Aalto University
School of Science

Matias Haapalehto

Two-dimensional site percolation with periodic boundary conditions

Course project report

PHYS-E0412 — Computational Physics

Report submitted : Spring 2020

1 Background

The statement of site percolation is quite simple: every site in a lattice is either occupied with a probability p, or not with probability $1-p$. Neighboring occupied sites are grouped into clusters. At a specific value of p, we can observe an interesting phenomenon: the formation of a cluster spanning the entire system. This is called percolation. In the limit of an infinite lattice, there is a finite threshold probability p_c such that [1]:

- for $p < p_c$ no spanning cluster exists and all clusters are finite
- for $p > p_c$ a spanning cluster exists
- for $p = p_c$ a spanning cluster exists with a finite probability larger than zero and less than one.

No exact results are known for the percolation threshold of a square lattice, which makes the problem an interesting model to solve computationally.

The model originates from studies in percolation in materials such as rock or concrete [2]. Other applications outside physics include modeling resistor networks or forest fires [2].

2 Theory and methodology

The percolation threshold is defined as the site occupation probability at which a spanning cluster first appears in an infinite lattice [1]. Since we can only simulate a finite lattice, we must choose a definition of the spanning cluster. For example, we can define it as a cluster that (i) spans the lattice either horizontally or vertically, (ii) spans the lattice in a fixed direction, (iii) spans the lattice both horizontally or vertically or (iv) spans the lattice only in one direction. The value of p_c depends also on the type of lattice and its dimension. The lattice we use in this study is the two-dimensional square lattice, where each site has four nearest neighbors.

In our simulation, we want to estimate an observable quantity over a range of values of p. Now we would have to repeat the simulation at many values of p, which makes the computation slow. Instead, it is faster to measure the observable for fixed numbers of occupied sites in the range of interest [2]. Let us refer to the ensemble of states of a system with n occupied sites as the *microcanonical percolation ensemble*. On the other hand, the case where the occupation probability p is fixed is called the *canonical percolation ensemble*. In the canonical ensemble, the probability of there being exactly n occupied sites is given by the binomial distribution:

$$B(N, n, p) = \binom{N}{n} p^n (1-p)^{N-n} \tag{1}$$

Where N is the total number of sites. Therefore, if we can measure the set of observables Q_n within the microcanonical ensemble, then the value in the canonical ensemble is given

by a convolution of the set of measurement with the binomial distribution:

$$Q(p) = \sum_{n=0}^{N} B(N, n, p) Q_n = \sum_{n=0}^{N} \binom{N}{n} p^n (1-p)^{N-n} Q_n \tag{2}$$

Direct evaluation of the binomial coefficients using factorials is not possible, therefore we will use the following method. The binomial distribution reaches a maximum for a given N when $n = n_max = pN$. We set this value to 1 for the time being. Then we compute the rest of the values of $B(N, n, p)$ iteratively using:

$$B(N, n, p) = \begin{cases} B(N, n-1, p) \frac{N-n+1}{n} \frac{p}{1-p}, & n > n_{max} \\ B(N, n+1, p) \frac{n+1}{N-n} \frac{1-p}{p}, & n < n_{max} \end{cases} \tag{3}$$

Lastly, we normalize the distribution by dividing each value by the normalization constant $C = \sum_n B(N, n, p)$.

One observable of interest is the probability $R_L(p)$. This is defined as the probability, for a value of p, that there is a cluster spanning completely a square of linear dimension L with periodic boundary conditions. There are several possible ways in which wrapping can occur, and therefore multiple definition for R_L [2]:

- $R_L^{(h)}$ is the probability that the system contains a cluster which wraps around the horizontal direction
- $R_L^{(e)}$ is the probability that the system contains a cluster which wraps around either the horizontal or vertical direction, or both
- $R_L^{(b)}$ is the probability that the system contains a cluster which wraps around both horizontal and vertical directions
- $R_L^{(1)}$ is the probability that the system contains a cluster which wraps around either direction, but not both.

An example of a cluster wrapping in either direction is given in Figure 1.

