Johnathan von der Heyde – 2019

Project 8 – Diffusion Limited Aggregation

In this project we simulate and study Diffusion Limited Aggregation (DLA), which, reasonably enough, is any diffusive system limited by aggregation (the sticking together of particles). More aptly though is the idea of aggregate systems limited by diffusion. The difference simply puts emphasis on what is limiting what, and in the latter case, it's aggregation which is limited – as the system still diffuses as though it weren't aggregating at all. We will see such a difference when we come to Eden clusters next project.

Here above we see two examples of DLA. On the left is a copper sulfate cluster, which grows in solution collecting charged particles. On the right is a *Litchtenberg figure* capturing the effect of electric discharge in plexiglas. In either case we see fractal tree-like figures, where the structure *diffuses* out to fill the available space, avoiding only itself. This should remind us of stochastic systems and feedback loops.

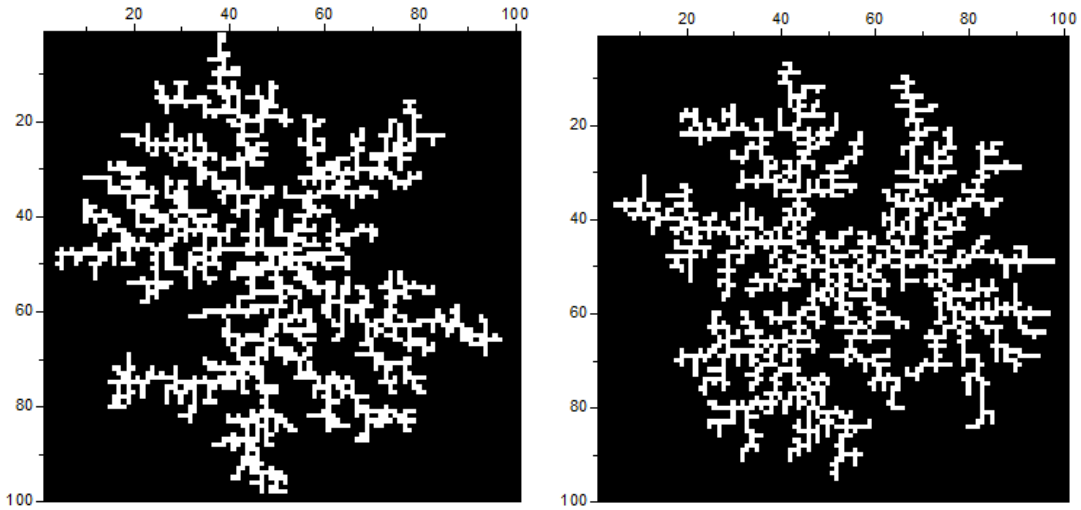
As in the previous projects, the basic method stems from 2 main types: Either we track the particles as they grow, or we track the grid within which they grow. And having gotten familiar with stochastic and diffusive systems, we find the simulation reasonably straightforward, with the only challenging part being the conditional statements which prevent the structure from crossing over itself.

This bit of code can be used for either method (here for gridding), whereby a new particle can be placed in a random new spot if and only if that spot has *n* neighbors.

