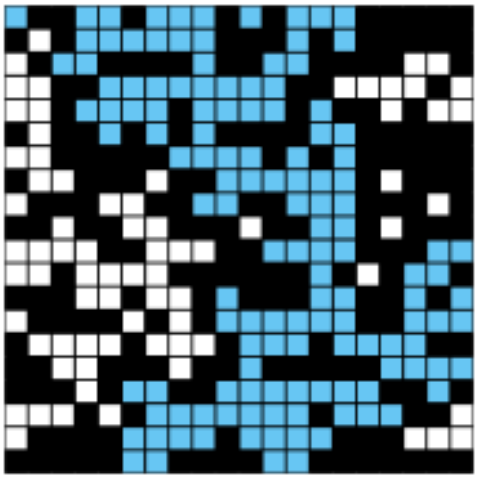
Percolation

Group 9:
Study of a complex problem with the computer

Tfeil El Houssein,
Ragnhild Aaraas Hånde,
Aigars Langins,
David O'Neill



Supervisors: Leticia Cugliandolo and Alessandro Tartaglia
Laboratoire de Physique Théorique et Hautes Energies
Université Piere et Marie Curie
Paris, France
March 2017

Contents

1 Introduction

Percolation is an interesting phenomenon which is deeply intertwined with several concepts in the world of physics. The word "percolate" descends from Latin and means "to filter" or "to trickle through." A good way to visualize percolation is to take a jar completely filled with dirt, and to begin pouring water into the jar. The question of interest is whether or not the water will reach the bottom of the jar, or come to rest somewhere in the middle. If the former is true, the water is said to have "percolated" through the dirt.

When considering the quantitative theory of percolation, it is possible to establish several parallels to statistical physics, specifically phase transitions and critical phenomena. These properties are in turn used to model a wide variety of occurrences in nature, such as forest fires, ecological migration, material conductivity, and information propagation. As it turns out, percolation is a highly complex problem to which very few analytical solutions exist, and is therefore typically modelled with numerical simulations.

2 Theory

2.1 Properties of the lattice

Percolation theory describes the behaviour of connected clusters in a graph [7]. This paper will take a closer look at percolation theory involving square lattices (or regular graphs). To be able to delve deeper into specific aspects of the theory it is necessary to first define some key concepts:

- **Lattice:** a symmetric, periodic structure of sites in D dimensions, where $D \in \mathbb{N}$.
- **Site:** a position in the lattice that can be "occupied" or "unoccupied" (as determined by the site occupation probability of the lattice).
- **Site occupation probability:** the probability that a site in a lattice is occupied (denoted hereafter p).
- **Nearest neighbors:** the set of sites immediately surrounding a particular site in each coordinate direction of the lattice (that is, in each of the D dimensions).
- **Cluster:** a group of connected, occupied sites.
- **Percolating lattice:** lattice containing a cluster that spans from one side of the lattice to the other, either horizontally or vertically.

It is interesting to look at how the properties of a lattice of a given size change when the site occupation probability changes. The first requirement is to define the probability for a lattice to percolate $\Pi(p)$. It is possible to estimate this probability by using Monte Carlo simulations of many lattices and finding the ratio given by

$$\Pi(p) = \frac{\text{number of percolating lattices}}{\text{total number of sampled lattices}}. \tag{1}$$

The critical probability p_c is defined as the site occupation probability p at which an infinite lattice starts to percolate. For a two dimensional lattice the critical probability is known and given by

$$p_c = 0.592746... \tag{2}$$

The infinite lattice thus undergoes a phase transition at the critical probability p_c. This means that for all $p > p_c$, the lattice will percolate, and for all $p < p_c$, there is no percolating cluster and the percolation probability will be a step function given by

$$\Pi(p) = \begin{cases} 0 & \text{for } p < p_c \\ 1 & \text{for } p > p_c. \end{cases} \tag{3}$$

To describe if an arbitrary site belongs to the infinite, percolating cluster, the strength of the network $P(p)$ is used. It is defined as the fraction of sites in the lattice that belong to the percolating cluster.

$$P(p) = \frac{\text{number of sites in percolating cluster}}{\text{total number of sites}}. \tag{4}$$

By definition, this formula holds only for $p > p_c$. If $p < p_c$, no site can belong to the percolating cluster, since the percolating cluster does not exist.

One other important property is the cluster number $n_s(s)$ which is the number of clusters of size s per lattice and is defined as

$$n_s(s) = \frac{\text{number of size } s \text{ clusters}}{\text{number of sites}}. \tag{5}$$

From the cluster number we can introduce the susceptibility $S(p)$ of a lattice. In general the susceptibility describes how the system responds to external stimuli, and in percolation theory it is defined by

$$S(p) = \frac{\sum_s n_s s^2}{\sum_s n_s s}. \tag{6}$$

The susceptibility can be thought of as the average cluster size in the lattice.