These probabilities satisfy the following equations:

$$R_L^{(e)} = 2R_L^{(h)} - R_L^{(b)} \tag{4}$$

$$R_L^{(1)} = R_L^{(h)} - R_L^{(b)} \tag{5}$$

Therefore only two of them are independent. The reason we are interesting in the wrapping probabilities is that their values can be calculated exactly for an infinite lattice [2]. The values to ten figures are given by:

$$R_\infty^{(h)}(p_c) = 0.521058290 \tag{6}$$

$$R_\infty^{(e)}(p_c) = 0.690473725 \tag{7}$$

$$R_\infty^{(b)}(p_c) = 0.351642855 \tag{8}$$

$$R_\infty^{(1)}(p_c) = 0.169415435 \tag{9}$$

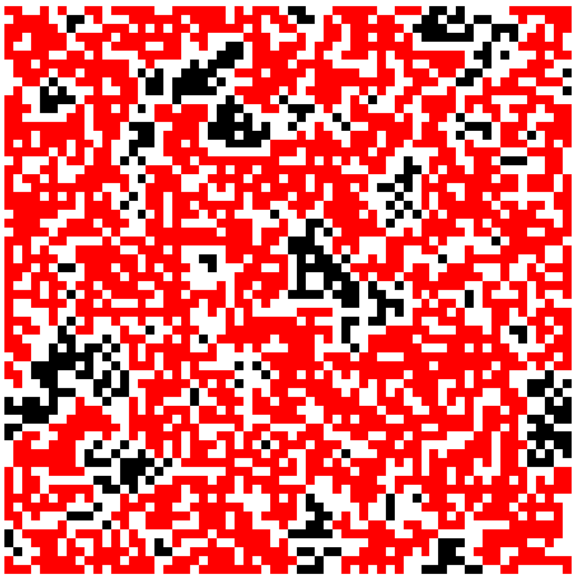

Figure 1: Sample lattice (L=64) where the spanning cluster is marked in red.

Now we can estimate p_c as the solution p of the following equation:

$$R_L(p) = R_\infty(p_c) \tag{10}$$

Our estimate of the mean of $R_L(p_c)$ over n runs is drawn from the binomial distribution which has standard deviation:

$$\sigma_{R_L} = \sqrt{\frac{R_L(p_c)[1 - R_L(p_c)]}{n}} \tag{11}$$

We can evaluate the error by approximating $R_L(p_c)$ by the known value of $R_\infty(p_c)$. Ref. [2] conjectures that $R_L(p_c)$ converges to $R_\infty(p_c)$ as L^{-2}. On the other hand, the width of the critical region decreases as $L^{-1/\nu}$, and therefore the gradient of $R_L(p)$ in the region goes as $L^{1/\nu}$. Thus the estimate of the critical occupation probability converges according to:

$$p - p_c \sim L^{-2-1/\nu} = L^{-11/4} \tag{12}$$

Since for percolation on a square lattice we have $\nu = \frac{4}{3}$.

2.1 Algorithm

The Newman-Ziff algorithm was employed to simulate the system within the micro-canonical ensemble. The steps of the algorithm are as follows [3]. The algorithm starts with an empty lattice . As the lattice is populated in a random order, clusters are given a label identifying them. When a bond connecting two sites is added, they are either a member of the same cluster or they are members of two different clusters. In this second

case, the labels need to be updated to reflect the fact that the bond has connected the clusters.

In the interest of efficiency, the clusters are stored in an array mimicking a tree structure in which a site is chosen to be the "root" of the cluster. All other sites in the cluster point either to the cluster root or another site in the cluster. Starting from any site, the root node can be found by following these pointers. Clusters can be connected together by a adding a pointer from the root of one to the root of the other.

This type of algorithm is called *weighted union–find with path compression*, weighted because the two trees are amalgamated by making the smaller a sub-tree of the other. Path compression signifies that the pointers of all traversed nodes are changed to point to the root node, in order to speed up the next searches. The steps taken to add one bond are roughly constant in time, therefore the algorithm adds N bonds in $\mathcal{O}(N)$ time.

In order find whether a configuration has percolated, additional work must be done. We add variables to each node which store the displacement to the parent node in both coordinates. Now the total displacement to the root node can be computed by traversing the tree and adding the displacements together. When a bond is added joining two sites i and j that are members of the same cluster, there are two paths from i to the root node. One that goes through the parent of i and another that consists of hopping to j and then going through j's parents. We sum the displacements along both of these paths, and percolation has occurred if the difference in displacement is equal to $\pm L$ in either of the coordinates [4].

