Aalto University
School of Science

Matias Haapalehto

Two-dimensional site percolation with periodic boundary conditions

Course project report

PHYS-E0412 — Computational Physics

Report submitted : January 30, 2020

1 Background

”A fast Monte Carlo algorithm for site or bond percolation” [1] - statement of the site percolation problem: every site is either occupied with probability p or not with probability $1-p$. - occupied sites form clusters - phase transition at a specific value of p, which is characterized by a cluster spanning the entire system from side to side - this phenomenon is called percolation - this model originates from studies in percolation in materials such as rock or concrete - other applications outside physics: modeling resistor networks, or forest fires - no exact results are known for the square lattice

”Introduction to Computer Simulation Methods” [2] - in the limit of a infinite lattice, there is a finite threshold probability p_c such that: - 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 probability greater than zero, and less than unity.

2 Theory and methodology

2.1 Algorithm

”Efficient Monte Carlo algorithm and high-precision results for percolation” [3] - start with a lattice in which no sites are occupied - as we populate the lattice in a random order, clusters are given a label identifying it - 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 changed to reflect the bond has amalgamated the two clusters. - the clusters are stored in an array resembling 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 - by following these pointers we can find the root starting from any site - clusters can be amalgamated by a adding a pointer from the root of one to the root of the other - the algorithm consists of adding repeatedly a random bond to the lattice, identifying the clusters the two corresponding sites belong to and, if necessary amalgamating the two clusters. - weighted union–find with path compression: weighted: the two trees are amalgamated by making the smaller a sub-tree of the other path compression: the pointers of all traversed nodes are changed to point to the root node - 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, we must do additional work. - we add variables to each node which store the displacement to the parent node. - the total displacement to the root node is computed by traversing the tree and adding the displacements together - when we add an internal bond to the lattice, we calculate such displacement and take their difference

”Continuum Percolation Thresholds in Two Dimensions” - 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

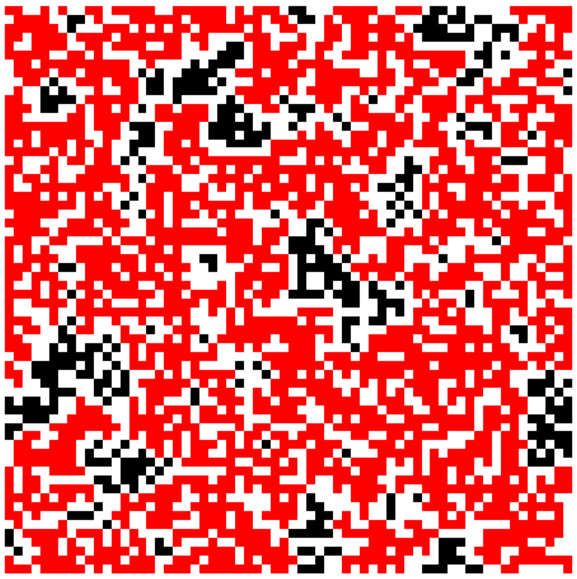


Figure 1: Sample lattice where the spanning cluster is marked in red.