The last concept that we are going to discuss is the correlation length $\xi(p)$ that gives a relevant length scale of the lattice. To determine the correlation length it is necessary to first find the correlation function $g(r)$ which is the probability that sites some distance r from a root site x belong to the same cluster as x. The correlation function can be defined as

$$g(r) = \frac{\text{number of sites in the same cluster at radius } r}{\text{number of sites at radius } r}. \tag{7}$$

With this in mind, it is possible to define the correlation length as

$$\xi^2(p) = \frac{\sum_r r^2 g(r)}{\sum_r g(r)}. \tag{8}$$

In other words the correlation length represents some average distance between two sites belonging to the same cluster.

2.2 Critical exponents

For an infinite lattice, the properties $P(p)$, $S(p)$ and $\xi(p)$ obey power laws when considering probability regions close to the critical probability p_c. These power laws are described by critical exponents, as can be seen below. We are interested in looking at the critical exponents for the strength of the lattice $P(p)$ given by β, the susceptibility $S(p)$ given by γ and the correlation length $\xi(p)$ given by ν.

The power laws are given by

$$P \propto |p - p_c|^{\beta} \tag{9}$$

$$S \propto |p - p_c|^{-\gamma} \tag{10}$$

$$\xi \propto |p - p_c|^{-\nu} \tag{11}$$

2.3 Finite size scaling

Percolation in general concerns infinite lattices, which are currently impossible to simulate in finite time on their computers. Fortunately there is a method known as finite size scaling which can be used to estimate the corresponding critical exponents. It can be used to eliminate finite size effects; for example, the strength of a lattice P should begin precisely at p_c on an infinite lattice, but in Fig. 3. it can be seen that the curves begin gradually rising before the critical probability. That is a purely finite size effect.

There is only one relevant length scale, and it is the correlation length ξ. As long as $L \gg \xi$, the finite size effects should not be relevant. On the other hand, when $L \ll \xi$, L will determine the length scale of the lattice.

In general, if a quantity X is predicted to scale as $|p - p_c|^{-\chi}$ it is expected to obey the general scaling law

$$X(L, \xi) = \xi^{\chi/\nu} x_1(L/\xi) \propto \begin{cases} \xi^{\chi/\nu}, & L \gg \xi \\ L^{\chi/\nu}, & L \ll \xi \end{cases} \tag{12}$$

If done correctly, finite size scaling should eliminate any finite size effects. For example, for the strength of a lattice P, plotting $P(p, L) \cdot (p - p_c)^{-\beta}$ against $(p - p_c)/L^{-1/\nu}$ should collapse all curves for different lattice sizes L into a single *master curve*.

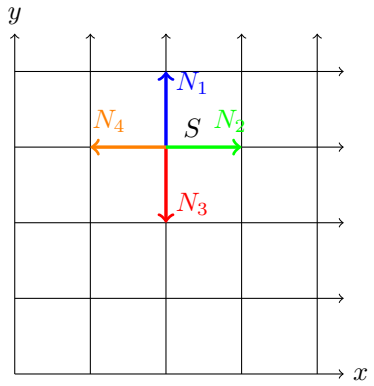
3 Algorithms

Although deeply tied to statistical physics, percolation is at an abstract level a sub-domain of graph theory. Therefore, the problem can be approached as a study of a connectivity network. There are a wide variety of well-known algorithms which can be used to systematically scan the lattice for clusters of occupied sites. For the sake of simplicity, we will consider a non-recursive breadth-first search.

3.1 Breadth First Search

Simply searching the sites in order is not sufficient to determine which sites comprise a cluster. For that reason, we implement a breadth-first search (BFS) algorithm. In 2D, a site can have up to four nearest neighbors, with as little as two in the cases of corner-sites. In Figure 1, the neighbors are denoted N_1, N_2, N_3, and N_4.

Figure 1: Representation of the lattice.



We first establish an empty queue data structure (first-in-first-out) which will hold visited nodes. The search begins at a root node (typically the site in the upper-left corner of the lattice), and explores the neighbors for clusters according to the pseudo-code in Algorithm 1.

Algorithm 1 BFS

```
1:  function BFS(lattice, root)
2:      q = new Queue
3:      c = new Cluster
4:      add root to c
5:      q.enqueue(root)
6:      while q is not empty do
7:          s = q.dequeue()
8:          for each neighbor n of s do
9:              if n was not visited and n is occupied then
10:                 add n to c
11:                 q.enqueue(n)
12:             end if
13:         end for
14:     end while
15:     lattice.addCluster(c)
16: end function
```

Each time a new site is visited, it is added to the queue, and it's neighbors are examined. The queue is then systematically emptied, and when there are no longer any sites contained in the structure, this indicates that a cluster has been found. The BFS is then repeated from a new root node, and the process continues until every site in the lattice has been examined and all clusters are found. The implementation of the BFS is given in Appendix A.1.