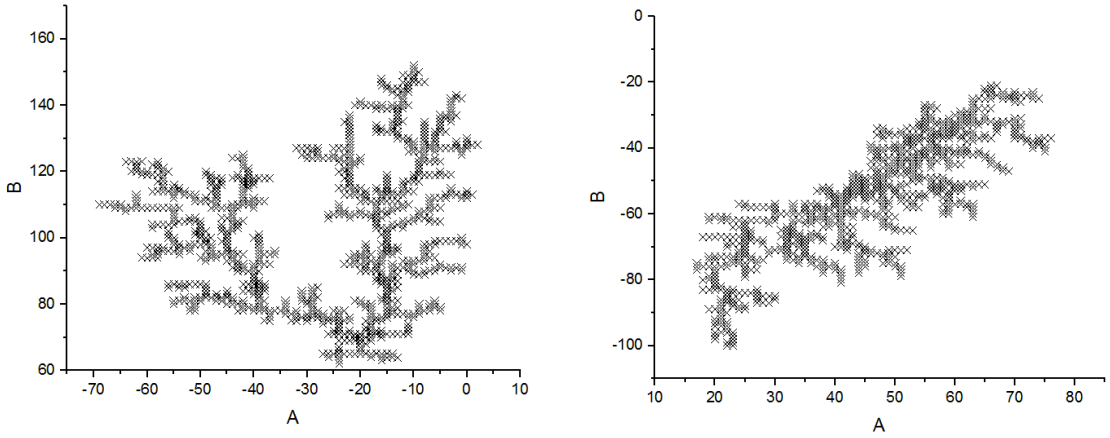
```
static int checkNeighbors(int x, int y){

        int neighbors = 0;
        if (x > 0) neighbors += grid[x-1][y];
        if (x < max-1) neighbors += grid[x+1][y];
        if (y > 0) neighbors += grid[x][y-1];
        if (y < max-1) neighbors += grid[x][y+1];
        return neighbors;
}
```

The if-conditions merely ignore boundary cases, as any point inside will satisfy all 4.

To be clear, such a method does not by itself guarantee no overlap – for that we must demand the return value be 1. Yet as we'll see, so long as the return value is greater than 0, the results are similarly good. Also, removing this condition yields diffusive systems – it is *the* parameter for our aggregation. Below we see some overlap, left, and no overlap, right – each with 2000 iterations (or points).



These were *seeded* (to start the structure) in the middle, thus they tend to radiate. They can be seeded from the bottom, so as to grow a (face-on) tree-like structure.



Above we see how they can grow (via tracking method), as seeded from bottom (left), and how even middle seeds don't necessarily grow radially (right). This is all thanks to their stochastic nature. To put simply, as I've said before, all it takes for these structures to form is that they *grow where they've grown before*.

To qualify these structures, we use what is known as *fractal dimensionality*, which essentially asks how much of their dimension do they fill? The term fractal here is apt, as it simply short for 'fractional'.

To calculate the fractional dimension, $\mathcal{F}$, we first take note of the fact that dimensional space d has ordinal differentiable elements ϵ

$$d(\epsilon) \propto r^d$$

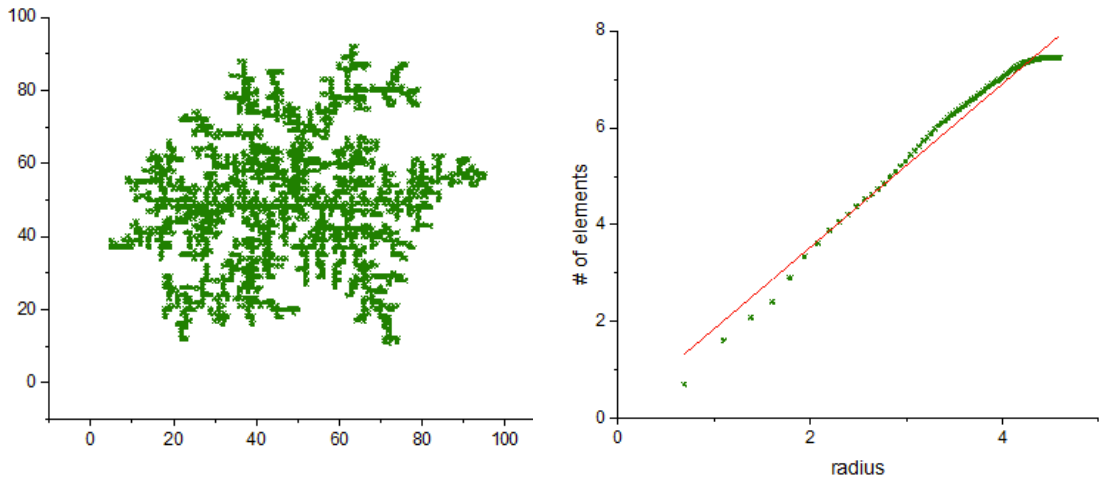
where r is the integration scope, itself a function of ϵ.

