FINAL REPORT
DIFFUSION LIMITED AGGREGATION

0. TEAM MEMBERS:

- Nguyen Van Thanh Tung – 20190090
- Pham Quang Hieu – 20194432
- Nguyen Vu Thien Trang – 20194459

1. PROBLEM FORMULATION

1.1. INTRODUCTION

In many cases one is only interested in the final steady state concentration field and not so much in the transient behavior, i.e., the route towards the steady state is not relevant. This may be so because the diffusion is a very fast process in comparison with other processes in the system, and then one may neglect the transient behavior.

Setting all time derivatives to zero in the diffusion equation we find the time independent diffusion equation:

$$\nabla^2 c = 0$$

This is the famous Laplace equation that is encountered in many places in science and engineering, and therefore very relevant to study in some more detail.

Consider the two-dimensional domain and the discretization of the previous section. The concentration c no longer depends on the time variable, so we denote the concentration on the grid point here as $c_{l,m}$. Substituting the finite difference formulation for the spatial derivatives results in:

$$c_{l,m} = \frac{1}{4}\left[c_{l+1,m} + c_{l-1,m} + c_{l,m+1} + c_{l,m-1}\right] \ , \ \forall (l,m)$$

1.2. DIFFUSION LIMITED AGGREGATION

Diffusion Limited Aggregation (DLA) is a model for non-equilibrium growth, where growth is determined by diffusing particles. It can model e.g. a Bacillus subtilis bacteria colony in a petri dish. The idea is that the colony feeds on nutrients in the immediate environment, that the probability of growth is determined by the concentration of nutrients and finally that the concentration of nutrients in its turn is determined by diffusion. The basic algorithm is:

```
1. Solve Laplace equation to get distribution of nutrients,
assume that the object is a sink (i.e. c = 0 on the object)
2. Let the object grow
3. Go back to (1)
```

The first step in Algorithm 5 is done by a parallel **SOR** iteration. Step 2, growing of the object, requires three steps:

1. determine growth candidates;
2. determine growth probabilities;
3. grow.

A growth candidate is basically a lattice site that is not part of the object, but whose north -, or east -, or south -, or west neighbor is part of the object. In Figure 18 a possible configuration of the object is shown; the black circles form the current object, while the white circles are the growth candidates.

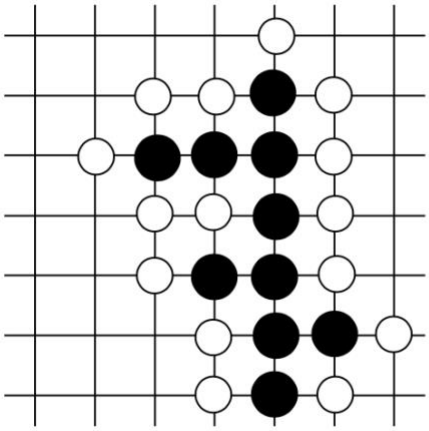


Figure 18: The object and possible growth sites.

The probability for growth at each of the growth candidates is calculated by

$$p_g\left((i,j)\in\circ\rightarrow(i,j)\in\bullet\right)=\frac{\left(c_{i,j}\right)^{\eta}}{\sum_{(i,j)\in\circ}\left(c_{i,j}\right)^{\eta}}.$$

The parameter η determines the shape of the object. For $\eta = 1$ we get the normal DLA cluster, i.e. a fractal object. For $\eta < 1$ the object becomes more compact (with $\eta = 0$ resulting in the Eden cluster), and for $\eta > 1$ the cluster becomes more open (and finally resembles say a lightning flash).

Modeling the growth is now a simple procedure. For each growth candidate a random number between zero and one is drawn and if the random number is smaller than the growth probability, this specific site is successful and is added to the object. In this way, on average just one single site is added to the object.

The results of a simulation are shown in Figure 19. The simulations were performed on a 256^2 lattice. SOR was used to solve the Laplace equation. As a starting point for the SOR iteration we used the previously calculated concentration field. This reduced the number of iterations by a large factor. For standard DLA growth ($\eta = 1.0$) we obtain the typical fractal pattern. For Eden growth a very compact growth form is obtained, and for $\eta = 2$, a sharp lighting type of pattern is obtained.