The implementation of the algorithm was done in the C language, while Python libraries were used for the plots. The Mersenne Twister was the pseudo-random number generator employed. The program was tested with the value of L=2, since the results are easy to predict exactly.

3 Results from simulations

The number of Monte Carlo samples simulated for each value of L is given in Table 1. The results for the wrapping probabilities are given in Figure 2 and results of the finite size scaling estimation are given in Figure 3. Results for the percolation threshold estimates are given in Table 2. The error was estimated using the variance of the linear regression.

L	n
32	10^9
64	$2 \cdot 10^8$
128	$5 \cdot 10^7$
256	10^7

Table 1: Number of simulated Monte Carlo samples n for each value of the linear dimension L of the square lattice

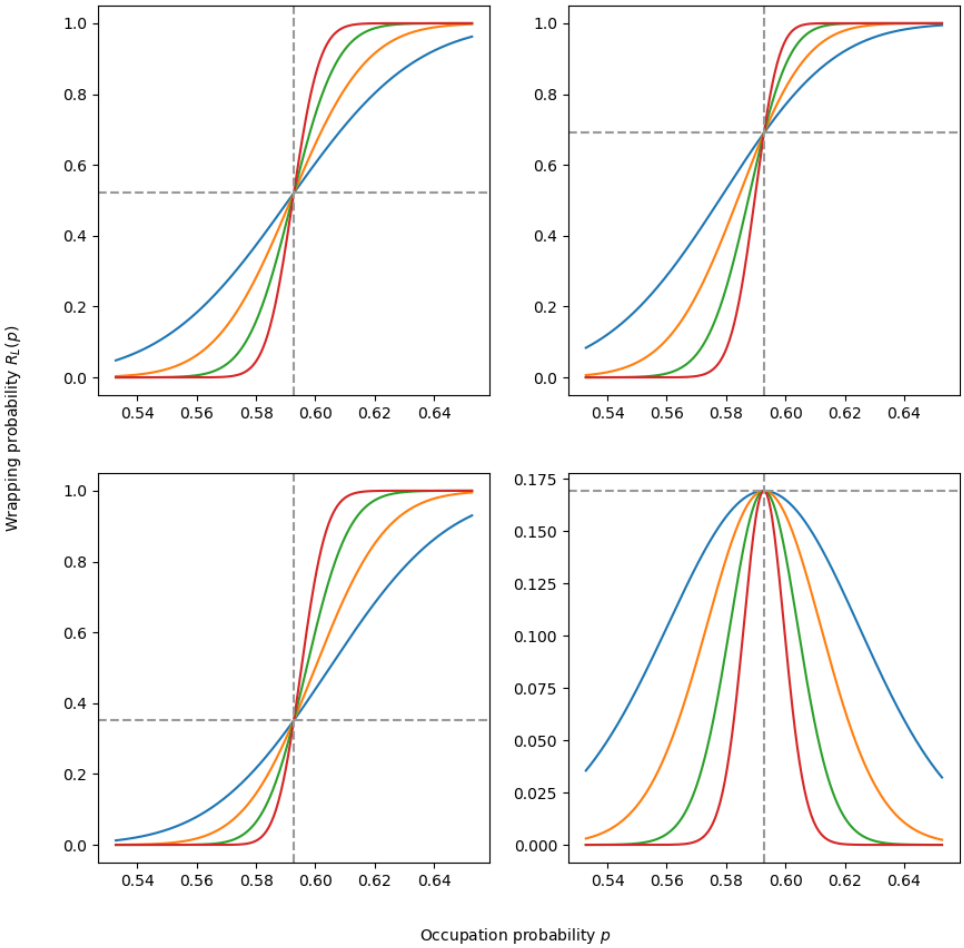

Figure 2: Plots of the wrapping probability functions $R_L(p)$ for (blue) $L = 32$, (orange) $L = 64$, (green) $L = 128$, (red) $L = 256$. The plots are zoomed on the region of critical percolation: (top left) $R_L^{(h)}$, (top right) $R_L^{(e)}$, (bottom left) $R_L^{(b)}$, (bottom right) $R_L^{(1)}$

$R_L^{(h)}$	0.5927467(20)
$R_L^{(e)}$	0.5927471(24)
$R_L^{(b)}$	0.5927464(15)
$R_L^{(1)}$	0.5927493(18)

Table 2: The percolation threshold determined using the different wrapping probability functions: $R_L^{(h)}$ along the horizontal axis, $R_L^{(e)}$ along either axis, $R_L^{(b)}$ along both axes, $R_L^{(1)}$ along a single axis.

