Percolation

Mahyar Albalan

403201223

1. Percolation

First, I create a function, *Percolation*, for generating the initial matrix. This function places 1 with a probability p and 0 otherwise. Then, I represent 1s with blue color and 0s with red blocks. To determine whether there is a possible path from the right side to the left side, there are several fundamental graph traversal methods, such as **Breadth-First Search (BFS)**, **Depth-First Search (DFS)**, and so on. Here, I used the **DFS algorithm**, which is easier to implement. It basically starts from one of the non-zero blocks on the right side and checks every possible direction. If it reaches the left side, it returns 1; otherwise, it returns 0. DFS is not the most optimized algorithm for finding the shortest path, but for this problem, it is good enough. Now, let's take a look at the output for $L = 2, 3, 10, 100$.

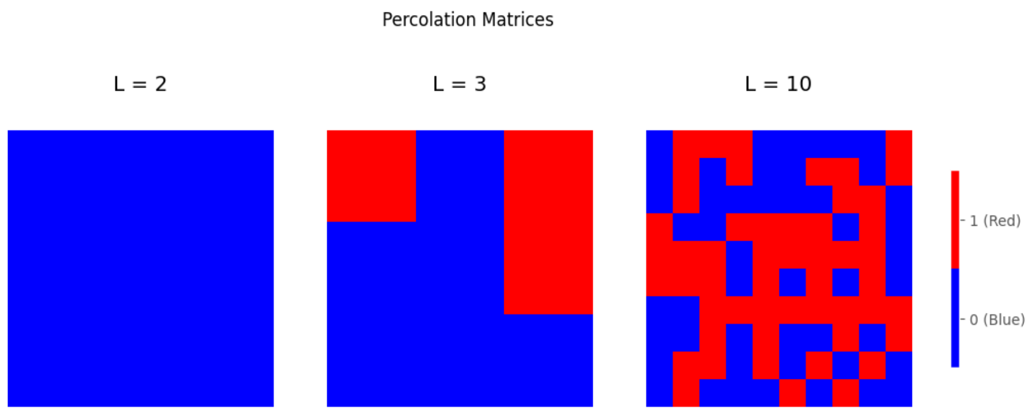

Figure 1: Output for $L = 2, 3, 10$.

Here, we see that only for $L = 10$, percolation occurs. Here is the result for $L = 100$ (after running the `can_traverse_matrix` function with this matrix):

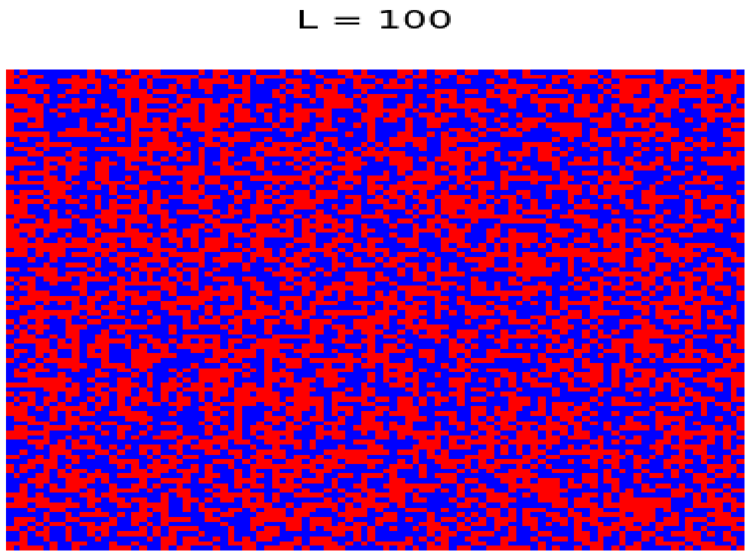

Figure 2: Output for $L = 100$.

2. Coloring Algorithm

This problem is pretty straightforward. I will follow the algorithm as described in the text. Here is the result for $L = 100$. First, I plot the output matrix, which contains different labels. Then, for better visualization, I assign `0` to all labels other than `1`.