3.2 Newman-Ziff

For a single lattice with a given occupation probability p, the BFS runs at time complexity $\mathcal{O}(M)$, where M is the total number of bonds (or sites). However, in considering some observable Q (for example, the average cluster size or critical occupation probability), it is almost always necessary to measure Q over a range of values of p. For this reason, a multitude of repeated simulations (and therefore repeated BFS calls) is required.

In order to combat these time inefficiencies for repeated lattice simulations, M. E. J. Newman and R. M. Ziff have defined a very useful algorithm which simplifies the time complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$ - which is quite a drastic improvement in the case of large lattice sizes [3]. The philosophy of the Newman-Ziff algorithm is remarkably straight-forward: Repeating the BFS over a range of different values of p requires the re-initialization of an entirely new lattice at each iteration. However, consider a valid lattice configuration with n occupied sites; a correct sample state with $n + 1$ occupied states is attained by simply adding one extra occupied site at a (pseudo-) random location. Therefore, a set of configurations is attainable by simply adding, one-by-one, occupied sites to an originally empty lattice at random locations. After each addition of an occupied site, the cluster information is updated and any observable Q can be recalculated. The pseudo-code in Algorithm 2 encapsulates briefly this concept.

Algorithm 2 Newman Ziff

```
    function NZ(L)                                   ▷ L := linear length of the lattice
2:      lattice = new empty Lattice(L)
        permutation = array containing random site addition sequence
4:      for i → length(permutation) do
            site = lattice.sites[permutation[i]]
6:          site.occupied = true
            for each neighbor n of site do
8:              updateClusterInformation(n)
            end for
10:         calculateObservable(lattice)
        end for
12: end function
```

As each site is added to the empty lattice, the algorithm also makes use of a secondary algorithm known as a union/find algorithm in order to perform cluster merging. This is encapsulated in line 8 of the pseudo-code. If a new site is added at the intersection of two pre-existing clusters, the two clusters must be amalgamated.

4 Results and Discussion

The observables introduced in theory were numerically determined, and the results can be consulted in the figures 2 - 6 and in Table 1.

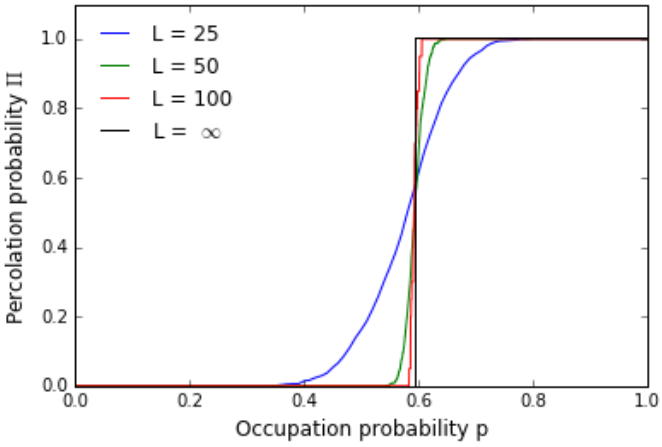
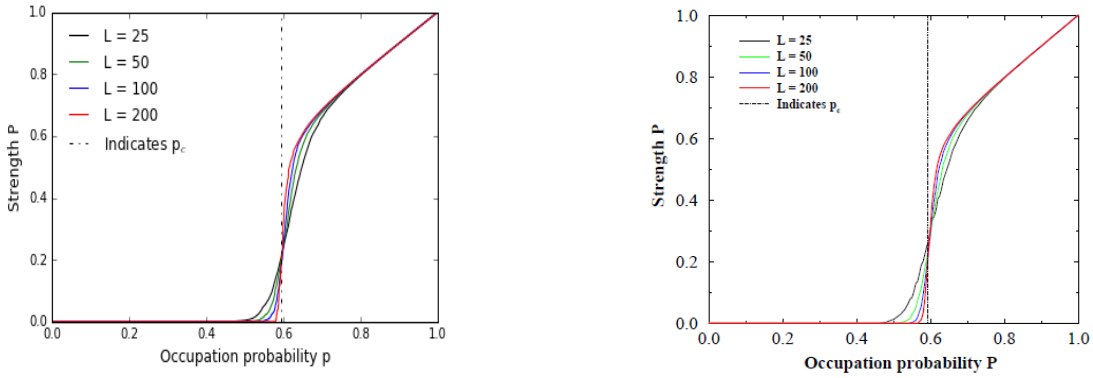


Figure 2: Percolation probability as a function of occupation probability.

Figure 2 gives the percolation probability as a function of the site occupation probability which was defined in equation (1) and numerically found by the Newman-Ziff algorithm. We know from equation (3) that for an infinite lattice, the percolation probability will be a step-function, and the step will be at the critical probability p_c, given by (2). This behaviour is quite evident in the plot. For the smallest lattice size ($L = 25$) there is a large spread about the critical occupation probability; this is the expected phenomena for such a small lattice size and is due to the finite size effects. However, it is clear that the data rapidly develops a resemblance to a step-like phase transition as the lattice size increases in the vicinity of p_c. At $L = 100$ the data closely mimics the theoretical curve for $L \to \infty$.



