:,0], particle_stick[:,1], alpha=0.5,\
                                        color='blue')
```

```
                                        color='blue')
                            plt.scatter(stick_path[m,0],stick_path[m,1], alpha=0.5, color='blue')
                            plt.savefig(file_name.format(m+count))
                            plt.pause(pause)
                        stick_coords = stick_path[-1:]
                        stuck = True

            if stuck:
                if boundary[0] < stick_coords[:,0] < boundary[1]:
                    if boundary[0] < stick_coords[:,1] < boundary[1]:
                        particle_stick = np.concatenate([particle_stick,stick_coords])
                        break

        if stick_coords[:,0] == 0 and stick_coords[:,1] == 0:
            break

        plt.xticks(np.arange(boundary[0],boundary[1]+1,5))
        plt.xlim(boundary[0],boundary[1])
        plt.yticks(np.arange(boundary[0],boundary[1]+1,5))
        plt.ylim(boundary[0],boundary[1])
        plt.scatter(particle_stick[:,0], particle_stick[:,1], alpha=1, color='blue')
        plt.savefig(file_name.format(k+1+count))

        count += file_number_adjustment

    frames = []
    path_dir= '/Users/paulzebarth/Desktop/Animation/*.png'

    for filename in sorted(glob.glob(path_dir)):
        frame = cv2.imread(filename)
        height, width, layers = frame.shape
        size = (width,height)
        frames.append(frame)

    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter('rect_att.mp4', fourcc, fps=30, frameSize = size)

    for i in range(len(frames)):
        out.write(frames[i])
    out.release()