For instance, if $d = 1$, the number of elements is proportional to the line, r^1.
And if $d = 2$, the number of elements is proportional to the plane, r^2.
Etc.

Therefore, for $\mathcal{F}$, we simply solve for d, and allow it to be fractional.

$$\mathcal{F} \propto \frac{\ln\left(\mathcal{F}(\epsilon)\right)}{r}$$

This means our fractional dimension can be calculated as the log of the number of elements per radii. And to simplify further, we realize that if we consider instead $ln(r)$, we have a log-log plot, the slope of which is $\mathcal{F}$.

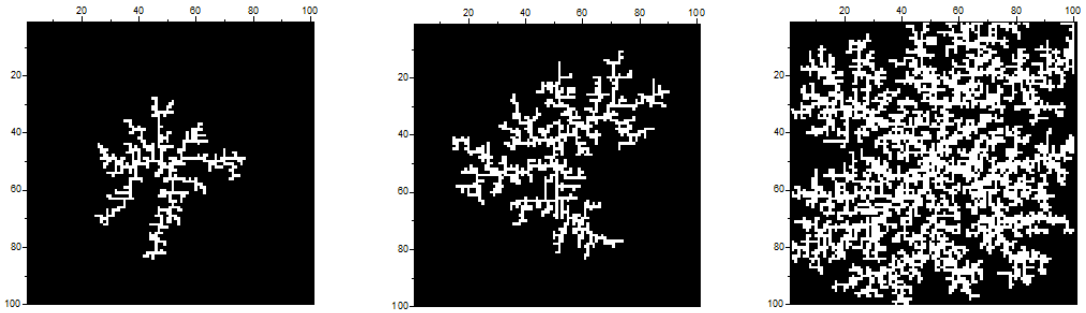


Here we take a nice olive cluster and plot its corresponding fractional dimension $\mathcal{F}$. This one has a measure of 1.687. For these particular clusters, the dimension is usually between 1.2 and 1.9. That's because they generally fill half the 2d space – and so measure about 1.5 dimensions. As you may notice, the log/log plot tends to peter out at the tip, where empty space begins. We may cut this data out, to get a better measure for $\mathcal{F}$, but in this case it's good enough.

Below are 2 algorithmic implementation of the $\mathcal{F}$ computation – in Java and Fortran. Both functions sample and count atoms within growing radii. The Java function does this by spiraling out from the center of the grid; the Fortran program already knows where the atoms are, since it is a tracking method, so it simply cycles through them.

```
static void calcFrac(){

    int x=0, y=0, d=1, rad=1, sum=0, mid=max/2;

    while (rad<max-1){
      while (2*x*d < rad){
        if (grid[x+mid][y+mid]!=0) sum++;
        x+=d;
      }
      while (2*y*d < rad){
        if (grid[x+mid][y+mid]!=0) sum++;
        y+=d;
      }
      //print out log(rad) and log(sum) here
      d*=-1; rad++;
    }
}
```

```
Subroutine calcFrac(){

    do i=2,max
      rad=i
      is=0
      do j=1,atoms
        x=iclx(j)
        y=icly(j)
        if(sqrt(x*x+y*y)<=rad) then
          is=is+1
        endif
      enddo
      sum=is
      !print out log(rad) and log(sum) here
    enddo
}
```