(a) Obtained numerical results.

(b) Results of Christensen [1].

Figure 3: Comparison of lattice strengths.

From the definition of the strength of the lattice given by (6) and with the help of the Newman-Ziff algorithm, the strength of the lattice where calculated for different sizes of the lattice, as shown in figure 3a. We can compare this result with the well accepted results produced by Christensen [1], which can be seen in Figure 3. The results we were able to produce match the Christensen results very well. The sharpening of the curves with an increase of lattice size is evident, and the linear nature at large values of p is to be expected. This is due to the fact that adding a site to an essentially full lattice will most likely belong to the percolating cluster.

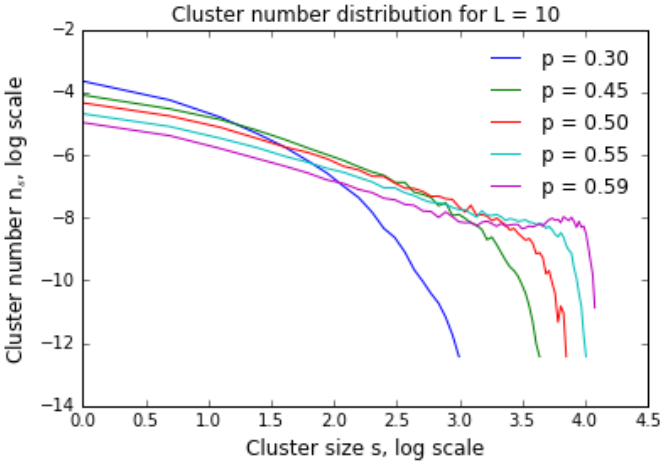


Figure 4: Cluster number distribution, log scale.

In Figure 4 we can see the cluster number as a function of cluster sizes, calculated with the Newman-Ziff algorithm. According to the theory, the cluster number n_s distribution should follow a power law for cluster sizes up to the "cut-off" size s_m. The s_m is defined as the value of s at which the cluster number curve quickly deteriorates. The figure obtained confirms this as the curves are approximately linear up to a certain cluster size. It is also expected that this "cut-off" size becomes larger as p approaches p_c, which can also be observed in the figure.

For the curve representing $p = 0.59$, a bump appears for large s, which can be explained due to the fact that close to p_c there will be a large, *almost* percolating cluster, which in turn significantly raises the n_s value.

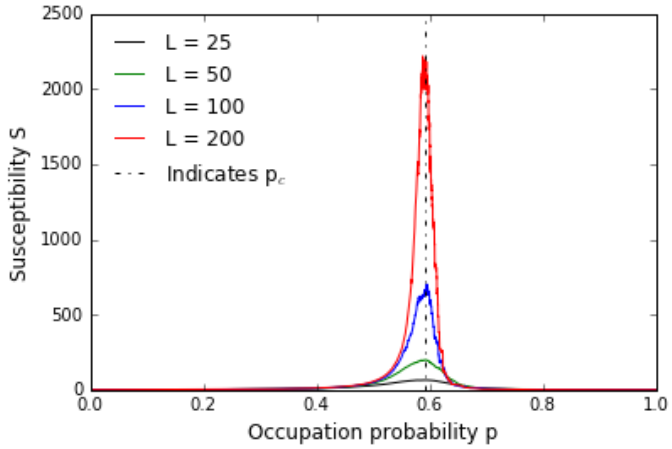


Figure 5: Susceptibility as a function of occupation probability.

We have seen from the previous plots that for $L \sim 100$ and larger, the lattice behavior is close to the behavior of the infinite lattice. In Figure 5 of the susceptibility $S(p)$, we can see a similar behavior, as the peak around p_c becomes more pronounced as L gets bigger. The first remark we can make is that the susceptibility seems to be more sensitive to the size of the lattice, as there is a considerable difference between the $L = 100$ and $L = 200$ results for the strength of the lattice (contrary to the susceptibility results). The peak tends to become larger and thinner as the size of the lattice is increased.

Very close to p_c there is once again an *almost* percolating cluster, that will raise the susceptibility significantly, as it is proportional to s^2. Thus, for an infinite cluster, we conclude $S \to \infty$ as $p \to p_c$.

