

# **Percolation by a class of ballistic deposition and their universality classes**



**Muhammad Shahnoor Rahman**

Department of Physics  
University of Dhaka

This dissertation is submitted for the degree of  
*Masters of Physics*

December 2018



I would like to dedicate this thesis to my loving parents ...



## **Declaration**

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Muhammad Shahnoor Rahman

December 2018



## **Acknowledgements**

And I would like to acknowledge ...



## **Abstract**

This is where you write your abstract ...



# Table of contents

<b>List of figures</b>	<b>xv</b>
<b>List of tables</b>	<b>xvii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation and Objective . . . . .	1
1.2 Method of Study . . . . .	1
1.3 Organization of Chapters . . . . .	1
<b>2 Scaling, Scale-Invariance and Self-Similarity</b>	<b>3</b>
2.1 Dimensions of Physical Quantity . . . . .	3
2.2 Buckingham $\pi$ Theorem . . . . .	4
2.2.1 An Example . . . . .	7
2.3 Similarity and Self-Similarity . . . . .	8
2.3.1 An example . . . . .	9
2.3.2 Diving Into Similarity . . . . .	10
2.3.3 Self-Similarity . . . . .	11
2.4 Scaling Hypothesis . . . . .	14
2.4.1 Dynamic Scaling . . . . .	14
2.4.2 Finite Size Scaling . . . . .	15
2.5 Homogeneous Functions and Scale-Invariance . . . . .	17
2.5.1 Generalized Homogeneous Function . . . . .	18
<b>3 Phase Transition</b>	<b>21</b>
3.1 Classification . . . . .	23
3.1.1 First Order . . . . .	23
3.1.2 Second Order . . . . .	23
3.2 Thermodynamic Quantities . . . . .	23
3.2.1 Entropy . . . . .	23

---

3.2.2	latent heat . . . . .	28
3.2.3	Specific Heat . . . . .	28
3.2.4	Order Parameter . . . . .	29
3.2.5	Susceptibility . . . . .	29
3.3	Shapes of the Thermodynamic Quantities . . . . .	30
3.3.1	Calculus to determine the shapes . . . . .	30
3.3.2	Free Energy . . . . .	30
3.3.3	Entropy and Specific Heat . . . . .	31
3.3.4	Order Parameter and Susceptibility . . . . .	32
3.4	Response Functions . . . . .	34
3.5	Critical Exponents . . . . .	34
3.5.1	List of Thermodynamic Quantities that Follows Power Law . . . . .	39
3.5.2	Rushbrooke Inequality . . . . .	39
3.5.3	Griffiths Inequality . . . . .	39
3.6	Ising Model . . . . .	40
<b>4</b>	<b>Percolation Theory</b>	<b>43</b>
4.1	Percolation Phenomena . . . . .	44
4.2	Historical Overview . . . . .	45
4.3	Classifications and Playground . . . . .	46
4.3.1	Types of Percolation . . . . .	46
4.3.2	Types of Playground . . . . .	48
4.4	Basic Elements . . . . .	50
4.4.1	Occupation Probability . . . . .	50
4.4.2	Cluster . . . . .	51
4.4.3	Spanning cluster . . . . .	51
4.5	Observable Quantities . . . . .	51
4.5.1	Percolation Threshold, $p_c$ . . . . .	52
4.5.2	Spanning Probability, $w(p,L)$ . . . . .	53
4.5.3	Entropy, $H(p,L)$ . . . . .	53
4.5.4	Specific Heat, $C(p,L)$ . . . . .	56
4.5.5	Order Parameter, $P(p,L)$ . . . . .	57
4.5.6	Susceptibility, $\chi(p,L)$ . . . . .	58
4.5.7	Mean Cluster Size, $S$ . . . . .	58
4.5.8	Cluster Size Distribution Function, $n_s$ . . . . .	58
4.5.9	Correlation Function, $g(r)$ . . . . .	59
4.5.10	Correlation length, $\xi$ . . . . .	59

4.5.11	Fractal Dimension, $d_f$	59
4.6	Exact Solutions	59
4.6.1	One Dimension	59
4.6.2	Infinite Dimension	63
4.7	Algorithm	69
4.8	Relation of Phase Transition with Percolation	69
4.9	Application	69
4.9.1	Epidemiology	69
4.9.2	Neural Network and Cognitive Psychology	70
4.9.3	Ferromagnetism	71
4.9.4	Cosmology	72
4.9.5	Square Lattice	73
4.9.6	Site Percolation	73
4.9.7	Bond Percolation	75
<b>5</b>	<b>Ballistic Deposition on Square Lattice</b>	<b>77</b>
5.1	Site Percolation Redefined	78
5.2	Structure and Algorithm	78
5.3	Finite Size Scaling in Percolation Theory	79
5.4	Finding Numerical Values	79
5.4.1	Critical occupation probability, $p_c$	79
5.4.2	Spanning Probability and finding $1/v$	80
5.4.3	Entropy, Specific Heat and finding $\alpha$	81
5.4.4	Order Parameter and finding $\beta$	82
5.4.5	Susceptibility and finding $\gamma$	84
5.4.6	Cluster Size Distribution	85
5.4.7	Cluster Rank Size Distribution	86
5.4.8	Order-Disorder Transition	86
5.4.9	Fractal Dimension	87
<b>6</b>	<b>Summary and Discussion</b>	<b>89</b>
<b>References</b>		<b>91</b>
<b>Appendix A</b>	<b>Percolation</b>	<b>99</b>
A.1	Algorithm	99
A.2	Code	99

A.2.1	Index . . . . .	99
A.2.2	Site . . . . .	104
A.2.3	Bond . . . . .	107
A.2.4	Lattice . . . . .	112
A.2.5	Exception . . . . .	133
A.2.6	Cluster . . . . .	135
A.2.7	Percolation . . . . .	140
A.2.8	Utilities . . . . .	192
A.2.9	Test . . . . .	196
A.2.10	Main . . . . .	200
A.2.11	CMakeLists . . . . .	202
A.2.12	complete code . . . . .	203
<b>A</b>	<b>Appendix B Convolution</b>	<b>205</b>
B.1	Algorithm . . . . .	205
B.2	Code . . . . .	205
<b>C</b>	<b>Appendix C Finding Exponents</b>	<b>207</b>
C.1	Algorithm . . . . .	207
C.1.1	Specific Heat and Susceptibility . . . . .	207

# List of figures

2.1	Application of Buckingham $\Pi$ theorem in a right triangle . . . . .	9
2.2	Self-Similarity in Cauliflower . . . . .	12
2.3	Self-Similarity examples . . . . .	13
3.1	Two competing Carnot engines, pressure (P) versus volume (V) - diagrams [10]	25
3.2	shape of $f(x)$ from $f'(x)$ . . . . .	30
3.3	shape of $f(x)$ from $f''(x)$ . . . . .	31
3.4	Shape of the free energy . . . . .	31
3.5	Shape of entropy . . . . .	32
3.6	Shape of Specific Heat . . . . .	33
3.7	figure required . . . . .	35
3.8	Function showing power law . . . . .	36
4.1	percolation phenomena in square structure. No current if $p < p_c$ . . . . .	45
4.2	Different types of cubic lattice [2] . . . . .	49
4.3	Different types of cubic lattice . . . . .	49
4.4	Scale Free Network [5] . . . . .	51
4.5	A square lattice of length $L = 10$ . Demonstrating the appearance of the wrapping cluster . . . . .	52
4.6	A system of 12 cluster . . . . .	55
4.7	entropy of a system of 8 particles . . . . .	57
4.8	One Dimensional Lattice. Empty ones are white and filled ones are black. .	59
4.9	Bethe Lattice for $z = 3$ . . . . .	65
4.10	NGC 4414, a typical spiral galaxy in the constellation Coma Berenices, is about 55,000 light-years in diameter and approximately 60 million light-years away from Earth. . . . .	72
4.11	Square Lattie (empty) of length 6 . . . . .	74
4.12	Site and Bond symbol (empty and occupied). . . . .	74

4.13	Growth of a Cluster in Site Percolation on square Lattice . . . . .	75
4.14	Growth of a Cluster in Bond Percolation on square Lattice . . . . .	76
5.1	Spanning Probability, $w(p,L)$ vs Occupation Probability, $p$ . . . . .	80
5.2	$w(p,L)$ vs $(p - p_c)L^{1/\nu}$ . . . . .	81
5.3	Entropy, $H(p,L)$ vs Occupation Probability, $p$ . . . . .	81
5.4	Specific Heat, $C(p,L)$ vs Occupation Probability, $p$ . . . . .	82
5.5	$C(p,L)$ vs $(p - p_c)L^{1/\nu}$ . . . . .	82
5.6	$CL^{-\alpha/\nu}$ vs $(p - p_c)L^{1/\nu}$ . . . . .	83
5.7	Order Parameter, $P(p,L)$ vs Occupation Probability, $p$ . . . . .	83
5.8	Order Parameter, $P(p,L)$ vs Occupation Probability, $p$ . . . . .	84
5.9	$P(p,L)$ vs $(p - p_c)L^{1/\nu}$ . . . . .	84
5.10	$PL^{\beta/\nu}$ vs $(p - p_c)L^{1/\nu}$ . . . . .	84
5.11	Susceptibility, $\chi(p,L)$ vs Occupation Probability, $p$ . . . . .	85
5.12	$\chi(p,L)$ vs $(p - p_c)L^{1/\nu}$ . . . . .	85
5.13	$\chi L^{\gamma/\nu}$ vs $(p - p_c)L^{1/\nu}$ . . . . .	86
5.14	Number of cluster of size $s$ , $n_s$ vs size of the cluster $s$ . . . . .	86
5.15	$\log(n_s)$ vs $\log(s)$ . . . . .	86
5.16	$H(p,L)/H(0,L)$ or $P(p,L)/P(1,L)$ vs $p$ . . . . .	87
5.17	$\log(S)$ vs $\log(L)$ . . . . .	88
C.1	Best collapsing on either right or left . . . . .	209
C.2	Minimizing exponent for scaling $x$ -values . . . . .	210

# List of tables

4.1	Percolation Threshold for Some Regular Lattices . . . . .	53
6.1	List of combined exponents . . . . .	90
6.2	List of exponents . . . . .	90



# **Chapter 1**

## **Introduction**

Percolation is one of the most studied problems in statistical physics. Its idea was first conceived by Paul Flory in the early 1940s in the context of gelation in polymers. Later, in 1957 it acquires the mathematical formulation due to the work of engineer Simon Broadbent and mathematician John Hammersley. Ever since then percolation theory has been studied extensively by scientists in general and physicists in particular.

### **1.1 Motivation and Objective**

### **1.2 Method of Study**

### **1.3 Organization of Chapters**



# **Chapter 2**

## **Scaling, Scale-Invariance and Self-Similarity**

In physics we observe a natural phenomena and try to understand it using the existing knowledge. To do this we need to assign numbers to the observable quantities. If we can express it in numbers only then we can say we have acquired some knowledge about that quantity. And if we cannot do this then our knowledge is inadequate about that quantity. This reveals the fact that physics is all about observation and measurement of physical quantities with the desire to acquire some knowledge about it and then use that knowledge to predict something that is yet to observe. For example, Albert Einstein predicted the existence of gravitational waves in 1916 in his general theory of relativity and the first direct observation of gravitational waves was made on 14 September 2015 and was announced by the LIGO and Virgo collaborations on 11 February 2016. Now, in order to understand the observation of a natural or artificial phenomena we need some tools. Since here we are trying to understand phase transition, finite size scaling (FSS) hypothesis is a great tool for the investigation. In this chapter we will try to understand the fundamentals which is based on Buckingham  $\pi$  theorem, self similarity and homogeneous functions.

### **2.1 Dimensions of Physical Quantity**

In order to express physical quantities in terms of numbers we need a unit of measurement, since a number times unit tells us how much the quantity is larger or smaller with respect to the unit. The units of measurement are described as fundamental and derivative ones. The fundamental units of measurement are defined arbitrarily in the form of certain standards, while the derivative ones are obtained from the fundamental units of measurement by virtue

of the definition of physical quantities, which are always indications of conceptual method of measuring them. An example involving fundamental and derivative unit of measurement is velocity. Since velocity is measured by how much distance an object travels per unit time, the unit of velocity is the ratio of distance or length over time. It is expressed as  $[v] = LT^{-1}$ , where  $L$  is the unit of length and  $T$  is the unit of time. The unit of length and time are fundamental here and the unit of velocity is the derivative of these two. A system of units of measurement is a set of fundamental units of measurement sufficient to measure the properties of the class of phenomena under consideration. For example the *CGS* system where length is measurement in terms of centimeter, mass is measured in terms of gram and the *SI* system where the mass is measured in terms of kilogram, length is measured in terms of meter and in both system time is measured in terms of second.

Dimension of physical quantity determines by what amount the numerical value must be changed if we want to go to another system of units of measurement. For instance, if the unit of length is decreased by a factor  $L$  and the unit of time is decreased by a factor  $T$ , the unit of velocity is smaller by the factor of  $LT^{-1}$  than the original unit, so the numerical value of velocity would scaled up by a factor of  $LT^{-1}$  owing to the definition of equivalence.

The changes in the numerical values of physical quantities upon passage from one systems of units of measurement to another within the same class are determined by their dimensions. The functions that determines the factor by which the numerical value of a physical quantity changes upon transition from systems of unit of measurement to another system within a given class is called the dimension function or, the dimension of that physical quantity. We emphasize that the dimension of a given physical quantity is different in different classes of systems of units. For example, the dimension of density  $I$  in the *MLT* class is  $[\rho] = ML^{-3}$  whereas in the *FLT* class it is  $[\rho] = FL^{-4}T^{-2}$  ??[3].

## 2.2 Buckingham $\pi$ Theorem

A part of physics is about modeling physical phenomenon. And while doing it, the first thing is to identify the relevant variables, and then relate them using known physical laws. For simple phenomenon this task is not hard since it involves deriving some quantitative relationship among the physical variables from the first principles. But when dealing with complex systems we need a systematic way of dealing with the problem of reducing number of parameters. In these situations constructing a model in a systematic manners with minimum input parameters that can help analyzing experimental results has been a useful method. One of the simplest way is based on dimensional analysis. Its function is to reduce a large number of parameters into a manageable set of parameters. Buckingham  $\pi$  theorem is one of

the most suitable and studied mathematical process to deal with this kind of problems.

Buckingham  $\pi$  theorem describes dimensionless variables obtained from the power products of governing parameters denoted by  $\Pi_1, \Pi_2, \dots$  etc. When investigating a certain dimensional physical quantity (governed) that depend on other  $n$  dimensional variables then this theorem provide us a systematic way to reduce the degrees of freedom of a function. Using this theorem we reduce a function of  $n$  variables problems into a function of  $k$  dimensionless variable problem if each of the  $k$  dimensional variable of original  $n$  variables can be expressed in terms of the  $n - k$  dimensionally independent variable.

The relationship found in physical theories or experiments can always be represented in the form

$$a = f(a_1, a_2, \dots, a_n) \quad (2.1)$$

where the quantities  $a_1, a_2, \dots, a_n$  are called the governing parameters. It is always possible to classify the governing parameters  $a_i$ 's into two groups using the definition of the dependent and independent variables. Let the arguments  $a_{k+1}, \dots, a_n$  have the independent dimensions and the dimensions of the arguments  $a_1, a_2, \dots, a_k$  can be expressed in terms of the dimensions of the governing independent parameters  $a_{k+1}, \dots, a_n$  in the following way

$$\begin{aligned} [a_1] &= [a_{k+1}]^{\alpha_1} \dots [a_n]^{\gamma_1} \\ [a_2] &= [a_{k+1}]^{\alpha_2} \dots [a_n]^{\gamma_2} \end{aligned} \quad (2.2)$$

$$\vdots \quad (2.3)$$

$$[a_k] = [a_{k+1}]^{\alpha_k} \dots [a_n]^{\gamma_k}$$

The dimension of the governed parameter  $a$  must also be expressible in terms of the dimensionally independent governing parameters  $a_1, \dots, a_k$  since  $a$  does not have independent dimension and hence we can write

$$[a] = [a_{k+1}]^{\alpha} \dots [a_n]^{\gamma} \quad (2.4)$$

Thus, there exist number  $\alpha, \gamma$  such that 2.4 holds. We have set of governing parameters.

$$\Pi_1 = \frac{a_1}{[a_{k+1}]^{\alpha_1} \dots [a_n]^{\gamma_1}} \quad (2.5)$$

$$\Pi_2 = \frac{a_2}{[a_{k+1}]^{\alpha_2} \dots [a_n]^{\gamma_2}} \quad (2.6)$$

$$\vdots \quad (2.7)$$

$$\Pi_k = \frac{a_k}{[a_{k+1}]^{\alpha_k} \dots [a_n]^{\gamma_k}} \quad (2.8)$$

and a dimensionless governed parameter

$$\Pi = \frac{f(\Pi_1, \dots, \Pi_k, a_{k+1}, \dots, a_n)}{[a_{k+1}]^\alpha \dots [a_n]^\gamma} \quad (2.9)$$

The right hand side of equation 2.9 clearly reveals that the dimensionless quantity  $\Pi$  is a function of  $a_{k+1}, \dots, a_n, \Pi_1, \dots, \Pi_k$ , i.e.,

$$\Pi \equiv F(a_{k+1}, \dots, a_n, \Pi_1, \dots, \Pi_k) \quad (2.10)$$

The quantities  $\Pi, \Pi_1, \dots, \Pi_k$  are obviously dimensionless, and hence upon transition from one system of unit to another inside a given class their numerical values must remain unchanged. At the same time, according to the above, one can pass to a system of units of measurement such that any of the parameters of  $a_{k+1}, \dots, a_n$ , say for example,  $a_{k+1}$ , is changed by an arbitrary factor, and the remaining ones are unchanged. Upon such transition the first argument of  $F$  is unchanged arbitrarily, and all other arguments of the function remain unchanged as well as its value  $\Pi$ . Hence, it follows  $\frac{\delta F}{\delta a_{k+1}} = 0$  and entirely analogously  $\frac{\delta F}{\delta a_{k+2}} = 0, \dots, \frac{\delta F}{\delta a_n} = 0$ . Therefore, the relation 2.10 is in fact represented by a function of  $k$  arguments and proves it is independent of  $a_{k+1}, \dots, a_n$ , that is,

$$\Pi = \Phi(\Pi_1, \dots, \Pi_k) \quad (2.11)$$

and the function  $f$  can be written in the following special form

$$f(a_1, \dots, a_k, \dots, a_n) = a_{k+1}^\alpha \dots a_n^\gamma \Phi(\Pi_1, \dots, \Pi_k) \quad (2.12)$$

equation 2.12 is known as the Buckingham  $\Pi$  theorem. It constitutes one of the central statements in dimensional analysis and has great bearings on scaling theory.

### 2.2.1 An Example

An explicit example using this theorem is needed for better understanding it's application. The simplest example that can describe basic features of Buckingham  $\Pi$  theorem is the area of a right triangle and Pythagorean theorem.

Consider a right triangle where three sides are of size  $a, b$  and  $c$  and for definiteness, the smaller of its acute angles  $\theta$ . Assume that we are to measure the area  $S$  of the triangle. The area  $S$  can be written in the following form

$$S = S(a, b, c) \quad (2.13)$$

However, the definition of two governing parameters  $a$  and  $b$  can be expressed in terms of  $c$  alone since we have

$$[a] \sim [c] \text{ and } [b] \sim [c] \quad (2.14)$$

and so is true for the governed parameter  $S$  as we can write the dimensional relation  $[S] \sim [c^2]$ . We therefore can define two dimensionless parameters

$$\Pi_1 = \sin \theta = a/c \quad (2.15)$$

$$\Pi_2 = \cos \theta = b/c \quad (2.16)$$

and the dimensionless governed parameter

$$\Pi = \frac{S}{c^2} = c^{-2}S(c\Pi_1, c\Pi_2, c) \equiv F(c, \Pi_1, \Pi_2) \quad (2.17)$$

Now it is possible to pass from one unit of measurement to another system of unit of measurement within the same class and upon such transition the arguments  $\Pi_1, \Pi_2$  of the function  $F$  and the function itself remain unchanged. It implies that the function  $F$  is independent of  $c$  and hence we can write

$$\Pi = \phi(\Pi_1, \Pi_2) \quad (2.18)$$

However,  $\Pi_1$  and  $\Pi_2$  both depends on the dimensionless quantity  $\theta$  and hence we can write

$$S = c^2\phi(\theta) \quad (2.19)$$

where the scaling function  $\phi(\theta)$  is universal in character. In order to further capture the significance of equation 2.19 we rewrite it as

$$\frac{S}{c^2} \sim \phi(\theta) \quad (2.20)$$

This result has far reaching consequences. For instance, consider that we have a right triangle of any arbitrary sides  $a' \neq a$ ,  $b' \neq b$  and  $c' \neq c$  but have the same acute angle  $\theta$  as before. This can be ensured by choosing an arbitrary point on the hypotenuse of the previous triangle and drop a perpendicular on the base  $b$ . Consider that the area of the new triangle is  $S'$  yet we will have

$$\frac{S}{c^2} = \frac{S'}{(c')^2} \quad (2.21)$$

since the numerical value of the ratio of the area over the square of the hypotenuse depends on the angle  $\theta$ . It implies that if we plot the ratio of the area over the square of the hypotenuse as a function  $\theta$  all the data points should collapse onto a single curve regardless of the size of the hypotenuse and the respective areas of the right triangle. In fact, the details calculation reveals that

$$\phi(b/c) = \frac{1}{2} \sin \theta \quad (2.22)$$

This data collapse implies that if two or more right triangles which have one of the acute angle identical then such triangles are similar. We shall see later that whenever we will find data collapse between two different systems of the same phenomenon then it would mean that the corresponding systems or their underlying mechanisms are similar. Similarly, if we find that data collected from the whole system collapsed with similarly collected data from a suitably chosen part of the whole system then we can conclude that part is similar to the whole, implying self-similarity. On the other hand, if we have a set of data collected at many different times for a kinetic system and find they all collapse onto a single curve then we can say that the same system at different times are similar. However, similarity in this case is found of the same system at different times and hence we may coin it as temporal self-similarity.

## 2.3 Similarity and Self-Similarity

The concept of physical similarity is a natural generalization of the concept of similarity in geometry. For instance, two triangles are similar if they differ only in the numerical values of the dimensional parameters, i.e. the lengths of the sides, while the dimensionless parameters, the angles at the vertices are identical for the two triangles. Analogously,

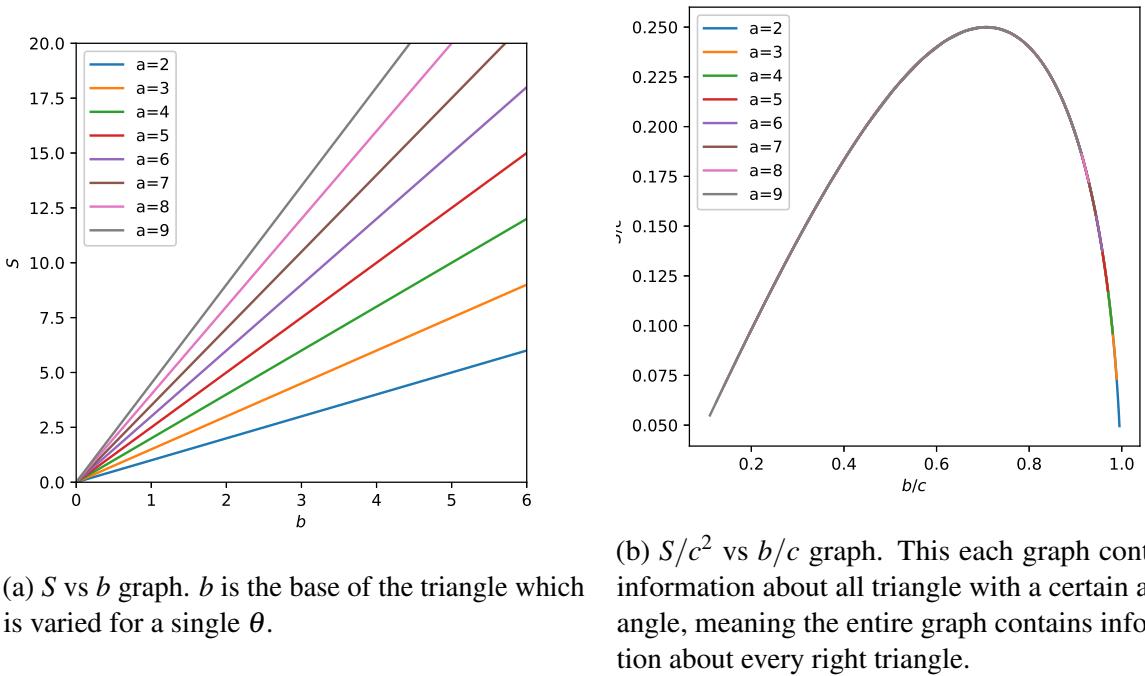


Fig. 2.1 Application of Buckingham  $\Pi$  theorem in a right triangle

physical phenomena are called similar if they differ only in their numerical values of the dimensional governing parameters; the values of the corresponding dimensionless parameters  $\Pi_1, \dots, \Pi_k$  are being identical. In connection with this definition of similar phenomena, the dimensionless quantities are called similarity parameters. The term *self-similarity* is, as the term itself suggests, a structure or process and a part of it appear to be the same when compared. This also means that a self-similar structure is infinite in theory. Therefore the fundamental principle of a self-similar structure is the repetition of a unit pattern on different scales.

