Fractal Generation in C

Brinda Venkataramani **Physics 2G03**

All code for this project can be found under `~/fractals`. The project can be compiled by typing `make` into the terminal, and run by typing `fractals` into the terminal afterwards. All output files generated can be plotted via `scatter.py`, `bestfit.py`, and `trendline.py` in the Jupyter hub. For more details on plotting, please see usage notes in the final section.

Introduction

In matter, small particles often clump together forming structures whose sizes are greater than the ranges of the forces holding them together[1]. Naturally, such structures are of interest in the physical sciences and engineering due to their unique seemingly force-defying conformations. It is of general interest what other properties they may exhibit. The purpose of this assignment was to generate such structures by implementing a variant of the Eden model for matter clumping in C, and to study their properties. This variant was first proposed and tested by Witten and Sander, in their 1981 paper, *Diffusion-Limited Aggregation, a Kinetic Critical Phenomenon* in *Physical Review Letters*[2].

Description of the Eden Model Variant

The diffusion-limited algorithm first proposed and implemented by Witten and Sander is described by them as follows[3]:

> "Our model is a variant of the Eden model whose initial state is a seed particle at the origin of a lattice. A second particle is added at some random site at large distance from the origin. This particle walks randomly until it visits a site adjacent to the seed. Then the walking particle becomes part of the cluster. Another particle is now introduced at a random distant point, and it walks randomly until it joins the cluster, and so forth. If a particle touches the boundaries of the lattice in its random walk it is removed and another introduced."

Essentially, the algorithm works as follows:

1. A point mass is introduced at the center of the grid.

[1] *Physical Review B*, **27**, 5686 (1983)
[2] *Physical Review Letters*, **47**, 1400 (1981)
[3] *Physical Review Letters*, **47**, 1400 (1981)

2. Next, a particle is randomly added some distance away from this point mass (which I shall refer to as the seed particle henceforth).

3. This particle performs a random walk until one of two things happen.

   (a) If the particle touches the boundary of the lattice/grid, it is removed and another reintroduced.

   (b) If the particle ends up adjacent to the mass (i.e. the clump), it becomes "stuck" and a new particle is introduced.

This process is repeated over N iterations, where N refers to the number of particles which are to be introduced to the lattice.

My implementation of the algorithm as well as a description of my implementation is given below.