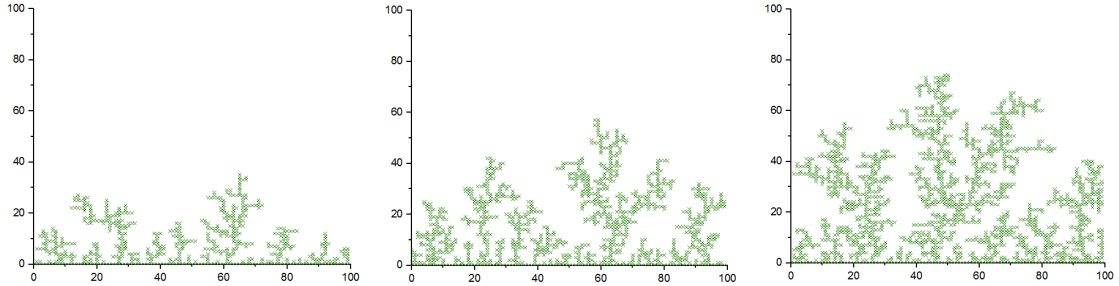


Here we see clusters fill their space, going from 500 to 1000 to 5000 iterations, with respective $\mathcal{F}$ measures of 1.45, 1.53 and 1.88.

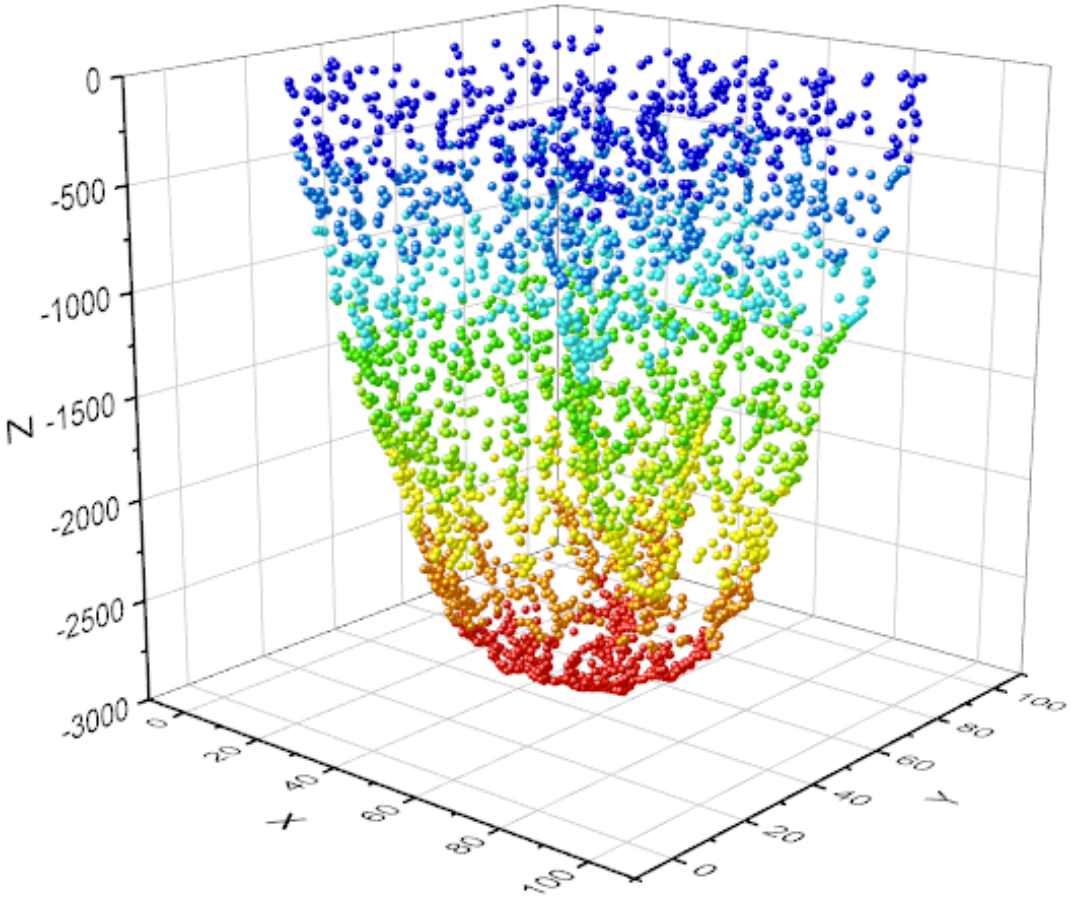
—

Since there isn't much more to learn here, the rest will be exploration.