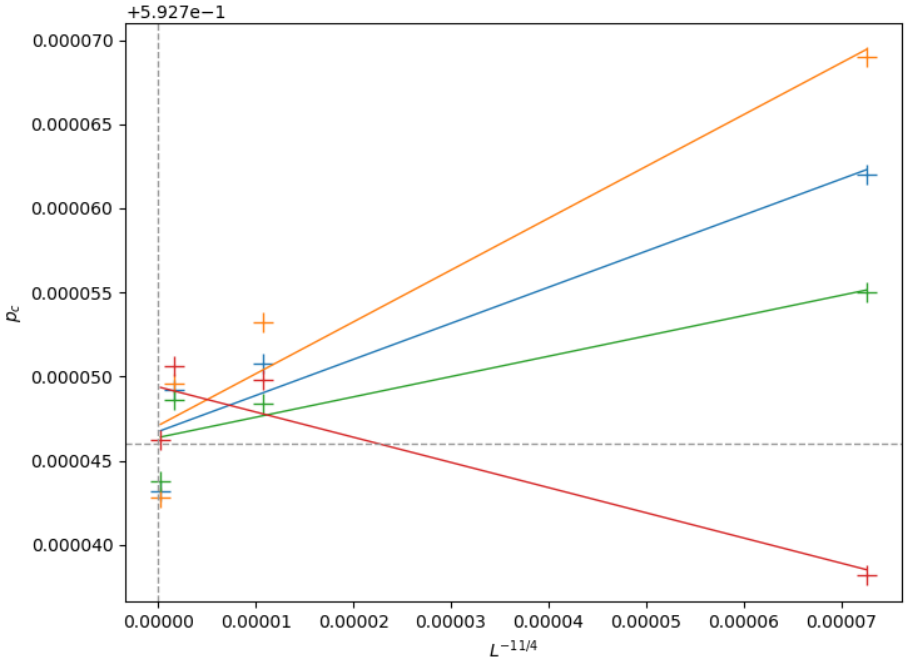

Figure 3: Finite size scaling estimate for the critical percolation threshold. The colors correspond to the various wrapping probability definitions: (orange) $R_L^{(e)}$, (blue) $R_L^{(h)}$, (green) $R_L^{(b)}$, (red) $R_L^{(1)}$

4 Conclusions and thoughts on further studies

Despite its simplicity, the percolation problem displays many interesting properties, such as phase transition at the percolation threshold. We simulated finite lattices using the Newman-Ziff algorithm, which is efficient way to solve the problem. Finally, we estimated the value of the percolation threshold using finite size scaling.

In the finite size estimation, we can see that the fit was quite poor due to the variance in the results. This is due to the lack of Monte Carlo samples, the number of which was heavily constrained by the time required to compute the simulation. A computation with more samples would yield an increased accuracy for the percolation threshold p_c. However, the computation duration was already in the order of tens of hours. Without parallelization and/or optimization, reproducing the results in [2] would take weeks or months. Still, our results are in reasonable agreement with the state-of-the-art, and we can conclude that this is an excellent method to estimate the percolation threshold.

Aside running the simulation longer, other interesting paths to take would be to simulate the system in higher dimensions, or on a different kind of lattice, such as a triangular lattice. A simulation in a continuum instead of a regular lattice would be interesting as well.

References

[1] H. Gould, J. Tobochnik, and W. Wolfgang Christian, *An Introduction to Computer Simulation Methods: Applications to Physical Systems*. Addison-Wesley, 2006. Third edition.

[2] M. E. J. Newman and R. M. Ziff, "Fast monte carlo algorithm for site or bond percolation," *Physical Review E*, vol. 64, Jun 2001.

[3] M. E. J. Newman and R. M. Ziff, "Efficient monte carlo algorithm and high-precision results for percolation," *Physical Review Letters*, vol. 85, p. 4104–4107, Nov 2000.