### 2.3.1 An example

At this point a real world example will be helpful. Suppose we need to build an airplane. Obviously it's a billion dollar project. If we try to build the airplane directly without first building a prototype then we will wasting time, money and man power. Because we can not build an actual working airplane without following certain steps. First we need to build a small-scale model of the airplane. Then we need to test the model for performance. If it is satisfying then we can start building a prototype using the knowledge obtained from the

experimentation on the model. Here the model and the prototype are similar to each other. That's why this method works. Since only the dimensional parameters are scaled.

### 2.3.2 Diving Into Similarity

Let us consider two similar phenomena, one of which will be called the prototype and the other the model. For both phenomena there is some relation of the form

$$\alpha = f(a_1, a_2, a_3, b_1, b_2) \quad (2.23)$$

where the function  $f$  is the same for both cases by the definition of similar phenomena, but the numerical values of the governing parameters  $a_1, a_2, a_3, b_1, b_2$  are different. Thus for prototype we have

$$\alpha_p = f(a_1^{(p)}, a_2^{(p)}, a_3^{(p)}, b_1^{(p)}, b_2^{(p)}) \quad (2.24)$$

and for model we have

$$\alpha_m = f(a_1^{(m)}, a_2^{(m)}, a_3^{(m)}, b_1^{(m)}, b_2^{(m)}) \quad (2.25)$$

where the index  $p$  denotes quantities related to the prototype and the index  $m$  denotes quantities related to the model. Consider that  $b_1$  and  $b_2$  are dependent variable and thus they are expressed in terms of  $a_1, a_2, a_3$  in both model and prototype systems. Using dimensional analysis we find for both phenomena

$$\Pi^{(p)} = \Phi(\Pi_1^{(p)}, \Pi_2^{(p)}) \quad (2.26)$$

and

$$\Pi^{(m)} = \Phi(\Pi_1^{(m)}, \Pi_2^{(m)}) \quad (2.27)$$

where the function  $\Phi$  must be the same for the model and the prototype. By the definition of similar phenomena the dimensional quantities must be identical in both the cases such as in the prototype and in the model, i.e.,

$$\Pi_1^{(m)} = \Pi_1^{(p)} \quad (2.28)$$

$$\Pi_2^{(m)} = \Pi_2^{(p)} \quad (2.29)$$

It also follows that the governed dimensionless parameter satisfies

$$\Pi^{(m)} = \Pi^{(p)} \quad (2.30)$$

Returning to dimensional variables, we get from the above equation

$$a_p = a_m \left( \frac{a_1^{(p)}}{a_1^{(m)}} \right)^{q_1} \left( \frac{a_2^{(p)}}{a_2^{(m)}} \right)^{q_2} \left( \frac{a_3^{(p)}}{a_3^{(m)}} \right)^{q_3} \quad (2.31)$$

which is a simple rule for recalculating the results of measurements on the similar model for the prototype, for which direct measurement may be difficult to carry out for one reason or another.

The conditions for similarity of the model to the prototype-equality of the similarity parameters  $\Pi_1, \Pi_2$  for both phenomena show that it is necessary to choose the governing parameters  $b_1^{(m)}, b_2^{(m)}$  of the model as to guarantee the similarity of the model to the prototype

$$b_1^{(m)} = b_1^{(p)} \left( \frac{a_1^{(m)}}{a_1^{(p)}} \right)^{\alpha_1} \left( \frac{a_2^{(m)}}{a_2^{(p)}} \right)^{\beta_1} \left( \frac{a_3^{(m)}}{a_3^{(p)}} \right)^{\gamma_1} \quad (2.32)$$

and

$$b_2^{(m)} = b_2^{(p)} \left( \frac{a_1^{(m)}}{a_1^{(p)}} \right)^{\alpha_2} \left( \frac{a_2^{(m)}}{a_2^{(p)}} \right)^{\beta_2} \left( \frac{a_3^{(m)}}{a_3^{(p)}} \right)^{\gamma_2} \quad (2.33)$$

whereas the model parameters  $a_1^{(m)}, a_2^{(m)}, a_3^{(m)}$  can be chosen arbitrarily. The simple definitions and statements presented above describe the entire content of the theory of similarity.

### 2.3.3 Self-Similarity

A system is called self-similar When a small part of the system is similar to the whole system. A self similar structure is created by repetition of a unit pattern or a simple rule over different size scales. Although the term "self-similarity" itself is explanatory, some example gives better insight.

The cauliflower head contains branches or parts, which when removed and compared with the whole found to be very much the same except it is scaled down. These isolated branches can again be decomposed into smaller parts, which again look very similar to the whole as well as of the branches. Such self-similarity can easily be carried through for about three to four stages. After that the structures are too small to go for further dissection. Of course, from the mathematical point of view the property of self-similarity may be continued through an infinite stages though in real world such property sustain only a few stages 2.2.

There are plenty of other examples of self similarity in nature. Snowflakes exhibit self-similar branching patterns 2.3. The growth in aggregating colloidal particles are statistically self-similar. A leafless tree branches 2.3 in a self-similar fashion, each length splitting into



(a) A cauliflower



(b) Dissection of cauliflower

Fig. 2.2 Self-Similarity in Cauliflower



(a) Leaf



(b) Snowflake



(c) Blood vessel



(d) Leafless Tree

Fig. 2.3 Self-Similarity examples

two or more branches. This branch pattern is repeated on smaller and smaller length scales till the tree top is reached. The veins of the leaves also branch in a self similar manner 2.3. The decimal number system is a construct that uses the idea of self-similarity. If we look a meter stick, we shall see that a decimeter range with its marks looks like a meter range with its marks, only smaller by a factor of 10. This pattern of meter stick makes it very easy to note readings. The human brain is also a complex network of neurons which organize in self-similar patterns. From quantum particle paths, lightning bolts, blood vessels 2.3, aggregation of bacteria all are example of self-similarity.

## 2.4 Scaling Hypothesis

### 2.4.1 Dynamic Scaling

A function  $f(x, t)$  is said to obey dynamic scaling if one of the variable  $t$  strictly denotes time and if it satisfies

$$f(x, t) \sim t^\theta \phi(x/t^z) \quad (2.34)$$

where  $\theta$  and  $z$  are fixed by the dimensional relation  $[t^\theta] = [f]$  and  $[t^z] = [x]$  respectively, while  $\phi(\xi)$  is known as the scaling function. Sometimes it is also written in the following form

$$f(x, t) \sim x^\omega \phi(x/t^z) \quad (2.35)$$

Buckingham  $\pi$ -theorem can provide a systematic processing procedure to obtain the dynamic scaling form and at the same time appreciate the fact that the second form is not mathematically sound. An interesting aspect of the structure of the dynamic scaling form given by equation 2.34 is that the distribution function  $f(x, t)$  at various moments of time can be obtained from one another by a similarity transformation

$$x \rightarrow \lambda^z x \quad (2.36)$$

$$t \rightarrow \lambda t \quad (2.37)$$

$$f \rightarrow \lambda^\theta \quad (2.38)$$

revealing the self-similar nature of the function  $f(x, t)$ .

To derive it one has to know first that one of the two governing parameters can be assumed to be independent. Let us assume that  $t$  is chosen to be an independent parameter and hence

$x$  can be expressed in terms of  $t$

$$x \sim t^z \quad (2.39)$$

It implies that we can choose  $t^z$  as unit of measurement or yard-stick and quantify  $x$  in terms of dimensionless quantity  $\xi = x/t^z$ . Here, the quantity  $\xi$  is a number that tells how many  $t^z$  we need to measure  $x$ . If  $t$  is independent quantity then we can also express  $f$  in units of  $t^\theta$  to obtain yet another dimensionless quantity  $\phi = f(x, t)/t^\theta$  where the exponent  $\theta$  is fixed by the dimensional requirement  $[f] = [t^\theta]$ . Since  $\phi$  is a dimensionless quantity its numerical value can only depend on dimensionless quantity  $\xi$  not on depend on a dimensional quantity  $t$ . We can then immediately obtain the scaling form given by Eq. 2.34. On the other hand, had we choose  $x$  to be independent parameter instead of  $t$  then following the same argument we would have the following scaling 2.35.

### 2.4.2 Finite Size Scaling

There exists another scaling hypothesis, known as the finite-size scaling (FSS), that has been extensively used as a very powerful tool for estimating finite size effects specially in the second order phase transition near the critical temperature  $T$ . The various response functions, typically the second derivative of the free energy  $F$ , in second order phase transition diverges. Such transitions are clasified by a set of critical exponents which characterize the critical point. The best known example of second order phase transition is the paramagnetic to ferromagnetic transition where

$$M \sim (T - T_c)^\beta \quad (2.40)$$

$$\chi_M \sim (T - T_c)^{-\gamma} \quad (2.41)$$

$$C_V \sim (T - T_c)^{-\alpha} \quad (2.42)$$

$$\xi \sim (T - T_c)^{-\nu} \quad (2.43)$$

where,  $M, \chi_M, C_V, \xi$  are Magnetization, Susceptibility, Heat capacity, Correlation length respectively. These relations are only true in the thermodynamic limit in the sense that the system size is infinite. However, we can work in simulation and experiment with finite size  $L^d$  where correlation length  $\xi \sim L$ . Finite size scaling thus provides a means of extrapolating various results for infinite systems.

According to finite size scaling (FSS) hypothesis, a function  $f(\epsilon, L)$  with  $\epsilon = T - T_c$  is said to obey finite size scaling if it can be expressed as

$$f(\epsilon, L) \sim L^{-\omega/nu} \phi(\epsilon L^{1/\nu}) \quad (2.44)$$

However, using the Buckingham  $\pi$ -theorem we not only obtain the correct scaling form but we also gain a deeper insight into the problem as it provides a systematic processing procedure. For instance, as we know that the correlation length  $\xi$  in the limit  $L \rightarrow \infty$  diverges like  $\xi \sim \varepsilon^\nu$  near the critical point and it bear the dimension of length. We therefore can either choose  $L$  as an independent parameter and measure  $\xi$ , i.e.,  $T - T_c$ , in unit of  $L$ . Consequently we can measure  $L$  in unit of  $T - T_c$  assuming it as an independent parameter. Choosing the later case we can define a dimensionless quantity

$$\pi = \frac{L}{\xi} = L(T - T_c)^\nu \quad (2.45)$$

and the corresponding dimensionless governing parameter is

$$\Pi = \frac{f(\varepsilon, L)}{\xi^\omega} = \phi(\pi) \quad (2.46)$$

Following the argument of the  $\pi$ -theorem we can immediately write that

$$f(\varepsilon, L) \sim (T - T_c)^{-\nu\omega} \phi(L(T - T_c)^\nu) \quad (2.47)$$

On the other hand had we chosen  $L$  as an independent parameter then the similar treatment would yield

$$f(\varepsilon, L) \sim L^\theta \phi(\{L(T - T_c)^\nu\}^{-1}) \quad (2.48)$$

Till to date neither of the two scaling forms obtained following  $\pi$  theorem are in use in their strict form. Instead, what is done traditionally are as follows. The important point is that if  $\pi = L/\xi$  is dimensionless then so is

$$\pi^{1/\nu} = (L/\xi^{1/\nu}) = (T - T_c)L^{1/\nu} \quad (2.49)$$

It also means that we can choose  $L$  as independent parameter and express  $(T - T_c)$  in unit of  $L^{-1/\nu}$  to make the dimensionless quantity coincide with  $\pi^{1/\nu}$ . Then  $f$  too can be expressed in unit of  $L^\theta$  which according to the prescription of  $\pi$ -theorem we have the following FSS scaling form

$$f(\varepsilon, L) \sim L^\theta \phi((T - T_c)L^{1/\nu}) \quad (2.50)$$

which is the same as the traditional scaling form given by 2.44 if we find  $\theta$  negative and it is related to the exponent  $\nu$  via  $\theta = -\omega/\nu$ .

A quantitative way of interpreting how the experimental data exhibits finite-size scaling is done by invoking the idea of data-collapse method - an idea that goes back to the original

observation of Rushbrooke. That is, the values of  $f(\varepsilon, L)$  for different system size  $L$  can be made to collapse on a single curve if  $fL^{\omega/\nu}$  is plotted against  $\varepsilon L^{1/\nu}$ . It implies that systems of different sizes are all similar that also include system where  $L \rightarrow \infty$ . The method of data-collapse therefore comes as a powerful means of establishing scaling. It is extensively used to analyze and extract exponents especially from numerical simulations. We shall elucidate it further in the upcoming chapters.

## 2.5 Homogeneous Functions and Scale-Invariance

A function is called scale-invariant or scale-free if it retains its form keeping all its characteristic features intact even if we change the measurement unit or scale. Mathematically, a function  $f(r)$  is called scale-invariant or scale-free if it satisfies

$$f(\lambda x) = g(\lambda) f(x) \quad \forall \lambda \quad (2.51)$$

where  $g(\lambda)$  is yet unspecified function. That is, one is interested in the shape of  $f(\lambda x)$  for some scale factor  $\lambda$  which can be taken to be a length or size rescaling. For instance dimensional functions of physical quantity are always scale-free since they obey power monomial law. It can be rigorously proved that the function that satisfies 2.51 should always have power law of the form  $f(x) \sim x^{-\alpha}$ .

Let us first set  $r = 1$  to obtain  $f(\lambda) = g(\lambda)f(1)$ . Thus  $g(\lambda) = f(\lambda)/f(1)$  and equation 2.51 can be written as

$$f(\lambda x) = \frac{f(\lambda)f(x)}{f(1)} \quad (2.52)$$

The above equation is supposed to be true for any  $\lambda$ , we can therefore differentiate both sides with respect to  $\lambda$  to yield

$$x f'(\lambda x) = \frac{f'(\lambda)f(x)}{f(1)} \quad (2.53)$$

where  $f'$  indicates the derivative of  $f$  with respect to its argument. Now we set  $\lambda = 1$  and get

$$x f'(x) = \frac{f'(1)f(x)}{f(1)} \quad (2.54)$$

This is a first order differential equation which has a solution

$$f(x) = f(1)x^{-\alpha} \quad (2.55)$$

where  $\alpha = -f(1)/f'(1)$ . There it is proven that the power law is the only solution that can satisfy 2.51. We can also prove that  $g(\lambda)$  has a power law form as well.

Power law distribution of the form  $f(x) \sim x^{-\alpha}$  are said to be scale free since the ratio  $\frac{f(\lambda x)}{f(x)}$  depends on  $\lambda$  alone. Thus the distribution does not need a characteristic scale. If we change the unit of measurement of  $x$  by a factor of  $\lambda$ , the numerical value of  $f(x)$  will change by a factor of  $g(\lambda)$ , without affecting the shape of the function  $f$ .

### 2.5.1 Generalized Homogeneous Function

A function  $f(x, y)$  of two independent variables  $x$  and  $y$  is said to be a generalized homogeneous function if for all values of the parameter  $\lambda$  the function  $f(x, y)$  satisfies,

$$f(\lambda^a x, \lambda^b y) = \lambda^p f(x, y) \quad (2.56)$$

where  $a, b$  are arbitrary numbers. In contrast to the homogeneous functions defined in the previous section ?? generalized homogeneous functions can not be written as  $f(\lambda x, \lambda y) = \lambda^p f(x, y)$ , because 2.56 can not be generalized any further to the following form,

$$f(\lambda^a x, \lambda^b y) = \lambda^p f(x, y) \quad (2.57)$$

choosing  $p = 1$  in the above equation yields

$$f(\lambda^{a/p} x, \lambda^{b/p} y) = \lambda f(x, y) \quad (2.58)$$

Similarly an statement converse is also valid and the equation above is no more general than the form in the equation 2.56. Another equivalent form of 2.56 is as follows,

$$f(\lambda x, \lambda^b y) = \lambda^p f(x, y) \quad (2.59)$$

Similarly

$$f(\lambda^a x, \lambda y) = \lambda^p f(x, y) \quad (2.60)$$

Note that there are at least two undetermined parameters  $a$  and  $b$  for a generalized homogeneous function. Now let us see what happens if we choose  $\lambda^a = 1/x$  to set in equation 2.56,

$$f\left(1, \frac{y}{x^{b/a}}\right) = x^{-\frac{p}{a}} \quad (2.61)$$

$$f(x, y) = x^{p/a} f(y/x^{b/a}) \quad (2.62)$$

This combining and hence the simplification of two variables  $x$  and  $y$  into a single term has far reaching consequence in Windex scaling ??in the theory of phase transition and critical phenomena.



# Chapter 3

## Phase Transition

**What is this** Phase transition is one of the most studied problem in physics. Phase transition is a process where below a critical point the system behaves in one way whereas above that point the system behaves in a completely different way. There is a control parameter in phase transition. It can be temperature  $T$  or magnetic field  $H$ . For example in ferromagnet to paramagnet transition temperature is the control parameter and for normal to superconductor transition both temperature and magnetic field are the control parameter.

The first explicit statement of the first law of thermodynamics, by *Rudolf Clausius* in 1850, referred to cyclic thermodynamic processes.

*In all cases in which work is produced by the agency of heat, a quantity of heat is consumed which is proportional to the work done; and conversely, by the expenditure of an equal quantity of work an equal quantity of heat is produced.*

$$\Delta E = Q + W \quad (3.1)$$

where,  $Q$  is the net quantity of heat supplied to the system by its surroundings and  $W$  is the net work done by the system. The IUPAC convention for the sign is as follows: All net energy transferred to the system is positive and net energy transferred from the system is negative. Clausius also stated the law in another form, referring to the existence of a function of state of the system, the internal energy, and expressed it in terms of a differential equation for the increments of a thermodynamic process.

*In a thermodynamic process involving a closed system, the increment in the internal energy is equal to the difference between the heat accumulated by the system and the work done by it.*

For quasi-static process

$$dE = dQ - PdV \quad (3.2)$$

$E$  is the internal energy. here  $W = -PdV$  since work done by the system on the environment if the product  $PdV$  whereas the work done on the system is  $-PdV$  for pressure  $P$  and volume change  $dV$ .

The term heat for  $Q$  means "that amount of energy added or removed by conduction of heat or by thermal radiation", rather than referring to a form of energy within the system. The internal energy is a mathematical abstraction that keeps account of the exchanges of energy that befall the system.

For quasi-static state we can write

$$dQ = TdS \quad (3.3)$$

where  $S$  is the entropy of the system and  $T$  is the temperature. Thus we can write for canonical ensemble

$$dE = TdS - pdV \quad (3.4)$$

such that  $E = E(S, V)$  and for grand canonical ensemble

$$dE = TdS - pdV + \mu dN \quad (3.5)$$

where  $E = E(S, V, N)$ . But a problem arises, since there is no device we currently posses that can measure entropy. So we use Legendre transformation to change variable dependency

$$\begin{aligned} dE &= TdS - pdV \\ &= TdS + SdT - SdT - PdV \\ d(E - TS) &= -SdT - PdV \\ dA &= -SdT - PdV \end{aligned} \quad (3.6)$$

where  $A = A(T, V)$  is the Helmholtz free energy. We can perform another Legendre transformation in 3.6 as follows

$$\begin{aligned} dA &= -SdT - PdV - VdP + VdP \\ d(A + PV) &= -SdT + VdP \\ dG &= -SdT + VdP \end{aligned} \quad (3.7)$$

where  $G = G(T, P)$  is the Gibbs free energy.

Let's take a break to talk about free energy. What is free energy?

## 3.1 Classification

### 3.1.1 First Order

1. Latent heat of nucleation in growth
2. Symmetry may or may not be broken
3. Discontinuous change in entropy

### 3.1.2 Second Order

1. Symmetry is always broken
2. No Latent heat or meta-stable state
3. Continuous change in entropy

## 3.2 Thermodynamic Quantities

### 3.2.1 Entropy

One of the most important concept in Statistical Mechanics is entropy. The notion of entropy was first introduced in physics by a German scientist Rudolf Clausius who laid the foundation for the second law of thermodynamics in 1850 by examining the relation between heat transfer and work. The second law of thermodynamics states that the total entropy of an isolated system can never decrease over time. Therefore the direction toward which entropy increases monotonically is the direction of time. Monotonically means that it can increase or keep constant but never decrease. The term entropy is used in many other branches of science, sometimes distant from physics or mathematics (such as sociology), where it no longer maintains its rigorous quantitative character. Usually, it roughly means disorder, chaos, decay of diversity or tendency toward uniform distribution of kinds [34].

Entropy is considered as a quantity about the disorderness of a system. This kind of disorder is the number of states a system can take on. So what are the states of a system? Imagine a cube of volume  $1\text{cm}^3$ , filled with one particular gas. At a particular time if we can

label all the molecules of the gas uniquely then at next moment most of the molecule will change their positions due to their random motion. Then we will not be able to identify each molecule with their previous label. This process of identifying the labels of the molecules are easier if it's a liquid and even more easier in it's solid form. Since temperature increases the random motion of the molecules, as the temprature rises it is more difficult to identify those molecules. Thus at high temperature a system has higher entropy. Another thing to mention about it's volume. If a larger volume is selected then obviously the number of possible states will increase therefore entropy will increase.

Example:

If we were ot compare the entropy of the moon and the sun, the above discussion tells us that the sun has higher entropy than the moon. The reason is that the sun is much much larger than the moon and has much higher temperature than the moon. Therefor the number of possible states will be larger for the sun than the moon.

In Classical Physics, entropy is seen as a magnitude which in every time is proportional to the quantity of energy that at that time cannot be transformed into mechanical work. Using the above interpretation, entropy plays a central role in the formulation of the second law of thermodynamics which states that in an isolated physical system, any transformation leads to an increase of its entropy.

In Probability Theory, the entropy of a random variable measures the uncertainty over the values which can be reached by the variable.

In Information Theory, the entropy of the compression of a message (for example, of a file from a computer), quantifies the content of the information of the message to have the minimum lost of information in the compression process previous to its transmission.

In Abstract Theory of Dynamical Systems, the entropy measures the exponential complexity of the system or the average flow of information per unit of time [12].

So it is already explicit that the concept of entropy is not easy to grasp and frequently entropy is seen as very mysterious quantity and that's why it received a very large number of interpretations, explications, applications. In order to understand how the complex concept of entropy emerged, in this chapter we will give a brief history and will review the works of Clausius, Boltzmann, Shannon and Rényi.

## Clausius Entropy

As mentioned Earlier in this section The concept and name of entropy originated in the early 1850s in the work of Rudolf Julius Emmanuel Clausius (1822 – 1888) and that work was at

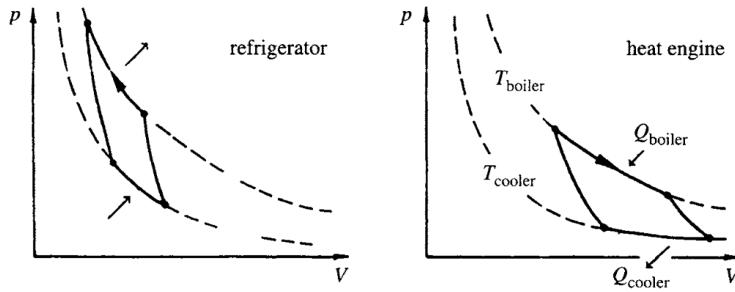


Fig. 3.1 Two competing Carnot engines, pressure (P) versus volume (V) - diagrams [10]

first primarily concerned with the question of which cycle is best suited for the conversion of heat into work and which substance is best used for the conversion [27].