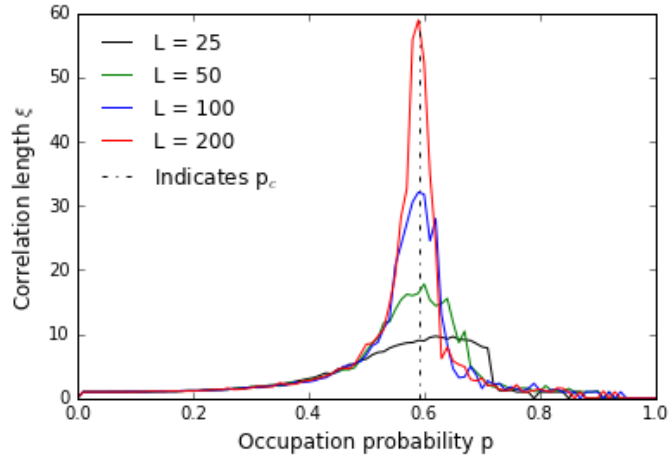


Figure 6: Correlation lengths for various lattice sizes.

By using the definition of the correlation function (7) and the correlation length (8), and with the help of the BFS algorithm, the correlation length was calculated as shown in Figure 6. A divergence is observed at the critical probability p_c. That is once again expected as the correlation length can be interpreted as the average distance between two sites of the same cluster and the average cluster size diverges close to p_c as well, as we have already seen in Figure 5.

Figure 6 shows that for the smaller lattices $L = 25, 50$ the curves are rather un-smooth. The effect disappears for larger lattices, as the lattice irregularities are averaged out. Another cause is that all sites are not equal in our implementation due to the fact that free boundary conditions were used, rather than periodic boundary conditions. The boundary is more apparent for smaller lattices.

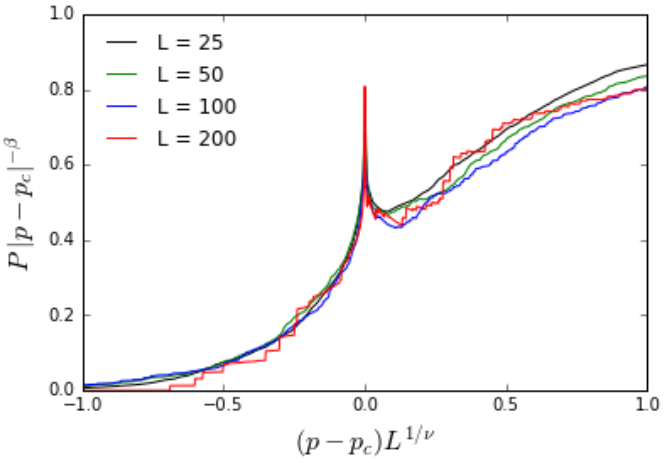


Figure 7: Finite size scaling of the strength of the lattice P.

In Fig 7. the results of the finite size scaling can be observed. Note that the accepted values for the critical exponents were used in creation of this plot. A master-curve was theoretically expected, and it can be seen that all the curves are rather close together, especially in the vicinity of p_c. The small discrepancies can be explained by the sample sizes used, which were rather costly in terms of time-complexity.

The peak at $p = p_c$ is unavoidable because of the expression being plotted; it can be seen that on the y axis values are multiplied by the distance from the critical probability, taken to be a negative power - $|p - p_c|^{-\beta}$.

The right-most portion of the plot of curves for $L = 100$ and $L = 200$ get very close to one another, which is indeed expected (master-curve).

Table 1: Critical exponents

Exponents	Calculated Value	Theoretical Value	Relative error
β	0,154	$\frac{5}{36}$=0,138	0,116
γ	2,122	$\frac{43}{18}$=2,388	0,111
ν	1,215	$\frac{4}{3}$=1,333	0,089

The critical exponents can be estimated from the obtained plots of strength P, susceptibility S and correlation length ξ, by fitting the data to a power law given by Equations (9) to (11).

These curves were fit for lattice sizes $L = \{25, 50, 100, 200\}$. The best matches came from the $L = 200$ lattice, and can be seen in the table below. It is however very hard to fit them precisely, as *very* large lattices and many samples are required. Ideally, the correlation length process would be implemented into the Newman-Ziff algorithm in order to eliminate the time complexity issues. Given in table 1 is also the relative error between the theoretical values and our calculated values of the exponents. We can see that the error is roughly 10%. This error could be minimized by sampling larger lattices and more sophisticated algorithmic/software approaches.

5 Applications

As mentioned in the introduction of this paper, percolation theory has a wide range of applications. Here we briefly discuss a few interesting applications.

We will begin with the manufacturing of gas masks [2]. The masks are made of porous carbon granules. In these granules, the pores form a random network of small, interconnected tunnels. If these pores are wide enough and sufficiently connected, the gas passes through the carbon. On the contrary, if the pores are too small or if they are imperfectly connected, the emanations can no longer pass through the filter. Consequently, the efficiency of the device depends on a critical point which is characteristic of the percolation phenomenon. In this example, the filtration is inactive below the critical threshold, because the gas can pass through the mask and it becomes active above the threshold since the gas can no longer pass through it.