[4] S. Mertens and C. Moore, "Continuum percolation thresholds in two dimensions," *Physical Review E*, vol. 86, Dec 2012.

A The Newman-Ziff algorithm implemented in the C language

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <assert.h>
#include "mtwister.h"

/* Site percolation for square LxL lattice
with periodic boundary conditions */

#define L 256 /* Linear dimension */
#define N (L*L)
#define EMPTY (-N-1) /* Marker */

#define STACKSIZE 100



/*
For non-root occupied sites, contains the label for
the site's parent in the tree Root sites are marked
with a negative value equal to minus the size of the cluster
For unoccupied states the value is EMPTY
*/
int ptr[N];          /* Array of pointers */

/*
Contains a list of the nearest neighbors,
change this to work with a different topology
*/
int nn[N][4];        /* Nearest neighbors */

/*
Contains the random occupation order
*/
int order[N];        /* Occupation order */

/* Difference vector from the site to the cluster root */
int dx[N];
int dy[N];

/*
Sets up the nearest neighbors according to the chosen topology
```

```
periodical boundary conditions
*/
void boundaries()
{
    int i;

    for (i=0; i<N; i++) {
        nn[i][0] = (i+1)%N; /* Right */
        nn[i][1] = (i+N-1)%N; /* Left */
        nn[i][2] = (i+L)%N; /* Down */
        nn[i][3] = (i+N-L)%N; /* Up */
        if (i%L==0) nn[i][1] = i+L-1; /* First column */
        if ((i+1)%L==0) nn[i][0] = i-L+1; /* Last column */
    }
}


/*
Generates the random order in which the sites are occupied,
randomly permuting the integers from 0 to N-1
*/
void permutation(MTRand *r)
{
    int i,j;
    int temp;

    for (i=0; i<N; i++) order[i] = i;

    /* Initialization of the displacements*/
    for (i=0; i<N; i++) {dx[i] = 0, dy[i] =0;};
    for (i=0; i<N; i++) {

        /* Permute the current integer with a randomly
         chosen integer from further in the list */

        double drand = genRand(r);
        j = i + (N-i)*drand;
        j = j % N;
        assert(0 <= drand && drand <= 1);
        assert(0 <= j && j < N);

        /* Swap i and j */
        temp = order[i];
        order[i] = order[j];
        order[j] = temp;
```

```
    }
}


/* Finds the root pointer given a site */
int findroot(int i)
{
    int r; /* Root */
    int sp=0; /* Stack pointer */
    int stack[STACKSIZE];

    /* Stack of displacements */
    int dx_s[STACKSIZE], dy_s[STACKSIZE];
    /* Cumulative displacement */
    int dx_cum = 0, dy_cum = 0;

    r = i;

    while (ptr[r]>=0) {

        dx_s[sp] = dx[r];
        dy_s[sp] = dy[r];
        stack[sp] = r;
        r = ptr[r];
        sp++;
    }

    while (sp) {
        sp--;
        dx_cum += dx_s[sp];
        dy_cum += dy_s[sp];

        dx[stack[sp]] = dx_cum;
        dy[stack[sp]] = dy_cum;
        ptr[stack[sp]] = r;
    }
    return r;
}


/* Populate the lattice anc check for percolation */
int percolate()
{
    int i,j;
    int s1,s2;
```

```
int r1,r2;

/* Initialize lattice*/
for (i=0; i<N; i++) ptr[i] = EMPTY;

for (i=0; i<N; i++) {

    /* s1: site which will be occupied */
    r1 = s1 = order[i];
    ptr[s1] = -1;       /* Cluster of size 1 */

    for (j=0; j<4; j++) {   /* Iterate over nearest neighbors */
        s2 = nn[s1][j];     /* s2: neighboring cluster*/

        if (ptr[s2]!=EMPTY) { /* Merge the clusters */
            r2 = findroot(s2);

            /* Merge the clusters */
            if (r2!=r1) {
                if (ptr[r1]>ptr[r2]) {
                    ptr[r2] += ptr[r1];

                    /* Compress the displacement*/
                    findroot(s1);
                    ptr[r1] = r2;

                    /* Update the displacements */
                    findroot(s2);
                    dx[r1] = - dx[s1] + dx[s2];
                    dy[r1] = - dy[s1] + dy[s2];