Listing 1: Implementation of diffusion limited aggregation (DLA) in C (`fractals.c`)

```c
#include <stdio.h> // For file-I/O.
#include <stdlib.h> // For random number generation.
#include <stdbool.h> // For boolean variables.
#include <time.h> // For time(0).
#include <math.h> // For cos() and sin() functions.

// This function returns either +/- 1.
// It is used when performing a random walk in the horizontal or vertical direction.

int returnMove() {
  int arr[] = {-1,1}; // Array with possible return values.
  int random = rand() % 2; // Returns 0 or 1.
  return arr[random]; // Returns +/- 1 depending on value of random.

}

// This is the main program.

int main() {
  const int max_iter = 1001; // This is the number of particles to introduce.
  const int R = 101; // This is the size of square lattice.
  int center = (R - 1)/2; // This is the origin of the square lattice.
  int grid[R][R]; // We declare a lattice with specified dimensions.

  // These are other variable declarations.
  // The variables i, j, and c, are counting variables.
  // The variables x, y, xn, yn, are used in the random walk.
```

```
28    // Choice is used to determine whether to move horizontally or ↩
          vertically.
29    // The variable theta is used to start particle at random (x,y).
30
31    int i, j, c, x, y, xn, yn, choice;
32    float theta;
33
34    // We declare a file pointer, for when we want to write the fractal ↩
          generated to a file.
35
36    FILE *output;
37
38    // We initialize the random seed.
39    // This is done so that we do not generate the same fractals over and ↩
          over.
40    // The sets the seed to the system time at which the program is started.
41
42    srand(time(0));
43
44    // Now we initialize the clumping grid.
45    // This grid tells us whether a point on the lattice is a part of the ↩
          mass or not.
46
47    for (i = 0; i < R; i++) {
48      for (j = 0; j < R; j++) {
49        grid[i][j] = 0;
50
51      }
52
53    }
54
55    grid[center][center] = 1; // Set the seed at the center.
56
57    // This is the main part of the program where we do random walks with ↩
          the particles we introduce up to a maximum number of particles, ↩
          max_iter.
58
59    for (c = 0; c < max_iter; c++) {
60
61      // The rand() function returns a value between 0 and RAND_MAX. So, ↩
            rand()/RAND_MAX casted to a float explicitly, will return some ↩
            value between 0 and 1. Then, multiplying by 2pi gives us a random ↩
            angle between 0 and 2pi.
62
63      theta = ((float)rand()/(float)(RAND_MAX))*2*M_PI;
64
65      // Then we can define x, y in terms of polar coordinates.
```

```
66      // We set random points on the circle defined by radius of half of the grid size.
67
68      x = center + 0.95*(center - 1)*cos(theta);
69      y = center + 0.95*(center - 1)*sin(theta);
70
71      // Now we do a random walk with the particle we have introduced.
72
73      for (;;) {
74        choice = rand() % 2; // Returns 0 or 1.
75
76        // If 0, move horizontally.
77
78        if (choice == 0) {
79          xn = x + returnMove();
80          if (xn > 0 && xn < (R - 1)) x = xn; // Only move if within bounds.
81
82        }
83
84        // If 1, move vertically.
85
86        if (choice == 1) {
87          yn = y + returnMove();
88          if (yn > 0  && yn < (R - 1)) y = yn; // Only move if within bounds.
89
90        }
91
92        bool found = false;
93
94        // Now we check if we should join the clump or not.
95        // Essentially, if any adjacent space (including diagonals) is part of the clump this particle joins the clump.
96
97        for (i = -1; i <= 1; i++) {
98          for (j = -1; j <= 1; j++) {
99            if (grid[x + i][y + j] == 1) {
100             found = true;
101             break; // Break out of first for.
102
103           }
104
105         }
106
107         if (found == true) break; // Break out of second for.
108
109       }
```

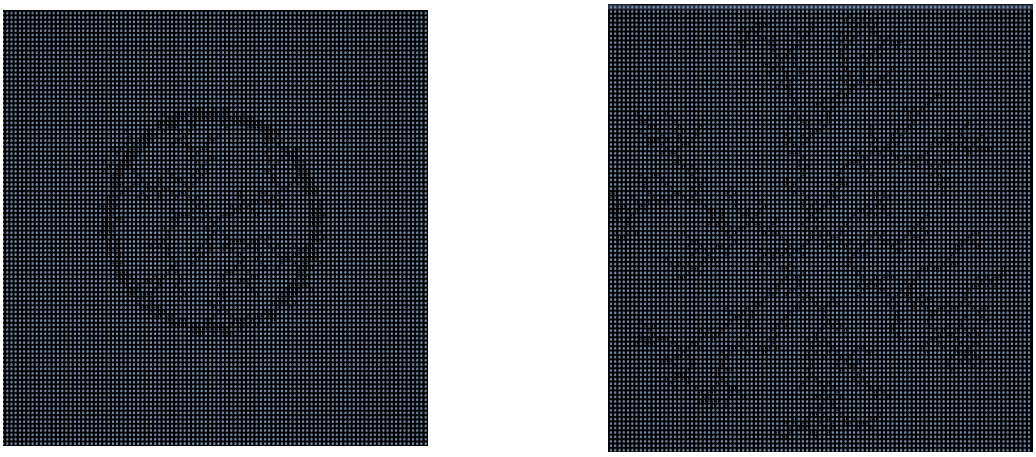
```
110
111         if (found == true) break; // Break out of random walk loop.
112
113       }
114
115       // Set the point at which the particle got stuck to be part of the ↩
              mass.
116
117       grid[x][y] = 1;
118
119       // Go back to the beginning and introduce a new particle.
120
121     }
122
123     // Finally, we output the clumping grid to an output file, "output.dat".
124
125     output = fopen("output.dat", "w");
126
127     for (i = 0; i < R; i++) {
128       for (j = 0; j < R; j++) {
129         fprintf(output, "%i ", grid[i][j]);
130
131       }
132
133       fprintf(output, "\n"); // This is so there is a newline between rows.
134
135     }
136
137     fclose(output); // Close the output file.
138
139 }
```

I'll discuss some algorithm design choices here; namely some constraints on different variables within the algorithm.

