Apple Bread Challenge Pt 3

Shankha Suvra Dam

2024-03-28

Section 1: What is Diffusion-Limited Aggregation (DLA)?

Diffusion-Limited Aggregation (DLA) is a phenomenon in which particles, driven by Brownian motion, gradually cluster together to form larger aggregates. This process occurs as individual particles undergo random walks and come into contact with each other, which leads to the formation of complex (and often fractal-like) structures. The clusters formed in DLA are called *Brownian trees*. These structures exhibit a fractal dimension (≈ 1.71 in 2D) and resemble natural patterns like snowflakes and plant growth.

DLA can be observed in various systems, including the likes of dielectric breakdown and mineral deposits. It also provides us valuable insights into how randomness and diffusion lead to complex, branching structures in nature and computer simulations.

Section 2: Algorithm used to simulate DLA

There are a lot of possible approaches, and hence algorithms used to simulate DLA, and I shall be explaining the one I have used in a bit, however I'd also like to elaborate on how theoretically one is supposed to simulate DLA.

So, we do this in $\mathbb{Z}^2$, and start off with releasing particles at *infinity*, and then perform nearest-neighbour random walks until the walk attaches at the first site visited in the neighbourhood of the cluster (which initially just consists of the origin). The tree keeps on developing and expanding as the number of random walks increases and we get the beautiful Brownian trees as a result.

Now, there's another concept which I've implemented, which is known as the `stickiness factor` (p), which is the probability of a walk to end at a site visited in the neighbourhood of the cluster.

Since we're simulating this on a computer and not just in theory, we need to place some restrictions on our simulation to make it feasible to represent.

1. The starting point cannot be infinity for obvious reasons, and we'll need to start from at some particular distance so as to be able to actually simulate the walk. So we shall be taking the distance in our case to 500, so as to get a good representation.
2. We'll need to limit the number of points we have in the graph, since firstly, we only have a finite number of starting points, and as the cluster grows, there will come a point after which the starting points themselves would be either directly in the neighbourhood of the cluster, or inside the cluster itself, which is not what we desire. Secondly, we have a limited amount of computing resources, which makes it imperative for us to limit the number of points as we cannot let the code just go on forever infinitely.

3. We will also need to limit the range of the walk, since if we allow the walk to be unrestricted and go throughout $\mathbb{Z}^2$, the time taken to simulate just one walk would be humongous, and here, to get a good idea of the tree, we require atleast a few hundred walks, if not thousands. Hence, I shall be restricting the range in which the random walk shall be taken, with it being limited by a set of distances (following taxicab metric), which if it tries to cross, it is just reflected back inside instead.

Now, here's the algorithm.

1. We generate a set of points at a particular distance d from the origin, and then choose one starting point at random from the same.
2. We start a random walk from that point, while it follows a set of limits (i.e. $1.25 \times d$, $0.8 \times d$, $0.6 \times d$, $0.4 \times d$, $0.2 \times d$). While the distance of a walker from origin at an instance is d_p, let l_i be the last limit for which $d_p \leq l_i$. Then the walker is restricted inside a boundary of points whose distance from origin is l_i.
3. When the walker arrives in the neighbourhood of the cluster, it has a probability p of stopping at that point in the neighbourhood. If it stops there then we go back to step 1, otherwise we continue to the walk till we arrive once again in the neighbourhood of the cluster, and follow 3 again.

This goes on till we have enough number of points for us to view a proper plot. The sole reason we have considered these many limits is to get the walks to converge faster.

Section 3: Code for simulating the model

Please note that we'll be considering the **Taxicab** distance of the starting point from the seed to keep it simple. In my code, my stickiness factor shall also be $\frac{1}{2}$

```
library(tidyverse)
```

```
## Warning: package 'tidyverse' was built under R version 4.3.3
```

```
## Warning: package 'ggplot2' was built under R version 4.3.3
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.0     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.0
## v purrr     1.0.2
## -- Conflicts ---------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become error
```

```
library(viridis)
```

```
## Loading required package: viridisLite
```

Here are the constants.

```
set.seed(1e+6)
seed = 0 + 0i
start.dist = 250
limit.dists = c(
    1.25 * start.dist,
    0.8 * start.dist,
    0.6 * start.dist,
    0.4 * start.dist,
    0.2 * start.dist
)
num.particles = 5000
cluster = c(seed)
stickiness.factor = 0.5
```

```
taxicab.distance = function(pt) {
    abs(Re(pt)) + abs(Im(pt))
    # abs(pt)
}
```

```
random.walk = function(cluster, start.pt, stickiness.factor = 1) {
    stop.neighbourhood = unique(c(cluster + 1, cluster - 1, cluster - 1i, cluster + 1i))
    current.pt = start.pt
    limit.idx.current = 1
    limit.idx.next = 2
    # debug.step.count = 1

    while(TRUE) {
        if(current.pt %in% stop.neighbourhood) {
            u = runif(1)
            if(u <= stickiness.factor)
                break
        }
        direction = sample(c(1, -1, 1i, -1i), size = 1, replace = T)
        current.pt = current.pt + direction

        if(taxicab.distance(current.pt) < limit.dists[limit.idx.next]) {
            limit.idx.current = limit.idx.next
            limit.idx.next = min(limit.idx.next + 1, length(limit.dists))
        }

        if(taxicab.distance(current.pt) > limit.dists[limit.idx.current]) {
            current.pt = current.pt - (2 * direction)
        }
        # debug.step.count = debug.step.count + 1

        #if(debug.step.count %% 10000 == 0) {
        #   print(current.pt)
        #   print(limit.idx.current)
        #   print(limit.idx.next)
        #}
    }

    current.pt
```

```
}
```

```
# Main code

## generating the list of possible starting points
start.pt.list = c(
    (1 * start.dist) + 0i,
    (-1 * start.dist) + 0i,
    0 + (1i * start.dist),
    0 + (-1i * start.dist)
)

for(j in 1:(start.dist - 1)) {
    looping.complex = j + ((start.dist - j) * 1i)
    start.pt.list = c(
        start.pt.list,
        looping.complex,
        Conj(looping.complex),
        -1 * looping.complex,
        -1 * Conj(looping.complex)
    )
}

## generating the cluster
for(i in 1:num.particles) {
    start.pt = sample(start.pt.list, size = 1, replace = TRUE)

    new.cluster.pt = random.walk(
        cluster,
        start.pt,
        stickiness.factor
    )
    cluster = c(cluster, new.cluster.pt)
    #print(i)
}
```

```
## plotting the cluster
df = data.frame(
    x.vals = Re(cluster),
    y.vals = Im(cluster),
    steps = 1:length(cluster)
)

ggplot( data = df ) +
    geom_point(
        mapping = aes(x = x.vals, y = y.vals),
        size = 0.6,
        show.legend = F,
        color = turbo(length(cluster))
    ) +
    theme(
        panel.grid = element_blank(),
        panel.background = element_rect(fill = "black"),
```

```
        plot.background = element_rect(fill = "black"),
        axis.ticks = element_blank(),
        axis.text = element_blank()
    )
```