                    /* Unit vector between the two sites */
                    switch (j) {
                        case 0:
                            dx[r1] += + 1;
                            break;
                        case 1:
                            dx[r1] += - 1;
                            break;
                        case 2:
                            dy[r1] += + 1;
                            break;
                        case 3:
                            dy[r1] += - 1;
                            break;
```

```
            }

            r1 = r2;

        } else {
            ptr[r1] += ptr[r2];
            /* Compress the displacements*/
            findroot(s2);
            ptr[r2] = r1;

            /* Update the displacements */
            findroot(s1);
            dx[r2] = - dx[s2] + dx[s1];
            dy[r2] = - dy[s2] + dy[s1];

            /* Find the unit vector between the two sites */
            switch (j) {
                case 0:
                    dx[r2] += - 1;
                    break;
                case 1:
                    dx[r2] += + 1;
                    break;
                case 2:
                    dy[r2] += - 1;
                    break;
                case 3:
                    dy[r2] += + 1;
                    break;
            }

        }
    }
    else {
        /* Internal bond, therefore we can test for percolation */
        int dx1 = 0, dy1 = 0, dx2 = 0, dy2 = 0;

        /* Compress the displacements */
        findroot(s1);
        findroot(s2);
        dx1 = dx[s1];
        dx2 = dx[s2];
        dy1 = dy[s1];
        dy2 = dy[s2];
```

```
/* Unit vector of the newly added bond */
switch (j) {
    case 0:
        dx1 += -1;
        break;
    case 1:
        dx1 += +1;
        break;
    case 2:
        dy1 += -1;
        break;
    case 3:
        dy1 += +1;
        break;
}

/* Check for multiply connected cluster */
if (abs(dx1-dx2) != 0  || abs(dy1-dy2) != 0 ) {


    /* Printing the percolated configuration to a file */
    /*
    FILE *f = fopen("sample.txt", "w");
    if (f == NULL) {
        printf("Error opening file!\n");
        exit(1);
    }

    for (int idx = 0; idx < N; idx++) {

        if (idx % L == 0 && idx) {
            fprintf(f, "\n");
        }

        findroot(idx);
        if (ptr[idx] == EMPTY) {
            // Site is empty
            fprintf(f, "0 ");
        }
        else if (ptr[idx] == r1 || idx == r1) {
            // Belongs to the percolating cluster
            fprintf(f, "1 ");
        }
        else {
            // Belongs to another cluster
```

```
                                fprintf(f, "2 ");
                            }
                        }

                        fclose(f);
                        */

                        /* Return the number of sites occupied*/
                        return (i+1);
                    }
                }
            }
        }
    }
    /* Should never happen since percolation always
    occurs before all the sites are occupied */
    return N;
}


int main()
{
    /* Pseudo-random number generator*/
    MTRand r = seedRand(42);

    /* Percolation probability as a function of n */
    int RL[N];
    for (int i = 0; i< N; i++) {
        RL[i] = 0;
    }

    /* Initialize the nearest neighbors */
    boundaries();

    /* Number of simulations to run */
    int NUM_SIM = 1;

    for (int j = 0; j < NUM_SIM; j++) {

        /* Permute the occupation order */
        permutation(&r);

        /* Number of occupied sites when percolation occurs */
        int nc = percolate();
```

```
        /* Update the result */
        for (int i = nc-1; i < N; i++)
                RL[i] += 1;
    }

    /* Compute the average and print result to the output */
    for (int i = 0; i < N; i++) {
        printf("%.9g %i\n", (RL[i])/(1.0*NUM_SIM), i+1);
    }