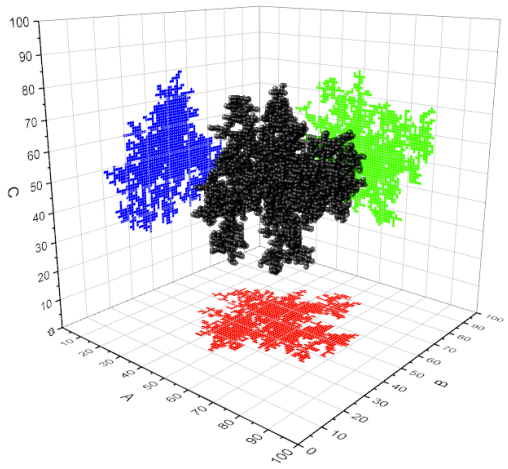
As the book suggests, we can also see how these manifest when seeded along a whole base line. As a result, they grow much like wild shrubbery, or perhaps coral.

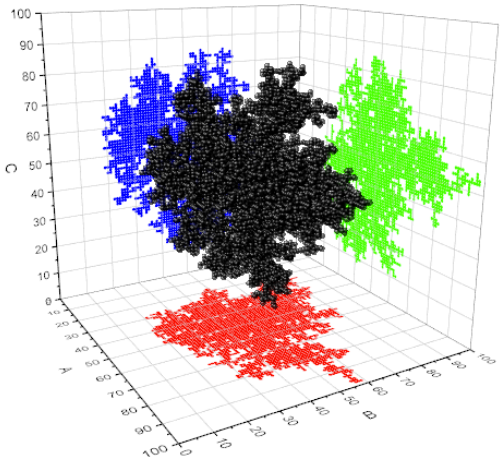


Additionally, we can see what's going on in 3D – firstly with time as the 3rd dimension. While it's certainly pretty, it unfortunately doesn't reveal much novelty.



Below we see what happens when they're allowed to grow in 3D.