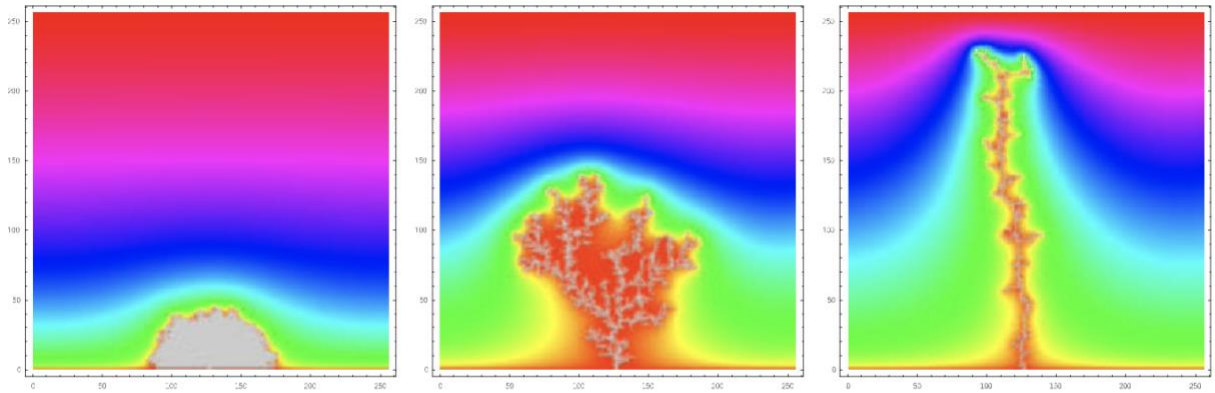


Figure 19: Results of DLA growth on a 256^2 lattice. The left figure is for $\eta = 0$, the middle for $\eta = 1.0$ and the right for $\eta = 2.0$.

1.3. 2D CONDITIONS

As in our problem, we are facing with 2D grid. Then we have a strictly boundary simulation:

- Periodic in columns direction, with a period equals to N (width of the grid)
- Constant in row direction

The initial object we can put anywhere on the grid, and this can be setting by change the source code.

2. METHOD

We choose to apply **Successive Order Relaxation (SOR)** method. SOR can be thought of as a smoothed version of Gauss Seidel Iterative method by using momentum.

2.1. THE JACOBI ITERATIVE METHODS

$$c_{l,m}^{(n+1)} = \frac{1}{4}\left[c_{l+1,m}^{(n)} + c_{l-1,m}^{(n)} + c_{l,m+1}^{(n)} + c_{l,m-1}^{(n)}\right] . \qquad [17]$$

As before, n is the iteration index. This iterative scheme is known as the Jacobi iteration. Note the relation of Equation [17] with the finite difference scheme for the time dependent diffusion equation, Eq. [11]. If we take the by the CFL condition allowed maximum time step, i.e. $D\delta x^2/\delta t = ¼$ and use that in Eq. [11] we reproduce the Jacobi iteration of Eq. [17]. In other words, we expect that the Jacobi iteration will also suffer from a relative large amount of iterations needed before convergence.

Algorithm 2 provides the code for the inner loops of a Jacobi iteration. We again assume the square domain with periodic boundaries in the x-direction and fixed boundaries in the y-direction. Furthermore, a specific stopping criterion is implemented. We demand that