Another example of an application to percolation theory is conductivity in a two-phase system, consisting of two phases A and B distributed randomly with concentrations p and $1 - p$ respectively [6]. Suppose that A has

a conductivity σ_a, while B is insulating ($\sigma_b = 0$). With a small concentration of A, the regions of A are isolated from one another and it is impossible to cross the medium without passing into the regions of B. The system is then insulating. Increasing the concentration p of A, the size of the phase A increases. At $p = p_c$, phase A covers just enough of the alloy, so that it becomes conducting and an insulator-conductor phase transition is observed.

Another great use of percolation theory can be found in the context of the universality of critical exponents - if two systems belong to the same universality class, their critical exponents are equivalent. It is sometimes possible to deduce the universality class of a physical system, but its critical exponent may be indeterminable. It is then sometimes possible to map this particular physical system to a percolation problem from which the critical exponents can be estimated numerically. By universality, the exponents of the physical system should then be the same. One example of this is the Potts model correspondence to a square bond percolation problem in the limit of $q \to 1$.

6 Conclusions

We have investigated and estimated different properties of two dimensional squared lattices of different sizes. With the Newman-Ziff and the Breadth First Search algorithms, we where able to compute the percolation probability $\Pi(p)$, the strength of the lattice $P(p)$, the cluster numbers $n_s(s)$, the susceptibility $S(p)$ and the correlation length $\xi(p)$ of a lattice. The obtained results correspond well to the expected results. The critical exponents were found to within about 10% accuracy of the known values. For small lattices some finite size effects were present, but approaching the limit of the infinite lattice caused the finite size effects to dissipate. The results were acceptably close to the theoretical infinite lattice results.

Acknowledgments

This project would not have been possible without the help of our primary advisors, Leticia Cugliandolo and Alessandro Tartaglia. They provided deep insight, expertise, and guidance to our efforts. For this, we thank them greatly.

References

[1] Christensen, K, Percolation Theory, 2002.

[2] Hammersley and Walsh, Percolation theory for mathematicians, 1980, p. 593, Springer.

[3] Newman, M. E. J and Ziff, R. M, Fast Monte Carlo algorithm for site or bond percolation, physical review E, volume 64, 016706, 2001.

[4] Sator, N, chapter 3 phase transition , http://www.lptl.jussieu.fr/user/sator/Chapitre3.pdf

[5] Stauffer, D and Aharony, A, Introduction to Percolation Theory, Taylor & Francis, second edition, 1994.

[6] Turban, L, Percolation et conduction, Visited 02. March 2017,http://physique.dep.univ-lorraine.fr/2013/06/24/percolation-et-conduction-document-de-cours-m2/

[7] Percolation Theory, Visited 19.March 2017, https://en.wikipedia.org/wiki/Percolation_theory

Appendix A Breadth First Search

A.1 The Algorithm

```
def bfs(lattice, root_pos):
    queue = []
    root = lattice.sites[root_pos]
    cluster = Cluster(lattice, root)
    queue.append(root)
    cluster.add_site(root)
    root.cluster = cluster
    while queue:
        site = queue.pop()
        for direction in range(1, 5):
            neighbor = site.get_nearest_neighbor(direction)
            if neighbor.occupied and not neighbor.visited:
                neighbor.visited = True
                queue.append(neighbor)
                cluster.add_site(neighbor)
                neighbor.cluster = cluster
    lattice.add_cluster(cluster)
```