Clausius based his argument on the plausible axiom that heat cannot pass by itself from a cold to a hot body. In order to exploit that axiom Clausius considered two competing Carnot cycles (see figure 3.1) working in the same temperature range, one as a heat engine and one as a refrigerator; the refrigerator consumes the work the engine provides. By comparing the amounts of heat  $Q$  passed from top to bottom and vice versa, he came to the conclusion that among all efficiencies the efficiency of a Carnot cycle is maximum and universal. 'Maximum' means that no cycle in the same range of temperature has a bigger efficiency than a Carnot cycle, and 'universal' means that all working substances provide the same efficiency in a Carnot cycle [24]. It is easy to calculate the efficiency of the Carnot engine of an ideal gas:

$$\begin{aligned}\eta &= \frac{W}{Q_{boiler}} \\ &= \frac{Q_{boiler} - Q_{cooler}}{Q_{boiler}} \\ &= 1 - \frac{Q_{cooler}}{Q_{boiler}} \quad (3.8)\end{aligned}$$

$$= 1 - \frac{T_{cooler}}{T_{boiler}} \quad (3.9)$$

where  $T$  is the Kelvin temperature. And, since by Clausius' result this efficiency is universal, it holds not only for ideal gases but for all substances, be they water, mercury, sulphur, or steel. Nicolas Léonard Sadi Carnot anticipated Clausius by 30 years, but no one could understand Carnot's reasoning. Carnot believed in the caloric theory of heat, by which the heat passing through the cycle from boiler to cooler is unchanged in amount. This is quite wrong and it is a kind of miracle that Carnot, despite his erroneous concepts, obtained a correct result. Carnot's trouble was that he did not know the balance of energy, or the first

law of thermodynamics, which states that

$$dU = dQ - dW \quad (3.10)$$

$$= dQ - PdV \quad (3.11)$$

since

$$dW = PdV \quad (3.12)$$

This equation holds, if the work is expended reversibly for the volume change. With that superior knowledge Clausius showed that it is not the heat that passes through the cycle from boiler to cooler unchanged in amount. He proved that from equation 3.9

$$\frac{Q}{T} \Big|_{\text{boiler}} = \frac{Q}{T} \Big|_{\text{heater}} \quad (3.13)$$

and decided to define the entropy change to be the ratio of heat flow to temperature: [80]

$$\Delta S_{\text{thermo}} = \frac{Q}{T} \quad (3.14)$$

He also showed that in an arbitrary process, the change of entropy, satisfies the inequality

$$dS \geq \frac{dQ}{T} \quad (3.15)$$

This important relation is known as the second law of thermodynamics. We will now discuss the significance of this law. First, we stick to reversible process, where the equality holds in equation 3.15. We may then eliminate  $dQ$  between the first and second laws, we obtain the Gibbs equation

$$dS = \frac{1}{T} (dU + PdV) \quad (3.16)$$

In an adiabatic irreversible process,  $dQ = 0$  and equation 3.15 one has the fundamental relation between entropy and irreversibility. In words

**In a closed system (adiabatic), the entropy cannot decrease, it remains constant or increase [15]** But the irreversible increase of entropy is not a property of the microscopic laws of nature. Because the microscopic laws of nature are time-reversal invariant. For example the Maxwell's equations are time reversible or the laws governing the motion of electrons of atoms are also time reversible. Thus the direction of time cannot be determined from microscopic laws of nature. But since entropy always increase monotonically, the direction in which entropy increases (or remains constant) is the direction of time [80].

### Boltzmann Entropy

Another intuitive interpretation of entropy is as a measure of the disorder in a system. Liquids have higher entropy than crystals intuitively because their atomic positions are less orderly [80]. Ludwig Boltzmann interpreted the entropy function as statistical entropy using probability theory. Around 1900 there was a fierce debate going on between scientists whether atoms really existed or not. Boltzmann was convinced that they existed and realized that models that relied on atoms and molecules and their energy distribution and their speed and momentum, could be of great help to understand physical phenomena. Because atoms were supposed to be very small, even in relatively small systems, one faces already a tremendous number of atoms. For example: one mole of water contains about  $6.023 \times 10^{23}$  molecules! Clearly it is impossible to track the energy and velocity of each individual atom. Therefore, Boltzmann introduced a mathematical treatment using statistical mechanical methods to describe the properties of a given physical system (for example the relationship between temperature, pressure and volume of one liter of air).

Boltzmann's idea behind statistical mechanics was to describe the properties of matter from the mechanical properties of atoms or molecules. In doing so, he was finally able to derive the Second Law of Thermodynamics around 1890 and showed the relationship between the atomic properties and the value of the entropy for a given system. It was Max Planck who formulated the expression of entropy of the ideal gas system based on Boltzmann's results [75]. The Boltzmann entropy formula or Boltzmann-Planck entropy formula is

$$S = k_B \log \Omega \quad (3.17)$$

$S$  is called the Planck entropy or Boltzmann entropy. Here  $k_B$  is the Boltzmann constant (ideal gas constant  $R$  divided by Avogadro's number  $N$ ) which equals to  $1.4 \times 10^{-23} \text{ J/K}$ , and  $\Omega$ , comes from the German *Wahrscheinlichkeit*, meaning probability, which is often referred to as disorder. In another way we can define it as the number of microstates (often modeled as quantum states) with the given macrostate. Macrostate is any particular arrangement of atoms where we look only on average quantities. Any individual arrangement defining the properties (e.g. positions and velocities) of all the atoms for a given macrostate is a microstate. For a microstate it matters what individual particles do, for the macrostate it does not.

The logarithm is used because it simplifies the computations and reproduced the property of additivity of entropy, in the sense that the entropy of two systems sums instead of multiplies. In equation 3.17, the entropy  $S$  increases when  $\Omega$  does. The more microstates are, the more disorder and more entropy are. Besides for only one possible microstate, entropy is zero.

The notion of disorder is an intuitive notion depending of the system to be considered. In the case of a gas, it is considered it in a ordered state if its molecules have a distribution of energies or positions very different of those random which means the Boltzmann distribution. The most disordered states are the most probable and as consequence have the most entropy. Another consequence is that in the universe the disorder tends to increase [12].

From there, it is a short conceptual jump to the second Law (total entropy tends to increase, and all spontaneous processes increase entropy). In summary, the thermodynamic definition of entropy provides the experimental definition of entropy, while the probabilistic definition of entropy extends the concept, providing an explanation and a deeper understanding of its nature [9].

These definitions of entropy is of no use in our model. However we can use the Boltzmann entropy with a few modification to get Shannon entropy [81]. This process of getting Shannon entropy from Boltzmann entropy and how does it help us is described in section 4.5.3.

### 3.2.2 latent heat

The latent heat associated with the phase transition is , where and are the entropies just above and below the phase transition. Since the entropy is continuous at the phase transition, the latent heat is zero. The latent heat is always zero for a second order phase transition.

### 3.2.3 Specific Heat

The specific heat is the amount of heat per unit mass required to raise the temperature by one degree Celsius. The relationship between heat and temperature change is usually expressed in the form shown below where  $C$  is the specific heat.

$$C = \frac{Q}{dT} \quad (3.18)$$

and we know

$$Q = TdS \quad (3.19)$$

where  $S$  is the entropy. we immediately get

$$C = T \frac{dS}{dT} \quad (3.20)$$

### 3.2.4 Order Parameter

### 3.2.5 Susceptibility

The first law of thermodynamics for a non magnetic system is ??

$$dE = TdS - PdV \quad (3.21)$$

for a magnetic system  $P \rightarrow h$  and  $V \rightarrow -m$ , where  $h$  is the magnetic field and  $m$  is the magnetization. The negative sign for the magnetization is because of the fact that the magnetization increases with the increase of the magnetic field whereas the volume decreases as the pressure increases. Since we relate pressure to the magnetic field, to relate volume to magnetization we need to put a minus sign with it. Thus we get

$$dE = TdS + hdm \quad (3.22)$$

$$= TdS + SdT - SdT + hdm \quad (3.23)$$

$$d(E - TS) = -SdT + hdm \quad (3.24)$$

$$dA = -SdT + hdm \quad (3.25)$$

$$dA = -SdT + hdm + mdh - mdh \quad (3.26)$$

$$d(A - mh) = -SdT - mdh \quad (3.27)$$

$$dG = -SdT - mdh \quad (3.28)$$

where  $A = A(T, m)$  is the Helmholtz free energy and  $G = G(T, h)$  is the Gibbs free energy for the magnetic system. From the definition of free energy for a magnetic system 3.28 3.25 we can find the expression for entropy and magnetization in terms of free energy.

$$S = -\left(\frac{\partial G}{\partial T}\right)_h = -\left(\frac{\partial A}{\partial T}\right)_m \quad (3.29)$$

$$m = -\left(\frac{\partial G}{\partial h}\right)_T \quad (3.30)$$

$$h = \left(\frac{\partial A}{\partial m}\right)_T \quad (3.31)$$

And since the susceptibility is defined as the derivative of the magnetization with respect to the magnetic field, we get

$$\chi = \left(\frac{\partial m}{\partial h}\right)_h = -\left(\frac{\partial^2 G}{\partial h^2}\right)_T \quad (3.32)$$

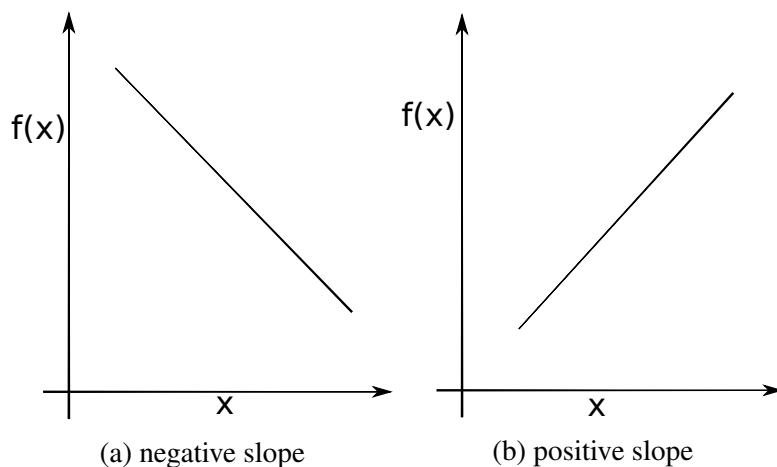


Fig. 3.2 shape of  $f(x)$  from  $f'(x)$

again we can write  $\chi$  as

$$\begin{aligned}\chi &= \frac{1}{\frac{\partial h}{\partial m}} \\ &= \frac{1}{\frac{\partial^2 A}{\partial m^2}}\end{aligned}\tag{3.33}$$

### 3.3 Shapes of the Thermodynamic Quantities

### 3.3.1 Calculus to determine the shapes

From the law of Calculus we can estimate the approximate shape of a function by it's first and second derivative. Say we have a function  $f(x)$  and we want to estimate it's shape. If the first derivative of this function is negative (positive) the function is said to have decreasing (increasing) slope 3.2.

If the second derivative of this function is negative (positive) the function is said to have concave (convex) shape 3.3. Thus if we know the sign of the first and second derivative we can approximate a shape of the function.

### 3.3.2 Free Energy

Now since we obtain from 3.7 that the entropy is the first derivative of the free energy 3.34 and from 3.39 specific heat is the second derivative of the free energy 3.35. Now since the entropy is a positive quantity and specific heat is also a positive quantity we get that the first and second derivative of the free energy is negative which implies from the laws of calculus

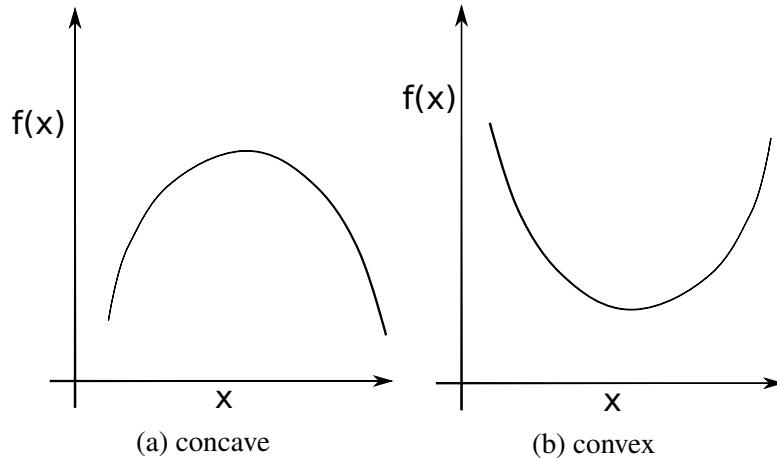
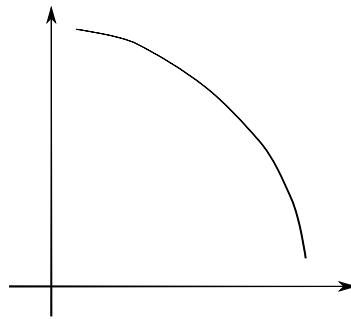
Fig. 3.3 shape of  $f(x)$  from  $f''(x)$ 

Fig. 3.4 Shape of the free energy

that the shape of the free energy should be a decreasing concave curve 3.4 in other words concave shaped with negative slope.

$$S = -\frac{\partial G}{\partial T} \quad (3.34)$$

$$C = -\frac{\partial^2 G}{\partial T^2} \quad (3.35)$$

### 3.3.3 Entropy and Specific Heat

Now that we know the shapes of the free energy, the very next thing to do is to find the shape of the entropy. Since we can get entropy from differentiating the free energy with respect to temperature  $T$  we get the following shape 3.5. From the definition of the transition order we know that the first order transition is discontinuous and the second order transition continuous.

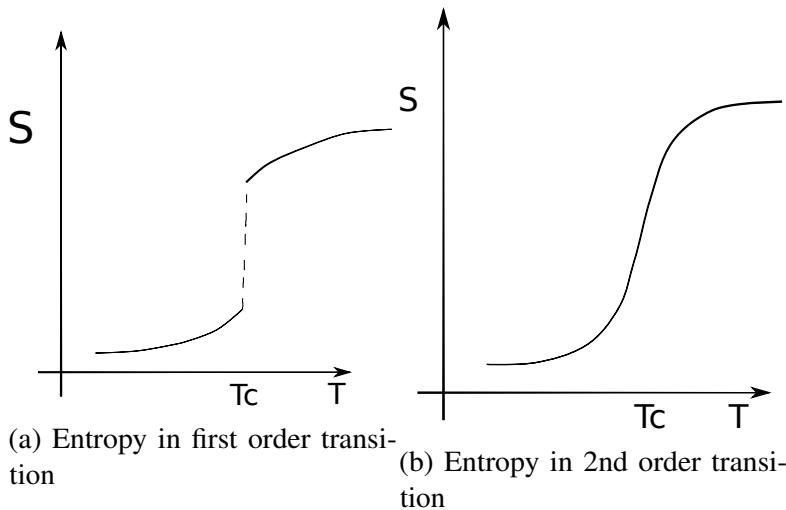


Fig. 3.5 Shape of entropy

Since the first order transition requires latent heat at the critical point the discontinuity is inevitable. Note that, since entropy measures disorderedness of a system it should increase with increasing temperature. Because the temperature is nothing but the average kinetic energy of the particles in the system. As the kinetic energy of the system increases with temperature, the particles tend to vibrate more and get higher average velocity. Thus making the system disordered. Therefore in a physical system entropy always increases with the increasing of temperature. If we find something different, as if, the entropy is decreasing with the increasing of temperature, there must be a problem somewhere. Just after knowing the shape of entropy one can anticipate the shape of specific heat, which is the derivative of the entropy with respect to temperature and is defined in 3.37. Using this and the shape of entropy we can get the shape of the specific heat immediately as follows

### 3.3.4 Order Parameter and Susceptibility

Since diamagnetic substances have negative susceptibilities ( $\chi < 0$ ); paramagnetic, and ferromagnetic substances have positive susceptibilities ( $\chi > 0$ ), and we are working for a phase transition model similar to paramagnet and ferromagnet transition, taking susceptibility to be positive is appropriate for our model. Now if susceptibility is positive then from 3.32 we get  $\left(\frac{\partial^2 G}{\partial h^2}\right) < 0$  which means that the shape of the free energy for this case is concave from the knowledge of 3.3.1. And from 3.30 and the fact that the magnetic field  $h$  can be positive or negative, as it can change direction, the Gibbs free energy is negative, since it is the lowest binding energy.

Also from 3.33 we can say  $\frac{\partial^2 A}{\partial m^2}$  is positive giving convex shape of the Helmholtz free energy.

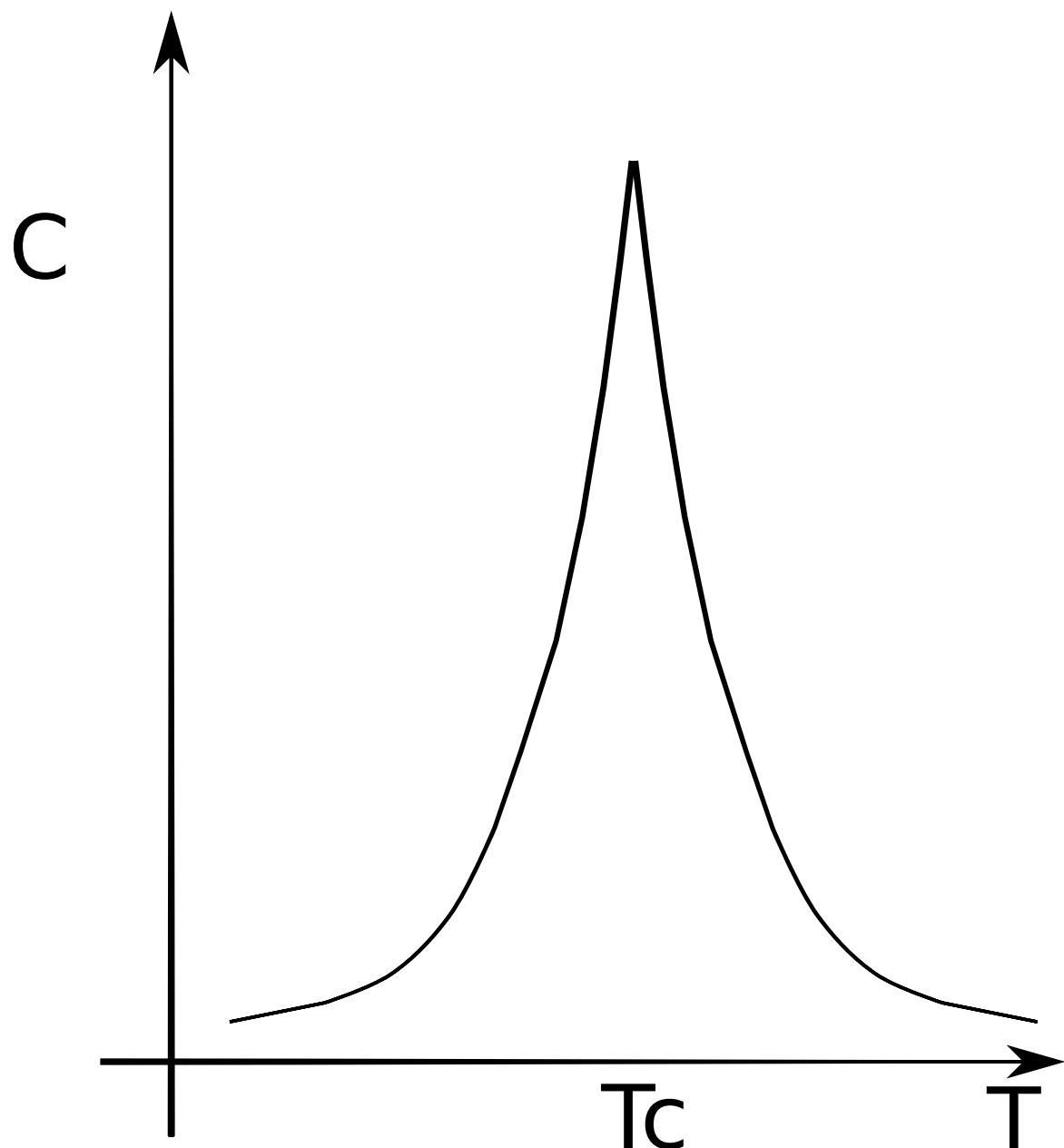


Fig. 3.6 Shape of Specific Heat

And 3.31 tells us that since  $h$  can be positive or negative, the Helmholtz free energy can have increasing or decreasing slope respectively.

### 3.4 Response Functions

The Specific heat and Susceptibility are called the response function in thermodynamics and the definitions are as follows. The specific heat  $C_p$  or  $C_v$  is the derivative of the enthalpy with respect to the temperature at constant pressure or volume respectively.

$$C_x = \left( \frac{dQ}{dT} \right)_x \quad (3.36)$$

here  $x$  can be  $P$  for pressure or  $V$  for volume. And from ?? we can write

$$C_x = T \left( \frac{dS}{dT} \right) \quad (3.37)$$

To give the following

$$C_v = -T \left( \frac{\partial^2 A}{\partial T^2} \right)_v \quad (3.38)$$

$$C_p = -T \left( \frac{\partial^2 G}{\partial T^2} \right)_p \quad (3.39)$$

Nearest neighbor interaction and 2nd nearest neighbor interaction. Old models of phase transition : Ising model 1D and 2D. Bragg Willium model.

### 3.5 Critical Exponents

Near the critical point there is, in general, a function that describes the behavior of the system that is mostly interesting. For thermodynamical system the temperature is the control parameter ??, thus that function depends on the temperature. But since we want the information at the critical point,  $(T - T_c)/T_c$ , instead of  $T$  is a better parameter to address. In equation

$$\epsilon = \frac{T - T_c}{T_c} = \frac{T}{T_c} - 1 \quad (3.40)$$

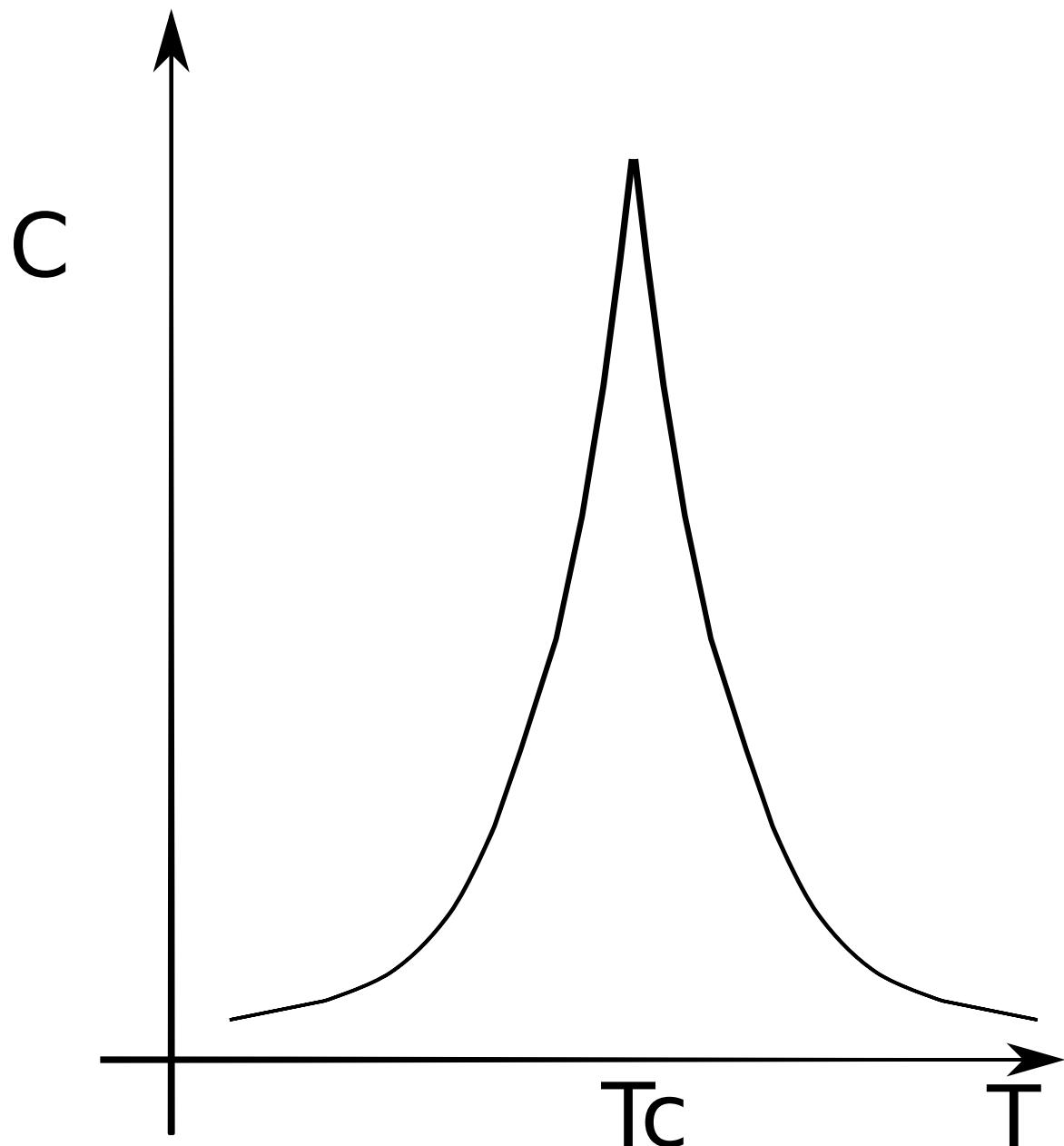


Fig. 3.7 figure required

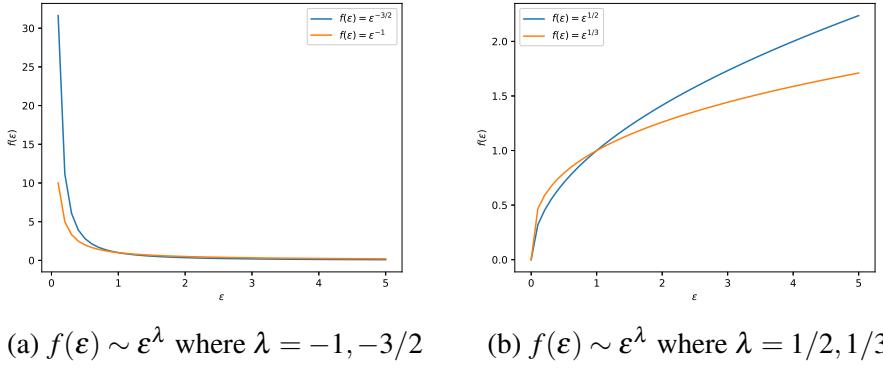
(a)  $f(\epsilon) \sim \epsilon^\lambda$  where  $\lambda = -1, -3/2$ (b)  $f(\epsilon) \sim \epsilon^\lambda$  where  $\lambda = 1/2, 1/3$ 

Fig. 3.8 Function showing power law

$\epsilon$  is considered a better parameter. Thus a function  $f(\epsilon)$  is used instead of  $f(T)$ . Using the fact that near  $T_c$  the function  $f(\epsilon)$  exhibits power law ??

$$f(\epsilon) \sim \epsilon^\lambda \quad (3.41)$$

The following figures shows some function that exhibit power law To see this more closely, we can expand any function  $f(\epsilon)$  as a power series of  $\epsilon$

$$f(\epsilon) = A\epsilon^\lambda (1 + B\epsilon^a + C\epsilon^b + \dots) \quad (3.42)$$

Near  $T_c$  only the first term is dominating ??.