$$\max_{ij}\left|c_{ij}^{(n+1)} - c_{ij}^{(n)}\right| < \varepsilon .$$

The difference in concentration between two iterations on all grid points should be smaller that some small number ε. This is a rather severe stopping condition. Others can also be used, e.g. calculating the mean difference and demanding that this should be smaller that some small number. Here we will not pay any further attention to the stopping criterion, but you should realize this is an important issue to be considered in any new application of an iterative method.

```
/* Jacobi update, square domain, periodic in x, fixed */
/* upper and lower boundaries                          */
do {
   δ = 0
   for i=0 to max {
      for j=0 to max {
         if(c_ij is a source) c_ij^(n+1) = 1.0
         else if(c_ij is a sink) c_ij^(n+1) = 0.0
         else {
            /* periodic boundaries */
            west = (i==0) ? c_max-1,j^(n) : c_i-1,j^(n)
            east = (i==max) ? c_1,j^(n) : c_i+1,j^(n)
            /* fixed boundaries */
            south = (j==0) ? c0 : c_i,j-1^(n)
            north = (j==max) ? cL : c_i,j+1^(n)
             c_ij^(n+1) = 0.25 * (west + east + south + north)
         }
         /* stopping criterion */
         if(|c_ij^(n+1) - c_ij^(n)| > tolerance) δ = |c_ij^(n+1) - c_ij^(n)|
      }
   }
while (δ > tolerance)
```

Algorithm 2: The sequential Jacobi iteration

2.2 GAUSS-SEIDEL ITERATIVE METHOD

The Jacobi iteration is not very efficient, and here we will introduce a first step to improve the method. The *Gauss-Seidel* iteration is obtained by applying a simple idea. In the Jacobi iteration we always use results from the previous iteration to update a point, *even* when we already have new results available. The idea of Gauss-Seidel iteration is to apply new results as soon as they become available. In order to write down a formula for the Gauss-Seidel iteration we must specify the order in which we update the grid points. Assuming a row-wise update procedure (i.e. we increment l while keeping m fixed) we find for the Gauss-Seidel iteration

$$c_{l,m}^{(n+1)} = \frac{1}{4}\left[c_{l+1,m}^{(n)} + c_{l-1,m}^{(n+1)} + c_{l,m+1}^{(n)} + c_{l,m-1}^{(n+1)}\right] . \qquad [18]$$

One immediate advantage of the Gauss-Seidel iteration lies in the memory usage. In the Jacobi iteration you would need two arrays, one to store the old results, and another to store the new results. In Gauss-Seidel you immediately use the new results as soon as they are available. So, we only need one array to store the results. Especially for large grids this can amount to enormous savings in memory! We say that the Gauss-Seidel iteration can be computed *in place*.

Is Gauss-Seidel iteration also faster than Jacobi iteration. According to theory it turns out that a Gauss-Seidel iteration requires a factor of two iterations less then Jacobi (see [3], section 17.5). This is also suggested by a numerical experiment. We have taken the same case as in the previous section, and for $N = 40$ we have measured the number of iterations needed for Gauss-Seidel and compared to Jacobi. The results are shown in Figure 13. The reduction of the number of iterations with a constant number is indeed observed, and this constant number is very close to the factor of two as predicted by the theory. This means that Gauss-Seidel iteration is still not a very efficient iterative procedure (as compared to direct methods). However, Gauss-Seidel is also only a stepping stone towards the Succesive Over Relaxation method (see next section) which *is* a very efficient iterative method.

The Gauss-Seidel iteration posses a next challenge to parallel computation. At first sight we must conclude that the parallelism available in Jacobi iteration is now completely destroyed by the Gauss-Seidel iteration. Gauss-Seidel iteration seems inherently sequential. Well, it is in the way we introduced it, with the row-wise ordering of the computations. However, this row-wise ordering was just a convenient choice. It turns out that if we take another ordering of the computations we can restore parallelism in the

Gauss-Seidel iteration. This is an interesting case where reordering of computations provides parallelism. Keep this in mind, as it may help you in the future in finding parallelism in algorithms!