A.2 The Classes

```
import random


# Represents a node in the lattice
class Site(object):
    def __init__(self, position, lattice):
        self.position = position  # the site's lattice position
        self.lattice = lattice  # the parent lattice
        self.occupied = False  # is the site occupied?
        self.visited = False  # has the site been visited by the DFS?
        self.row_number = None  # row position in the 2D lattice representation
        self.cluster = None  # the cluster this site belongs to (if any)
        self.left = False  # is the site a left-boundary node?
        self.right = False  # is the site a right-boundary node?
        self.top = False  # is the site a top-boundary node?
        self.bottom = False  # is the site a bottom-boundary node?

        self.check_boundary()

    # establishes the boundary nature of the site
    def check_boundary(self):
        x = self.position
        l = self.lattice.length
        if ((x + 1) % l) == 0:
            self.right = True
        if (x + 1) > (l * (l-1)):
            self.bottom = True
        if x % l == 0:
            self.left = True
        if (x + 1) < l:
            self.top = True

    # gets the set of closest neighboring sites (neighbors at r = 1)
    def get_nearest_neighbor(self, direction):
        x = self.position
        l = self.lattice.length

        if direction == 1:
            if self.right:
                return self.lattice.sites[x]
            return self.lattice.sites[x + 1]
        elif direction == 2:
            if self.bottom:
                return self.lattice.sites[x]
            return self.lattice.sites[x + l]
        elif direction == 3:
            if self.left:
```

```
48                  return self.lattice.sites[x]
49              return self.lattice.sites[x - 1]
50          elif direction == 4:
51              if self.top:
52                  return self.lattice.sites[x]
53              return self.lattice.sites[x - l]
54
55      # a general neighbor-finding function for a specified distance r
56      def find_neighbors_at_radius(self, r):
57          x = self.position
58          neighbors = []
59          for i in range(0, r):
60              try:
61                  if (self.lattice.sites[x + i * self.lattice.length].row_number ==
62                          self.lattice.sites[x + (r - i) + self.lattice.length * i].row_number):
63                      neighbors.append(self.lattice.sites[x + (r - i) + self.lattice.length * i])
64
65              except IndexError:
66                  pass
67              try:
68                  if (self.lattice.sites[x + self.lattice.length * (r - i)].row_number ==
69                          self.lattice.sites[x - i + self.lattice.length * (r - i)].row_number):
70                      neighbors.append(self.lattice.sites[x - i + self.lattice.length * (r - i)])
71              except IndexError:
72                  pass
73              try:
74                  if (self.lattice.sites[x - self.lattice.length * i].row_number ==
75                          self.lattice.sites[x - (r - i) - self.lattice.length * i].row_number):
76                      if (x - (r - i) - self.lattice.length * i) >= 0:
77                          neighbors.append(self.lattice.sites[x - (r - i) - self.lattice.length * i])
78              except IndexError:
79                  pass
80              try:
81                  if (self.lattice.sites[x - self.lattice.length * (r - i)].row_number ==
82                          self.lattice.sites[x + i - self.lattice.length * (r - i)].row_number):
83                      if (x + i - self.lattice.length * (r - i)) >= 0:
84                          neighbors.append(self.lattice.sites[x + i - self.lattice.length * (r - i)])
85              except IndexError:
86                  pass
87          return neighbors
88
89
90  # The 2D square lattice
91  class Lattice(object):
92      def __init__(self, length, occupation_probability):
93          self.length = length  # lattice dimension is (length x length)
94          self.occupation_probability = occupation_probability
95          self.size = self.length ** 2  # total number of sites in the lattice
96          self.clusters = []  # the collection of clusters present in the lattice
97          self.sites = []  # the collection of sites in the lattice
98
99          row_number = 0
100
101         for i in range(0, self.size):
102             site = Site(i, self)
103             self.sites.append(site)
104             if i % length == 0:
105                 row_number += 1
106             self.sites[i].row_number = row_number
107             if random.random() < self.occupation_probability:
108                 self.sites[i].occupied = True
109
110     def add_cluster(self, cluster):
111         self.clusters.append(cluster)
112
113
114 # Represents a cluster in the lattice. A cluster is a set of connected sites.
115 class Cluster(object):
116     def __init__(self, lattice, root):
117         self.lattice = lattice
118         self.sites = []
119         self.root = root
```

```
120             self.root_pos = root.position
121
122             self.sites.append(root)
123
124         def get_size(self):
125             return len(self.sites)
126
127         def is_percolating(self):
128             if self.get_size() < self.lattice.length:
129                 return False
130             top = False
131             bottom = False
132             left = False
133             right = False
134             for i in range(0, self.get_size()):
135                 if self.sites[i].top:
136                     top = True
137                     if bottom:
138                         return True
139                 elif self.sites[i].bottom:
140                     bottom = True
141                     if top:
142                         return True
143                 elif self.sites[i].left:
144                     left = True
145                     if right:
146                         return True
147                 elif self.sites[i].right:
148                     right = True
149                     if left:
150                         return True
151             return False
152
153         def add_site(self, site):
154             self.sites.append(site)
```