First, grid size, R, should be odd. That is, radius of circle inscribed should be odd so as to have a well-defined central point where we set the seed. Testing tells us that the number of iterations should be roughly 5% to 10% of the grid size. For example, 500-1000 steps with a grid size of 101 by 101. Initially, I started with an 11 by 11 grid, and introduced 31 particles, when running tests to ensure the algorithm was working.

Actually, the number of particles to introduce is also reliant on at what radius we are introducing new particles. Consider lines 68 and 69 of Listing 1. Using 0.95 as the scaling factor starts particles near the edge of the grid, so that we can introduce more particles before our branches reach the edges of the grid, where new particles spawn (and can hence aggregate). Consider

the comparison of the following two images both generated on a 101 by 101 grid (Figure 1).



(a) 1000 particles at 0.5 scaling.

(b) 1000 particles at 0.95 scaling.

Figure 1: Graphical relationship between scaling factor and number of particles to introduce.

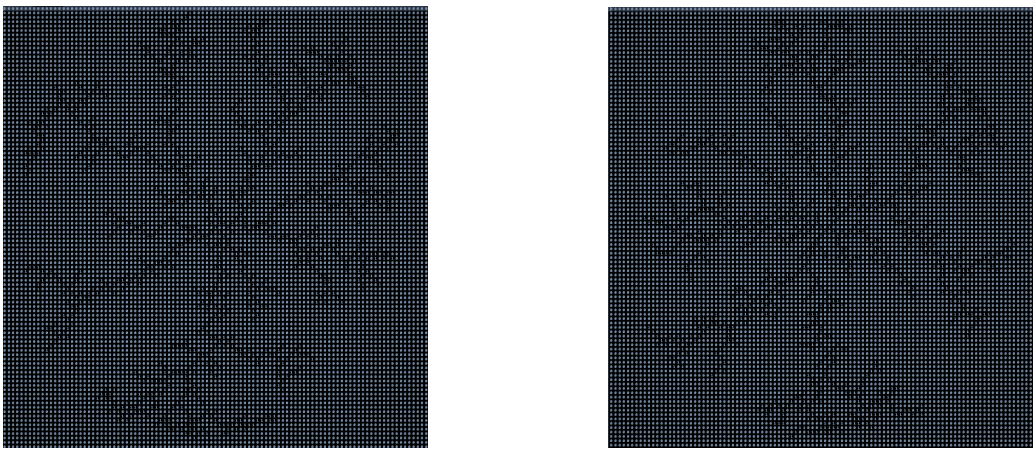
Due to where the particles are being introduced, we see massive clumping on the left image.

Hence, the relations for number of particles to introduce are as follows, where s is the scaling factor, d is the size of the lattice, and n is the number of particles to introduce:

$$n \propto s \tag{1}$$

$$n \propto d \tag{2}$$

Here are some other plain-text plots of some small fractals I generated during the testing phase (Figure 2).



(a) 1000 particles at 0.95 scaling.

(b) 1000 particles at 0.95 scaling.

Figure 2: Plots of 1000 particles at 0.95 scaling on a 101 by 101 grid.

Also of note is the fact that I am using the outer boundaries of the grid as the kill zone. For example, if my lattice size was 11, I only allowed particles to exist on zones 2 through 10 (or in terms of the array, from zones 0 to 9.) This is a purposeful design choice, as I did not want to have to write several cases to check that I was not searching out of bounds of my array for adjacent particles, when checking if the current particle should join the clump or not.