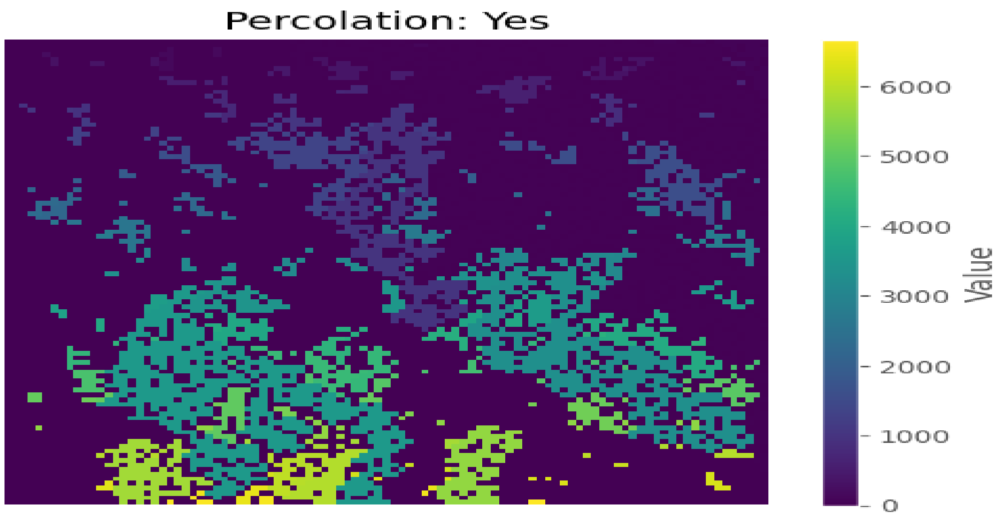

Figure 3: Colored matrix with different labels for $L = 100$ *and* $P = 0.675$.

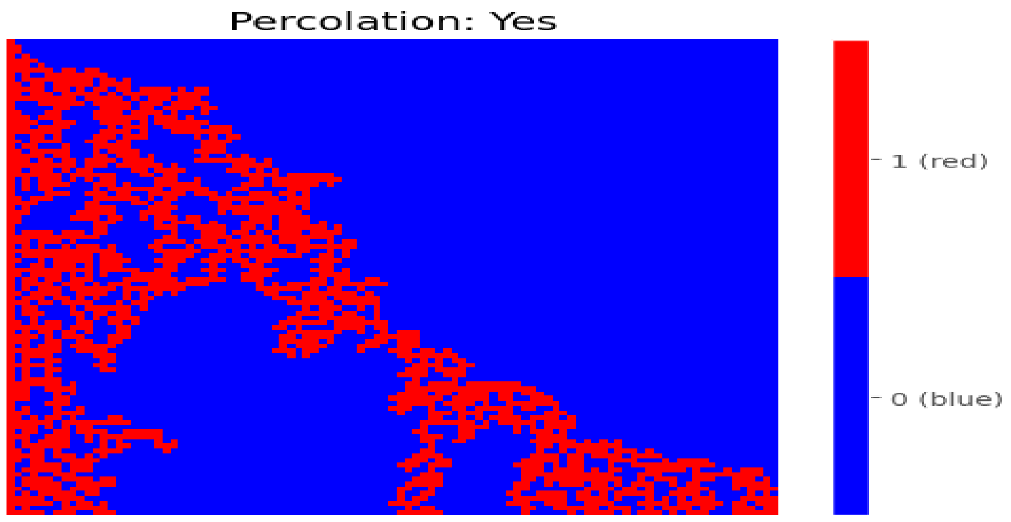

Figure 4: Colored matrix with labels other than `1` set to `0`.

This algorithm works almost instantly for $L = 100$, which is much better than the algorithm in Question 1. However, during the process of coloring the matrix, the resulting shape depends on the order in which we traverse the cells. This algorithm is faster because it has a time complexity of $O(N^2)$ and is not NP-complete.

