The Convergence of IDLA to a Circle

Jean-Luc Thiffeault and Ruojun Wang

August 29, 2018

1 An IDLA Simulation and the Boundary

1.1 An IDLA Simulation with N Particles

Internal diffusion-limited aggregation (IDLA) is (?) a cluster growth process in which particles start at one or more sources within a cluster, diffuse outward, and are added to the cluster at the first site outside it they reach. (refer?)

1.1.1 Particles move in 8 directions

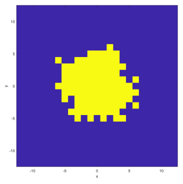
I started from MATLAB codes given by Professor Thiffeault. To beign, we prepare a grid quadrant where the particles can perform the IDLA process:

```
Ngrid = ceil(1.2*sqrt(Npart));
    grid0 = Ngrid+1;
    grid = zeros(2*Ngrid+1);
    x = -Ngrid:Ngrid;
```

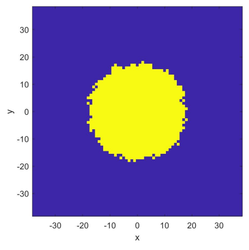
where `Npart` represents the total number of particles that take part into the simulation. `Ngrid` represents the length of one side of grid quadrant. We let `grid0` be the center of the grid quadrant and generate an array with size `2*Ngrid+1`; a `Ngrid*Ngrid` grid quadrant is then set up. Core codes to simulate IDLA process are as following:

```
drift = [0 0];
for i = 1:Npart
      X = [0 0];
  while 1
        X = X + (randi(3,1,2)-2) + drift;
        if ~grid(X(1)+grid0,X(2)+grid0)
          grid(X(1)+grid0,X(2)+grid0) = 1;
          break
    ...
```

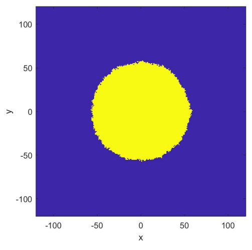
In this for loop, we start the first particle from the location `X = [0 0]`, which would locate at the center `grid0` of the quadrant we set previously. Then inside the while loop generates the coordinate that the next particle would locate. As the definition of IDLA process states, the next particle would move due to a random direction realized by `(randi(3,1,2)-2)`. A drift `drift` might be added to effect the final shape of the particles could form; the direction of drift can be given by `drift = [0 0]` before the for loop. The codes in if statement just says the particle would keep moving due to `(randi(3,1,2)-2)` until it finds an unoccupied grid to settle down. An occupied grid would be marked as "1" then. In this first simulation, we want the particles to move in 8 directions randomly (i.e. up, down, left, right, upper left, upper right, lower left, lower right). The results of the simulation are shown in Fig.1.1. We can observe that with the growth of the number of particles which participate in the IDLA process, the boundary of the shape tends to be circular, and hence such a convergence to a circle intrigues us to do further study.



(a) `Npart = 100`



(b) `Npart = 1000`



(c) `Npart = 10000`

Figure 1.1: IDLA simulation with 8 directions

1.2 Particles move in 4 directions

My first task is to restrict the motion of direction for each particles which involve in IDLA process. We would like to see if the boundary of the occupied region is still converging to a circle by restricting the particles to choose a random motion in 4 directions. That is, we eliminate the diagonal motion of particles (i.e. upper left, upper right, lower left, lower right). The MATLAB codes to realize this are:

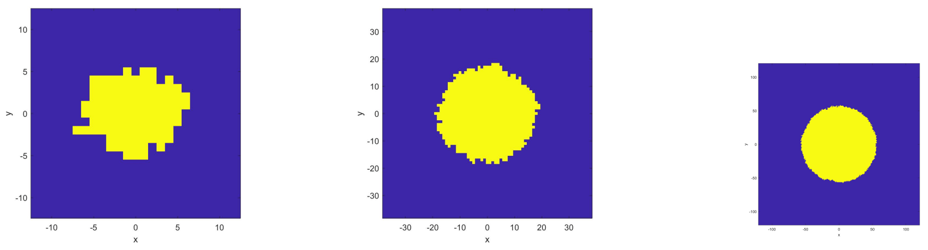
```
v_dir = [1 0; 0 1; -1 0; 0 -1];
n_dir = 4;

for i = 1:Npart
  X = [0 0];
  while 1
```

```
d = randi(n_dir);

X(1) = X(1) + v_dir(d, 1) + drift(1);
X(2) = X(2) + v_dir(d, 2) + drift(2);
...
```

Here, we prepare a 4*2 matrix containing 4 vectors which represent four directions that a particle can move along. Then similarly as the case shown in the motion with 8 directions, inside the for loop, we start from the center of the grid quadrant `[0 0]`. After determining a random number inside a list consisting 1 to 4, we are able to select a vector in the 4*2 matrix and then determine the direction of the particle moves along. The results of the simulation are shown in Fig.1.2.