# rect_att_animation(5)
```

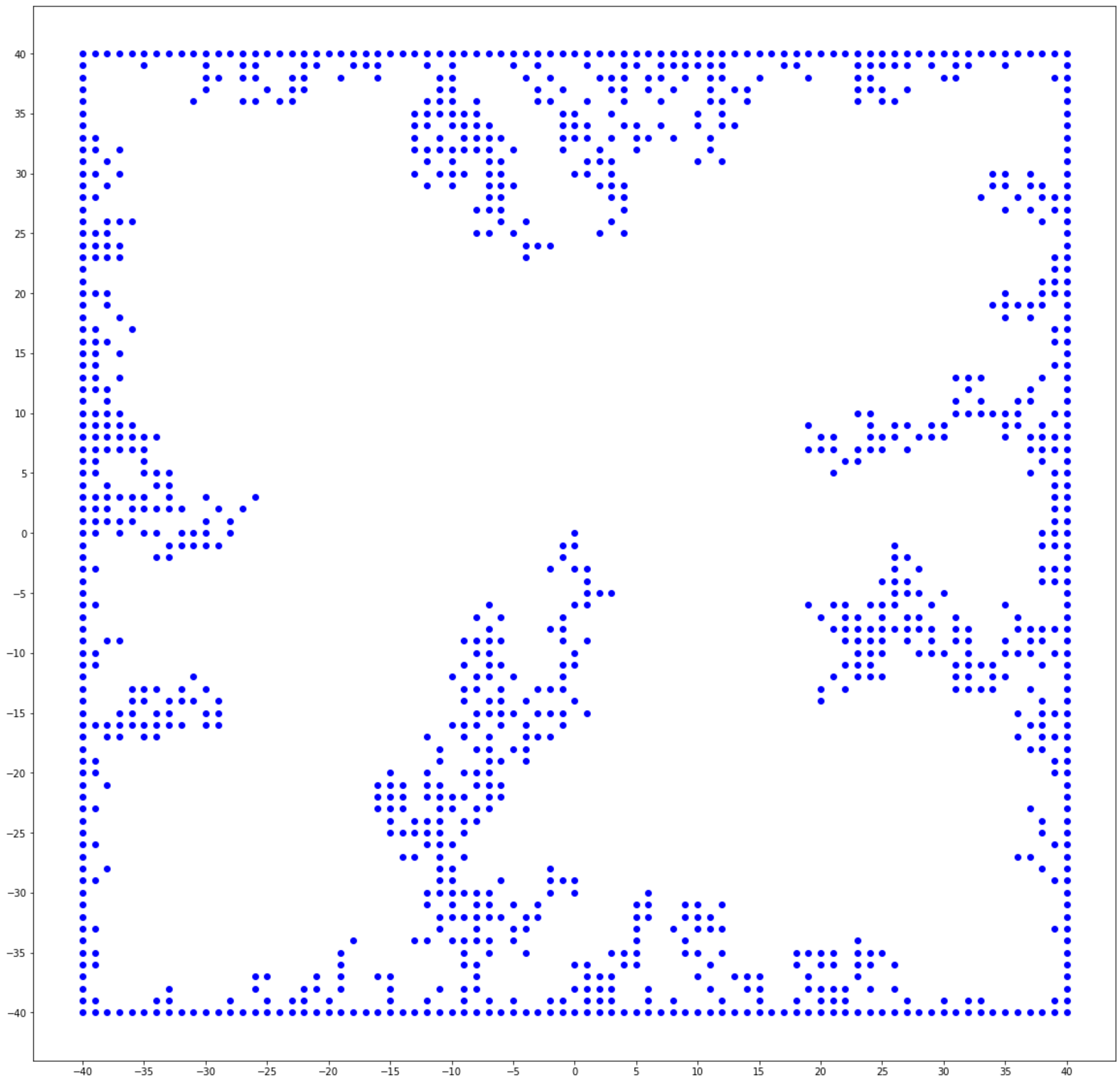
Appendix D



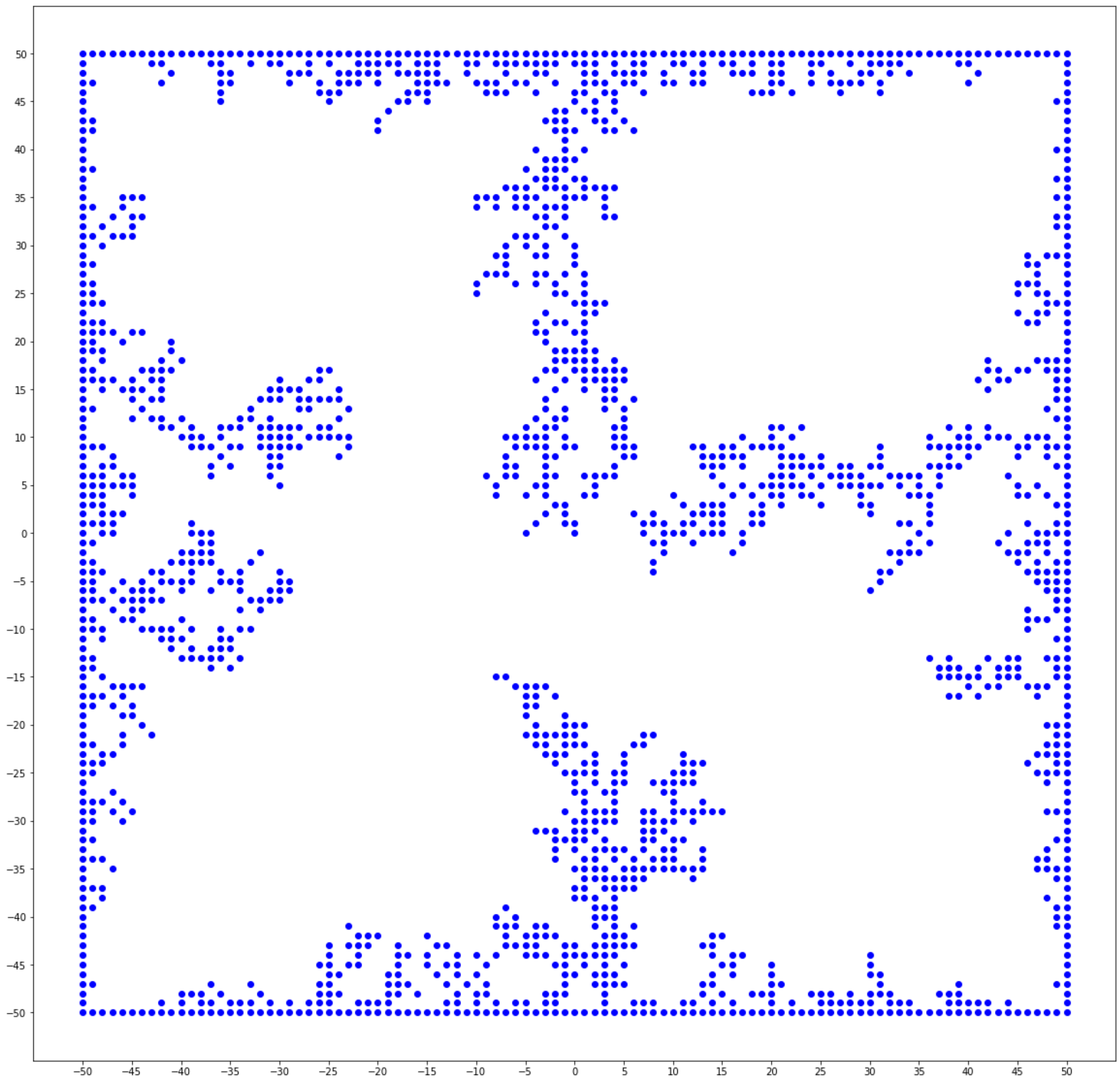

20
15
10
5
0
−5
−10
−15
−20
−20
−15
−10
−5
0
5
10
15
20

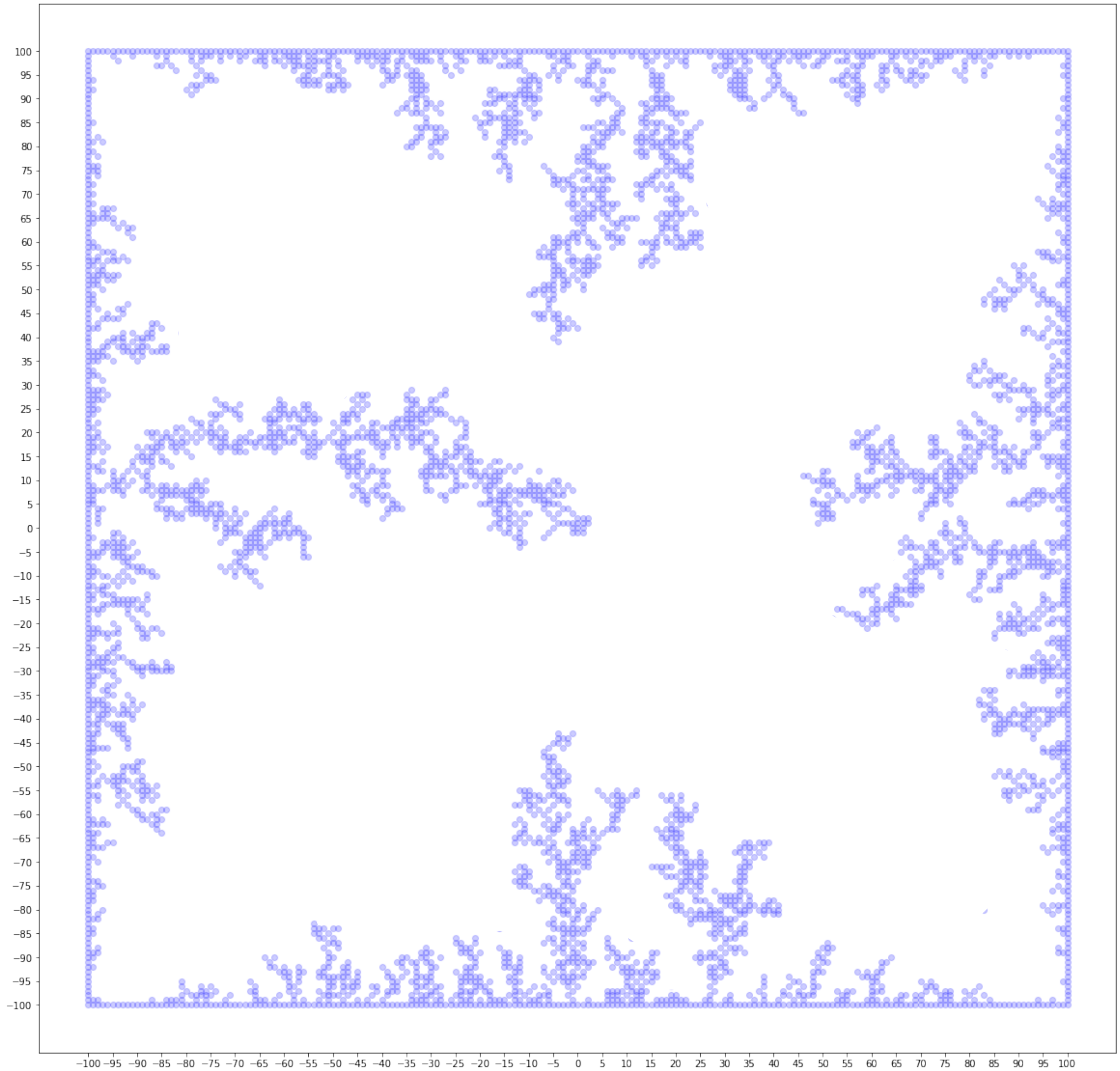


30
25
20
15
10
5
0
−5
−10
−15
−20
−25
−30
−30 −25 −20 −15 −10 −5 0 5 10 15 20 25 30



40
35
30
25
20
15
10
5
0
−5
−10
−15
−20
−25
−30
−35
−40
−40 −35 −30 −25 −20 −15 −10 −5 0 5 10 15 20 25 30 35 40


50
45
40
35
30
25
20
15
10
5
0
−5
−10
−15
−20
−25
−30
−35
−40
−45
−50
−50 −45 −40 −35 −30 −25 −20 −15 −10 −5 0 5 10 15 20 25 30 35 40 45 50


100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
−5
−10
−15
−20
−25
−30
−35
−40
−45
−50
−55
−60
−65
−70
−75
−80
−85
−90
−95
−100
−100 −95 −90 −85 −80 −75 −70 −65 −60 −55 −50 −45 −40 −35 −30 −25 −20 −15 −10 −5 0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100