3.Hoshen-Kopelman

This question is **quite** similar to the **previous** question. Here, I implement the algorithm described in the text. The time complexity of this algorithm is $O(N^2)$, where N is the length of the matrix. The result for $L = 100$ with a probability $P = 0.6$ of being active is as follows:

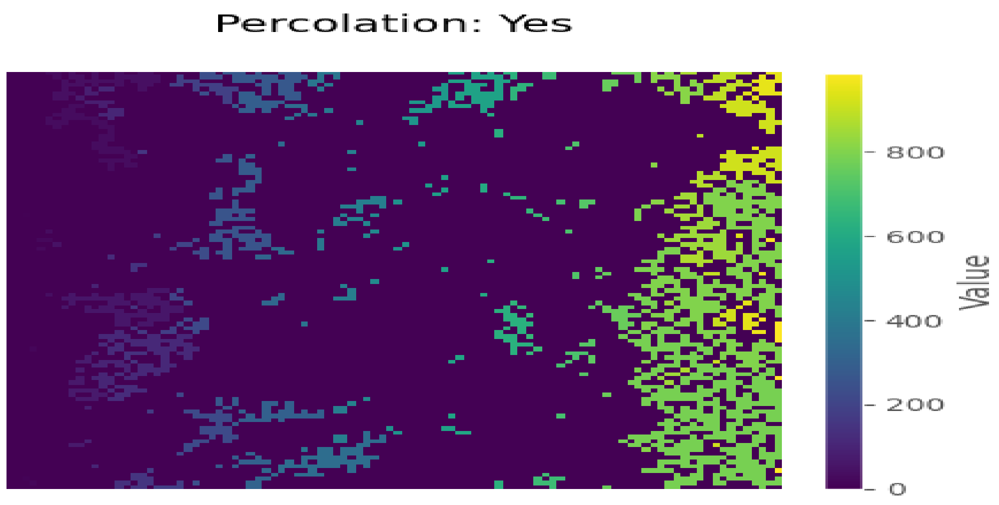

Figure 5: Hoshen-Kopelman output for $L = 100$ *and* $P = 0.6$.

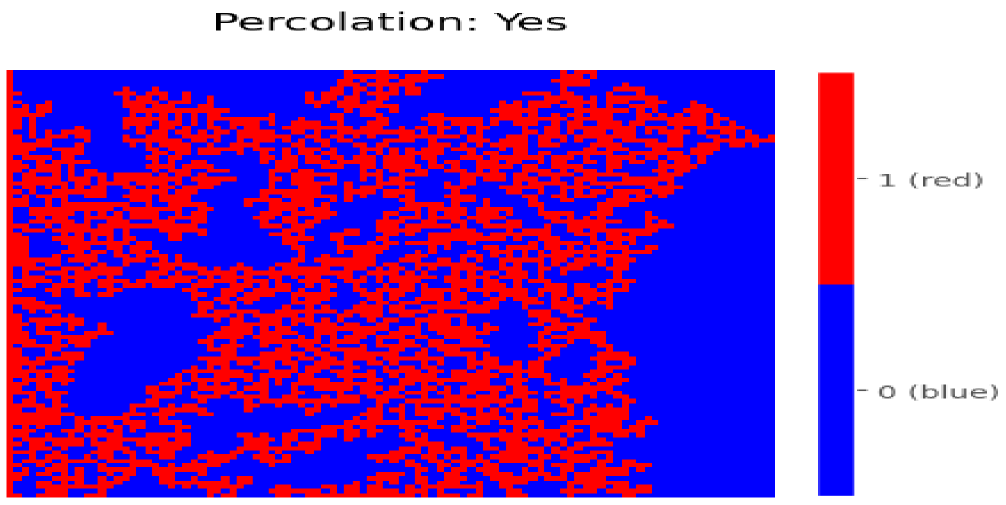

Figure 6: output labels other than `1` set to `0` for better visualization.

Both algorithms in this question and the previous question are fast and have approximately the same time complexity. For $L = 100$, they run almost instantly. For $L = 1000$ and $P = 0.6$, the runtime of the Hoshen-Kopelman algorithm is **1 minute 3 seconds**, and the runtime of the Coloring algorithm is **8.18 seconds**! The difference is because the algorithm in the text for calculating the size of each label is not optimized. We could calculate the whole matrix as before, and after constructing the matrix, we could use **NumPy**'s vectorized calculations and its functions to count the number of each label much faster.