We had the Gibbs free energy (for a magnetic system)

$$G = G(T, h) \equiv G(\epsilon, h) \quad (3.43)$$

since  $\epsilon$  is a better parameter. Assume that  $G$  is a generalized Homogeneous function. Therefore from ?? we can write

$$G(\lambda^{a_\epsilon} \epsilon, \lambda^{a_h} h) = \lambda G(\epsilon, h) \quad (3.44)$$

differentiating equation 3.44 with respect to  $h$

$$\begin{aligned} \frac{\partial G(\lambda^{a_\varepsilon} \varepsilon, \lambda^{a_h} h)}{\partial h} &= \lambda \frac{\partial G(\varepsilon, h)}{\partial h} \\ \frac{\partial G(\lambda^{a_\varepsilon} \varepsilon, \lambda^{a_h} h)}{\partial \lambda^{a_h} h} \lambda^{a_h} &= \lambda \frac{\partial G(\varepsilon, h)}{\partial h} \\ \lambda^{a_h} G'(\lambda^{a_\varepsilon} \varepsilon, \lambda^{a_h} h) &= \lambda G'(\varepsilon, h) \\ -\lambda^{a_h} m(\lambda^{a_\varepsilon} \varepsilon, \lambda^{a_h} h) &= -\lambda m(\varepsilon, h) \\ m(\lambda^{a_\varepsilon} \varepsilon, \lambda^{a_h} h) &= \lambda^{1-a_h} m(\varepsilon, h) \end{aligned} \quad (3.45)$$

$$(3.46)$$

Here  $m$  is the magnetization or order parameter.

### Figure of order general parameter

Setting  $h = 0$  in equation 3.46

$$\begin{aligned} m(\lambda^{a_\varepsilon} \varepsilon) &= \lambda^{1-a_h} m(\varepsilon) \\ m(1) &= \lambda^{1-a_h} m(\varepsilon) \\ m(1) &= \varepsilon^{-\frac{1-a_h}{a_\varepsilon}} m(\varepsilon) \\ m(\varepsilon) &= \varepsilon^\beta m(1) \end{aligned} \quad (3.47)$$

where

$$\beta = \frac{1-a_h}{a_\varepsilon} \quad (3.48)$$

Note that, setting  $\lambda^{a_\varepsilon} \varepsilon = 1$  gives  $\lambda = \varepsilon^{-1/a_\varepsilon}$ .

Setting  $\varepsilon = 0$  in equation 3.46

$$\begin{aligned} m(\lambda^{a_h} h) &= \lambda^{1-a_h} m(h) \\ m(1) &= h^{-\frac{1-a_h}{a_h}} m(h) \\ m(h) &= m(1) h^\delta \end{aligned} \quad (3.49)$$

where

$$\delta = \frac{a_h}{1-a_h} \quad (3.50)$$

Again, note that, setting  $\lambda^{a_h} h = 1$  gives  $\lambda = h^{-1/a_h}$

Now, Since the response functions are the second derivative of the free energy ??, by

differentiating 3.44 twice with respect to  $\varepsilon$  we get the Specific Heat

$$\begin{aligned}\lambda^{2a_\varepsilon} G''(\lambda^{a_\varepsilon} \varepsilon, \lambda^{a_h} h) &= \lambda G''(\varepsilon, h) \\ \lambda^{2a_\varepsilon} \frac{\partial G(\lambda^{a_\varepsilon} \varepsilon, \lambda^{a_h} h)}{\left(\frac{1}{T_c}\right)^2 \partial T^2} &= \lambda G''(\varepsilon, h) \\ \lambda^{2a_\varepsilon} C(\lambda^{a_\varepsilon} \varepsilon, \lambda^{a_h} h) &= \lambda C(\varepsilon, h)\end{aligned}\tag{3.51}$$

We have used  $\varepsilon = \frac{T}{T_c} - 1$  and  $d\varepsilon = \frac{1}{T_c} dT$ . Now setting  $h = 0$  we get from 3.51

$$\begin{aligned}\lambda^{2a_\varepsilon} C(\lambda^{a_\varepsilon} \varepsilon) &= \lambda C(\varepsilon) \\ C(1) &= \lambda^{1-2a_\varepsilon} C(\varepsilon) \\ &= \varepsilon^{-\frac{1-2a_\varepsilon}{a_\varepsilon}} C(\varepsilon) \\ C(\varepsilon) &= \varepsilon^{-\alpha} C(1)\end{aligned}\tag{3.52}$$

We have used the value of  $\lambda$  when we set  $\varepsilon \lambda^{a_\varepsilon} = 1$  and the exponent

$$\alpha = \frac{2a_\varepsilon - 1}{a_\varepsilon}\tag{3.53}$$

Again by differentiating 3.44 twice with respect to  $h$  we get the susceptibility. Then we set  $h = 0$  to get only  $\varepsilon$  dependency.

$$\begin{aligned}\lambda \frac{\partial^2 G(\varepsilon, h)}{\partial h^2} &= \lambda^{2a_h} \frac{\partial^2 G(\lambda^{a_\varepsilon} \varepsilon, \lambda^{a_h} h)}{\partial h^2} \\ \lambda \chi(\varepsilon, h) &= \lambda^{2a_h} \chi(\lambda^{a_\varepsilon} \varepsilon, \lambda^{a_h} h) \\ \chi(\varepsilon) &= \lambda^{2a_h - 1} \chi(\lambda^{a_\varepsilon} \varepsilon) \\ \chi(\varepsilon) &= \varepsilon^{-\gamma} \chi(1)\end{aligned}\tag{3.54}$$

Similar to previous case, We have used the value of  $\lambda$  when we set  $\varepsilon \lambda^{a_\varepsilon} = 1$  and the exponent is

$$\gamma = \frac{2a_h - 1}{a_\varepsilon}\tag{3.55}$$

### 3.5.1 List of Thermodynamic Quantities that Follows Power Law

**Critical Exponents at a glance** is it better title.

The exponent that scales Specific heat is called  $\alpha$

$$C \sim \varepsilon^{-\alpha} \quad (3.56)$$

The exponent that scales order-parameter is called  $\beta$

$$m \sim \varepsilon^\beta \quad (3.57)$$

Another exponent that scales order-parameter is called  $\delta$ , but it relates the order parameter with the magnetic field,  $h$

$$m \sim h^\delta \quad (3.58)$$

The exponent that scales susceptibility is called  $\gamma$

$$\chi \sim \varepsilon^{-\gamma} \quad (3.59)$$

Note that these quantities only follows power law near  $T_c$ .

### 3.5.2 Rushbrooke Inequality

$$\alpha + 2\beta + \gamma \geq 2 \quad (3.60)$$

but the equality is often obtain theoretically which is shown below in the present case

$$\begin{aligned} \alpha + 2\beta + \gamma &= \frac{2a_\varepsilon - 1}{a_\varepsilon} + \frac{2(1 - a_h)}{a_\varepsilon} + \frac{2a_h - 1}{a_\varepsilon} \\ &= \frac{2a_\varepsilon}{a_\varepsilon} \\ &= 2 \end{aligned}$$

Thus the Rushbrooke inequality is satisfied.

### 3.5.3 Griffiths Inequality

$$\alpha + \beta(1 + \delta) = 2 \quad (3.61)$$

Let's do a quick check to see if this is also satisfied.

$$\begin{aligned}
 \alpha + \beta(1 + \delta) &= \frac{2a_\varepsilon - 1}{a_\varepsilon} + \frac{1 - a_h}{a_\varepsilon} \left( 1 + \frac{a_h}{1 - a_h} \right) \\
 &= \frac{2a_\varepsilon - 1}{a_\varepsilon} + \frac{1}{a_\varepsilon} \\
 &= \frac{2a_\varepsilon}{a_\varepsilon} \\
 &= 2
 \end{aligned} \tag{3.62}$$

So the Griffiths Inequality is also satisfied.

## 3.6 Ising Model

For a very long time scientists have been trying to understand phase transition in different aspects. And for this purpose many different models have been provided for decades. The first model was provided by physicist Ernst Ising as a mathematical model of ferromagnetism in statistical mechanics. The Ising model was invented by the physicist Wilhelm Lenz (1920), who gave it as a problem to his student Ernst Ising. The one-dimensional Ising model has no phase transition and was solved by Ising (1925) himself in his 1924 thesis [50]. But he provided a model in 1 dimension and proved that in 1 dimension phase transition is not possible and concluded that it is not possible in higher dimension as well. But it was proved later that in dimension  $d \geq 2$  phase transition is possible [85, 62] which proved Ising wrong about his conclusion but he certainly gave a good insight for phase transition and hence the naming.

Consider a set  $\Lambda$  of lattice sites, each with a set of adjacent sites forming a  $d$ -dimensional lattice. For each lattice site  $k \in \Lambda$  there is a discrete variable  $\sigma_k$  such that  $\sigma_k \in \{+1, -1\}$ , representing the site's spin. A spin configuration  $\sigma = (\sigma_k)_{k \in \Lambda}$  is an assignment of spin value to each lattice site.

For any two adjacent sites  $i, j \in \Lambda$  there is an interaction  $J_{ij}$ . Also a site  $j \in \Lambda$  has an external magnetic field  $h_j$  interacting with it. The energy of a configuration  $\sigma$  is given by the Hamiltonian

$$H(\sigma) = - \sum_{\langle ij \rangle} J_{ij} \sigma_i \sigma_j - \mu \sum_j h_j \sigma_j \tag{3.63}$$

Where the first sum is over pairs of adjacent spins counting every pair only once. And we use  $\langle ij \rangle$  to indicate that  $i$  and  $j$  are nearest neighbors.  $\mu$  indicates the magnetic moment. The

negative sign on the second term is there by convention [11]. The configuration probability is given by the Boltzmann distribution with inverse temperature  $\beta \geq 0$

$$P_\beta(\sigma) = \frac{e^{-\beta H(\sigma)}}{Z_\beta} \quad (3.64)$$

where  $\beta = (k_B T)^{-1}$  and the normalization constant

$$Z_\beta = \sum_{\sigma} e^{-\beta H(\sigma)} \quad (3.65)$$

is the partition function. For a function  $f$  of the spins, which is the observable here, we denote

$$\langle f \rangle_\beta = \sum_{\sigma} f(\sigma) P_\beta(\sigma) \quad (3.66)$$

the expectation value of  $f$ . The configuration probabilities  $P_\beta(\sigma)$  represent the probability that the system is in a state with configuration  $\sigma$  in equilibrium.

The minus sign on each term of the Hamiltonian function  $H(\sigma)$  is conventional. Using this sign convention, the Ising models can be classified according to the sign of the interaction: if, for all pairs  $i, j$

$$J_{ij} \begin{cases} > 0, \text{ ferromagnetic interaction} \\ < 0, \text{ anti-ferromagnetic interaction} \\ = 0, \text{ non-interacting spins} \end{cases} \quad (3.67)$$

otherwise the system is called non-ferromagnetic.

In a ferromagnetic Ising model, spins tend to be aligned, meaning, the configurations in which adjacent spins are of the same sign have higher probability. In an anti-ferromagnetic model, adjacent spins tend to have opposite signs.

The sign convention of  $H(\sigma)$  also explains how a spin site  $j$  interacts with the external field. Namely, the spin site wants to line up with the external field.

$$h_j \begin{cases} > 0, \text{ the spin site } j \text{ desires to line up in the positive direction} \\ > 0, \text{ the spin site } j \text{ desires to line up in the negative direction} \\ = 0, \text{ no external influence on the spin site} \end{cases} \quad (3.68)$$

Ising models are often examined without an external field interacting with the lattice, that is,  $h = 0$  for all  $j$  in the lattice  $\Lambda$ . Using this simplification, our Hamiltonian becomes:

$$H(\sigma) = - \sum_{\langle ij \rangle} J_{ij} \sigma_i \sigma_j \quad (3.69)$$

When the external field is everywhere zero,  $h = 0$ , the Ising model is symmetric under switching the value of the spin in all the lattice sites; a non zero field breaks this symmetry.

Another common simplification is to assume that all of the nearest neighbors  $\langle ij \rangle$  have the same interaction strength. Then we can set  $J_{ij} = J$  for all pairs  $i, j$ . In this case our Hamiltonian is further simplified to:

$$H(\sigma) = -J \sum_{\langle ij \rangle} \sigma_i \sigma_j \quad (3.70)$$

Ising himself had provided the theory of his model for 1 dimensional case. He did not find any phase transition and concluded that phase transition is not possible in any dimension.

# Chapter 4

## Percolation Theory

Percolation theory potentially has been of great interest as it can describe many phenomena [73]. New models and variants of existing model is always welcome due to its importance and of wide interdisciplinary interests. In recent decades there has been a surge of research activities in studying percolation thanks to the emergence of network which has been used as the skeleton for percolation which can mimic structure of many natural and man-made systems.

There are several reason to study percolation. First, it is easy to formulate and simple to implement as there is only one control parameter, called occupation probability  $p$  4.4.1. Second, scientists use it as a theoretical model for phase transition, just like architects use geometric model before building large expensive structure, because of its simplicity. Third, it is well endowed with beautiful features and conjectures like finite-size scaling, universality just like its thermal counterpart. Fourth, besides being the paradigmatic model for phase transition, it has been found that the notion of percolation is omnipresent in a wide range of many seemingly disparate systems 4.9.

To study percolation theoretically, the first thing that one need is to choose a skeleton or playground, namely an empty lattice (or a graph/network), consisting of sites (or nodes) and bonds (or links). The definition of the percolation model is then so simple that it merely needs a sentence to define it. Each site or bond of the chosen skeleton, depending on whether we want to study site or bond type percolation, is either occupied with probability  $p$  or remains empty with probability  $1 - p$  independent of the state of its neighbors. Recently, percolation has received a renewed attention due to widening scope for using complex networks as a skeleton and due to widening extent of using various variants as a rule. In percolation most observable quantities this way or another is connected to clusters, group of contiguous

occupied sites form a cluster, or to their distribution function. As the occupation probability  $p$  is tuned starting from  $p = 0$ , one finds that at certain value of  $p = p_c$  the observable quantities undergoes a sudden and sharp change which is always regarded as a sign of phase transition. Indeed, the value at which such change occurs is called threshold or critical value which is equivalent to critical temperature of its thermal counterpart. The phase transition that percolation describes is purely geometrical in nature. It requires no consideration of quantum and many particle interaction effects and hence we can use it as a model for thermal Continuous Phase Transition (CPT) like architect use model before constructing large and complicated structure

## 4.1 Percolation Phenomena

The word 'Percolation' comes from the coffee percolator but it has nothing to do with coffee brewing. The only connection might be in the concept that lies behind the name. Let's start explaining the phenomena by example. Let's consider a square grid made of conductor of size  $L \times L$  containing  $L^2$  sites (insulator). Now we set up an arrangement to put a potential difference across the square grid and measure the effective resistance of the grid as well as the current. Suppose now we start taking of the sites one by one at a random and place a conductor there. We define a parameter called Occupation Probability,  $p$ , which is the fraction of the sites replaced as conductor to the total number of sites. We visit all the sites and generate a random number and only if  $r \leq p$  we replace the site with a conductor. We observe that the electric conductance of the grid will be found to increase with increasing sites being replaced by conductors. As the sites are being turned into insulators, the value of  $p$  changes. We see that at a certain value of  $p$ , the electricity starting flowing. This particular value is known as Percolation Threshold,  $p_c$ . This vanishing resistance occurs when a particular amount of sites turns into conductors from insulators that causes the system to get the connection between the two end across polarity. The exact value of  $p_c$  has been found to be close to 0.5927. One can never expect to get the value each time they perform this experiment, infact, the chance of getting this exact value at any experiment can be one in a million. So we do this same experiment a number of times and then average over the value to get a better result. This example shows a simple way of understanding the phenomenon that we call percolation. The theory which simulates this kind of phenomenon is known as Percolation Theory. It provides a quantitative description of the nature of continuous pathways through space. The usual objective of research implementing this phenomenon is to characterize some aspect of the critical phenomena of the phase transition between

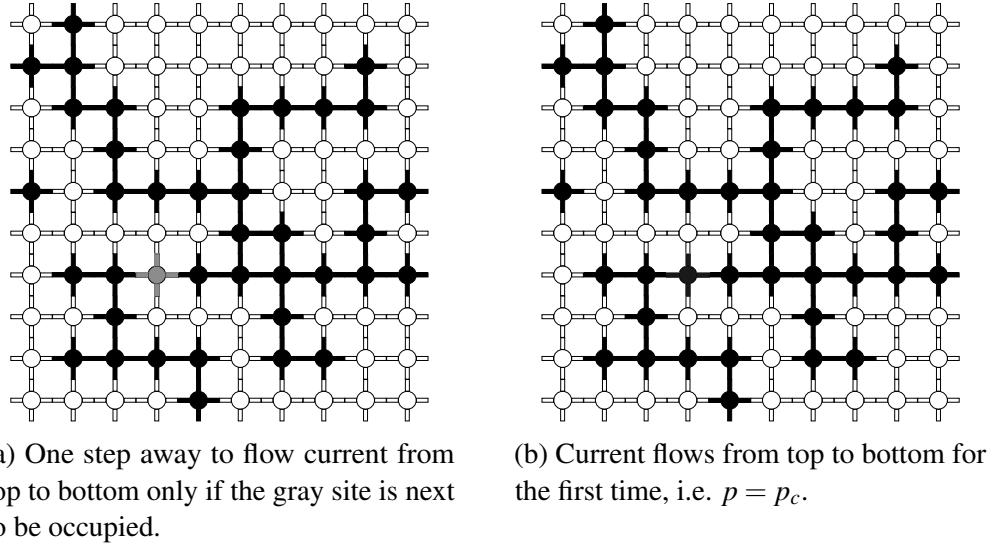


Fig. 4.1 percolation phenomena in square structure. No current if  $p < p_c$ .

finite and infinite range connectivity. This is actually non-trivial when the space is randomly disordered and the connected regions acquire fractal properties [86, 60].

## 4.2 Historical Overview

In 1941, the idea of percolation was first conceived by Flory [38] in the context of gelation transition. But as a mathematical model, it was formulated in the late fifties, to understand the motion of gas molecules through the maze of pores in carbon granules filling a gas mask. This seminal work was done by Simon Broadbent and John Hammersley [21], an engineer and a mathematician respectively. Later on, percolation problem was popularized in physics community by Cyril Domb, Michael Fisher, John Essam and M.F. Skyes [36, 37]. Another remarkable work was the observation of percolation to be the limiting case of the general Potts model, which includes the Ising model and can be solved exactly which is done by Fortuin and Kasteleyn in 1969 [51]. This work paved the way to many exact results in percolation, and also allowed the usage of powerful re-normalization group ideas [23]. Finding percolation threshold both exactly and by simulation has been an enduring subject of research in this field [76], as well as the development of algorithms such as those by Hoshen and Kopelman [47], by Leath [54], and by Newman and Ziff [65] (which we have used in our research). Finding rigorous proofs of exact thresholds and bounds has also been an enduring area of research for mathematicians (Kesten [52], Wierman [82], Bollobas and Riordan [19] etc.) Another infusion of interest in percolation came from the surging field of network theory which goes back to the study of random and complete graphs by Erdos-Renyi(1959). In ER

model, an observation of made of formation of a giant component, this giant component is exactly analogous to the formation pattern in percolation [1, 35]. It was then revitalized by interest in the small world phenomenon and Scale free networks. In 2000, Newman and Moore found the critical point for a random graph in the limit of large size, in which case the system is effectively a Bethe lattice, and this result connects to the early work of Flory, Fisher and Essam, but it was found with a general degree distribution [63]. In the field of random networks, the model of explosive percolation was first introduced by Achlioptas, D’Souza and Spencer [7], and this has been another fascinating problem which has led to a wave of new interest in percolation.

## 4.3 Classifications and Playground

Any percolation problem have two major parts, one is the rule which states how and what to occupy and connect and another is a playground on which we apply the rules. The term spanning cluster is the cluster that connects the opposite boundary of a playground. If, however, the system has no boundary then it is measured in terms of largest cluster and is called Giant Connected Component or GCC and it only appears if the system has reached or passed the threshold.

### 4.3.1 Types of Percolation

Since the first percolation model studied was Bernoulli percolation (In this model all bonds are independent. This model is called bond percolation by physicists) through time physicists have defined different types of percolation model in different types of skeleton or playground. Here we discuss some of the most used percolation types.

#### Bond Percolation

In bond (or edge, link) percolation we occupy the bonds with probability  $p$  and we use sites to connect the bonds together. The cluster size is measured in terms of the number of sites in the cluster. If  $p$  is below a critical value  $p_c$  then there is no spanning cluster (or GCC) and if  $p \geq p_c$  then we will have a spanning cluster or GCC.

#### Site Percolation

In site (or vertex, node) percolation we occupy the sites. According to the definition of site percolation we use sites to measure the cluster size. But in order to keep it consistent with

the laws of thermodynamics we have changed the definition [71]. Now we use bonds to measure the cluster size while we occupy sites. This change of definition does not effect the exponent that describe the phase transition. Before the critical point where  $p < p_c$  there is not spanning cluster and if  $p = p_c$  the spanning cluster appears for the first time and it remains as the largest cluster of the system. All the property that describes the phase transition are present at  $p = p_c$  and this is where we study them.

### Explosive Percolation

In 2009 Achlioptas et al. proposed a biased occupation rule, known as the Achlioptas process (AP), that encourages slower growth of the larger clusters and faster growth of the smaller clusters instead of random occupation in classical percolation [7]. According to this rule a pair of links or, bonds are first picked uniformly at random from all possible distinct links. However, of the two, only the one that satisfies the pre-selected rule is finally chosen to occupy and the other one is discarded. The preset rule is usually chosen so that it discourages the growth of the larger clusters and encourages the growth of the smaller clusters. As a result, the percolation threshold is delayed and hence the corresponding  $p_c$  is always higher than the case where only one bond is always selected. Furthermore, it is natural to expect that close to  $p_c$  nearly equal sized clusters, waiting to merge, are so great in number that occupation of a few bonds results in an abrupt global connection and thus the name "Explosive Percolation" (EP). Investigation of Achlioptas processes on different types of substrate networks and with different choices of rules especially with the aim of developing rules that do not use global information about a graph is an active area of research.

### K-Core Percolation

The  $K$ -core of an unweighted, undirected network is the maximal subset of nodes such that each node is connected to at least  $K$  other nodes [79]. Determining  $K$ -cores computationally is fast and they are insightful in many situations [33, 31]. Every undirected, unweighted network has a  $K$ -core decomposition.  $K$ -shell of a network is defined as the set of all nodes that belong to the  $K$ -core but not to the  $(K + 1)$ - core.  $K$ -core is given by the union of all  $c$ -shells for  $c \geq K$  and the  $K$ -core decomposition is the set of all of its  $c$ -shells. One can examine the  $K$ -core of a network as the limit of a dynamical pruning process. Starting with the initial network we delete all nodes with fewer than  $k$  neighbors. After this pruning, the degree of some of the remaining nodes will have been reduced. Repeating this process to delete nodes will give a network which will have fewer than  $K$  remaining neighbors. Iterating this process until no further pruning is possible leaves the  $K$ -core of the network [70].

## Bootstrap Percolation

Bootstrap percolation is an infection-like process in which nodes becomes infected if sufficiently many of their neighbors are infected [8]. In bootstrap percolation, sites on an empty lattice are first randomly occupied and then all occupied sites with less than a given number  $m$  of the occupied neighbors are successively removed until a stable configuration reached. On any lattice for sufficiently large  $m$ , the ensuing clusters can only be infinite. On a Bethe lattice for  $m \geq 3$ , the fraction of the lattice occupied by infinite clusters discontinuously jumps from zero at the percolation threshold. From an analysis of stable and metastable ground state of the dilute Blume-Capel model [16], it is concluded that the effects like bootstrap percolation may occur in some real magnets [25].

## Other

Apart from the type of percolation process described briefly above there are numerous other types of percolation. In *Limited Path Percolation*, one construes "connectivity" as implying that sufficiently short path still exists after some network components have been removed [56]. The percolation of  $K$ -cliques (completely connected sub-graphs of  $K$  nodes) has been used to study the algorithmic detection of dense sets of nodes known as "communities" [68]. Various percolation processes have also been studied in several different types of multi-layer networks (e.g. multiplex networks and interdependent networks) [17, 53]. Another class of models results if we remove the restriction of a lattice and allow particles to occupy positions which vary continuously in space. *Continuum Percolation* [44, 69, 18], as it is called, suffers from the added complication that tricks which can sometimes be used on lattice models cannot be applied. A quite different process as *Invasion Percolation* [43, 59, 84]; its invention was prompted by attempts to understand flow in porous media. Random numbers are assigned to each site of a lattice. Choose a site, or sites, on one side of the lattice and draw a bond to the neighbor which has the lower random number assigned to it. (The growing cluster represents the invading fluid with the remainder of the sites representing the initial, or defending, fluid). This process continues until the cluster reaches the other side [55].