The Makefile for this code is given below.

Listing 2: Makefile for DLA implementation

```
fractals: fractals.c
    gcc -o fractals fractals.c -lm
```

Plotting Fractals in Python

Because of the way the algorithm was implemented, I had to rewrite a plotting script in Python to plot the fractals. (That is, none of the provided scripts worked to plot the fractals.) First, I had to get the set of coordinates we needed to plot from the clumping grid. This is achieved as in Listing 3.

Listing 3: Function to write mass points to .dat file

```
FILE *coords;

coords = fopen("coords.dat", "w"); // Open the file coords.dat in write mode.

// If the location is part of the clump, return the coordinate.

for (i = 0; i < R; i++) {
    for (j = 0; j < R; j++) {
        if (grid[i][j] == 1) {
            fprintf(coords, "%i", arr[i][j]);

        }

    }

}

fclose(coords);
```

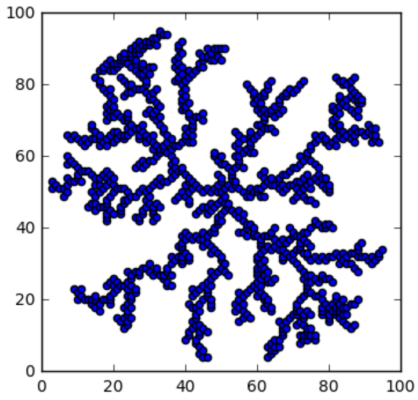
Note that the coordinates generated by this function are such that the fractal is inverted horizontally. This does not have an overall impact on any calculations performed.

Now that there were coordinates, I simply had to write a quick code using `matplotlib` in Python to plot the coordinates as a scatter plot (Listing 4).

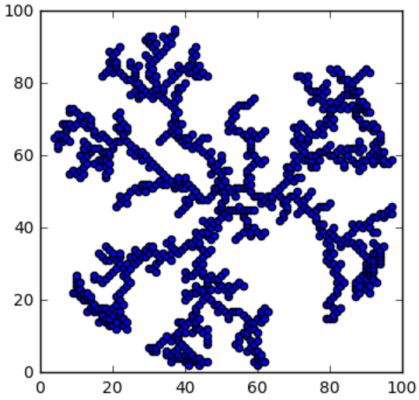
Listing 4: Function to plot fractals using `matplotlib`

```
from matplotlib import pyplot as plt
import matplotlib

f = open("coords.dat","r") # Open the coordinate file.
x, y = [], []

# Get coordinates into lists.

for l in f:
    row = l.split()
    x.append(row[0])
    y.append(row[1])

plt.xlim(0, 100) # Sets range of x.
plt.ylim(0, 100) # Sets range of y.
plt.gca().set_aspect('equal', adjustable='box') # Square plot.
plt.scatter(x, y)
plt.show()
```

Here are some plots generated in Jupyter hub (Figure 3, 4).

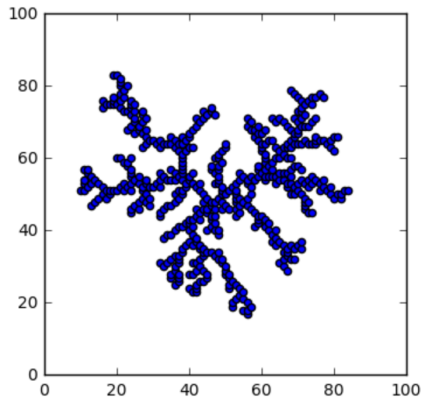


(a) 1000 particles at 0.95 scaling.

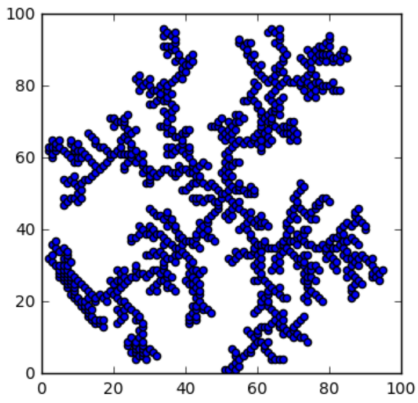


(b) 1000 particles at 0.95 scaling.