4.Infinite Cluster Probability

For the first part of this question, I used the **coloring** function from the second question. The result is as follows:

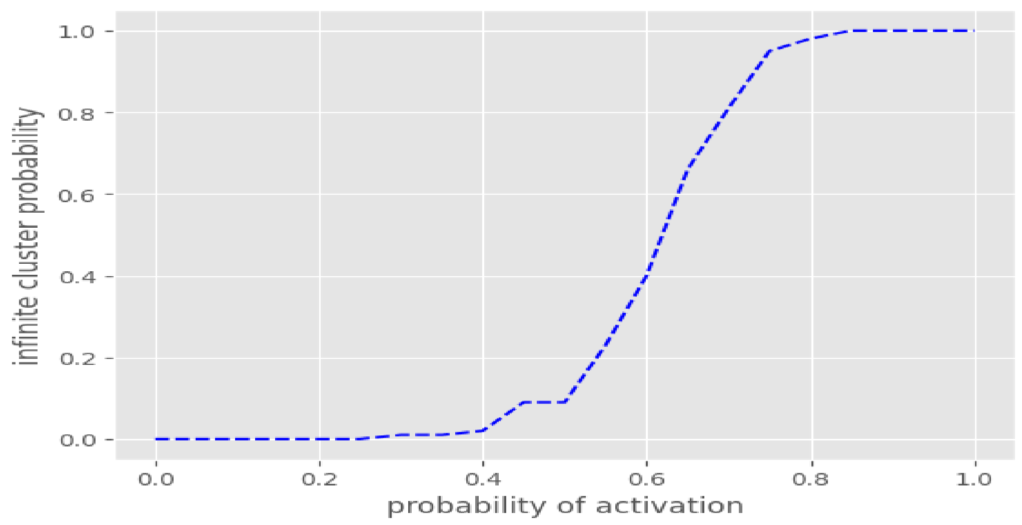

Figure 7: infinite cluster probability for L=10 at different activation probability.

As you can see in the top plot, for P values between approximately 0.4 and 0.8, the infinite cluster probability increases from 0 to 1, behaving as expected (similar to a sigmoid function).
Here are the results for $L = \{10, 20, 40, 80, 160\}$:

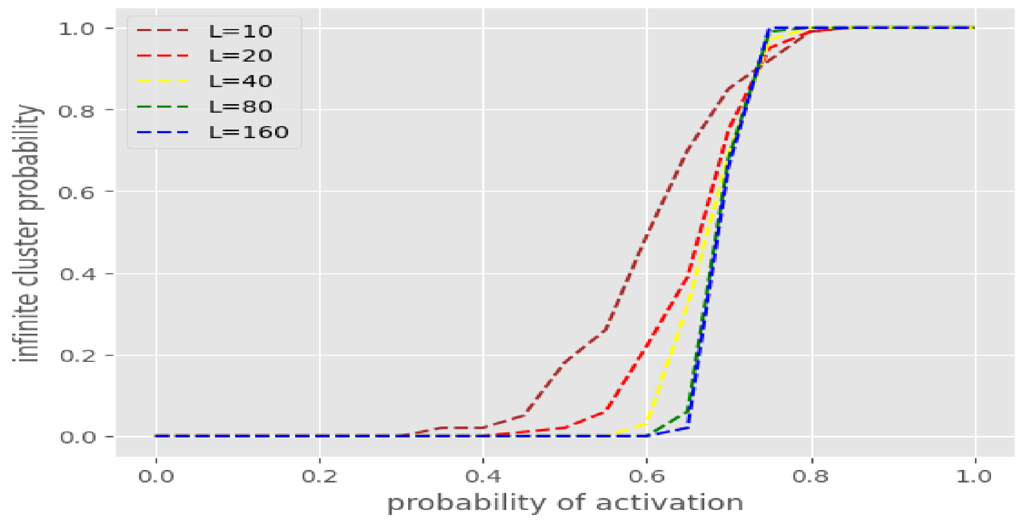

Figure 8: infinite cluster probability for L = [10,20,40,80,160] at different activation probability.

Here again, we can see that as the size of the matrix increases, the plot grows sharper, which is the exact behavior we expected. Moreover, for an infinitely sized sample, we can expect this growth to be instantaneous.