### 4.3.2 Types of Playground

Apart from the rules we need a playground to study percolation. Most used playground is described in the following.

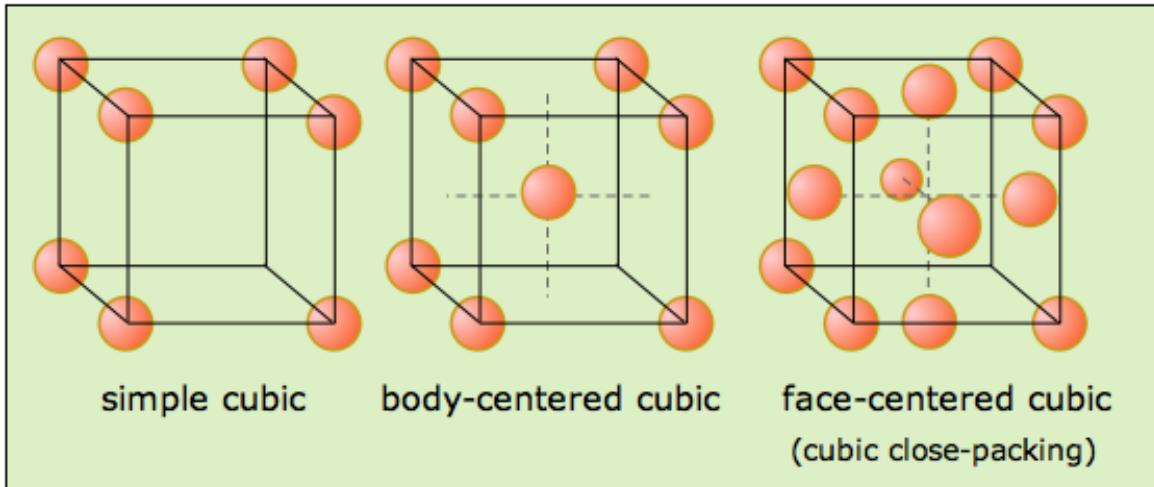


Fig. 4.2 Different types of cubic lattice [2]

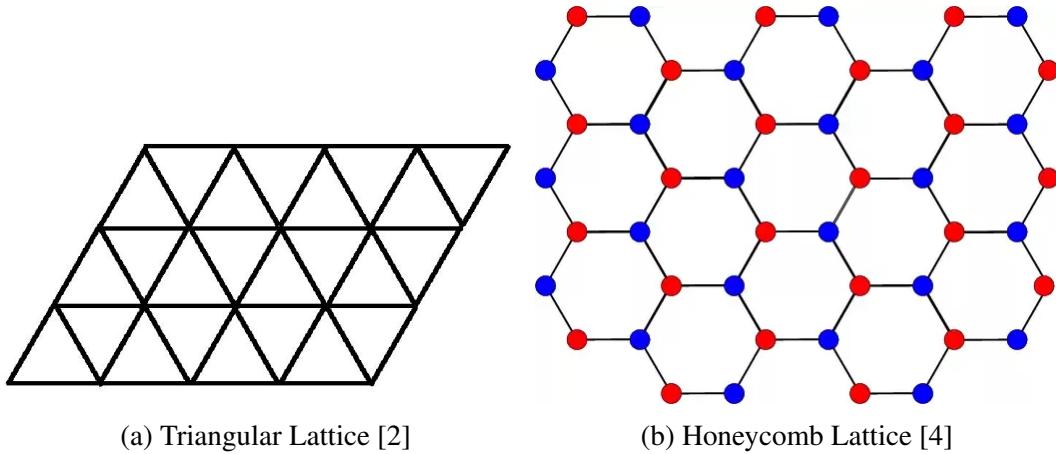


Fig. 4.3 Different types of cubic lattice

## Lattice

Percolation was mainly studied on different types of lattice. For example, Honeycomb lattice, Bethe lattice, Simple Cubic lattice, Body-Centered-Cubic lattice, Face-Centered-Cubic lattice. In 1998 Christian D. Lorenz et al. did extensive Monte Carlo simulation to study bond percolation on the simple cubic, face-centered-cubic and body-centered-cubic lattices using epidemic approach. These simulations provide very precise values of the critical thresholds [57]. They calculated Fisher exponent,  $\tau$ , the finite-size correction exponent,  $\Omega$  and the scaling function exponent,  $\sigma$  confirmed to be universal. They also did percolation on the HCP (hexagonal closed packed) lattice [58]. Before that in 1981 JC Wierman studied bond percolation on honeycomb and triangular lattices [83].

Furthermore, an exact solution on Bethe lattice for high density percolation was done by G. R. Reich and P. L. Leath in 1978 [72] and J. Chalupa et al. did bootstrap percolation on Bethe lattice in the following year [25].

### **Graph or Network**

A new playground was added to the world of percolation in 1959 when two prominent mathematicians of all time, Paul Erdos and Alfred Renyi introduced Random Graphs []. Before that, it was mostly popular among the mathematicians but not that much fascinating for most of the physicists because of its abstraction. But with the advent of scale free networks [6] which mimics the real life networks such as citation network, social network, protein-protein interaction, in 1999, physicists became much more interested in random networks. The difference in the terminology like graphs and networks are just graphs are mostly used by mathematicians and computer scientists and networks is mainly referred by physicists to the same concept. Duncan S. Callaway and his group did percolation study to find robustness and fragility [22]. Clique percolation is also studied on random networks [32]. A critical phenomenon analysis on random networks for core percolation is done by Bauer [14]. In early 80's, before the birth of scale free networks, C McDiarmid did two significant works which you can find in the references [].

Reuvan Cohen et al. studied scale free networks close to the percolation threshold [28]. N. Schwartz et al. worked on percolation problem in directed scale free networks [77]. Filippo Radicchi and Santo Fortunato studied scale-free networks constructed via a cooperative Achlioptas Growth Process [7]. They showed that networks constructed via this biased procedure show a percolation transition which strongly differes from the one observed in standard percolation, where links were introduced just randomly.

## **4.4 Basic Elements**

In this section we discuss some of the basic elements of percolation. All the observable quantities depends on these elements.

### **4.4.1 Occupation Probability**

The probability at which each site (bond) is occupied is called occupation probability in site (bond) percolation and it is denoted as  $p$ . In section 4.7 we discuss who we determine  $p$  in

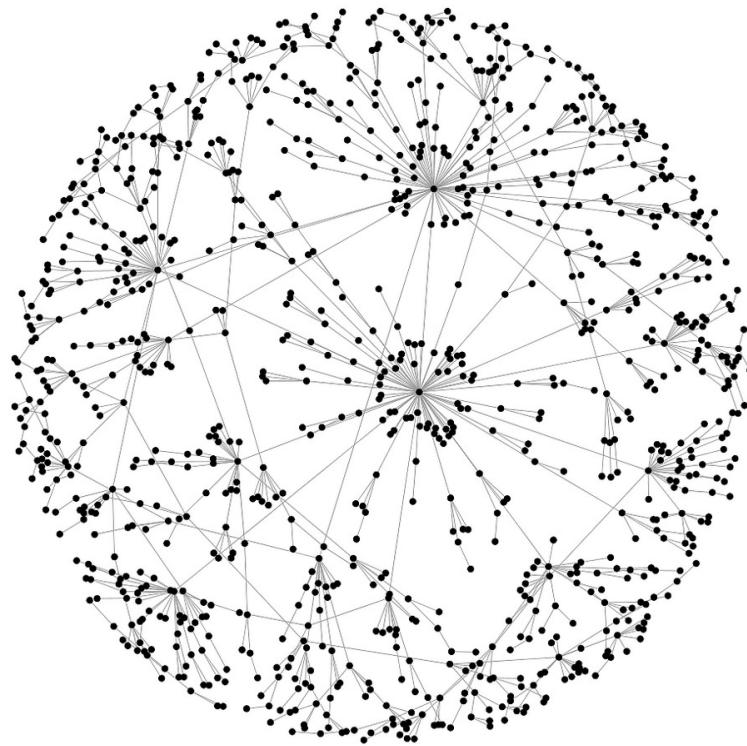


Fig. 4.4 Scale Free Network [5]

percolation. Simply saying, instead of fixing  $p$  for one experiment we can find all quantity for each values of  $p$  using the famous Ziff algorithm [64, 65].

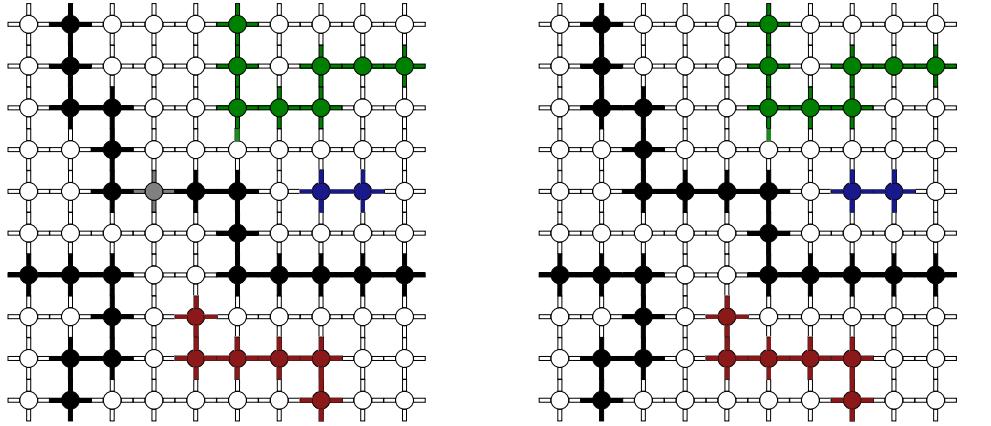
#### 4.4.2 Cluster

Cluster is a collection of sites and bonds. In bond percolation we occupy bond and measure cluster size in terms of the number of sites in it. And following this idea we measure the cluster size in terms of the number of bonds in it. Which is the new definition of site percolation [71]. Measuring cluster size in terms of the number of bonds in it in site percolation reproduces all known results and it is consistent with laws of thermodynamics.

#### 4.4.3 Spanning cluster

### 4.5 Observable Quantities

In percolation we study formation of clusters, their properties such as how they are distributed as a function of control parameter  $p$ . A cluster is a group of occupied bonds (sites) in site (bond) percolation [71] With no gap or unoccupied object between them. In a network cluster



(a) One step away from threshold. Since the gray site is not occupied. The moment it gets occupied we reach percolation threshold

(b) The gray site gets occupied and a wrapping cluster appears for the first time. We have reached percolation threshold

Fig. 4.5 A square lattice of length  $L = 10$ . Demonstrating the appearance of the wrapping cluster

is defined as a group of nodes that connects the links. In this section we discuss some of the most used observable quantities in percolation.

#### 4.5.1 Percolation Threshold, $p_c$

The idea of percolation threshold is a mathematical concept that deals with formation of long range connectivity in random systems. While there are no existence of a giant connected component that are talked about below the threshold, its existence is well present above it. As the occupation probability increases from 0 toward 1, average cluster size also increases and among all cluster one cluster pops up to be the spanning cluster at  $p_c$ . Hence  $p_c$  is the percolation threshold or critical point. In the thermodynamic limit this cluster becomes infinitely large. Thus for  $p < p_c$  there is no spanning cluster and for  $p \geq p_c$  there is one. The spanning cluster is a special type of cluster which spans the entire lattice. In periodic case we call it wrapping cluster. A figure illustrating this process is given here 4.5. Percolation threshold has been determined for different types of lattices. The table 4.1 shows some of them.

One simple way of obtaining  $p_c$  is to perform  $M$  number of independent experiment for a particular  $L$  and them take the average

$$p_{c_{avg}} = \lim_{M \rightarrow \infty} \frac{1}{M} \sum_i p_{c_i} \quad (4.1)$$

Lattice	$p_c$ , site percolation	$p_c$ , bond percolation
Body Centered	0.246	0.1803
Face Centered	0.198	0.119
Simple Cubic	0.3116	0.2488
Diamond	0.43	0.388
Honeycomb	0.6962	0.65271
Triangular	0.50	0.34729
Square	0.592746	0.50

Table 4.1 Percolation Threshold for Some Regular Lattices

But performing infinite number of experiment is not possible. So we use another approach which is described in section 4.5.2 and 5.4.1

### 4.5.2 Spanning Probability, $w(p, L)$

The best quantity for finding the critical exponent  $v$  is the spanning probability  $W(p, L)$ . It describes the likelihood of finding a cluster that spans across the entire system of length  $L$  either horizontally or vertically at a given occupation probability  $p$ . To find  $W(p, L)$  we perform say  $M$  independent realizations under the same identical conditions. In each realization for a given finite system size we take record of the  $p_c$  value at which there appears a spanning cluster for the first time. If there is a spanning cluster at  $p = p_{ci}$  then it means that there exists a spanning cluster for all  $p_{ci} \leq p \leq 1$ . To find a regularity or a pattern among all the  $M$  numbers of  $p_c$  values recorded, one usually looks at the relative frequency of occurrence within a class or width  $\Delta p$ . To find  $W(p, L)$ , we can process the data containing  $M$  number of  $p_c$  values to plot histogram displaying normalized relative frequency as a function of class of width  $\Delta p$  chosen as per convenience [71]. Figure 5.1 shows spanning probability  $w(p, L)$  as a function of  $p$  and  $L$ . Clearly all curves meet at a specific point regardless of system length  $L$ . Thus we can say if  $L \rightarrow \infty$  the curve would still go through that point and that's how we get the  $p_c$  value. Then if we apply scaling on the  $x$ -axis by  $(p - p_c)L^{1/v}$  we would get a perfect data collapse for a specific  $1/nu$  that is our exponent. This process is discussed further in section 5.4.1 and 5.4.2.

### 4.5.3 Entropy, $H(p, L)$

Entropy is one of the key feature of a phase transition model. But thermodynamic entropy, e.g. Boltzmann entropy or Clausius entropy, cannot be calculated for any model that depends on a probabilistic theory such as our percolation on a square lattice model. So we take

our approach in another way. Information entropy or the *Shanon entropy* is best suited for our model. The concept of Shannon entropy in information theory describes how much information there is in a signal or event. This concept of information entropy was introduced by Claude Shannon in his 1948 [81]. The seminal work of Shannon based on papers by Nyquist [66, 67] and Hartley [45]. He rationalized these early efforts into a coherent mathematical theory of communication.

The Definition of Shannon entropy is

$$H = - \sum_i \mu_i \log \mu_i \quad (4.2)$$

where,  $\mu_i$  is the probability of getting  $i$ -th element.

As we have seen in section 3.2.1 that the entropy measures disorderness of a system. It is also easy to understand order and disorder in solid to liquid transition. But it is not that easy in percolation since the idea of order and disorder in percolation is not yet clear. To understand disorder, let us consider that at  $p = 0$  we have 12 isolated sites and each has different color to identify them visually, figure 4.6. Thus each color corresponds to one distinct cluster. Occupation of a bond means merging of two colors into one. If one of the components is bigger in size than the other then the newly merged cluster will take the color of the bigger cluster and if they are equal then we choose one at random. If we now continue to occupy all the frozen bonds then we will finally have one cluster of one color. Initially at  $p = 0$  we have 12 different colors and hence it can easily be regarded as the most disordered state. On the other hand, at the other extreme we will have only one cluster represented by one color which can be regarded as the ordered state. We can easily extend the problem to a system that contains  $N$  number of sites but colored with 12 colors only such that  $n_1, n_2, \dots, n_{12}$  of them are red, orange, ..., violet respectively and hence we can define  $\mu_i = n_i/N$ . We now make  $M$  number of independent attempts to pick one site at each attempt from  $N$  sites at random with uniform probability. Say, that we found  $n'_1$  times red,  $n'_2$  times orange, and so on such that  $\sum_{i=1}^m n'_i = M$ . We assume that both  $N$  and  $M$  are sufficiently large and  $M \approx N$ . The total number of ways we could have  $M$  outcomes are

$$\Omega = \frac{M!}{(M_{\mu_1})! (M_{\mu_2})! \dots (M_{\mu_m})!} \quad (4.3)$$

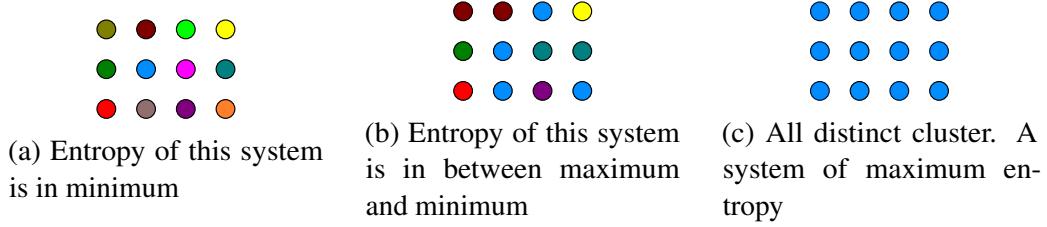


Fig. 4.6 A system of 12 cluster

since  $n'_i \approx M_{\mu_i}$ . Taking log on both side of the above equation and using the Stirling's approximation  $\log(M!) = M \log M - M$  for very large  $M$  we get

$$\begin{aligned}
 \log \Omega &= \log M! - \log \left( \prod_i M_{\mu_i} \right) \\
 &= M \log M - M - \sum_i \log (M_{\mu_i}!) \\
 &= M \log M - M - \sum_i [M_{\mu_i} \log M_{\mu_i} - M_{\mu_i}] \\
 &= M \log M - \sum_i M_{\mu_i} \log M_{\mu_i} \\
 &= M \log M - M \sum_i \frac{M_{\mu_i}}{M} \left[ \log \left( \frac{M_{\mu_i}}{M} \right) + \log M \right] \\
 &= M \log M - M \sum_i \mu_i \log \mu_i + M \sum_i \frac{M_{\mu_i}}{M} \log M
 \end{aligned} \tag{4.4}$$

Finally we have,

$$\log \Omega = -M \sum_{i=1}^m \mu_i \log \mu_i = M H(\mu) \tag{4.5}$$

where  $H(\mu)$  is nothing but the Shannon entropy  $H(p)$  and clearly it is the degree of uncertainty per attempt. Total entropy or information is therefore equal to  $NH$  if we consider  $M = N$ . In the case when each site has distinct color then each site will have the equal chance of being picked. It means  $\mu_i = 1/N \forall i$  and hence  $H(\mu) = N \log(N)$  which is the average entropy or degree of disorder and the total entropy is  $NH(\mu)$ . Now initially if we had  $N$  distinct color then the system would have the maximum entropy  $S = N \log N$ . On the other hand, if all the sites had the same color then the system would have minimum entropy  $S = 0$ . Thus percolation is indeed an order-disorder transition where disorder is equivalent to degree of confusion [71]. Now recall the definition of Boltzmann entropy in equation 3.17. If we consider  $K_B = 1$  then we get Shannon entropy to represent Boltzmann entropy.

### worked out example

Say we have a coin with a head and a tail. If the coin is unbiased then the probability of getting the head or the tail is 50% or 0.5. Now what's the entropy of this system? The Shannon entropy is the one that can be used here. For convenience  $\log_2$  will be used for evaluating logarithms here, after all  $\log_2 = \text{const.} \log_{10} = \text{const.} \log_e$ . Here,  $\mu_i = 0.5$  and  $\log_2(0.5) = -1$ , Therefore we have

$$\begin{aligned} H &= -(0.5 \log_2(0.5) + 0.5 \log_2(0.5)) \\ &= -(0.5 \times (-1) + 0.5 \times (-1)) \\ &= 1 \end{aligned}$$

so in this system entropy is 1.

Now take a new system where there is 4 identical object. Since all objects are identical, each have probability  $\mu_i = 1/4$  and  $\log_2(1/4) = -2$ . Then the total entropy of that system is 2. So this is clear that the entropy increases with the increase of the system size. Here system size is determined by the number of particles in it. Now, let's take a non-uniform system, where there are total 8 particles and there are 4 cluster. Cluster 1 has 4 particles and cluster 2 has 2 particles and other 2 cluster of 1 particle each. Probability of getting cluster 1 is  $\mu_1 = 4/8$  and for cluster 2 it is  $\mu_2 = 2/8$  and for other clusters it is  $\mu_3 = \mu_4 = 1/8$ . Now the entropy of the system becomes,

$$\begin{aligned} H &= -\sum_i \mu_i \log_2 \mu_i \\ &= -(4/8 \log_2(4/8) + 2/8 \log_2(2/8) + 1/8 \log_2(1/8) + 1/8 \log_2(1/8)) \\ &= -(-1/2 - 1/2 - 3/8 - 3/8) \\ &= 7/4 \end{aligned}$$

which is less then the entropy of a system where all 8 particles are disconnected, i.e.,  $H = 3$ . So we can see that as the particles are joined together entropy reduces, which is in agreement with the experimental results.

#### 4.5.4 Specific Heat, $C(p, L)$

According to the definition of specific heat in thermodynamics 3.20 we can find specific heat if we know the temperature and entropy. In percolation theory we measure the Shannon

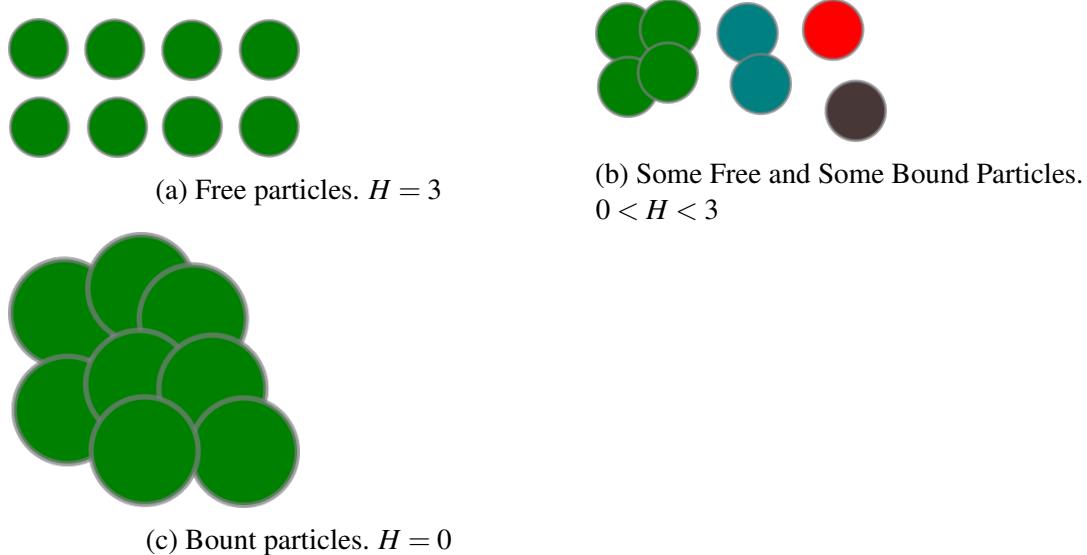


Fig. 4.7 entropy of a system of 8 particles

entropy,  $H(p, L)$ . And we use  $(1 - p)$  as the analogue of temperature which gives

$$C = (1 - p) \frac{dH}{d(1 - p)} \quad (4.6)$$

$$= -(1 - p) \frac{dH}{dp} \quad (4.7)$$

using this definition we can easily find the specific heat of the percolating system. And from specific heat we obtain the critical exponent  $\alpha$ .

#### 4.5.5 Order Parameter, $P(p, L)$

From above discussion we can say that if  $p \geq p_c$  the spanning cluster exists but we still cannot say if a randomly chosen site belongs to the spanning cluster. Therefore we need to quantify the strength of the spanning cluster. Percolation strength or  $P$  is defined as the probability to find the site that belongs to the spanning cluster, meaning randomly pick a site and what is the probability that the selected site will belong to the spanning cluster. And this quantity should depend on occupation probability  $p$  and system size  $L$ . We call it Percolation Strength or sometimes Order Parameter as it describes the measure of order in the system. Order means the likeliness of not getting confused. If all clusters of same size, i.e. identical, we will get confused which cluster we have chosen ( $p = 0$  case) but if there is only one cluster there is no chance of confusion ( $p = 1$  case).

We define the percolation strength as

$$P = \frac{\text{number of sites in the spanning cluster}}{\text{total number of sites in the lattice}} \quad (4.8)$$

But in a system where there are no boundary, e.g. a network or Bethe lattice, the idea of spanning cluster is not valid. Then we use the largest cluster to define the percolation strength

$$P = \frac{\text{number of nodes in the largest cluster}}{\text{total number of nodes in the network}} \quad (4.9)$$

Both definition, though looks a bit different when plotted, gives the same critical exponent  $\beta$ . How to find this exponent is shown in section 5.4.4.