(a) `Npart = 100` (b) `Npart = 1000` (c) `Npart = 10000`

Figure 1.2: IDLA simulation with 4 directions

We could then observe that a IDLA simulation also produce a final occupied region which converges to a circle, as the number of particles which participated grows (Fig.1.2).

1.3 Boundary of the Occupied Region

From the graph generated by MATLAB codes, we observe that the boundary of the shape constructed by the IDLA process tends to be smooth and to imitate a circle, as the value of `Npart` (the number of particles) become larger. We want to understand to what extent the shape constructed by the process converges to a circle. We start by selecting out the pixels which constitute the boundary of the occupied region and mainly focus on these pixels. Three different algorithms are tried to remain the boundary only as the IDLA process evolves.

1.3.1 Three Algorithms to Build the Boundary

Build entire boundary. The idea here is to start from an occupied grid on the boundary (for example, one on the boundary which has the same horizontal coordinate as the center grid), to examine its neighbor occupied grids, to choose the next one which is on the boundary, to examine the neighbors based on this next grid, and hence to find all the grids on the boundary of the occupied region.

(use image maybe?)

```
for k=1:size(bdP,1)
  if ~(bdP(k,1) == 0 && bdP(k,2) == 0)
  finish = 0;
  neighb = [bdP(k,1)+1 bdP(k,2); bdP(k,1)-1 bdP(k,2);
    bdP(k,1) bdP(k,2)+1; bdP(k,1) bdP(k,2)-1;
    bdP(k,1)+1 bdP(k,2)+1; bdP(k,1)+1 bdP(k,2)-1;
    bdP(k,1)-1 bdP(k,2)+1; bdP(k,1)-1 bdP(k,2)-1];

    for j=1:8
      if (grid(neighb(j,1), neighb(j,2))==1 &&
      gridB(neighb(j,1), neighb(j,2))==0)

        if ~((grid(neighb(j,1)+1, neighb(j,2))==1) &&
        (grid(neighb(j,1)-1, neighb(j,2))==1) &&
        (grid(neighb(j,1), neighb(j,2)+1)==1) &&
        (grid(neighb(j,1), neighb(j,2)-1)==1))

          bdP = [bdP; neighb(j,1) neighb(j,2)];
          gridB(neighb(j,1), neighb(j,2))=1;
          ...
```

In the first for loop, we prepare the coordinates of all eight neighbors for one particular occupied grid: Since the boundary of the occupied region can be constructed by grids which are only connected diagonally, so to examine the total eight closest grids for one is necessary. If an occupied grid is marked as "1" and an unoccupied one is marked as "0," we can determine the next occupied grid on the boundary by seeing if all its horizontally and vertically connected neighbors are marked as "1."

Build boundary incrementally. This algorithm is designed to examine all the grids on the grid quadrant and select out the grids which construct the boundary only as the occupied region grows. The idea is to come up with a method to tell the difference between the grids inside the region and outside it: the grids which are not on the boundary are surrounded by grids along its four sides, while the grids which are on the boundary have at least one side which does not touch another grid. Hence, as one more particle is added to the process, we could eliminate those which are not considered on the boundary. For example, in Fig.1.3, when the fifth particle is added to the process, we can determine the grid at the center is no longer a grid on the boundary and it should be removed. The MATLAB codes are shown as following:

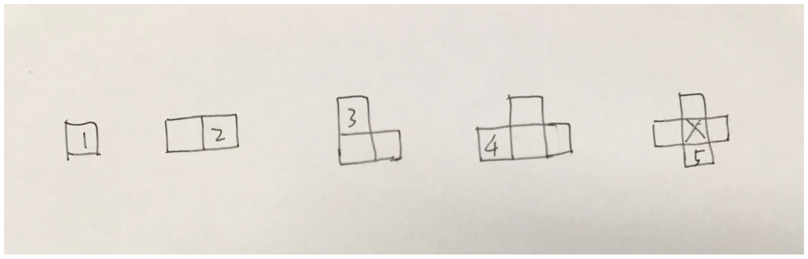


Figure 1.3: remove a grid which is not considered on the boundary any longer

```
for k = 2:(2*grid0-1)
  for l = 2:(2*grid0-1)
    if grid(k, l) == 1 && grid(k, l+1) == 1 &&
    grid (k, l-1) == 1 && grid (k+1, l) == 1 &&
    grid (k-1, l) == 1

      gridB(k, l) = 0;
      ...
```

where the grids which are occupied are marked as "1," and those which are not occupied are denoted as "0." By examining if all four neighbors of one grid are marked as "1," we are able to determine whether one grid is considered on the boundary.