5.Probability of being part of an infinite cluster

This question is exactly like the last question, with the only difference being that we have to calculate the probability of a point being part of an infinite cluster. We can achieve this by calculating the ratio of the number of 1s (where 1s represent points on the infinite cluster) to the total number of points in the matrix when percolation occurs. Here, instead of using the **Coloring** algorithm, I used the **Hoshen_Kopelman 2** function, which also calculates the size of each cluster. This function implements the real Hoshen-Kopelman algorithm. In the previous Hoshen-Kopelman algorithm question, the code and solution were biased because the text assigned 1 to the right column, which was an unnecessary condition. In this new function, I removed this condition and, with a slight adjustment, created this improved version. Here are the results, as before:

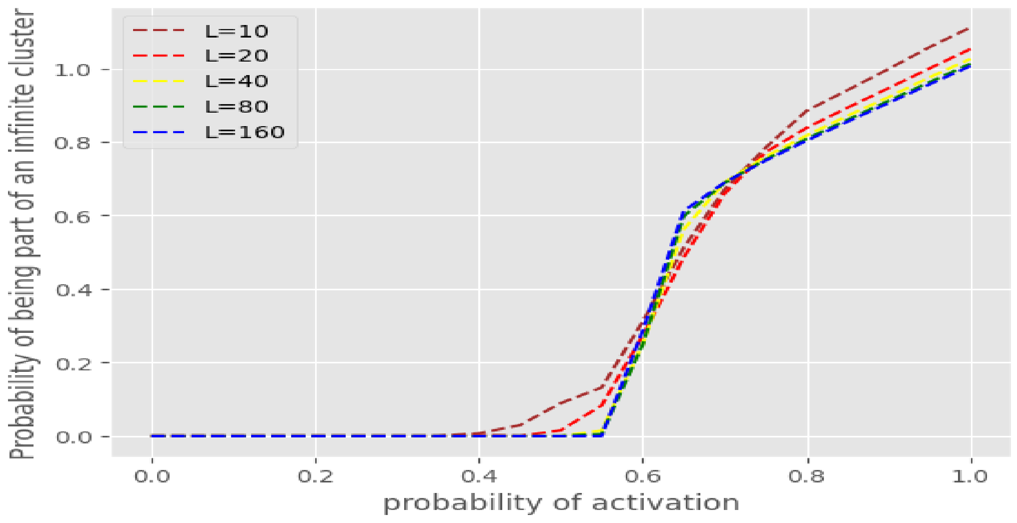

Figure 9: Probability of being part of an infinite cluster for L = [10,20,40,80,160] at different activation probability.

6.Correlation length

For this question, I first create a `correlation_length` function, which calculates the correlation length of a given sample by computing the radius of gyration for each finite cluster. Here are the results for different sample sizes:

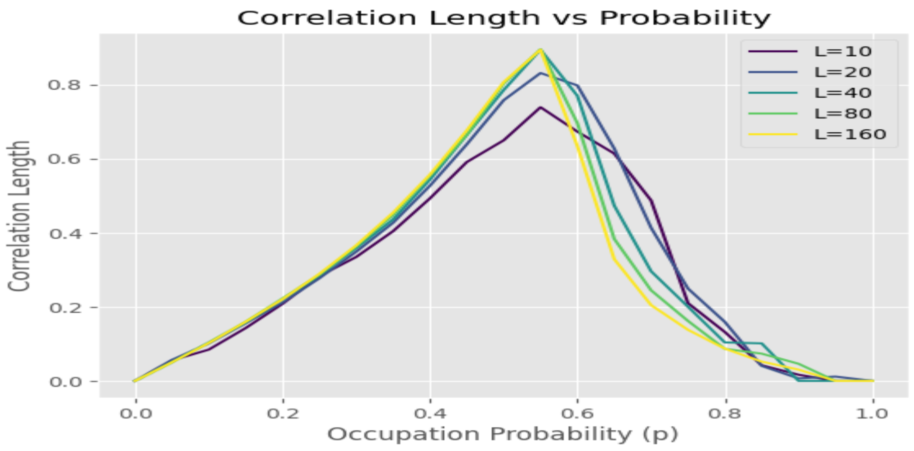

Figure 10: Correlation length for L = [10,20,40,80,160] at different activation probability.