Figure 3: Plots of 1000 particles at 0.95 scaling on a 101 by 101 grid using matplotlib in Python.



(a) 500 particles at 0.99 scaling.



(b) 1000 particles at 0.99 scaling.

Figure 4: Plots of a variable number of particles at varying high scalings on a 101 by 101 grid using matplotlib in Python.

Implementing Stickiness

Implementing stickiness is fairly simple. It only remains to introduce a single variable and an extra clause into the if clauses. Essentially, we want to modify two aspects of our codes:

1. First, we want to introduce the concept of stickiness. We can do this by setting an additional condition, that we must not only be adjacent to the mass, but also that we must exceed a certain sticking probability, denoted stickiness in the main code.

2. Next, because we don't necessarily join the mass if we are adjacent, we acknowledge that there is a probability that the particle could try and move onto the mass itself. To counter this, we introduce an additional constraint when doing the random walk, that we cannot walk onto a spot where the clumping grid is 1 (i.e. a spot which is already occupied).

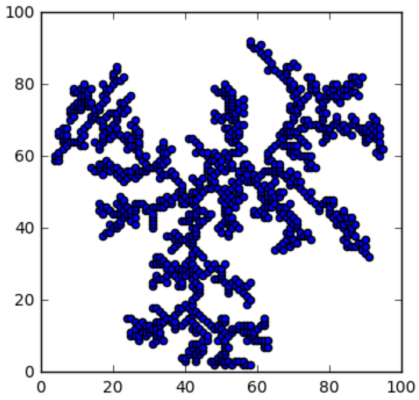
The above can be achieved by the following changes to Listing 1 (Listing 5).

Listing 5: Modification to main code to add stickiness

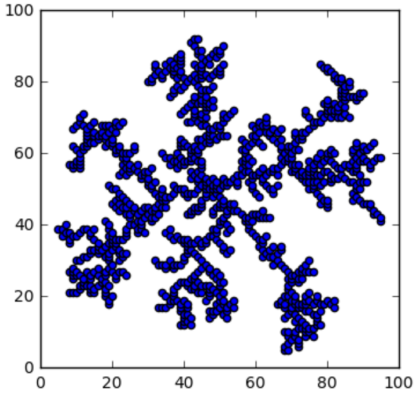
```
1  const float stickiness = 0.3; // Input desired stickiness. Higher ↩
       stickiness makes it harder for the particle to join the clump.
2  float stick; // Random stick factor calculated in random walk loop. Used ↩
       in if validation.
3
4  .
5  .
6  .
7
8  // If 0, move horizontally.
9
10 if (choice == 0) {
11      xn = x + returnMove();
```

```
12      if (xn > 0 && xn < (R - 1) && grid[xn][y] != 1) x = xn; // Only move ↩
            if within bounds and not already occupied.
13
14  }
15
16  // If 1, move vertically.
17
18  if (choice == 1) {
19      yn = y + returnMove();
20      if (yn > 0  && yn < (R - 1) && grid[x][yn] != 1) y = yn; // Only move ↩
            if within bounds.
21
22      }
23
24  .
25  .
26  .
27
28  stick = (float)rand()/(float)(RAND_MAX);
29
30  for (i = -1; i <= 1; i++) {
31      for (j = -1; j <= 1; j++) {
32          if (grid[x + i][y + j] == 1 && stick > stickiness) {
33              found = true;
34              break; // Break out of first for.
35
36          }
37
38  .
39  .
40  .
```

Here are some plots with stickiness greater than 0 (Figure 5, 6).