Build boundary by Discrete Laplace. The third algorithm using Discrete Laplace has the similar idea to the second one. A Discrete Laplace can be written as:

$$(\Delta f)_{i,j} = 4f_{i,j} - f_{i+1,j} - f_{i-1,j} - f_{i,j+1} - f_{i,j-1} \tag{1.1}$$

Similarly to the second algorithm, the grids which are occupied are marked as "1," and those which are not occupied are denoted as "0." Hence, if $(\Delta f)_{i,j} > 0$, the grid with coordinate (i, j) is considered on the boundary. The MATLAB codes can be written as:

```
for k = 2:(2*Ngrid)
  for l = 2:(2*Ngrid)
    delta_f = 4*grid(k, l) - grid(k, l+1) - grid(k, l-1)
    - grid(k+1, l) - grid(k-1, l);

    if delta_f == 0
```

```
gridB(k, l) = 0;
...
```

In order to see if those three algorithms return the same results regarding the grids on the boundary which are selected out, I also further compare the discretized boundary generated by three algorithms and it turns out they agree with each other.

1.3.2 The Numerical Standard Deviation

The standard deviation from the simulation, which is marked by σ_{sim}, consists of two errors, the geometric error σ_{geom} and the statistical error σ_{stat}.

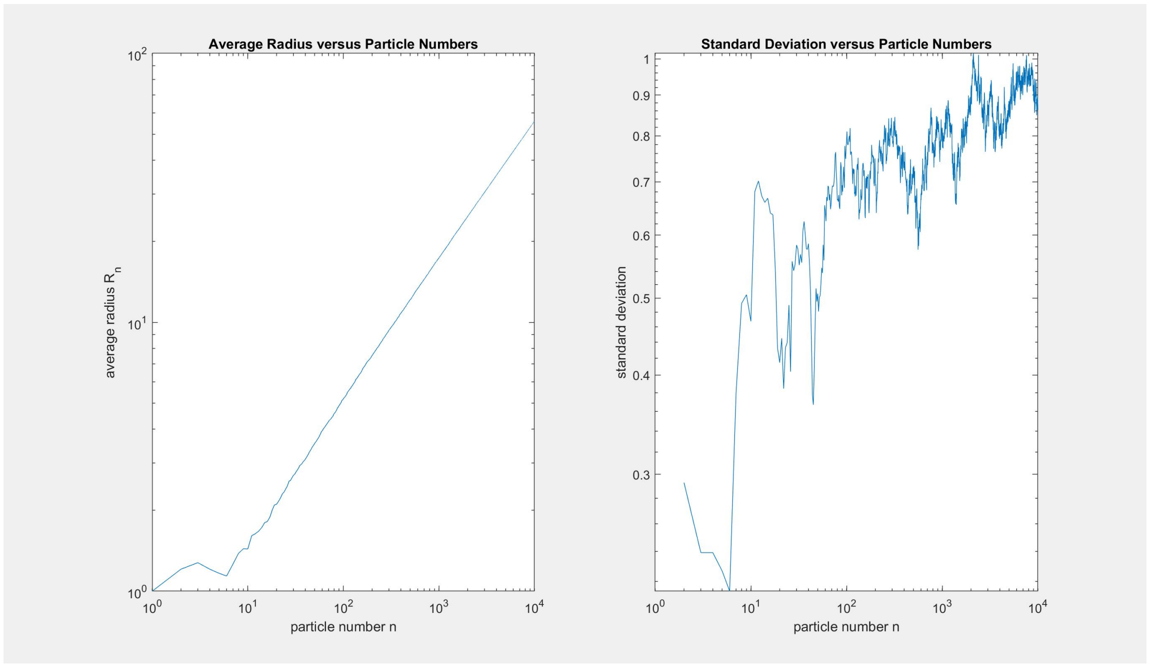


Figure 1.4: Average Radius versus Particle Numbers and Standard Deviation versus Particle Numbers when `Npart=10000`

2 The Geometric Error

A discretized circle can be constructed by two different kinds of algorithms, in order to obtain the geometric error σ_{geom}.

2.1 The Numerical Standard Deviation

2.1.1 The "Nonremove" Case

We consider a grid (a lattice of circle) in the plane with centers at coordinates $p = (m, n) \in \mathbb{Z}^2$. The center of circle drawn locates at the origin $(0, 0)$. Take a circle of radius R, centered on the origin. A continuous discretization of the circle is an ordered set of distinct pixels which the boundary of the circle passes through

$$\mathcal{D}_R = (p_i)_{0 \leq i \leq N-1} = (m_i, n_i)_{0 \leq i \leq N-1}, \tag{2.1}$$

where m_i denotes the horizontal coordinate of the center of a pixel and n_i denotes the vertical one of that.