For better visualization, near the critical point, I used smaller steps. The results are as follows:

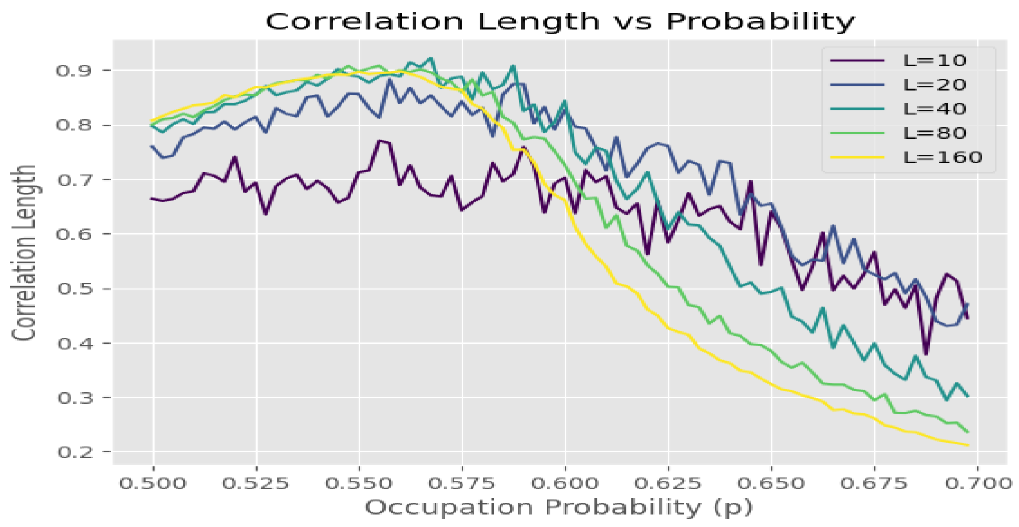

Figure 11: Correlation length for L = [10,20,40,80,160] at different activation probability near critical point.

Here, we can see that the critical value for $L = 160$ is 0.56. For site percolation, the value for an infinite-sized sample is 0.5927. Using this value, I calculated ν using the relation:

$$\log(L) = -\nu \cdot \log\left(|p_c(L) - p_c(\infty)|\right) + c$$

This gives $\nu \approx 1.337$, which is fairly close to the theoretical value $\nu = 1.333$. Therefore, for an infinite-sized sample, our critical point would be very close to $p_c = 0.597$.

7.critical exponent for binding percolation

This question is similar to the previous one, but in this case, we must use the **bond percolation** algorithm. With a slight modification of the `Hoshen-Kopelman2` function, I created the `Hoshen-Kopelman binding` function (you can check the details in the code). After this, as before, I used the `correlation_length` function to calculate the correlation length for different sample sizes and activation probabilities. The results are as follows:

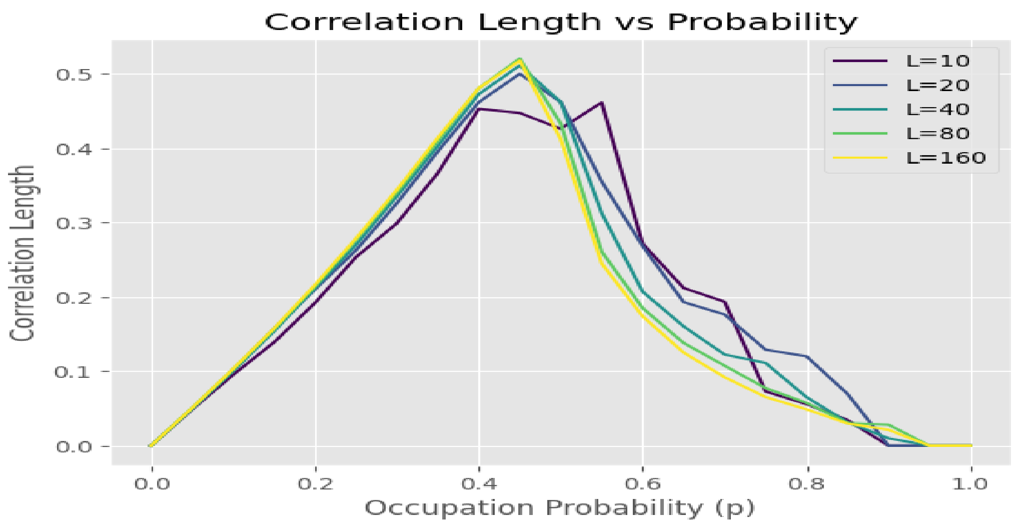

Figure 12: Correlation length for L = [10,20,40,80,160] at different activation probability near critical point.