Mathematically

$$P(p, L) = \frac{K}{\sum_i k_i} \quad (4.10)$$

where  $K$  is the size of the spanning cluster and  $k_i$  is the size of the  $i-th$  cluster. Percolation strength is the Order parameter of the system which is the measure of Order of a system. Note that, with periodic boundary condition we have

$$\sum_i k_i = L^2 \quad (4.11)$$

and without periodic boundary condition

$$\sum_i k_i = L(L - 1) \quad (4.12)$$

#### 4.5.6 Susceptibility, $\chi(p, L)$

#### 4.5.7 Mean Cluster Size, $S$

#### 4.5.8 Cluster Size Distribution Function, $n_s$

$$n_s(p_c) \sim s^{-\tau} \quad (4.13)$$



Fig. 4.8 One Dimensional Lattice. Empty ones are white and filled ones are black.

### 4.5.9 Correlation Function, $g(r)$

### 4.5.10 Correlation length, $\xi$

### 4.5.11 Fractal Dimension, $d_f$

## 4.6 Exact Solutions

Percolation problem can be solved exactly in 1 and  $\infty$  dimension. In dimension  $1 < d < \infty$  there is not analytical solution, it can only be solved approximately using simulations. Analytic solution in dimension greater than 1 and less than  $\infty$  is a still to be solved problem. Interestingly, many of the features found in one dimension seem to be valid for higher dimensions too. Thus using the insight of these exact solutions in 1 and  $\infty$  dimension we get a window into the world of phase transitions, scaling and critical exponents.

### 4.6.1 One Dimension

#### Threshold

The simplest lattice one can think of is the one dimensional lattice. It consists of many sites arranged at an equidistant positions along a line. Each site of the lattice can either be occupied with probability  $p$  or remain empty with probability  $1 - p$ . Thus there are only two possible states of each site. A cluster is a group of neighboring occupied sites which contains no empty sites in between. A single empty sites splits a cluster into two clusters. If we find  $n$  successive occupied sites, we say that it forms a cluster of size  $n$ . We want to find the probability at which an infinite cluster appears for the first time, i.e., the critical occupation probability.

Let  $\omega(p, L)$  is the probability that a linear chain of size  $L$  has percolating cluster at probability  $p$ . Note that, if two sites form one cluster, the probability that we find such cluster is  $p^2$ . Similarly if we want a cluster containing  $L$  sites the probability is  $\omega(p, L) = p^L$ , means  $L$  successive sites are occupied independent of each other.

$$\lim_{L \rightarrow \infty} \omega(p, L) = \begin{cases} 0, & \forall p < 1 \\ 1 & \text{only if } p = 1 \end{cases} \quad (4.14)$$

For  $p = 1$  all sites of the lattice are occupied and a percolating cluster spans from  $-\infty$  to  $\infty$  so that each and every site of the lattice belong to the percolating cluster. For  $p < 1$  we will have on the average  $(1 - p)^L$  empty sites. So if  $L \rightarrow \infty$ , we have  $(1 - p)^L \rightarrow \text{const.}$  revealing that there will be at least one, if not more, empty site somewhere in the chain. Which proves that as long as  $p < 1$  there is no spanning cluster. Thus the percolation threshold or the critical occupation probability in one dimension is

$$p_c = 1 \quad (4.15)$$

### Cluster Size

A cluster of size  $s$ , a.k.a.  $s$ -cluster, is formed when  $s$  successive sites are occupied and they are surrounded by two empty sites. Probability of  $s$  successive sites being occupied is  $p^s$  and 2 sites are unoccupied is  $(1 - p)^2$ . Thus the probability of picking a cluster at random that belongs to an  $s$ -cluster is

$$n_s = p^s (1 - p)^s \quad (4.16)$$

$n_s$  is also the number of  $s$ -clusters per lattice site. Note that the state of one particular site is independent of any other sites, that's why we multiply probabilities. Further manipulation of equation 4.16 gives

$$n_s = (1 - p)^2 \exp(s \ln p) = (1 - p)^2 \exp(-s/\xi) \quad (4.17)$$

where  $\xi$  is the correlation length and defined as

$$\xi = -\frac{1}{\ln p} = -\frac{1}{\ln(p_c - (p_c - p))} \sim (p - p_c)^{-1} = (p - p_c)^\nu \quad (4.18)$$

in the limit  $p \rightarrow p_c$  and since  $p_c = 1$ .

### Mean Cluster Size

The probability that an arbitrary site is in  $s$ -cluster is larger by a factor of  $s$ . This site can be any of the sites in the  $s$ -cluster. The probability that an arbitrary chosen site belongs to a cluster of size  $s$  is  $n_s s$ , since  $n_s$  is known to be the number of  $s$ -clusters per lattice site. Every occupied site must belong to one cluster even if it is a cluster of only one site, i.e., a cluster of size unity. The probability that an arbitrary site belongs to a cluster is therefore proportional

to the probably  $p$  that it is occupied.

$$\sum_{s=1}^{\infty} sn_s = \frac{\text{number of total occupied sites}}{\text{number of total lattice sites}} = p \quad (4.19)$$

A quick check of the validity of equation 4.19 can be performed using equation 4.16.

$$\begin{aligned} \sum_{s=1}^{\infty} sn_s &= \sum_s s(1-p)^2 p^s \\ &= (1-p)^2 \sum_s sp^s \\ &= (1-p)^2 \sum_s p \frac{d(p^s)}{dp} \\ &= (1-p)^2 p \frac{d \sum_s p^s}{dp} \\ &= (1-p)^2 p \frac{dp(1-p)^{-1}}{dp} \\ &= (1-p)^2 p \left( \frac{1}{1-p} + \frac{p}{(1-p)^2} \right) \\ &= p \end{aligned} \quad (4.20)$$

Here we have used the series sum 4.21.

$$\begin{aligned} \sum_s p^s &= p + p^2 + p^3 + \dots \\ &= p(1 + p + p^2 + \dots) \\ &= p(1 - p)^{-1} \end{aligned} \quad (4.21)$$

An important question one can ask is that what is average size of the cluster that we are hitting. Since  $n_s s$  is the probability that an arbitrary site belongs to an  $s$ -cluster and  $\sum_s n_s s$  is the probability that it belongs to any cluster. Thus we define  $w_s$  as

$$w_s = \frac{n_s s}{\sum_s n_s s} \quad (4.22)$$

$w_s$  is the probability that the cluster to which an arbitrary occupied site belongs contain exactly  $s$  sites. The average cluster size  $S$  is therefore

$$S = \sum_s w_s s \quad (4.23)$$

This equation is very much similar to

$$\bar{x} = \int xp(x)dx \quad (4.24)$$

using equation 4.22 we get

$$\begin{aligned} S &= \frac{\sum_s n_s s^2}{\sum_s n_s s} \\ &= \sum_{s=1}^{\infty} \frac{s^2 n_s}{p} \\ &= \frac{(1-p)^2}{p} \sum_{s=1}^{\infty} s^2 p^s \\ &= \frac{(1-p)^2}{p} \left( p \frac{d}{dp} \right)^2 \left( \sum_{s=1}^{\infty} p^s \right) \\ &= \frac{(1-p)^2}{p} \left( p \frac{d}{dp} \right)^2 (p(1-p)^{-1}) \\ &= p(1-p)^2 \frac{d^2}{dp^2} (p(1-p)^{-1}) \\ &= p(1-p)^2 \frac{d}{dp} (1-p)^{-2} \\ &= p(1-p)^2 2(1-p)^{-3} \\ &= \frac{2p}{1-p} \\ &= \frac{1+p}{1-p} \end{aligned} \quad (4.25)$$

we can write

$$S(p) = \frac{1-p}{1+p} = \frac{p_c + p}{p_c - p} \quad (4.26)$$

using the fact that  $p_c = 1$  in 1D lattice. This equation reveals that the mean cluster size diverges for  $p \rightarrow p_c$  where the minus sign signifies that we are approaching from below  $p_c$ . This is in sharp contrast with higher dimensional ones where we can approach to  $p_c$  from either end while in one dimension we cannot have access to the state  $p > p_c$ . We thus find the mean cluster size diverges following power law as we have [46]

$$S(p) \sim (p_c - p)^{-1} \quad (4.27)$$

We encounter the similar behaviour in the higher dimensions also.

### Correlation Function and Correlation Length

The correlation function or pair connectivity  $g(r)$  is the probability that a site at position  $r$  from an occupied site belongs to the same finite cluster. We are not including the contribution of the infinite cluster. This is valid infinite cluster does not exists as long as  $p < 1$ . Let  $r = 0$  then  $g(r = 0) = 1$  since the site at  $r = 0$  is the selected occupied site by definition. For 1D case a site at  $r$  to be occupied and belongs to the same finite cluster, we will need  $r$  subsequent sites and the probability of getting this is  $p^r$ . Therefore

$$g(r) = p^r \quad (4.28)$$

It can also be expressed in terms of correlation length  $\xi$

$$g(r) = \exp(\ln(p^r)) = \exp(-r/\xi) \quad (4.29)$$

where  $\xi$  is the correlation length 4.18.

Now that we have correlation function, we can define mean cluster size in terms of it

$$S = 1 + \sum_{r=1}^{\infty} g(r) \quad (4.30)$$

At this point it is evident that the cutoff cluster size  $s_\xi$ , mean cluster size  $S(p)$ , and correlation length  $\xi$  diverges at the percolation threshold. The divergence has the form of a simple power law of the distance from the critical occupation probability. In higher dimensional percolation problem this observation is also valid.

#### 4.6.2 Infinite Dimension

Apart from one dimension percolation problem can be solved in infinite dimension. For this we need a suitable playground such as Bethe lattice. Bethe lattice lattice is a special type of lattice where each site has  $z$  neighbors and each branch gives rise to  $(z - 1)$  other branches. Figure 4.9 shows the Bethe lattice for  $z = 3$ . Note that for  $z = 2$  we have nothing but the one dimensional lattice.

#### Properties of infinite dimensional object

For a 3D object the surface area is has dimension to  $L^2$  and volume as dimension  $L^3$ . The same pattern is true of object in any dimension. If we denote area by  $A$  and volume by  $V$  for

any dimension we have

$$A \propto V^{1-1/d} \quad (4.31)$$

now as  $d \rightarrow \infty$  we have

$$A \propto V \quad (4.32)$$

Therefore if we find that the area of any object is proportional to its volume we can say it is an infinite dimensional object.

### Bethe Lattice

In order to construct Bethe lattice for any  $z$  we start with a central point which will be connected to  $z$  sites. For example if  $z = 3$  then we will have a central site connected to 3 sites by a branch and when we go to next layer each branch will be divided to 2 more branches and this process will be continued up to  $r$  layers 4.9. Only at the surface of the lattice, where the branching is stopped, is only one bond or branch connecting the surface site to the interior. There is only open loops in this structure, which means that if we never change direction always reach new site if we never go back. Number of sites in the Bethe lattice increases exponentially with the distance from the origin, whereas in any  $d$ -dimensional lattice structure it would increase with distance  $d$ . In the case of Bethe lattice with  $z = 3$ , the origin is surrounded by a shell of three sites ("first generation"), in the second shell we have six sites followed by a third generation of twelve sites, etc. After  $r$  generation the total number of sites in the Bethe lattice is

$$1 + 3 \times (1 + 2 + \dots + 2^{r-1}) = 3 \cdot 2^r - 2 \quad (4.33)$$

The number  $3 \times 2^{r-1}$  is the number of sites at the surface. Here we have used the following finite series sum

$$1 + 2 + 2^2 + 2^3 + \dots + 2^r = 2^{r+1} - 1 \quad (4.34)$$

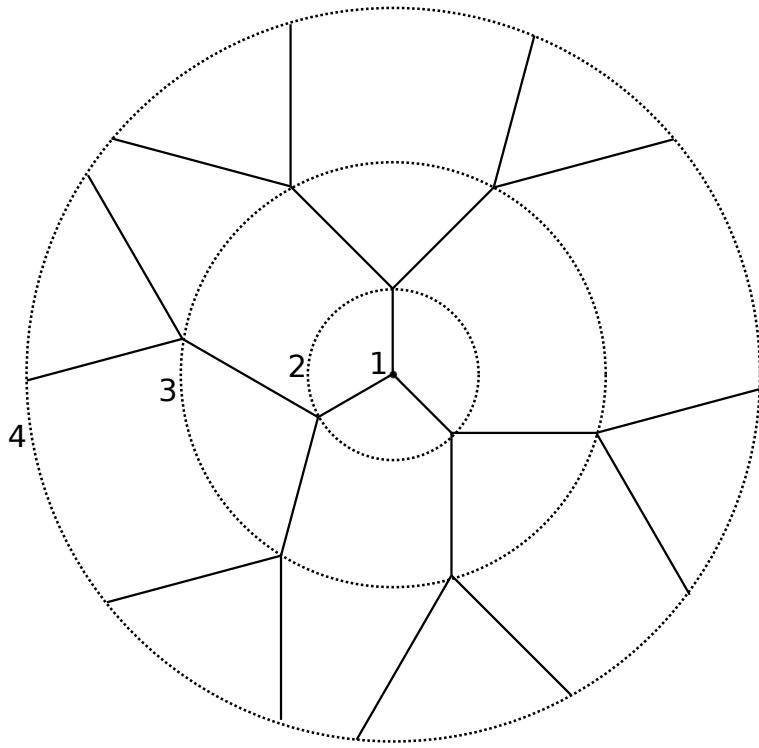
And if we measure the surface to volume ratio we get

$$\frac{A}{V} = \frac{\text{number of sites in the surface}}{\text{total number of sites}} = \frac{3 \times 2^{r-1}}{3 \times 2^r - 2} \quad (4.35)$$

as  $r \rightarrow \infty$  we get

$$\frac{A}{V} \sim \frac{3 \times 2^{r-1}}{3 \times 2^r} = \frac{1}{2} = \text{constant} \quad (4.36)$$

Therefore Bethe lattice is indeed an infinite dimensional lattice.

Fig. 4.9 Bethe Lattice for  $z = 3$ 

### Percolation Threshold

Percolation threshold of Bethe lattice is the occupation probability at which an infinite cluster appears for the first time. To find it we start walking from the origin and after one step we have  $z - 1$  new bonds that is connected to  $z - 1$  new sites in those direction. On the average there will be  $(z - 1)p$  occupied sites. And for each site there will be another  $z - 1$  branch and those bonds are connected to  $(z - 1)p$  sites on the average and so on. After  $r$  step we will have an infinite cluster at probability  $((z - 1)p)^r$ . Since  $r \rightarrow \infty$  we have  $((z - 1)p)^r = 0$  if  $(z - 1)p < 1$ . Thus we choose  $(z - 1)p = 1$  so that we will get an infinite cluster. That lead us to the desired critical occupation probability

$$(z - 1)p_c = 1$$

$$p_c = \frac{1}{z - 1} \quad (4.37)$$

For  $z = 3$  we have  $p_c = 1/2$ .

## Percolation Strength

Percolation strength of an infinite cluster is the probability of any arbitrary site to be the part of the infinite cluster. For the sake of calculation, for  $p > p_c$  in the Bethe lattice, we introduce a new quantity  $Q$  as the probability that an arbitrary site is note connected to the infinite cluster through a fixed branch originating from this site. Restricting ourselves to the lattice with  $z = 3$  and using basic probability theory, the strength

$$P(p) = p(1 - Q^3) \quad (4.38)$$

Here  $p$  is the probability that the site is occupied and  $(1 - Q^3)$  is the probability that at least one branch is connected to infinity.

The probability that the two subbranches which start at the neighbor are not both leading infinity is  $Q^2$ . The quantity  $pQ^2$  is the probability that this neighbor is occupied but not connected to infinity by any of its two subbranches. This neighbor is empty with probability  $(1 - p)$ , in which case even well connected subbranches do not help it. This gives us,

$$Q = (1 - p) + pQ^2 \quad (4.39)$$

This is the probability that this fixed branch does not lead to infinity, either because the connection is already broken at the first neighbor, or because later something is missing in the subbranch. So the solution of this quadratic equation is

$$Q = 1, \frac{1-p}{p} \quad (4.40)$$

For  $z$  neighbors, in general we have

$$Q = 1, 1 - \frac{2p(z-1) - 2}{p(z-1)(z-2)} \quad (4.41)$$

for  $p < p_c$ , there are no infinite clusters, fo with probability 1 there are no connection to infinity. Now we use Taylor expansion for  $P(p)$  around  $p = p_c = 1/2$

$$P(p) = \begin{cases} 0 & \text{for } p < p_c \\ p \left(1 - \left(\frac{1-p}{p}\right)^3\right) & \text{for } p \geq p_c \end{cases} \quad (4.42)$$

Let,

$$f(p) = \left(\frac{(1-p)}{p}\right)^3 \quad (4.43)$$

Then

$$\begin{aligned} f'(p) &= -3p^{-4}(1-p)^3 - 3p^{-3}(1-p)^2 \\ &= -\frac{3}{p} \left( \frac{(1-p)}{p} \right)^3 - \frac{3}{p} \left( \frac{(1-p)}{p} \right)^2 \end{aligned} \quad (4.44)$$

$$P(p) = P(p_c) + (p - p_c)P'(p_c) + \dots \quad (4.45)$$

$$= 0 + (p - p_c)(1 - f(p_c) - pf'(p_c)) + \dots \quad (4.46)$$

$$= 6(p - p_c) + \dots \quad (4.47)$$

$$(4.48)$$

Therefore we get

$$P(p) \propto (p - p_c) \text{ for } p \rightarrow p_c^+ \quad (4.49)$$

the critical exponent  $\beta$  is defined by

$$P(p) \propto (p - p_c)^\beta \quad (4.50)$$

Thus in Bethe lattice  $\beta = 1$ .

### Mean Cluster Size

In case of Bethe lattice the mean cluster size is defined as the average number of sites to which the origin belongs. Let  $T$  be the mean cluster size for one branch, that is the average number of sites to which the origin is connected and which belongs to one branch. Again, subbranches have the same mean cluster  $T$  as the branch itself. If the neighbor is empty the cluster size for this branch is zero. If the neighbor is occupied, it contributes its own mass to the cluster which is unity and adds the mass  $T$  for each of its two subbranches. Thus,

$$T = (1 - p) \times 0 + p(1 + 2T) \quad (4.51)$$

Solving this we get

$$T = \frac{p}{1 - 2p} \quad (4.52)$$

for  $p < p_c$ .

The total cluster size is zero if the origin is empty and  $(1 + 3T)$  if the origin is occupied.

Therefore the mean cluster size  $S(p)$  is

$$S(p) = 1 + 3T \quad (4.53)$$

$$\begin{aligned} &= \frac{1+p}{1-2p} \\ &= \frac{1+p}{2(p_c-p)} \\ &= \frac{1+p}{2}(p_c-p)^{-1} \end{aligned} \quad (4.54)$$

Thus the critical exponent  $\gamma = 1$  for Bethe lattice. This is the exact result for mean cluster size and we notice that it diverges for  $p \rightarrow p_c$ .

### Correlation Function and Correlation Length

The radial correlation function  $g(r)$  is the average number of occupied sites within the same cluster at a distance  $r$  from an arbitrary occupied site. The probability that a site at distance  $r$  from the origin is occupied and the sites in between are occupied too is equal to  $p^r$ . Now if we think about a shell of radius  $r$  then the number of all the sites enclosed by this shell is  $z(z-1)^{r-1}$ . Thus

$$g(r) = z(z-1)^{r-1} p^r \quad (4.55)$$

$$= \frac{z}{z-1} (p(z-1))^r \quad (4.56)$$

$$= \frac{z}{z-1} \exp[\log[p(z-1)]] \quad (4.57)$$

The value of percolation threshold for Bethe lattice can be found by analyzing the behaviour of the correlation function at large distances, i.e. at  $r \rightarrow \infty$ . For  $p(z-1) < 1$ ,  $g(r)$  decreases exponentially, on the other hand for  $p(z-1) > 1$ , the correlation function diverges which signifies the existence of an infinite cluster. Mathematical treatment yields the correlation length from 4.18

$$\begin{aligned} \xi &= \frac{-1}{\log[p(1-z)]} \\ &= \frac{-1}{\log(p/p_c)} \\ &= (p - p_c)^{-1} \end{aligned} \quad (4.58)$$

as  $p$  approaches  $p_c$ , that is  $v = 1$ .

Clearly the 1D lattice and Bethe lattice exhibits power law while we approach a critical value which suggests the same phenomena in other variants of such problems.

## 4.7 Algorithm

In the classical Hosen-Kopelman algorithm for percolation model [47], one first choose an occupation probability  $p$  and then generate a number  $r$  for each site of the lattice. The site is occupied if  $r \leq p$  and remain empty if  $r > p$ . One therefore create an entire new state of a given lattice size for every different value of  $p$ . Note that the number of occupied sites  $n$  for a given  $p$  may vary in each realization. However, the expected or ensemble average over  $M$  experiments will give  $n = pM$  in the limit  $M \rightarrow \infty$ . Thus the number of occupied bonds or sites is also a measure of  $p$ . Using this idea Ziff and Newman [65] proposed an algorithm which generate states for each value of  $n$  from zero up to some maximum value  $n = L^2$  for site percolation on  $L \times L$  square lattice for instance. In this way, one can save some effort by noticing the fact that a new state with  $n + 1$  occupied sites or bonds can be created by adding one extra randomly chosen site or bond to the state containing  $n$  sites or bonds. The first step of their algorithm is to decide an order in which the bonds or sites are to be occupied. That is, every attempt to occupy a bond/site is successful.

## 4.8 Relation of Phase Transition with Percolation

## 4.9 Application

Ever since the percolation phenomena was discovered it has been extensively used in wide range of science. Some of it is discussed here. Though limited, this discussion gives us the significance of percolation and enables us to appreciate it.

### 4.9.1 Epidemiology

Many diseases spread through human populations via close physical interactions. The interpersonal contact patterns that underlie disease transmission can naturally be thought to form a network, where links join individuals who interact with each other. During an outbreak, disease then spreads along these links. All epidemiological models make assumptions about the underlying network of interactions, often without explicitly stating them.

Percolation has long been an important tool in infectious disease epidemiology [13]. L Meyers gave a wonderful insight of contact network epidemiology, a more powerful approach that applies bond percolation on random graphs to model the spread of infectious disease through heterogeneous populations [61].

Before that in 2000, Moore et al. studied some simple models of disease transmission on small-world networks [63]. They showed that the resulting models display epidemic behavior when the infection or transmission probability rises above the threshold for site or bond percolation on the network, and they gave exact solutions for the position of this threshold in a variety of cases.

Alessandro Vespignani and his team studied wifi networks and malware epidemiology. They developed an epidemiological model that takes into consideration prevalent security flaws on these routers. They simulated spread of such a contagion on real-world data for georeferenced wireless routers [48].

Sander et al. considered a spatial model related to bond percolation for the spread of a disease that includes variation in the susceptibility to infection. They worked on a lattice with random bond strengths and showed that with strong heterogeneity, i.e. a wide range of variation of susceptibility, patchiness in the spread of the epidemic is very likely, and the criterion for epidemic outbreak depends strongly on the heterogeneity. Their results were qualitatively different from those of standard models in epidemiology, but correspond to real effects [74].

After the birth of scale free network the study of epidemics has become even more popular among statistical physicists.

#### **4.9.2 Neural Network and Cognitive Psychology**

Percolation is used to understand the way activation and diffusion of neural activity occur within neural networks [42]. Iian Breskin's team in 2006 studied living neural networks by measuring the neurons' response to a global electrical stimulation. They showed through analysis that neural connectivity is lowered by reducing the synaptic strength, chemically blocking neurotransmitter receptors. They used a graph-theoretic approach to show that the connectivity undergoes a percolation transition. This occurs as the giant component disintegrates, characterized by a power law with an exponent  $\beta \simeq 0.65$  [20].

It is easiest to understand percolation theory by explaining its use in epidemiology [63]. Individuals that are infected with a disease can spread it knowingly or unknowingly via social or physical interaction. It's easy to say that the more social a person is the more it is likely that he will spread more disease than the unsocial one. Therefore some factors, e.g.,

occupation, size of social circle etc, influence the rate of infection.

Now, if one were to think of neurons as the individuals and synaptic connections as the social bonds between people, then one can determine how easily messages between neurons will spread [42] When a neuron fires, the message is transmitted along all synaptic connections to other neurons until it can no longer continue.

Synaptic connections are considered either open or closed (like a social or unsocial person) and messages will flow along any and all open connections until they can go no further. Just like occupation and sociability play a key role in the spread of disease, so does the number of neurons, synaptic plasticity [49] and long-term potentiation when talking about neural percolation. Percolating clusters are a single large group of neurons that are all connected by open bonds and take up the majority of the network. Any signals that originate at any point within the percolating cluster will have a great impact and diffusion across the network than signals that original outside of the cluster. This is much like how a teacher is more likely to spread an infection to a whole community through contact with the students and subsequently with the families than an isolated businessman that works from home.