Below is the full Java code for gridding in 2d, followed by a tracking method pulled from provided Fortran code. For reasons beyond me, the tracking method prefers not to be radially symmetric and does not take well to isolated seeds. Perhaps something to do with its unique sampling method. Nevertheless this uniqueness provides very special clusters.

```
//Johnathan von der Heyde 2019
//Simulates DLA cluster growth
import java.security.SecureRandom;
import java.util.Scanner;
import java.io.FileWriter;
import java.io.IOException;
import java.lang.Math;

public class x2d {

  static int max = 100;
  static int iteration = 5000;
  static double grid[][] = new double[max][max];
        static SecureRandom rand = new SecureRandom();

  public static void main(String[] args) {

        int x=max/2, y=max/2, count=0;

        grid[x][y] = 1;

        while (count<iteration){

                x = returnRandomInt(0, max-1);
                y = returnRandomInt(0, max-1);

                while (checkNeighbors(x, y) == 0){

                        switch (returnRandomInt(1, 4)){
                                case 1: x = (x+1)%max;      break;
                                case 2: x = (x+max-1)%max;      break;
                                case 3: y = (y+1)%max;          break;
                                case 4: y = (y+max-1)%max;      break;
                        }
                }

                grid[x][y]++;

      System.out.println(count);

                count++;
        }

   calcFrac();

        writeGrid();
}
```

```
static int checkNeighbors(int x, int y){

        int neighbors = 0;
        if (x > 0) neighbors += grid[x-1][y];
   else grid[x][y]=0;
        if (x < max-1) neighbors += grid[x+1][y];
   else grid[x][y]=0;
        if (y > 0) neighbors += grid[x][y-1];
   else grid[x][y]=0;
   if (y < max-1) neighbors += grid[x][y+1];
   else grid[x][y]=0;
        return neighbors;
}

static void calcFrac(){

   try { FileWriter out = new FileWriter("dimesnion");

      int x=0, y=0, d=1, rad=1, sum=0, mid=max/2;

      while (rad<max-1){
         while (2*x*d < rad){
            if (grid[x+mid][y+mid]!=0) sum++;
            x+=d;
         }
         while (2*y*d < rad){
            if (grid[x+mid][y+mid]!=0) sum++;
            y+=d;
         }
         d*=-1; rad++;
         out.write(Math.log(rad) + "," + Math.log(sum) + "\n");
      }
      out.close();
   }
   catch (IOException e) {e.printStackTrace();}
}

static void writeGrid(){

        int count = 0;

        try { FileWriter out = new FileWriter("grid");
                for (int i=0; i<max; i++) {
                        for (int j=0; j<max; j++) {
           out.write(grid[i][j] + ",");
                        }
        out.write("\n");
                }
           out.close();
   }
   catch (IOException e) {e.printStackTrace();}
}

static int returnRandomInt(int a, int b) {
   return rand.nextInt(b-a+1)+a;
```

```
   }
} //END

//Johnathan von der Heyde 2019
import java.security.SecureRandom;
import java.util.Scanner;
import java.io.FileWriter;
import java.io.IOException;
import java.lang.Math;

public class dla2 {

   static SecureRandom rand = new SecureRandom();
   static int max = 1000;
   static int walkers = 5000;
   static int clusterX[] = new int[max];
   static int clusterY[] = new int[max];

   static int atoms=0, atomX, atomY, t, steps=1000;

   public static void main(String[] args) {

         int i, j, rand;
         double radius, diffX, diffY, doubX;

         clusterX[0] = clusterY[0] = 1;

         while (atoms<max-1) {

               System.out.println(atoms);

               radius=0.0;

               for (i=0; i<atoms; i++) {

                     if (Math.abs(clusterX[i])>radius) {
                           radius=Math.abs(clusterX[i]);
                     }
                     if (Math.abs(clusterY[i])>radius) {
                           radius=Math.abs(clusterY[i]);
                     }
               }

               radius+=5;

          converge:

               for (int w=0; w<walkers; w++) {

                     doubX = (returnRandomDouble()*radius);
                     atomX = (int)doubX;
                     if (returnRandomInt(0,1)==1) atomX *= -1;
                     atomY = (int)(Math.sqrt(radius*radius - doubX*doubX));
                     if (returnRandomInt(0,1)==1) atomY *= -1;
```

```
                        for (t=0; t<steps; t++) {

                                switch (returnRandomInt(1,4)) {
                                        case 1: atomX++; break;
                                        case 3: atomX--; break;
                                        case 2: atomY++; break;
                                        case 4: atomY--; break;
                                }

                                for (i=0; i<atoms; i++) {

                                        diffX = atomX - clusterX[i];
                                        diffY = atomY - clusterY[i];

                                        if (Math.sqrt((diffX*diffX)+(diffY*diffY))<=1.00001) {
                                                break converge;
                                        }
                                }
                        }
                }

                        atoms++;

                clusterX[atoms] = atomX;
                clusterY[atoms] = atomY;
        }

        writeGrid();
        }

  static void writeGrid(){

        try { FileWriter out = new FileWriter("2dout");
                for (int i=0; i<atoms; i++) {
                        out.write(clusterX[i] + "," + clusterY[i] + "\n");
                } out.close();
        }
    catch (IOException e) {e.printStackTrace();}
  }

  static int getInt() {
            Scanner scr = new Scanner(System.in);
            while (!scr.hasNextInt())
              scr.next();
            return scr.nextInt();
  }

  static Double returnRandomDouble() {
    return rand.nextDouble();
  }

  static int returnRandomInt(int a, int b) {
    return rand.nextInt(b-a+1)+a;
  }
} //END
```