now lets see its change near critical point with smaller steps:

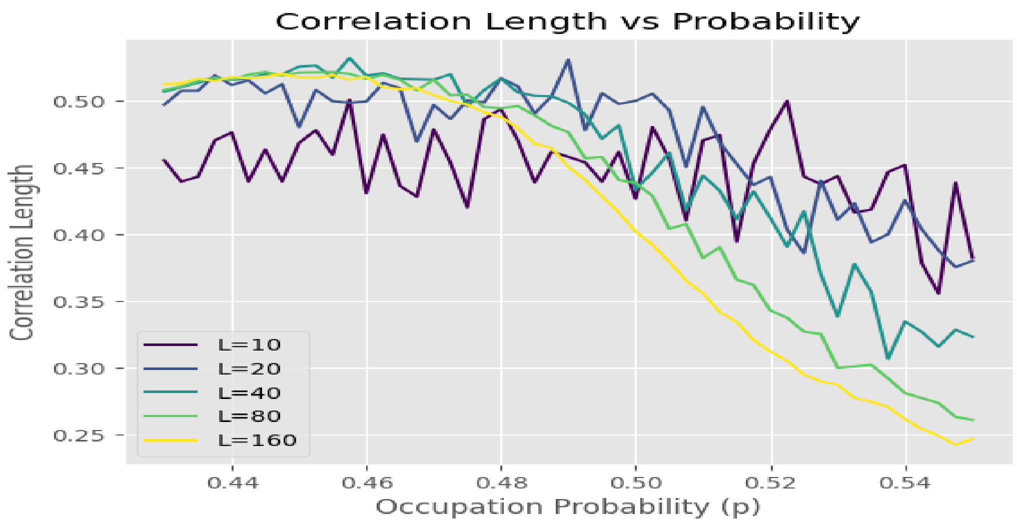

Figure 13: Correlation length for L = [10,20,40,80,160] at different activation probability near critical point.

Here, we can see that the critical value for $L = 160$ is 0.46. For binding percolation, the value for an infinite-sized sample is 0.5. Using this value, I calculated ν using the relation:

$$\log(L) = -\nu \cdot \log\left(|p_c(L) - p_c(\infty)|\right) + c$$

This gives $\nu \approx 1.362$, which is fairly close to the theoretical value $\nu = 1.333$(only 2.2% error),for cite percolation our estimation was much better(only 0.3% error).

8.Cluster Growth

This algorithm is fairly simple and easy to implement. It creates only one cluster. I implemented it as mentioned in the text, starting the cluster formation from the middle of the matrix. For each probability, I repeated the process 100 times. At each iteration, I calculated the **radius of gyration** and the **size of the cluster** (you can check the code for more details). The log-log plot is as follows:

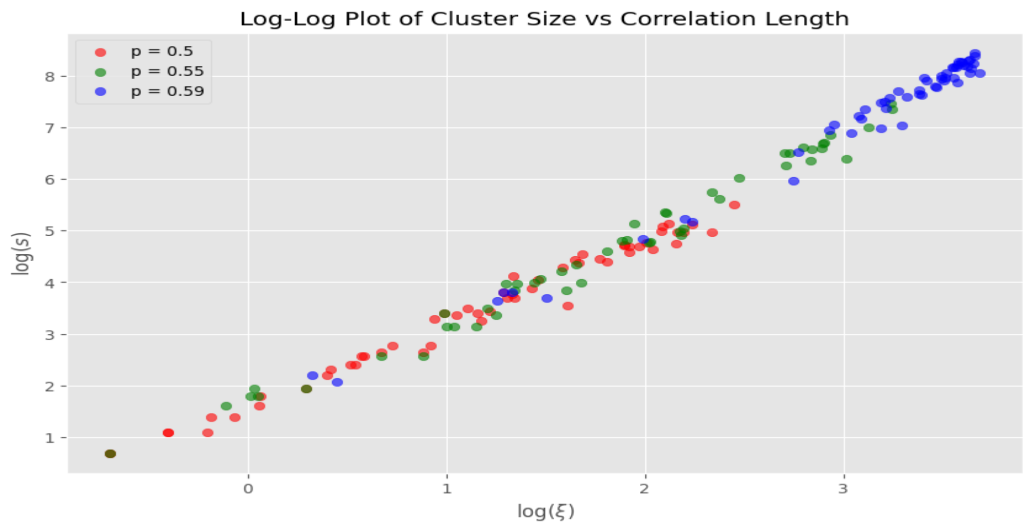

Figure 14: log(size of clusters) vs log(Correlation length) for P= [50,55,59]

as you can see the plot is linear as expected.

9.Transformation

Here, I use the same approach as in the text for calculating the probability of obtaining an inactive cell (0 cell). If, in the transformed lattice, we denote the probability of having an active cell by p', then the probability of having an inactive cell is $1 - p'$. We can obtain an inactive cell in **nine different variations** from the original lattice:

- One case where all the cells are inactive (0),
- Four cases where there is exactly **one active cell**, and
- Four cases where there are exactly **two active cells**.

Thus, the resulting expression is:

$$1 - p' = (1-p)^4 + 4 \cdot p^2 \cdot (1-p)^2 + 4 \cdot p \cdot (1-p)^3$$

Alternatively, if we express the **inactive cell** probability using p' and p, we have:

$$p' = p^4 + 4 \cdot p^2 \cdot (1-p)^2 + 4 \cdot p^3 \cdot (1-p)$$

By setting $p' = p$, we obtain **three non-negative solutions**:

$$p' = 0, \quad p' = 1, \quad p' \approx 0.618$$

You can check the code where I used the **fsolve** function from the **Scipy** library to calculate the **non-obvious** solution. Here is a plot for this equation:

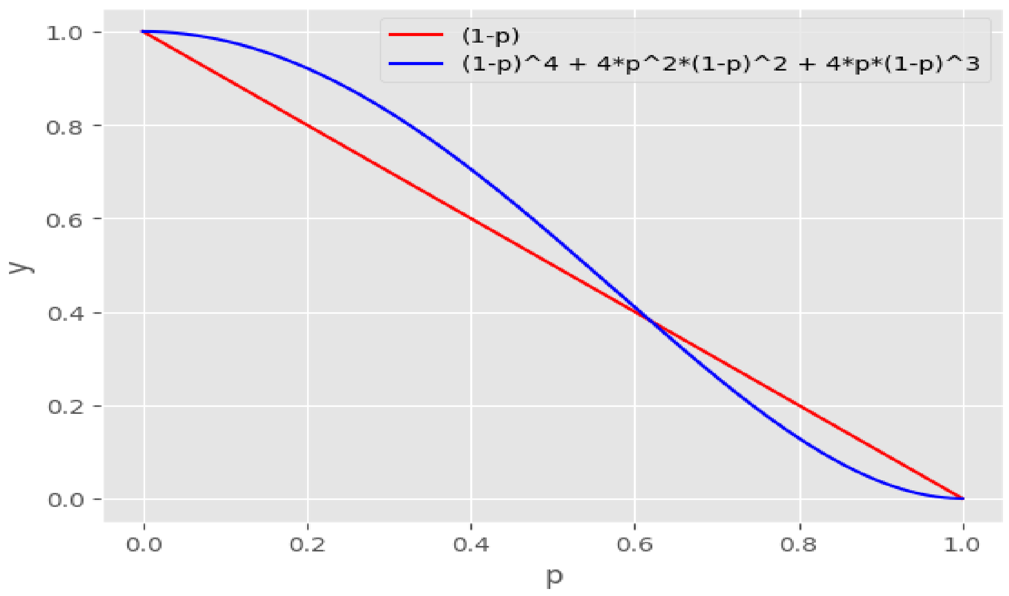

Figure 15: two sides of the first equation.