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

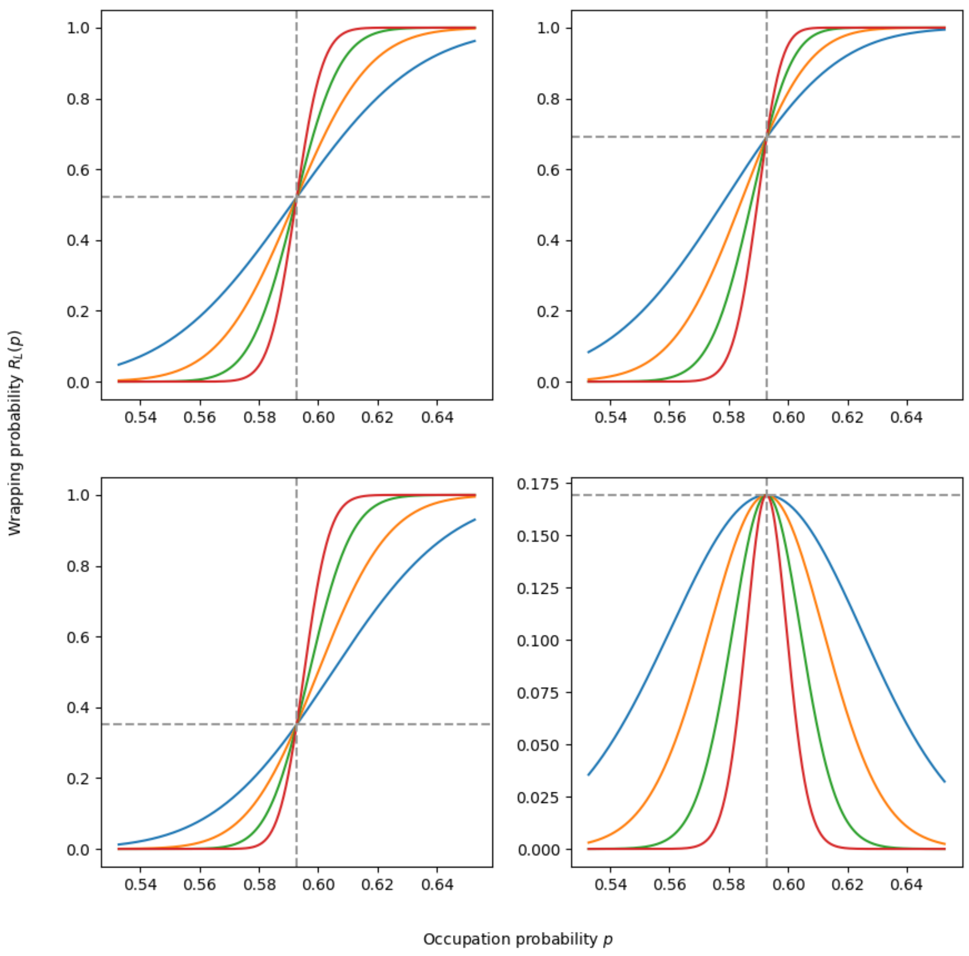
2.2 Implementation

- C - mersenne twister - plotting with python - binomial calculation - tested by computing L=2

2.3 Methodology

3 Results from simulations

4 Conclusions and thoughts on further studies



Wrapping probability $R_L(p)$
Occupation probability p
0.54
0.56
0.58
0.60
0.62
0.64
0.0
0.2
0.4
0.6
0.8
1.0
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175

Figure 2

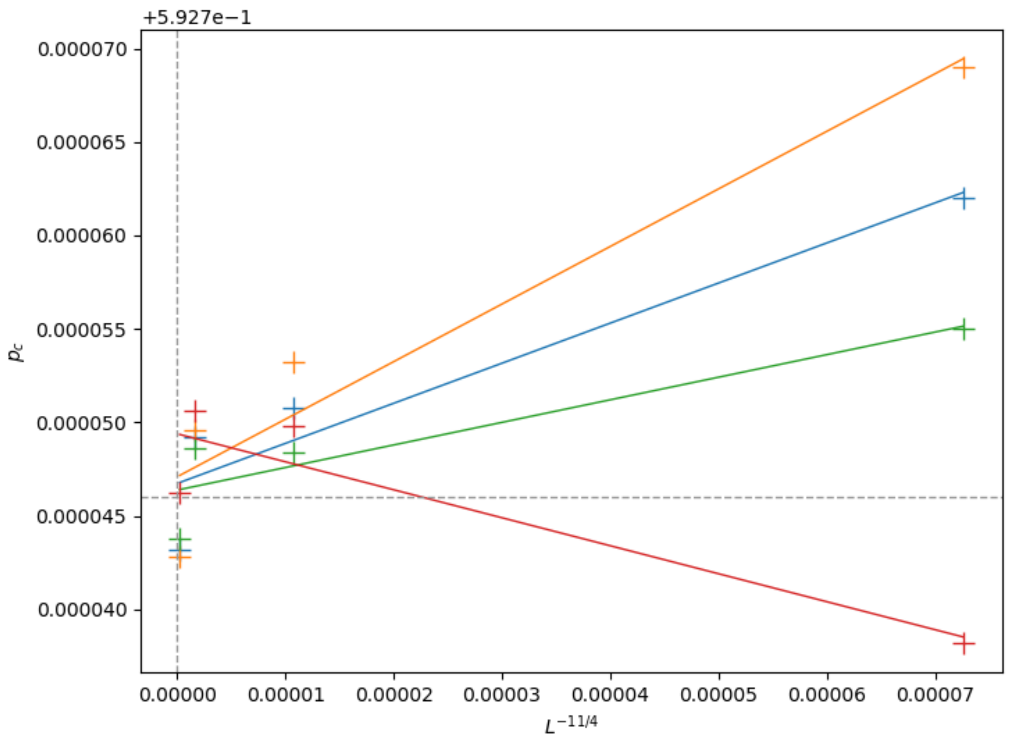

+5.927e−1
0.000070
0.000065
0.000060
0.000055
0.000050
0.000045
0.000040
p_c
0.00000
0.00001
0.00002
0.00003
0.00004
0.00005
0.00006
0.00007
$L^{-11/4}$

Figure 3

References

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

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

[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.

A The Newman-Ziff algorithm implemented in the C language

```c
#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);
    }

}
```