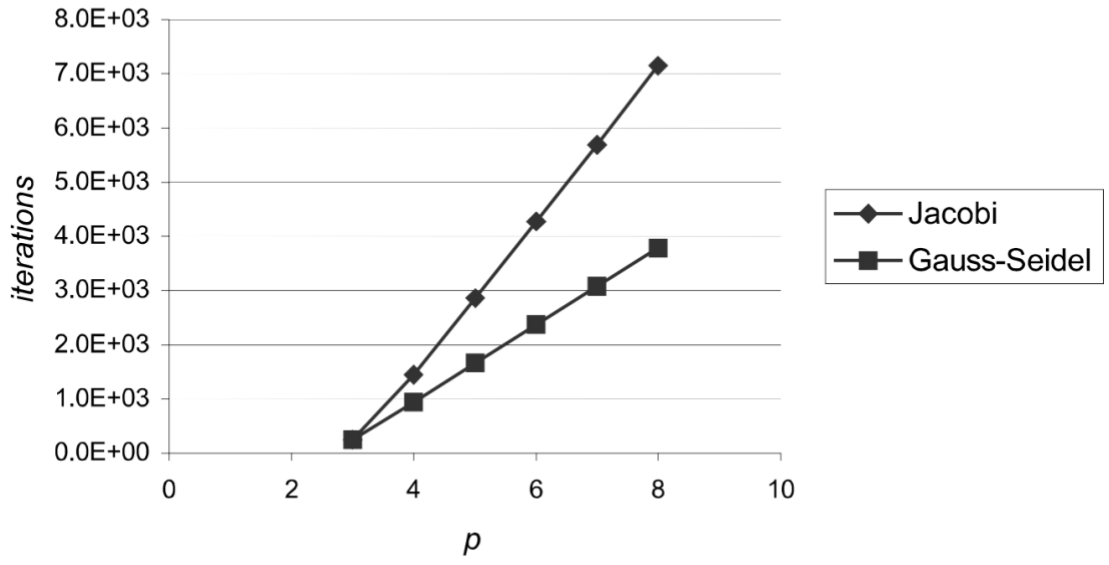


Figure 13: The number of iterations for the Jacobi and Gauss-Seidel iteration as a function of the stopping condition for N = *40.*

2.3. SUCCESSIVE OVER RELAXATION

The ultimate step in the series from Jacobi and Gauss-Seidel is to apply a final and as will become clear very efficient idea. In the Gauss-Seidel iteration the new iteration results is completely determined by its four neighbors. In the final method we apply a correction to that, by mixing the Gauss-Seidel result with the current value, i.e.

$$c_{l,m}^{(n+1)} = \frac{\omega}{4}\left[c_{l+1,m}^{(n)} + c_{l-1,m}^{(n+1)} + c_{l,m+1}^{(n)} + c_{l,m-1}^{(n+1)}\right] + (1-\omega)\, c_{l,m}^{(n)}. \qquad [19]$$

The parameter ω determines the strength of the mixing. One can prove (see e.g. [2]) that for $0 < \omega < 2$ the method is convergent. For $\omega = 1$ we recover the Gauss-Seidel iteration, for $0 < \omega < 1$ the method is called Successive Under Relaxation. For $1 < \omega < 2$ we speak of *Successive Over Relaxation*, or SOR.

3 IMPLEMENTATION DETAIL

3.1. LANGUAGE

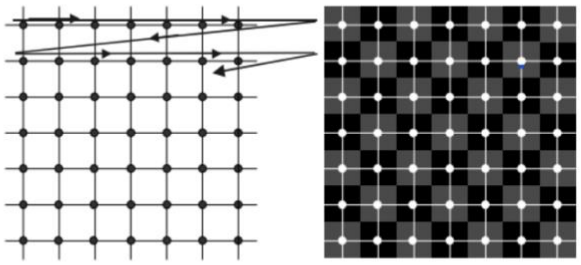
We implemented the **SOR** iterative method with Message Passing Interface (MPI). The programming language is C.

3.2. DETAIL

The input grid is NxN, we divide it into n_p process, each process handle n_r = N/n_p row. The task of each processor is passing message to get information from the boundary, estimating the diffusion process in each cell, export data.

❖ Each processor handles some row of the grid, with red-black ordering.

```
/* only the inner loop of the parallel Gauss-Seidel method with */
/* Red Black ordering */
   do {
      exchange boundary strips with neighboring processors;
      for all red grid points in this processor {
         update according to Gauss-Seidel iteration;
      }
      exchange boundary strips with neighboring processors;
      for all black grid points in this processor {
         update according to Gauss-Seidel iteration;
      }
      obtain the global maximum δ of all local δ_i values
   }
   while (δ > tolerance)
```



Algorithm 4: The pseudo code for parallel Gauss-Seidel iteration with red-black ordering.