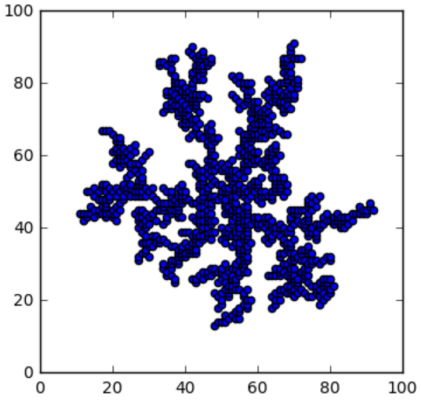


(a) Stickiness of 0.6.

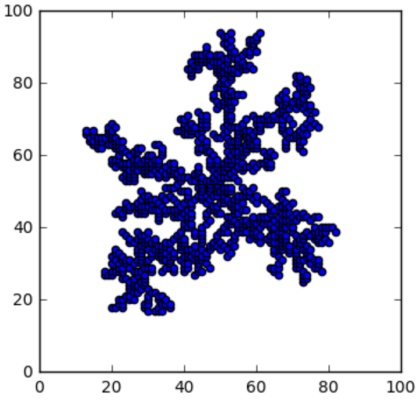


(b) Stickiness of 0.6.

Figure 5: 1000 particles on a 101 by 101 lattice with 0.95 scaling and 0.6 stickiness.



(a) Stickiness of 0.9.



(b) Stickiness of 0.9.

Figure 6: 1000 particles on a 101 by 101 lattice with 0.95 scaling and 0.9 stickiness.

Stickiness evidently results in greater aggregation of particles along branches, resulting in heavier clumping. This is explored when calculating fractal dimension in the next section.

Calculating Fractal Dimension

Using the file `coords.dat`, it was very easy to calculate fractal dimension. It was simply a matter of calculating the distance from each point in the mass to the seed at the origin, and checking whether or not that distance exceeded the radius. If not, then the number of points with that set radius should be incremented. A function to do this exactly, is given below (Listing 6). (Note that the main loop is included in the main loop in the main project file.)

Listing 6: Function to calculate fractal dimension

```
1 // This calculates the distance between two points on the xy-plane.
2
3 float getDistance(int x, int y, int x_0, int y_0) {
4   float d = sqrt(((x-x_0)*(x-x_0))+((y-y_0)*(y-y_0)));
```

```
  return d;

}

// This is included in the main loop of fractals.c.
// Function remains the same, but some modifications are made to variable ↩
    names.

int main() {
  FILE *input;
  int xr, yr;
  const float r = 50; // This is r in C(r).
  int t = 0; // Tally for number of points within set radius, r.
  int x0 = center;
  int y0 = center;

  input = fopen("coords.dat", "r");

  // Read until empty line which I know is in coordinates.dat.

  for (;;) {
    int nread = fscanf(input, "%i %i\n", &x, &y);
    if (!(nread >= 1)) break;
    if (getDistance(x, y, x0, y0) < r) {
      t++;

    }

  }

  fclose(input);

  printf("%i", t); // Print r^D.
```

It is easier to append this to a file, along with r, and then to plot them against each other with a line of best fit using matplotlib in Python. We can also plot r^2 on the same axis to demonstrate that fractal dimension is r^D, with $1 < D < 2$. We append to the file by modifying the print statement to the following (Listing 7):

Listing 7: Modification to print statement for plotting with matplotlib

```
// printf("%i", t); // Print r^D.

// Do the following instead.

```

```
5  fprintf(fractaldim, "%i $i %i %i", (int)r, t (int)r, (int)r^2); // Note ↩
       fractaldim is open in "a" mode.
```

It remains to plot some trends. Figure 7 shows plots of r against r, r^2, and $C(r)$.
We can use the following Python code to plot trendlines (Listing 8):

Listing 8: Plotting trendlines of fractal dimension using matplotlib

```
1  from matplotlib import pyplot as plt
2  import matplotlib
3
4  f = open("dimension.dat","r")
5  r, ry, t, r2 = [], [], [], []
6
7  for l in f:
8      row = l.split()
9      r.append(row[0])
10     ry.append(row[1])
11     t.append(row[2])
12     r2.append(row[3])
13
14 fig, ax = plt.subplots()
15 plt.plot(r, ry, label = "r")
16 plt.plot(r, r2, label = "r^2")
17 plt.plot(r, t, label = "C(r)")
18 legend = ax.legend(loc='upper left', shadow=True) // Set legend for figure↩
       .
19 plt.show()
```

Here is a plot of r against, r, r^2, and $C(r)$, showing that fractal dimension $1 < D < 2$ (Figure 7).