A.3 The Correlation Length

```
1  import numpy as np
2  import time
3  import matplotlib.pyplot as plt
4  from percolation_utils import *
5
6  length = 50
7  steps = 100
8  step = 1 / steps
9  samples = 100
10 probabilities = np.zeros(steps + 1)
11
12
13 # Runs a correlation-length analysis
14 def run_correlation():
15     print("Beginning run...")
16     start_time = time.time()
17     xi_plot = correlation_length()
18     print("Finished. Elapsed time: ", time.time() - start_time)
19     plot(probabilities, xi_plot)
20
21
22 # Plots results
23 def plot(p, xi_array):
24     plt.figure(0)
25     plt.title("Correlation Length vs Occupation Probability")
26     plt.xlabel("p")
27     plt.ylabel(r'$\xi$(p)')
28     plt.plot(p, xi_array)
29     plt.grid()
30     plt.show()
31
32
33 # Finds the correlation function for the center site in the lattice
34 def correlation_function(lattice):
```

```
35      cf = np.zeros(2 * lattice.length)
36      center = (lattice.length * (lattice.length - 1) + (lattice.length - 1)) // 2
37      center_site = lattice.sites[center]
38      if center_site.occupied and not center_site.cluster.is_percolating():
39          distance = 1
40          neighbors = center_site.find_neighbors_at_radius(distance)
41          while distance < (2 * lattice.length) and neighbors:
42              if distance == 1:
43                  cf[distance - 1] = 1
44                  distance += 1
45                  neighbors = center_site.find_neighbors_at_radius(distance)
46                  continue
47              numerator = 0
48              for i in range(0, len(neighbors)):
49                  neighbor = neighbors[i]
50                  if neighbor.occupied and id(neighbor.cluster) == id(center_site.cluster):
51                      numerator += 1
52              if numerator == 0:
53                  break
54              cf[distance - 1] = numerator / len(neighbors)
55              distance += 1
56              neighbors = center_site.find_neighbors_at_radius(distance)
57      return cf
58
59
60  # Finds the correlation length for a set of sample lattices through a probability range
61  def correlation_length():
62      xi_squared = []
63      for i in range(0, steps + 1):
64          probabilities[i] = step * i
65          if i != 0:
66              print("Preparing new sample set...")
67          print("Probability: ", probabilities[i])
68          correlation_matrix = []
69          for j in range(0, samples):
70              lattice = Lattice(length, probabilities[i])
71              for k in range(0, lattice.size):
72                  if lattice.sites[k].occupied and not lattice.sites[k].visited:
73                      bfs(lattice, k)
74              cf = correlation_function(lattice)
75              correlation_matrix.append(cf)
76          avg = np.mean(correlation_matrix, axis=0)
77          xi_val = 0
78          sum_cf = sum(avg)
79          if sum_cf != 0:
80              for j in range(0, len(avg)):
81                  xi_val += (j + 1) * (j + 1) * avg[j]
82              xi_val /= sum_cf
83              xi_squared.append(xi_val)
84          else:
85              xi_squared.append(0)
86      xi = np.sqrt(xi_squared)
87      return xi
88
89  run_correlation()
```

Appendix B Newman-Ziff

B.1 The Algorithm

```
1
2   def NZ(L, testing):
3       lattice = Lattice(L, 0)
4       size = L**2
5       permutation = np.random.permutation(size)
6       size += 1 # new size to account for the full cluster
7
8       clusters = {} # dictionary containing cluster objects
9       numbers = np.zeros(size-1) # array containing cluster numbers
10      all_numbers = []
11      occ_nbs = np.zeros(size) # array containg the probability values / x axis values in the plots
```

```
12      percs = np.zeros(size)
13      avg_sizes = np.zeros(size)
14      strengths = np.zeros(size)
15      percolates = 0
16
17
18      def avg_size():
19          sum1 = 0
20          sum2 = 0
21          if percolates:
22              copy = np.copy(numbers)
23              index = np.where(copy)[0].max()
24              copy[index] = 0
25              s = np.arange(1, len(numbers) + 1)
26              s2 = np.square(s)
27              sum1 += np.dot(s2, copy)
28              sum2 += np.dot(s, copy)
29              S = sum1 / sum2
30              if np.isnan(S):
31                  S = 0
32              return S
33          s = np.arange(1, len(numbers) + 1)
34          s2 = np.square(s)
35          sum1 += np.dot(s2, numbers)
36          sum2 += np.dot(s, numbers)
37          return sum1 / sum2
38
39
40      def strength(clusters):
41          if not percolates:
42              return 0
43          else:
44              return percolating_cluster.size / (size - 1)
45
46
47      for i in range(size): # define the occupation probability array
48          occ_nbs[i] =  i/(size-1)
49
50      for i in range(len(permutation)):
51          site = lattice.sites[permutation[i]]
52          cluster = Cluster(lattice, site, site)
53          site.cluster = cluster
54          last_cluster = cluster
55          clusters[id(cluster)] = cluster
56          numbers[0] += 1
57
58          lattice.addSite(site)
59          nearest = neighbours(site)
60
61          for n in nearest:
62              if site.cluster == n.cluster:
63                  last_cluster = site.cluster
64                  continue
65
66              else:
67                  del clusters[id(site.cluster)], clusters[id(n.cluster)]
68                  if site.cluster.size < n.cluster.size:
69                      numbers[site.cluster.size-1] -= 1
70                      numbers[n.cluster.size-1] -= 1
71                      n.cluster.addCluster(site.cluster)
72                      numbers[n.cluster.size-1] += 1
73                      clusters[id(n.cluster)] = n.cluster
74                      last_cluster = n.cluster
75                  else:
76                      numbers[n.cluster.size-1] -= 1
77                      numbers[site.cluster.size-1] -= 1
78                      site.cluster.addCluster(n.cluster)
79                      numbers[site.cluster.size-1] += 1
80                      clusters[id(site.cluster)] = site.cluster
81                      last_cluster = site.cluster
82
83          if not percolates:
```

```
84              if last_cluster.size >= L:
85                  if isPercolating(last_cluster):
86                      percolating_cluster = last_cluster
87                      percolates = 1
88
89          percs[i+1] = percolates
90          avg_sizes[i+1] = avg_size()
91          strengths[i+1] = strength(clusters)
92
93      return [occ_nbs, percs, avg_sizes, all_numbers, strengths]
```