4. EXPERIMENT RESULT

We grew the object with different initialization ways:

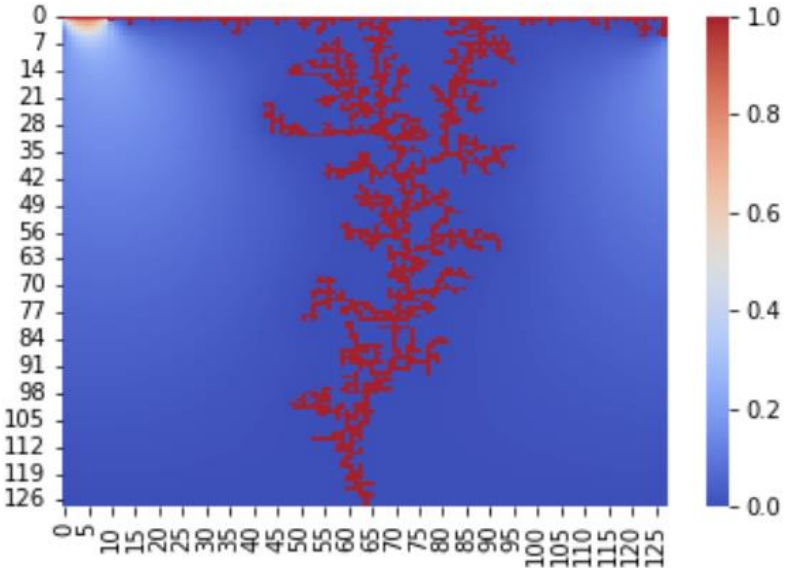


Fig1. Middle Bottom

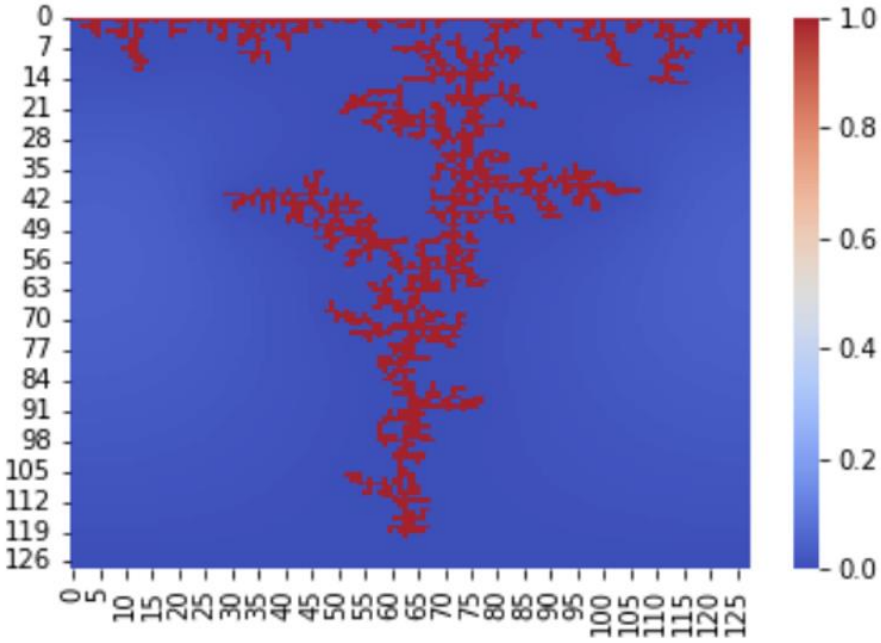


Fig2. Middle

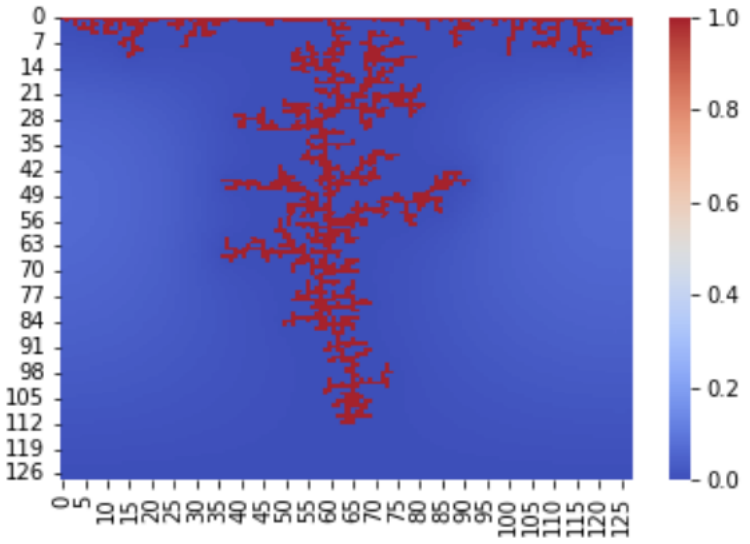