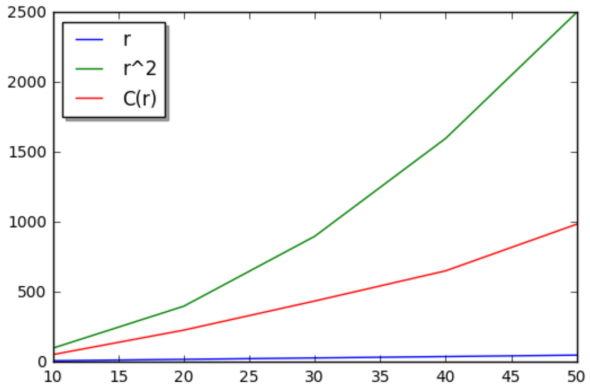
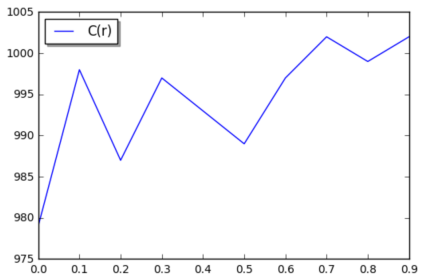
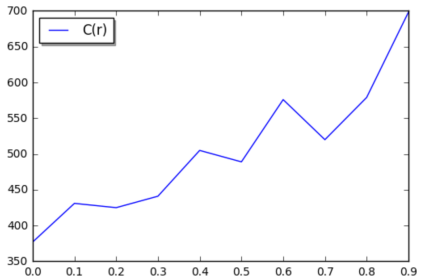


Figure 7: Plot of r, r^2, and $C(r)$ for a 1000 particle cluster on a 101 by 101 grid with no stickiness. $r < C(r) = r^D < r^2$ demonstrating that $1 < D < 2$ as desired.

We can also plot trendlines for fractal dimension against stickiness (Figure 8).



(a) C(50)



(b) C(30)

Figure 8: Fractal dimension with 1002 particles on 101 by 101 lattice with varying stickiness. Positive correlation seems to exist between increasing stickiness due to the clumping illustrated in Figures 5, 6, and fractal dimension.

It is also possible to plot fractal dimension against stickiness using a line of best fit. This is achieved with the following Python script (Listing 9). See usage notes for more details on how to generate the most optimal plots.

Listing 9: Plotting lines of best fit for r vs $C(r)$ and stickiness versus $C(r)$

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # Get values.
5
6  r, ry, t, r2, s = np.loadtxt("dimension.dat", usecols=(0,1,2,3,4), ↩
       delimiter =" ", unpack = True)
7  fig, ax = plt.subplots()
8
9  # Comment out first three for stickiness plots.
10 # Comment last line for fractal dimension plots.
11
12 plt.scatter(r, ry)
13 plt.scatter(r, t)
14 plt.scatter(r, r2)
15 # plt.scatter(s, t)
16
17 # Comment out first line for C(r) versus r graphs.
18 # Comment out second line for stickiness versus C(r) graphs.
19
20 # plt.plot(np.unique(s), np.poly1d(np.polyfit(s, t, 1))(np.unique(s)), ↩
       label= "C(r)")
21 plt.plot(np.unique(r), np.poly1d(np.polyfit(r, ry, 1))(np.unique(r)), ↩
       label = "r")
22 plt.plot(np.unique(r), np.poly1d(np.polyfit(r, t, 1))(np.unique(r)), label↩
        = "C(r)")
23 plt.plot(np.unique(r), np.poly1d(np.polyfit(r, r2, 1))(np.unique(r)), ↩
```