The algorithm can be realized by MATLAB codes as following: ... Given $R = 10000$, $\sigma_{geom} = 0.3729$.

2.1.2 The "Remove" Case

Another ways to construct a discretized consisting of less pixels occupied than those in the first algorithm. The algorithm can be realized by MATLAB codes: ... When $R = 10000$, $\sigma_{geom} = 0.2624$, which is smaller than the value obtained from the first algorithm.

2.2 The Upper Bound of L_2 Error of the Discretization

(A derivation of L_2 Error of the discretization.)

$$\mathrm{Err}_2\ \mathcal{D}_R = (\frac{1}{N} \sum_{i=0}^{N-1} (m_i^2 + n_i^2) - R^2)^{\frac{1}{2}}. \tag{2.2}$$

In both algorithm to construct a discretized circle, σ_{geom} can be approximated by $\mathrm{Err}_2\ \mathcal{D}_R$ when $R \to \infty$, since ... $\mathrm{Err}_2\ \mathcal{D}_R$ can be considered as the distance between the center of each pixel p_i and the boundary of the circle which passes through this pixel. We denote $\mathrm{Err}_2\ \mathcal{D}_R$ as d in this case. By computation(...), if the small piece is randomly distributed, the expectation value $\mathbb{E}\ d^2 \approx 0.1667$.

For both algorithms, a analytical upper bound can be found for d^2, which approximately equal to σ_{geom}^2 when $R \to \infty$. $\sigma_{geom}^2 \leq \frac{1}{2} \approx 0.5$ and then $\sigma_{geom} \leq 0.7071$.

(We sort pixels into 4 different types as following to lower the upper bound of d ...)

2.2.1 Sort Pixels into 4 Different Types

According to the ways that the boundary of the circle passes through each pixel, we can sort N pixels into 4 different types for the upper right quarter of the circle (other three quarters would just be the mirror images of the upper right one with respect to different axises of symmetry). As the algorithm provided above, the center of each pixel is represented by (m_i, n_i). (graph?) The boundary of the circle passing by would have 2 different intersections with a pixel p_i. Hence, the four different type of grids can be given by the inequalities restricting the coordinates (x, y) of the intersections. The first kind of pixel is provided by

$$\text{one intersection: } x = m_i - \frac{1}{2};\ n_i - \frac{1}{2} \leq y \leq n_i - \frac{1}{2},\ \text{where } y = \sqrt{R^2 - (m_i - \frac{1}{2})^2}; \tag{2.3}$$

... (other inequalities)

(A deviation to find upper bound by evaluating these four types of pixels.)

$$\sigma_{geom}^2 \leq \frac{1}{2} \approx 0.5 \tag{2.4}$$

Also, $\sigma_{geom} \leq 0.7071$. (The upper bound is still the same. We need a further improvement.)

2.2.2 An Observation of the Relation between the Fraction of Each Type of Pixels and Multiples of R

(By derivation, $(1 - \frac{1}{\sqrt{2}})R \approx 0.29289R$ and $\frac{1}{\sqrt{2}}R \approx 0.70711R$)

The "Nonremove" Case. Given by MATLAB codes, the numerical fraction of the four different type of pixels are

$$\text{fraction of type } 1 = 0.29291; \tag{2.5}$$
$$\text{fraction of type } 2 = 0.29286; \tag{2.6}$$
$$\text{fraction of type } 3 = 0.20711; \tag{2.7}$$
$$\text{fraction of type } 4 = 0.20711; \tag{2.8}$$

(By observation, the value in (2.5) and $(1 - \frac{1}{\sqrt{2}})$ are close. A derivation to find the relation between these two. The upper bound of d^2 can be determined then: $d^2 \leq p_1\ (\frac{1}{\sqrt{2}})^2 + p_2\ (\frac{1}{2})^2$.)

The "Remove" Case. Given by MATLAB codes, the numerical fraction of the four different type of pixels (for $R = 10000$) are

$$\text{fraction of type } 1 = ...; \tag{2.9}$$
$$\text{fraction of type } 2 = ...; \tag{2.10}$$
$$\text{fraction of type } 3 = 0.29291; \tag{2.11}$$
$$\text{fraction of type } 4 = 0.29291; \tag{2.12}$$

By observation, the values in (2.11), (2.12), and $(1 - \frac{1}{\sqrt{2}})$ are close. (However, to sort pixels into 4 types cannot help construct a relation between the fraction of each type of pixels and multiples of R as we did for the "nonremove" case. We need to sort pixels into 6 different types to obtain an analytical solution.)

2.2.3 Sort Pixels into 6 Different Types

(A derivation to obtain $d^2 \leq p_1\ (\frac{1}{\sqrt{2}})^2 + p_2\ (\frac{1}{2})^2$.)
(conclusion?)
(reference?)