}
```

B The plotting script in Python

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from timeit import timeit


# Read wrapping probabilities from file and compute the
# convolution with the binomial distribution
# Returns R_L^h(p), R_L^e(p), R_L^b(p), R_L^1(p)
def conv_from_file(filename, p_values):

    with open( filename, 'r') as f:
        lines = f.readlines()
        yh = np.array([float(line.split()[1]) for line in lines])
        yb = np.array([float(line.split()[2]) for line in lines])

    N = len(yh)

    # This is the clear bottleneck of this computation
    # (optimize using cython??)
    RLh = np.array([ np.sum(binomial(N, p)*yh) for p in p_values])
    RLb = np.array([ np.sum(binomial(N, p)*yb) for p in p_values])

    return RLh, 2*RLh-RLb, RLb, RLh-RLb


# Computes the binomial distribution using the recursive definition
def binomial(N, p):

    nmax = np.floor(N*p).astype(int)
    binom = np.zeros(N+1)
    binom[nmax+1] = 1.0

    for i in range(nmax,-1, -1):
        binom[i] = binom[i+1]*(i+1)*(1-p)/(N-i)/p

    for i in range(nmax, N+1):
        binom[i] = binom[i-1]*(N-i+1)*p/i/(1-p)

    binom = binom/np.sum(binom)
    return binom[1:]
```

```
# Find the index nearest of the array item that is nearest to the value
def find_nearest(array, value):
    array = np.asarray(array)
    idx = (np.abs(array - value)).argmin()
    return idx


# Find the index of maximum in the array
def find_max(array):
    array = np.asarray(array)
    idx = (np.abs(array)).argmax()
    return idx


def main():

    # General plot

    # 'Exact' values of the percolation threshold and
    # the wrapping probabilities
    pc = 0.592746
    Rinf = [0.521058290, 0.690473725,  0.351642855, 0.169415435]

    pmin = pc-0.06
    pmax = pc+0.06
    pn = 301
    pp =  np.linspace(pmin,pmax,pn)

    # Output of the Newman-Ziff algorithm
    fnames1 = [ "perc32_1e9.dat","perc64_2e8.dat",
                "perc128_5e7.dat", "perc256_1e7.dat"]

    fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2,  figsize=(10, 10))


    # Compute the wrapping probability
    # with respect to the occupation probability
    for idx, fname in enumerate(fnames1):

        RLh, RLe, RLb, RL1 = conv_from_file(fname, pp)

        ax1.plot(pp, RLh)
        ax2.plot(pp, RLe)
        ax3.plot(pp, RLb)
        ax4.plot(pp, RL1)
```

```
# Draw the lines for R_L = R_inf(p_c) and p = p_c
axes = [ax1, ax2, ax3, ax4]
for R, ax in zip(Rinf, axes):
    ax.axhline(y = R, color = '0.6', linestyle = '--')
    ax.axvline(x = pc, color = '0.6', linestyle = '--')

fig.text(0.5, 0.04, r'Occupation probability $p$',
                    ha='center')
fig.text(0.04, 0.5, r'Wrapping probability $R_L(p)$',
                    va='center', rotation='vertical')
fig.savefig("plots/plot1.png")


# Finite-size scaling plot
pmin = pc-0.00002
pmax = pc+0.00004
pn = 301
pp =  np.linspace(pmin,pmax,pn)

# Output of the Newman-Ziff algo
fnames2 = [ "perc32_1e9.dat","perc64_2e8.dat",
            "perc128_5e7.dat", "perc256_1e7.dat"]
pcrit = []

# Compute the wrapping probabilities
for fname in fnames2:

    RLh, RLe, RLb, RL1 = conv_from_file(fname, pp)

    # Find the p_c estimates
    pcrit.append([  pp[find_nearest(RLh, Rinf[0])],
                    pp[find_nearest(RLe, Rinf[1])],
                    pp[find_nearest(RLb, Rinf[2])],
                    pp[find_max(RL1)]])


pcrit = np.array(pcrit).T
Lscale = (np.array([32, 64, 128, 256]))**(-11/4)
colors = ['C0', 'C1', 'C2', 'C3']

fig, ax = plt.subplots(figsize=(8, 6))

# Fit a line to the p_c estimates scaling as L^{-11/4}
for plist, c in zip(pcrit, colors):
```

```
        coef, V = np.polyfit(Lscale, plist, 1, cov=True)

        print("Intercept: {} +/- {}".format(coef[1], np.sqrt(V[1][1])))

        poly = np.poly1d(coef)
        ax.plot(Lscale, poly(Lscale), c, linewidth=1)
        ax.plot(Lscale, plist, '+' + c, markersize=12)

    ax.axvline(x = 0, color = '0.6', linestyle = '--', linewidth=1)
    ax.set_ylabel(r'$p_c$')
    ax.set_xlabel(r'$L^{-11/4}$')
    fig.savefig("plots/plot2.png")
    plt.show()

print(timeit(main, number=1))
```