### 4.9.3 Ferromagnetism

One of the most studied phase transition phenomena of physics is that para-magnetic to ferromagnetic transition. The magnetic spins of a magnetic material, e.g., nickel, interact with each other: the energy is lower if the two spins on adjacent nickel atoms are parallel than if they are anti-parallel. This lower energy tends to cause the spins to be parallel and below a temperature called the Curie temperature,  $T_c$ , most of the spins in the nickel are parallel, their magnetic moments then add up constructively and the piece of nickel has a net magnetic moment: it is a ferromagnetic. Above the Curie temperature on average half the spins point in one direction and the other half in the opposite direction. Then their magnetic moments cancels out, and the nickel has no net magnetic moment. It is then a para-magnet. Thus at  $T_c$  the nickel goes from having no magnetic moment to having a magnetic moment. This is a sudden qualitative change and when this happens we say that a phase transition has occurred. Here the phase transition occurs when the magnetic moment goes from zero to non-zero. It is from the para-magnetic phase to the ferromagnetic phase. So, this critical value  $T_c$  of the temperature marking the borderline between the existence and non-existence of so called spontaneous magnetization. A standard mathematical model for this phenomenon is the Ising model [26]. It turns out that there are two parameters which specify the conditional probabilities: the 'external magnetic field'  $h$ , and the strength  $J$  of interaction between neighbors. If  $J = 0$ , the states of different vertices are independent,



Fig. 4.10 NGC 4414, a typical spiral galaxy in the constellation Coma Berenices, is about 55,000 light-years in diameter and approximately 60 million light-years away from Earth.

and the process is equivalent to site percolation [30]. The relationship between the Ising model and bond percolation is rather strong. It turns out that they are linked via a type of 'generalized percolation' called the other considerations [29, 78]. Studying the random-cluster model, one can obtain conclusions valid simultaneously for percolation and the Ising model. This discovery was made in 1970 by Fortuin and Kastelyn [41, 39, 40], and it has greatly influenced part of the current view of disordered physical systems

#### 4.9.4 Cosmology

We know mass distribution and motions of the components of a galaxy are determined by gravity, it has not been clear what is responsible for the striking morphology of a spiral galaxy such as shown in the figure. The spiral arms extend over 16,000 parsecs (1 parsecs equals 3.26 light-years), and the traditional view is that it is necessary to have a long-range interaction like gravity, which interacts with objects that have mass, to create such long-range order. However, in condensed matter physics it is well known that long-range order can be induced by a short-range interaction, and this is a characteristic feature of a continuous phase transition [48].

The structural features of spiral galaxies arise from a percolation phase transition that underlies the phenomenon of propagating star formation. According to this view, the

appearance of spiral arms is a consequence of the differential rotation of the galaxy and the characteristic divergence of correlation lengths for continuous phase transitions.

Other structural properties of spiral galaxies, such as the distribution of the gaseous components and the luminosity, arise directly from a feedback mechanism that pins the star formation rate close to the critical point of the phase transition. At least for some galaxies, morphological and other features are already fixed by general properties of phase transitions, irrespective of detailed dynamic or other considerations [78, 29].

Definitions of some quantities used in percolation. Basic Algorithm

#### 4.9.5 Square Lattice

A square lattice is an ideal playground for percolation. If it has length  $L$  then number of sites in the lattice is  $L^2$  and number of bonds in the lattice is  $2L^2$ . All sites are equally separated from each other at a certain distance. And all sites have exactly four neighbor. Since with the periodic boundary condition the sites in the left edge are connected with the sites in the right edge and same rule for sites in the top and bottom edges. If the sites are densely spaced the experiment will be accurate, meaning, the larger the size of the lattice the accurate the results will be. It implies that a lattice of infinite size should be used which is practically impossible. The simple solution to this problem is to use number of large lengths, (say  $L = \{L_1, L_2, \dots, L_n\}$ , where  $n$  is a finite number and  $L_1 < L_2 < \dots < L_n$ ), and extrapolate the results for infinite lattice. The visual structure of the square lattice is as follows 4.11. This is an empty lattice structure. Filled circles are for occupied site and filled bonds are for occupied bonds 4.12.

#### 4.9.6 Site Percolation

The algorithm for site percolation is as follows,

1. take a square lattice of length  $L$ .
2. fill all  $2L^2$  bonds initially.
3. occupy a randomly chosen site and it will join some clusters.
4. each time a site is occupied, it will get connected to four neighboring bonds and will form a cluster of size 4. Note that we define cluster size by number of bonds in it.
5. if a site is occupied and right next to it there is another occupied site and a cluster of size 7 will be formed.

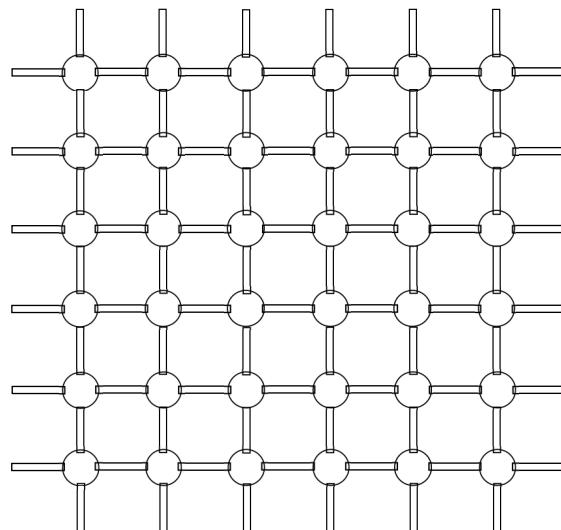


Fig. 4.11 Square Lattice (empty) of length 6



Fig. 4.12 Site and Bond symbol (empty and occupied).

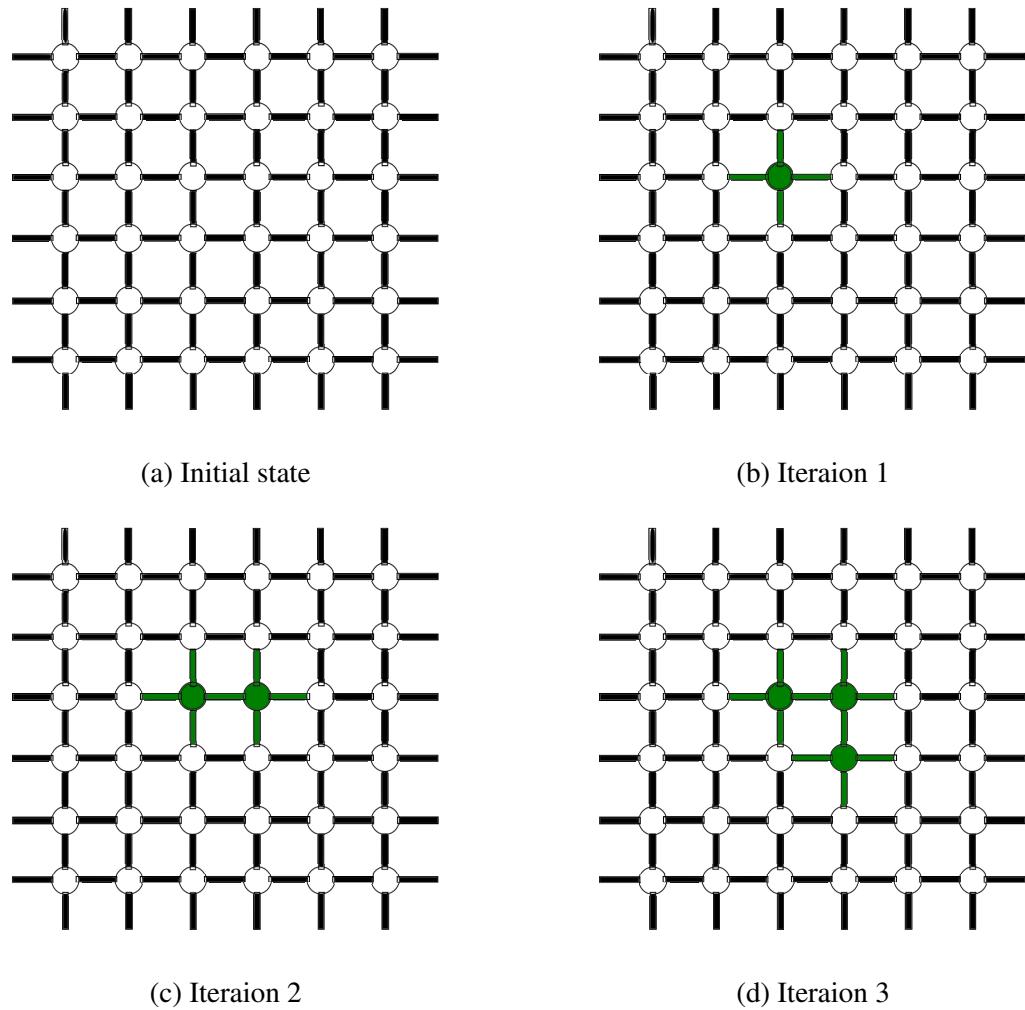


Fig. 4.13 Growth of a Cluster in Site Percolation on square Lattice

6. this process is repeated until all the sites are occupied and only one cluster remains

The formation of cluster is visualized in the figure 4.13.

#### 4.9.7 Bond Percolation

The algorithm for bond percolation is as follows,

1. take a square lattice of length  $L$ .
2. fill all  $L^2$  the sites initially.
3. occupy a randomly chosen bond and it will join some clusters.

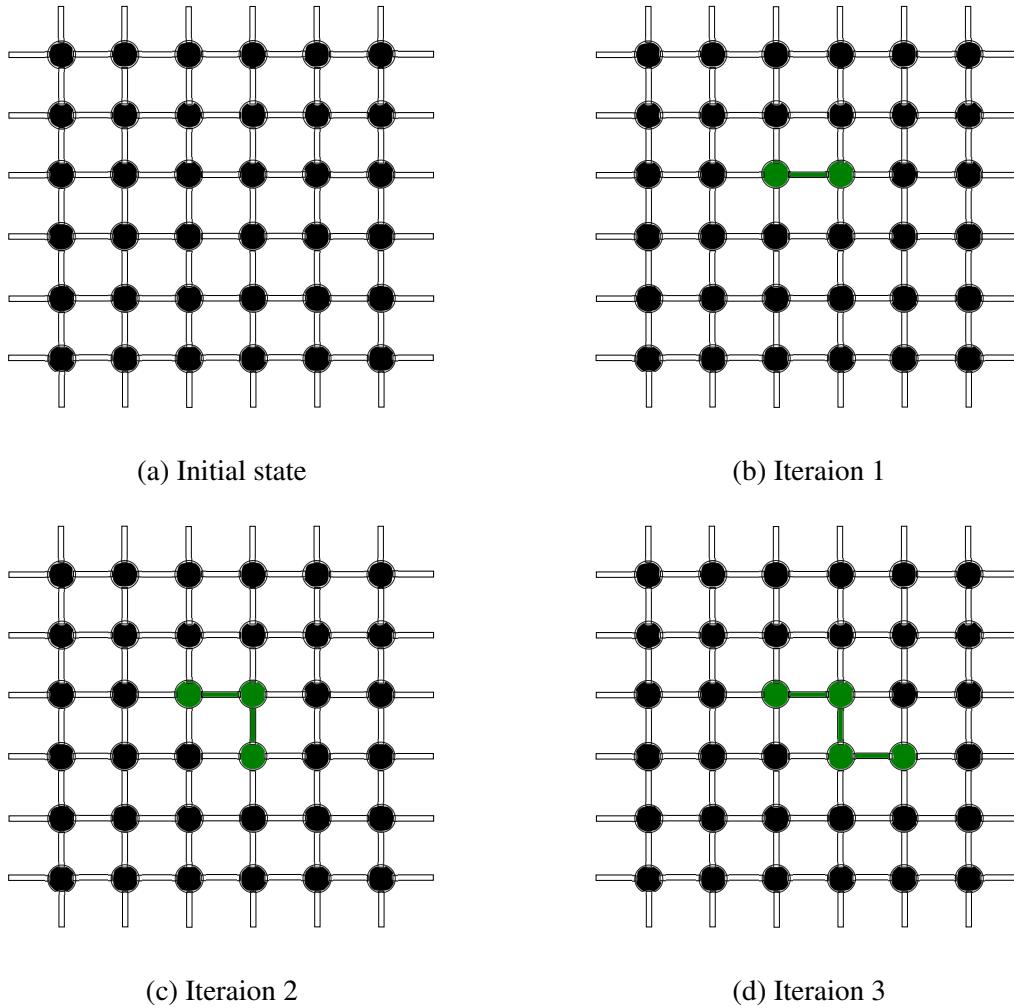


Fig. 4.14 Growth of a Cluster in Bond Percolation on square Lattice

4. each time a bond is occupied, it will get connected to two neighboring sites and will form a cluster of size 2. Here cluster size by number of sites in it.
5. if a bond is occupied and right next to it there is another occupied bond and they are connected by a site and a cluster of size 3 will be formed.
6. this process is repeated until all the bonds are occupied and only one cluster remains

The formation of cluster is visualized in the figure 4.14.

# Chapter 5

## Ballistic Deposition on Square Lattice

We investigate percolation by random sequential ballistic deposition (RSBD) on a square lattice with interaction range upto second nearest neighbors. The critical points  $p_c$  and all the necessary critical exponents  $\alpha, \beta, \gamma, \nu$  etc. are obtained numerically for each range of interactions. Like in its thermal counterpart, we find that the critical exponents of RSBD depend on the range of interactions and for a given range of interaction they obey the Rushbrooke inequality. Besides, we obtain the exponent  $\tau$  which characterizes the cluster size distribution function  $n_s(p_c) \sim s^{-\tau}$  [4.5.8] and the fractal dimension  $d_f$  that characterizes the spanning cluster at  $p_c$  [4.5.11]. Our results suggest that the RSBD for each range of interaction belong to a new universality class which is in sharp contrast to earlier results of the only work that exist on RSBD.

Imagine a spherical shaped object, say marble, is thrown on top of a 2D square lattice structure. First possible scenario is that the marble will be deposited in the first encountered site in the lattice. Now if the first encountered site is not empty, the possible scenario is that the marble will go in any of the four direction,  $+x, -x, +y$  or  $-y$ , assuming no other direction in between is allowed in the lattice. Now if the first neighbor is not empty then the marble will continue to go on in the previously selected direction and choose the next neighbor. This is the main theme of this thesis.

In our experiment, we occupy a randomly chosen site if it is empty else we select one of its four neighbor to occupy if it is empty else select next neighbor in that direction and occupy that site if it is empty else ignore current iteration and choose another site randomly. This process is repeated until there is no empty site in the lattice. We call this process the ballistic deposition on the square lattice. We introduce 1st and 2nd nearest neighbor interaction in this way.

## 5.1 Site Percolation Redefined

## 5.2 Structure and Algorithm

Random percolation (RP) model can also be seen as a random sequential adsorption (RSA) process of particles on a given substrate to form monolayers of clusters of complex shape and structures. In RSA, a site is first picked at random and it is occupied if it is empty and the trial attempt is rejected if it is already occupied. We shall first show that this process too reproduce all the existing results of the CRP including the  $p_c$  value. We can modify the rejection criterion. First, we assume that the adsorbing particles are hard sphere and impenetrable. Then we assume that if a particle fall onto an already adsorbed particle it is not straightaway rejected. Instead, it is allowed to roll down over the already deposited particle to one of its nearest neighbours at random following the steepest descent path. The particle is then adsorbed permanently if the nearest neighbour is empty else the trial attempt is rejected. This is known as the ballistic deposition (BD) model for  $l = 1$ . We also consider the case that if the nearest neighbour is occupied then the incoming particle attempt to push the neighbour to its next neighbour site along the same line to make room for itself. However, the trial attempt of pushing the neighbour is successful if the next neibouring site along the same line is empty else the trial attempt is discarded. We regard it as BD model for  $l = 2$  while the classical percolation correspond to BD model with  $l = 0$ . Our primery goal is to prove that the critical exponents of percolation changes as changes as we increase the range of interaction like we find in its thermal counterpart. We numerically find the various necessary critical exponents and find that BD for each different range of interaction belong to different universality class and each universality class obeys the Rusbrooke inequality.

Percolation is all about configuration of clusters of deposited particles and the investigation of the emergence of a large-scale connected path created by clusters formed by contiguous diposoted particles. We use extensive Monte Carlo simulation on a square lattice with the usual periodic boundary condition to study site percolation according to RSBD rule.

The algorithm of the percolation by RSBD can be described as follows. We first label all the sites row by row from left to right starting from the top left corner. That is, we first label the first row from left to right as  $i = 1, 2, \dots, L$ , the second row again from left to right as  $i = L + 1, L + 2, \dots, 2L$  and we continue this till we reach the bottom row which we label as  $i = (L - 1)L + 1, \dots, L^2$ . Then at each step we pick a discrete random number  $R$  from  $1, 2, \dots, L^2 - 1, L^2$  using uniform random number generator and check if the site it represents

is already occupied or not. If it is empty we occupy it straightaway and move on to the next step. Else we pick one of its neighbours at random. The second attempt in the same step, that mimic the roll over mechanism, is successful if the neighbour it picks is empty and if not the trial attempt to deposit is rejected permanently and we move on to the next step anyway. This process is repeated over and over again till we want it to stop. We call it RSBD of degree one. We also consider the case of RSBS of degree two where the trial attempt is made to occupy the second nearest neighbour too. In this case if the incoming particle that fall onto an already occupied site and find its neighbour is occupied too but the next nearest neighbour site is empty then the neighbour move to the empty site to make space for the incoming particle to be deposited there.

1. take a square lattice of length  $L$ .
2. choose a site randomly
3. if the chosen site is empty then occupy it else choose one of the four neighbor randomly
4. if the chosen neighbor is empty then occupy it else choose second nearest neighbor in that direction
5. if the second nearest neighbor is empty then occupy the site else ignore this step

## 5.3 Finite Size Scaling in Percolation Theory

## 5.4 Finding Numerical Values

### 5.4.1 Critical occupation probability, $p_c$

When we occupy sites of the square lattice, initially, there are only cluster of size 1. As we keep occupying different size of clusters starts to appear. At a certain point a special cluster appears which spans the entire lattice. We call this cluster the *Spanning Cluster*. It should be noted that the spanning cluster is a special property of the lattice and the appearance of the spanning cluster is the result of phase transition. At the point where the spanning cluster first appears is called the critical point and denoted by  $p_c$ , meaning the occupation probability at the critical point. The figure 5.1 shows the graph of the spanning probability,  $w(p,L)$ , versus the occupation probability  $p$  for different interactions ( $L_0, L_1, L_2$ ) for different lengths. We have found that  $p_c$  is 0.5927, 0.5782, 0.5701 for  $L_0, L_1, L_2$  respectively. Thus for long range interaction  $p_c$  is smaller than for short range interaction. The quantity  $w(p,L)$  is called

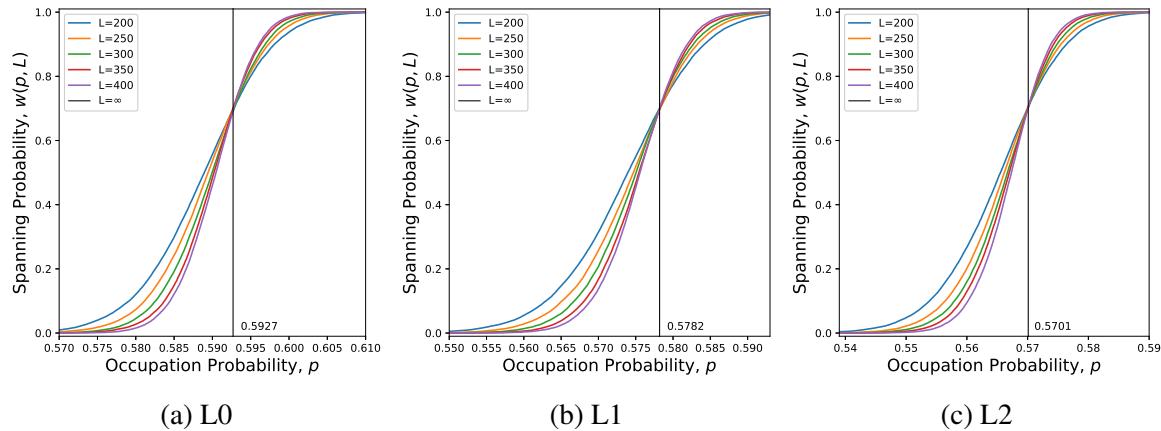


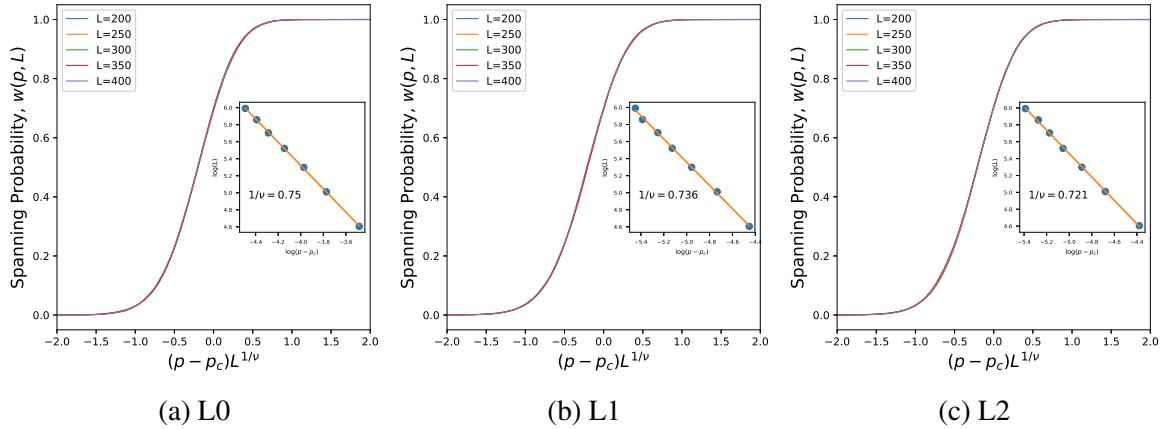
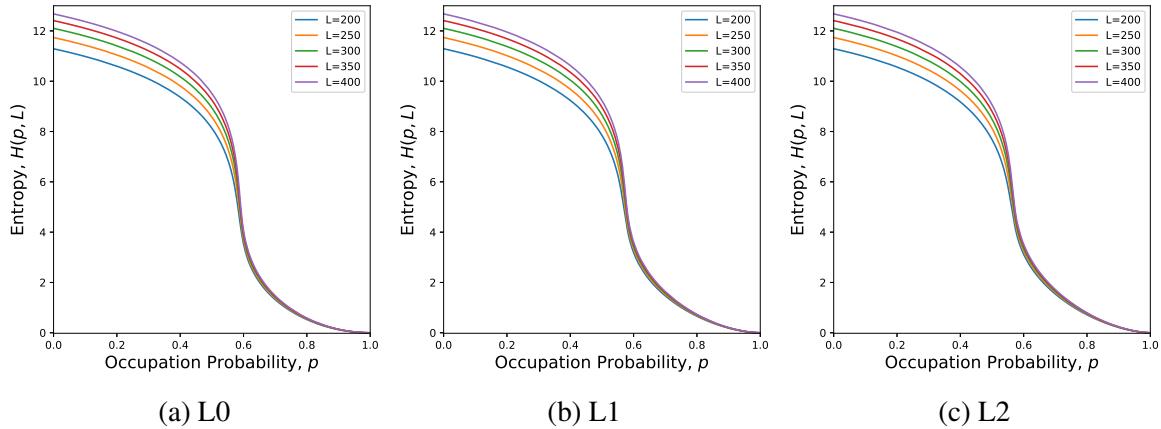
Fig. 5.1 Spanning Probability,  $w(p, L)$  vs Occupation Probability,  $p$

The spanning probability in a non periodic case and wrapping probability in a periodic case. The question is how do we find the wrapping probability, given that we have a list of  $p_c$  for different length. Note that for a certain length  $L$ , in each realization the  $p_c$  is not exact value, instead it is a range of values that contains the  $p_c$ . For example, say we have a lattice of length 200 and for that we will have different  $p_c$  at each realization. After an infinite number of experimentation we will have to take an average and that will give the exact value of  $p_c$ . But since it is practically impossible, we can find  $p_{c_{avg}}$  for different lattice size then from the graph we can extrapolate the exact value of  $p_c$  for infinite lattice. Here  $p_{c_{avg}}$  is calculated from a finite set of  $p_c$  for a certain length. This process is not good enough. Since it requires  $p_{c_{avg}}$ 's for a number of lattice size which is very costly to obtain. But from the data, list of  $p_c$ 's for different length, we can find the cumulative frequency distribution. It is astonishing that for all length the wrapping probability coincide at a specific point. This implies that if we could have an infinite system, the wrapping probability for that system would have gone through this same point. This means we got our critical point,  $p_c$ , as the intersection of the  $w(p, L)$  for different lengths.

### 5.4.2 Spanning Probability and finding $1/v$

From the figure 5.1 we can see that, as we increase the length of the lattice the wrapping probability  $w(p, L)$  moves closer and closer to the critical point. And if we draw a horizontal line at a certain height, say  $y = 0.1$ , and find the intersection of this line with  $w(p, L)$  for each length and note the  $p$  and if we plot  $\log(L)$  vs  $\log(p - p_c)$  we get the slope  $1/\nu$ . If we now use the finite size scaling (FSS) hypothesis

$$w = (p - p_c)L^{1/\nu} \quad (5.1)$$

Fig. 5.2  $w(p,L)$  vs  $(p - p_c)L^{1/\nu}$ Fig. 5.3 Entropy,  $H(p,L)$  vs Occupation Probability,  $p$ 

we get a very good data collapse for  $L0, L1, L2$  and got  $1/\nu$  as  $0.75, 0.736, 0.721$  respectively which is shown in figure 5.2. That is they are self-similar ??.