```
    label = "r^2")
legend = ax.legend(loc='upper left', shadow=True)

plt.show()
```

Here are some more plots generated (Figure 9). Again, we can see that $1 < D < 2$.

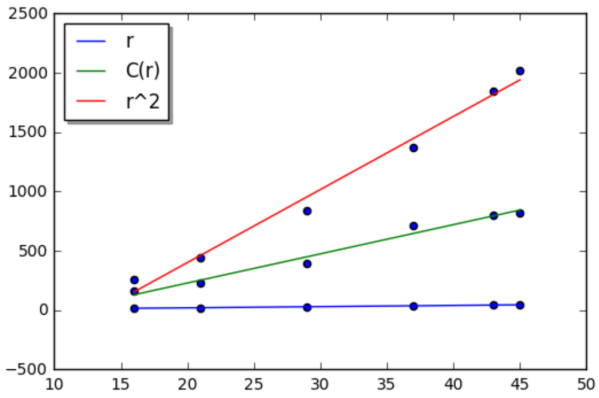


Figure 9: Plot of r, r^2, and $C(r)$ with lines of best fit for a 1000 particle cluster on a 101 by 101 grid with no stickiness. $r < C(r) = r^D < r^2$ demonstrating that $1 < D < 2$ as desired again.

We can also see that increasing stickiness results in higher $C(r)$ (Figure 10).

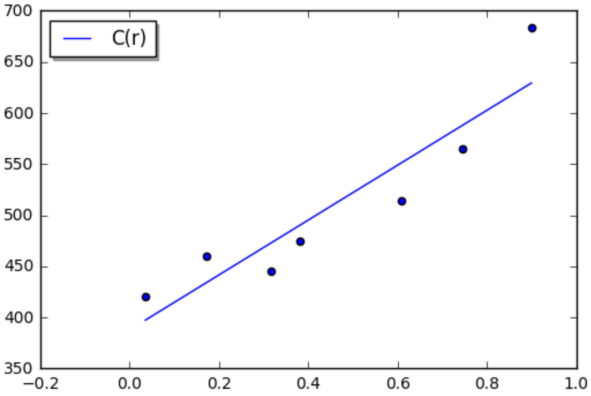


Figure 10: Plot of stickiness versus and $C(30)$ with line of best fit on 1000 particle cluster on a 101 by 101 grid with variable stickiness. Positive correlation between stickiness and fractal dimension is demonstrated.

Usage Notes and Limitations

This section outlines how to best generate different plots.

Plotting Fractals

Compile and run the program. Then, use `\%load scatter.py` in Jupyter hub. Running the program multiple times will replace the current fractal with a new one.

Trendlines

Note do not randomize r or stickiness when attempting to plot trends. To plot trends, do the following:

1. In `fractals.c` Make sure you are not randomizing stickiness or r.
2. Compile and run the program at desired r or stickiness.
3. Edit `fractals.c` to another desired r or stickiness and run again.
4. Run as many times as desired.
5. Then, load `trendline.py`, comment/uncomment the appropriate lines and run the script.

Note that before changing what you are plotting (i.e. r or stickiness), delete, `dimension.dat`.

I do not recommend plotting anything with this as it is slow and clunky. The `bestfit.py` script achieves the same results much more neatly.

Lines of Best Fit

To generate plots with lines of best fit, do the following:

1. In `fractals.c`, choose whether you want to randomize stickiness or r by commenting/uncommenting the appropriate lines. Do not randomize both.
2. Compile and run the program multiple times. For best results run the program more than three times.
3. Load `bestfit.py` and comment/uncomment the appropriate lines.
4. Run `bestfit.py`.

Note that before changing what you are plotting (i.e. r or stickiness), delete, `dimension.dat`.

Limitations

The only real limitation of the program is the inability to automatically run it N iterations. Aside from that, all results approach theoretical/experimental limits as demonstrated in this report.