Fig3. Middle of First Row

Reduce number of iterations:

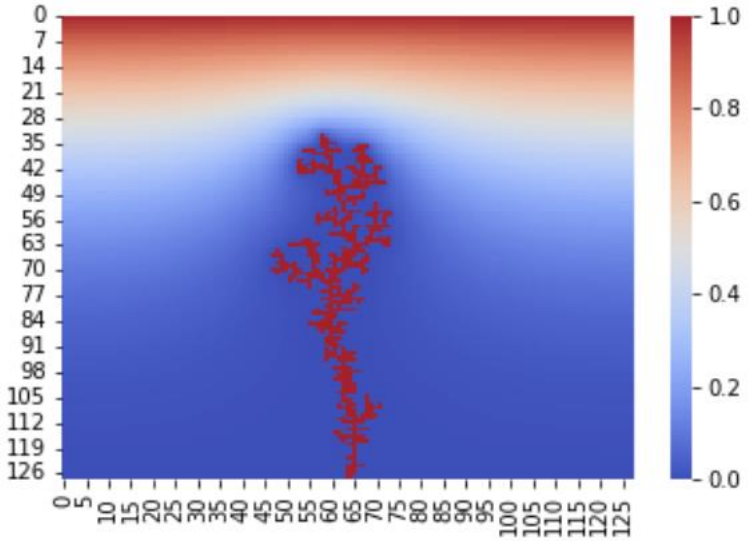


Fig 4. N_iteration=500

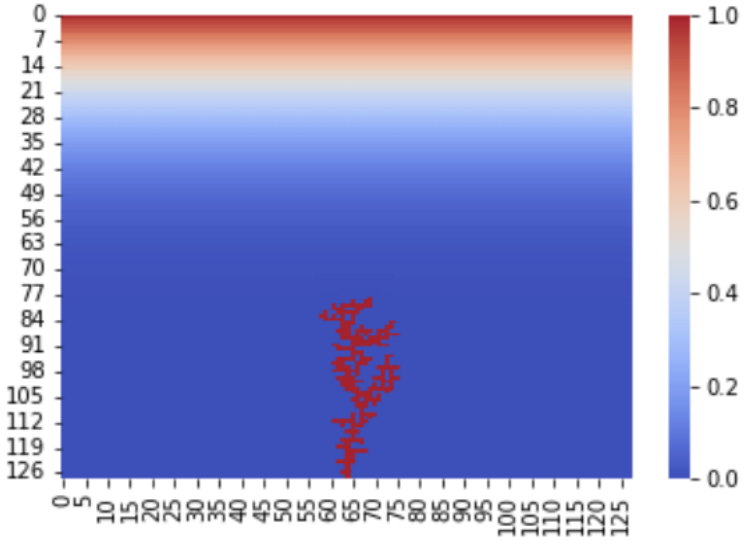


Fig5. N_iteration=250

Different omega:

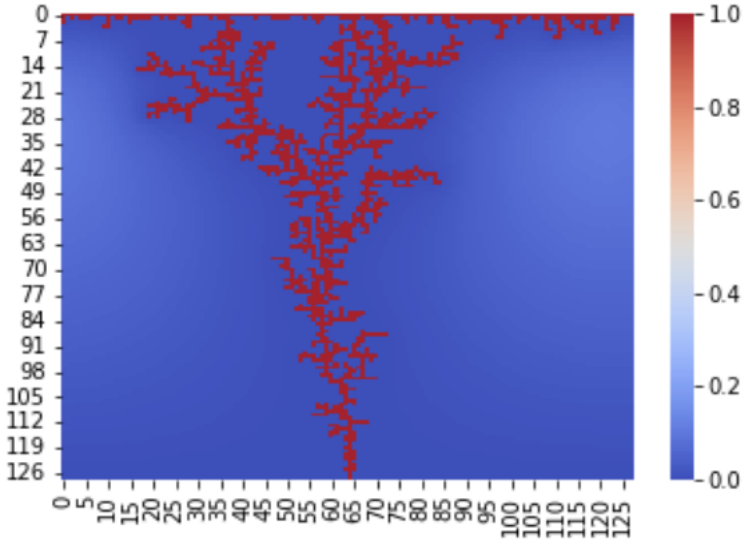


Fig6. Omega=1.0

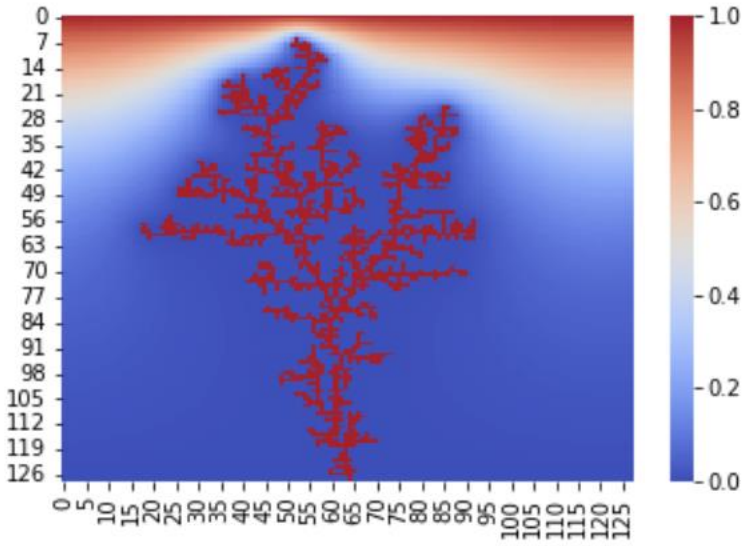


Fig7. Omega=1.3

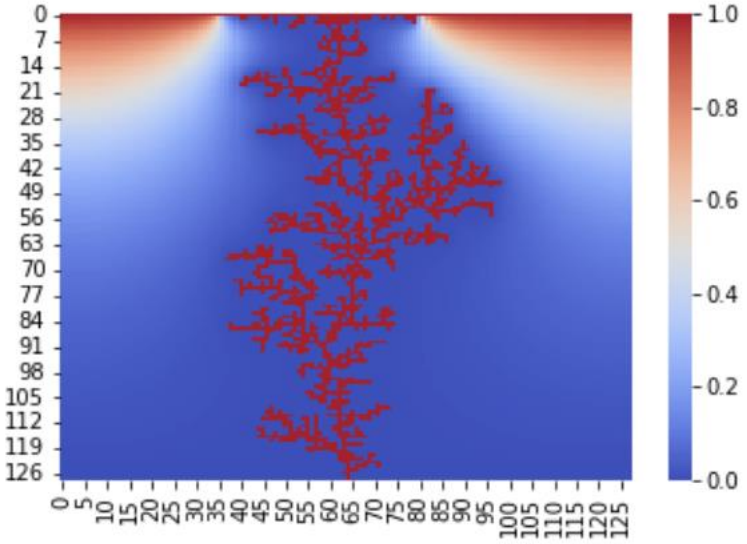


Fig8. Omega=1.6