### 5.4.3 Entropy, Specific Heat and finding $\alpha$

For any phase transition model the entropy is crucial. In percolation theory we use Shannon Entropy [81]. Using the definition ?? we get the figure 5.3 And since the specific heat  $C(p,L)$  is nothing but the derivative of entropy, by simply differentiating entropy we get the specific heat (although we need to perform convolution?? in order to get a smooth curve) shown in figure 5.4. From specific heat we can find the exponent  $\alpha$ . To do this first we need to scale the  $x$ -values of the specific heat data using the exponent  $1/\nu$  obtained from 5.4.2 and get the graph as in figure ???. From this graph we will not the height of each line and call it  $C_h$ . Since each line corresponds to a specific length  $L$  we can plot  $\log(C_h)$  versus  $\log(L)$  and the

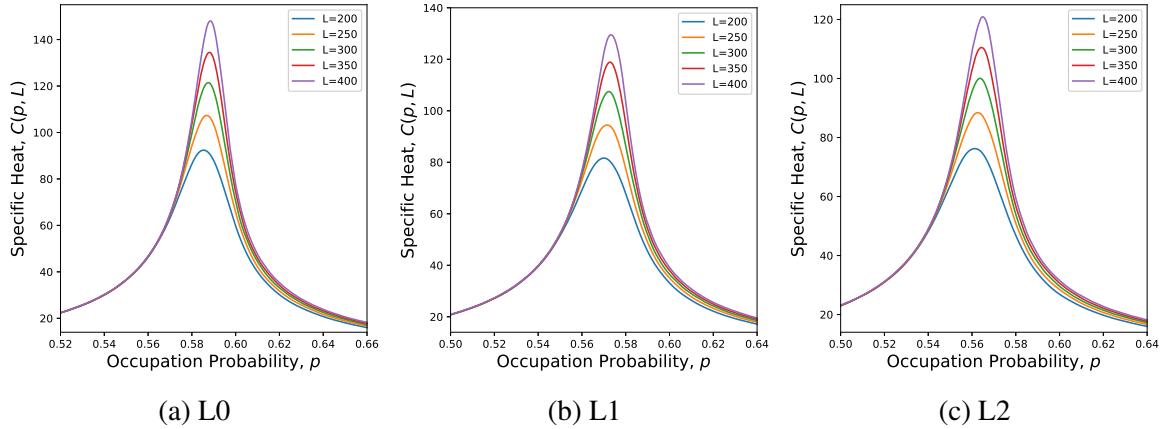


Fig. 5.4 Specific Heat,  $C(p,L)$  vs Occupation Probability,  $p$

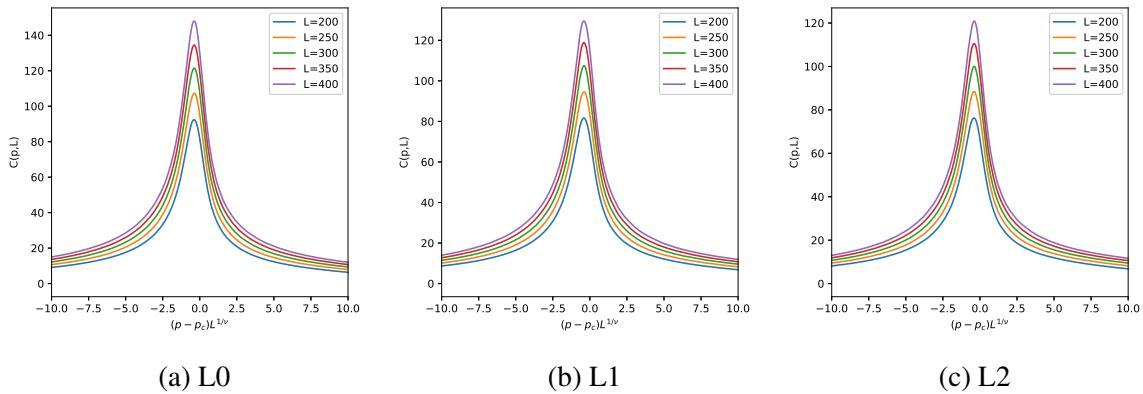


Fig. 5.5  $C(p,L)$  vs  $(p - p_c)L^{1/v}$

absolute value of the graph will give  $\alpha/\nu$  and from that we can find the exponent  $\alpha$  simply by dividing  $\alpha/\nu$  by  $1/\nu$ . We get  $\alpha$  values 0.906, 0.911, 0.919 for  $L0, L1, L2$  correspondingly. and using this value we can apply FSS hypothesis to get data collapse. If we plot  $CL^{-\alpha/\nu}$  vs  $(p - p_c)L^{1/\nu}$  we get perfect data collapse for  $L0, L1, L2$  and it is shown in figure 5.6

#### 5.4.4 Order Parameter and finding $\beta$

Order parameter, also known as the percolation strength, is , along with entropy, an important quantity in the study of phase transition. It is denoted as  $P(p,L)$ . Using the definition ?? we obtain the order parameter for our system and it is shown in the figure 5.7. Since using spanning cluster and the largest cluster gives the same exponent, it really does not matter which one we use. But in our case there is a boundary of the system, which we define as periodic. Hence using spanning cluster is appropriate. Using the exponent  $1/\nu$  obtained in

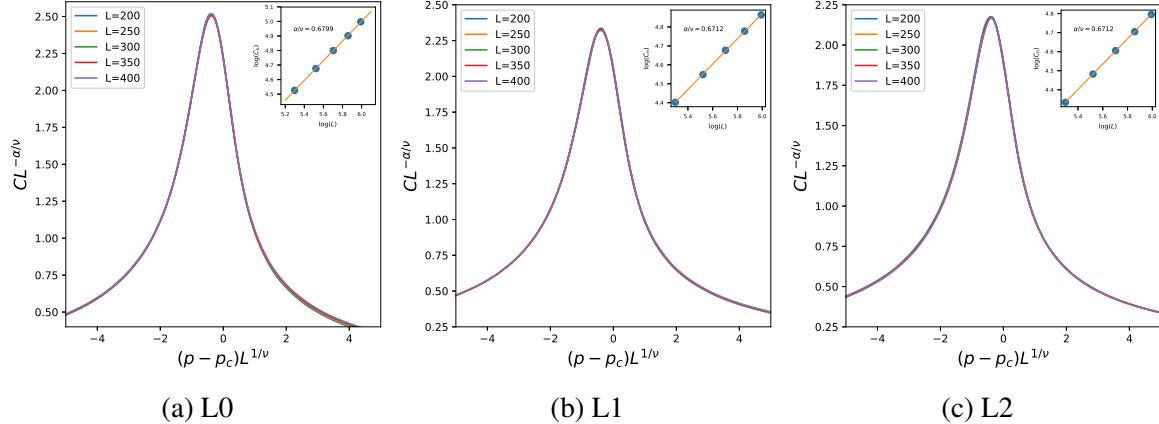


Fig. 5.6  $CL^{-\alpha/v}$  vs  $(p - p_c)L^{1/v}$

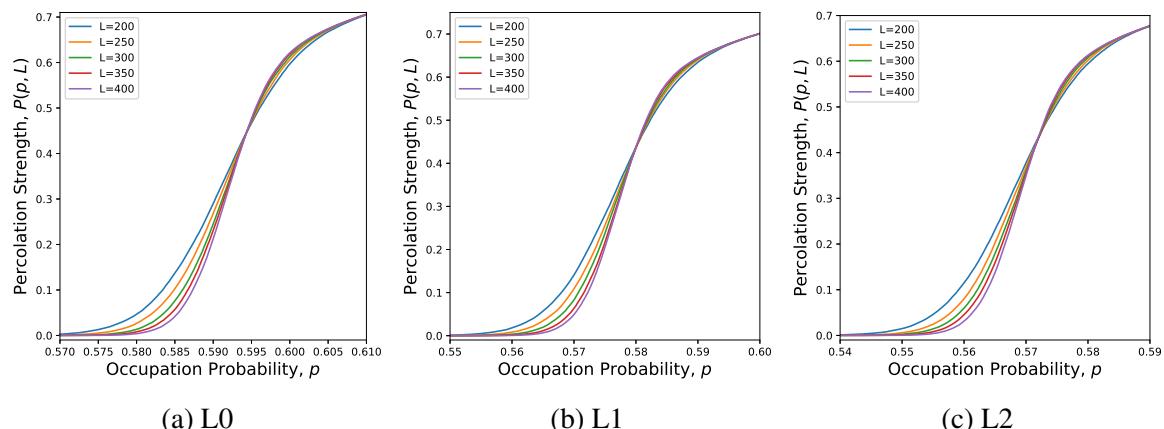


Fig. 5.7 Order Parameter,  $P(p,L)$  vs Occupation Probability,  $p$

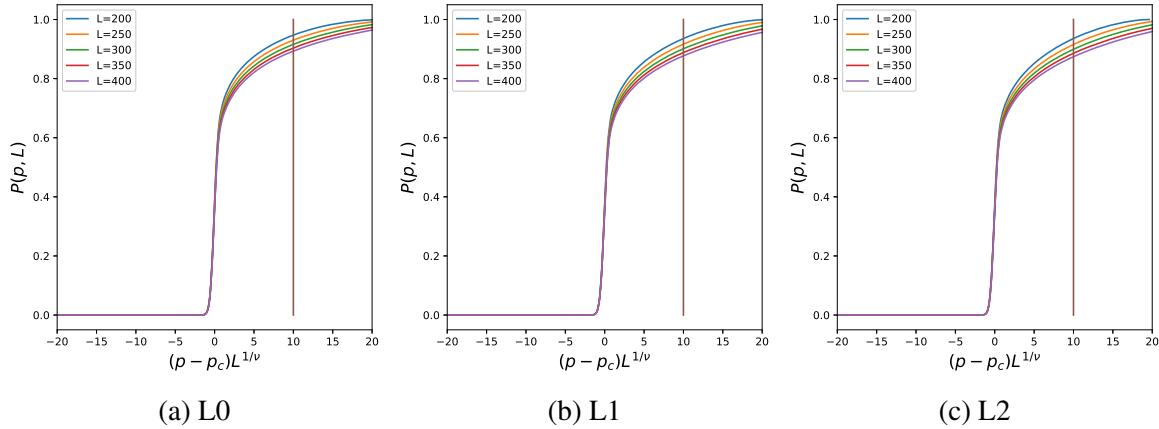


Fig. 5.8 Order Parameter,  $P(p,L)$  vs Occupation Probability,  $p$

Fig. 5.9  $P(p,L)$  vs  $(p - p_c)L^{1/\nu}$

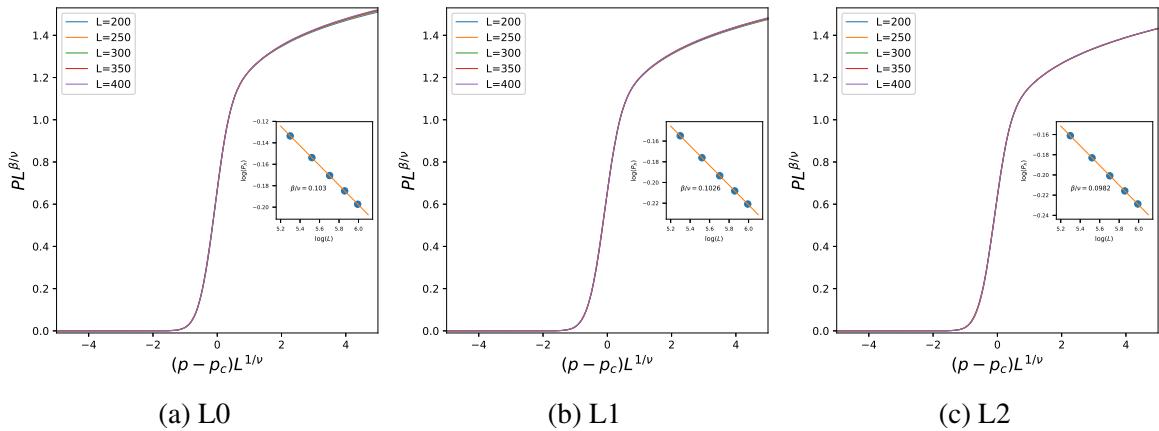


Fig. 5.10  $PL^{\beta/\nu}$  vs  $(p - p_c)L^{1/\nu}$

section 5.4.2 we scale the  $x$ -values as  $(p - p_c)L^{1/v}$  and get the following figure 5.8. Then in figure 5.8 we draw a vertical line where there are several horizontal lines. We measure the height of the lines and call it  $P_h$  and after plotting  $\log(P_h)$  vs  $\log(L)$  (inset of figure 5.10) we get the exponent  $\beta/v$  from its slope and obtain exponent  $\beta$  by dividing  $\beta/v$  by  $1/v$ . Using the FSS hypothesis 2.4.2 we plot  $PL^{\beta/v}$  versus  $(p - p_c)L^{1/v}$  and get a good data collapse which is shown in figure 5.10.

### 5.4.5 Susceptibility and finding $\gamma$

Susceptibility is defined as the derivative of the order parameter  $P(p, L)$  with respect to the control parameter  $p$ , i.e.,  $\chi = \frac{dP}{dp}$ . Using this definition we obtain the graph of susceptibility 5.11. And if we scale the  $x$  values and plot  $\chi$  vs  $(p - p_c)L^{1/\nu}$  we get all the peak point

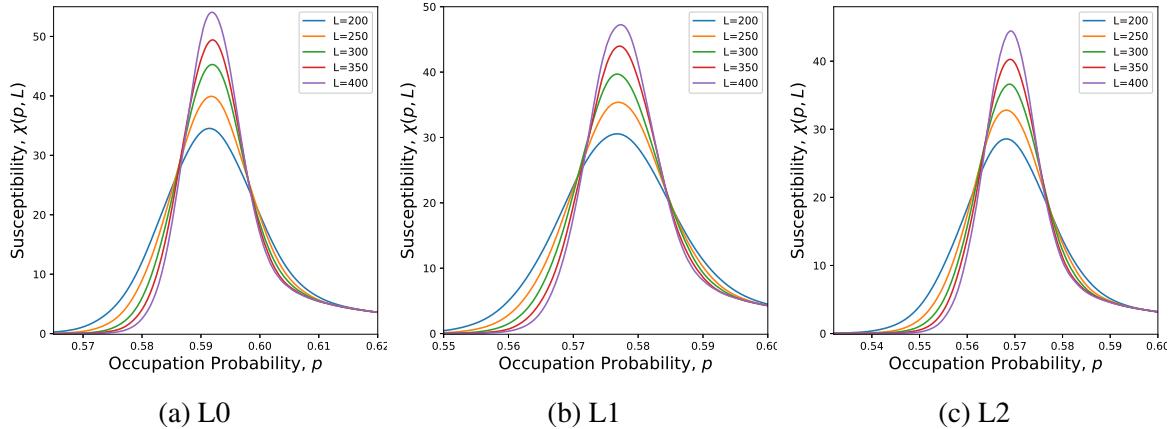


Fig. 5.11 Susceptibility,  $\chi(p,L)$  vs Occupation Probability,  $p$

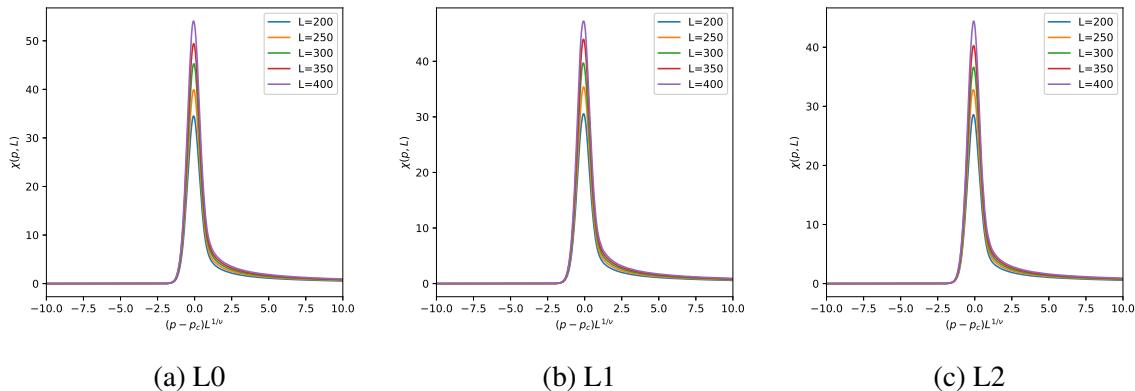
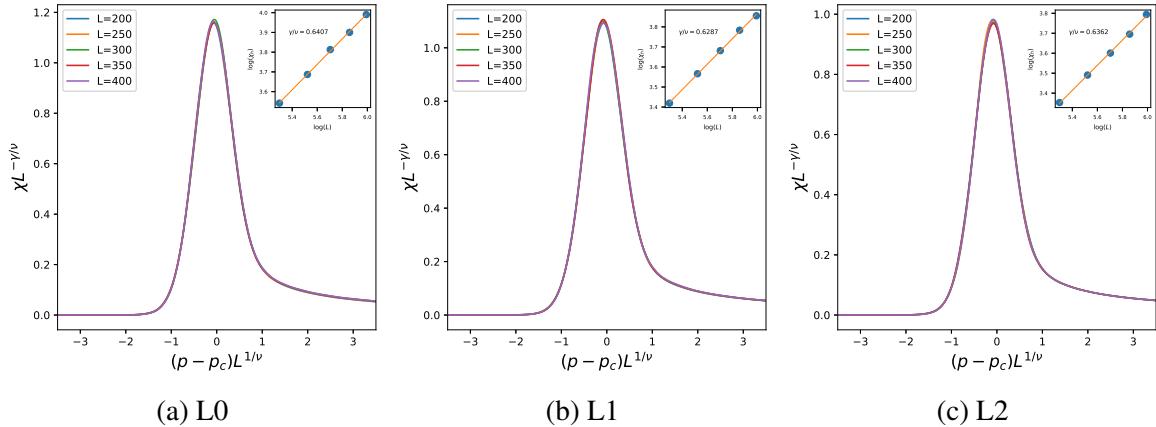


Fig. 5.12  $\chi(p,L)$  vs  $(p - p_c)L^{1/\nu}$

aligned (figure 5.12). Note that the value of  $1/\nu$  is known from section 5.4.2. Then we take the reading of the height of each line and call it  $\chi_h$ . Since each line represents a different lattice size, plotting  $\log(\chi_h)$  vs  $\log(L)$  gives the slope  $\gamma/\nu$ . And using the FSS hypothesis we plot  $\chi L^{-\gamma/\nu}$  vs  $(p - p_c)L^{1/\nu}$  and obtain a perfect data collapse. It is shown in figure 5.13. We obtain the values of  $\gamma$  to be 0.8543, 0.8542, 0.882.

### 5.4.6 Cluster Size Distribution

### $n_s$ vs $s$ graph

Fig. 5.13  $\chi L^{\gamma/\nu}$  vs  $(p - p_c)L^{1/\nu}$ Fig. 5.14 Number of cluster of size  $s$ ,  $n_s$  vs size of the cluster  $s$ 

### 5.4.7 Cluster Rank Size Distribution

### 5.4.8 Order-Disorder Transition

Phase transition is an order-disorder transition. There is a critical point which separates the two regions. Before the critical point the system is in disordered phase and after it is in ordered phase when we increase temperature in thermodynamics. Behavior of two phases are completely different. It's astonishing how the behavior changes. In percolation theory, this order disorder transition is different than in thermodynamics. Here disordered means uncertainty, since we are dealing with a system where probability is the control parameter (the occupation probability  $p$ ). When  $p$  is minimum all clusters are disconnected and have size of unity. This means that we can pick a cluster with probability  $\frac{1}{2L^2}$ , where  $L$  is the lattice size and  $2L^2$  is the number of bonds in the lattice. That's why entropy is maximum and order parameter is minimum in this region. Now as we keep occupying the lattice clusters of different size arises, and at some point a miracle happens. It is the critical point where the transition occurs. A cluster appears for the first time which spans the entire lattice either horizontally or vertically. And in case of periodic condition the cluster wraps the lattice all the way around it. This cluster is called the spanning (wrapping) cluster in non-periodic (periodic) condition. The probability of picking this cluster at random is always larger than picking any other clusters. Thus system goes to the ordered state. And if we keep occupying

Fig. 5.15  $\log(n_s)$  vs  $\log(s)$

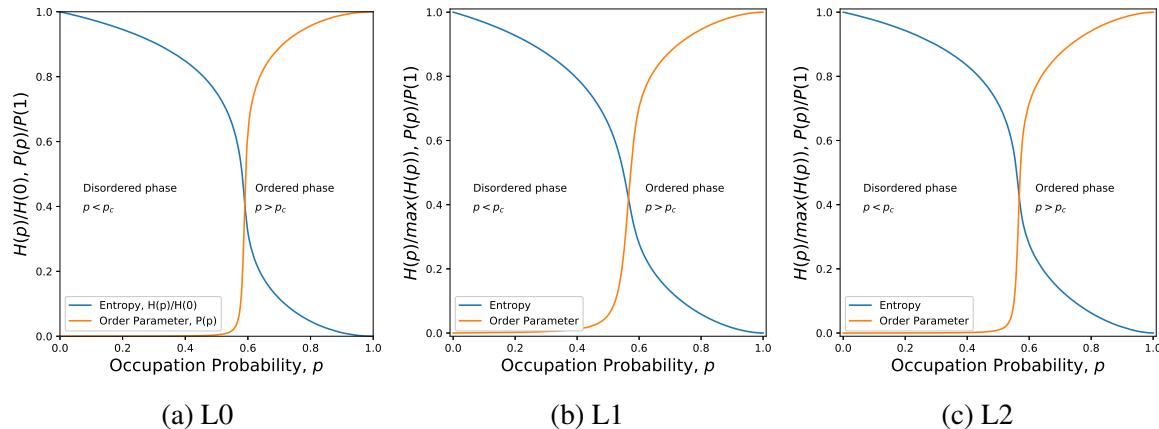


Fig. 5.16  $H(p,L)/H(0,L)$  or  $P(p,L)/P(1,L)$  vs  $p$

the lattice at some point all cluster are joined to form one cluster. Thus picking this cluster at random has no uncertainty, meaning we have reached the entirely ordered phase. Here entropy is minimum and ordered parameter is maximum. A graph ?? containing both entropy and order parameter can show this process. We have normalized the entropy (figure 5.3) to match with the order parameter (figure 5.7).

### 5.4.9 Fractal Dimension

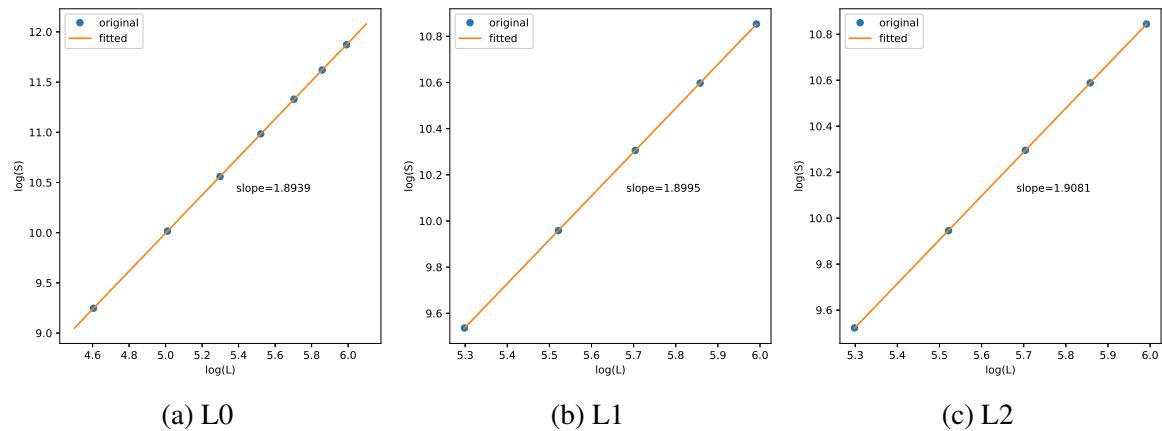
At critical point the square lattice shows the property of a fractal. A fractal is an object which occupies less space than it is embedded. For example a piece of cheese is a 3D fractal, since there are holes in the cheese which is empty. We use the relation

$$S \sim L^{d_f} \quad (5.2)$$

taking log we get

$$\log(S) = d_f \log(L) \quad (5.3)$$

Here  $S$  is the average size of the spanning cluster at critical point. Using this we get the figure ???. And we obtain fractal dimension  $d_f$  for  $L0, L1, L2$  which is listed in 6.2.

Fig. 5.17  $\log(S)$  vs  $\log(L)$

# Chapter 6

## Summary and Discussion

We have investigated percolation by random sequential ballistic deposition (RSBD) on a square lattice with interaction range upto second nearest neighbors. The critical points  $p_c$  and all the necessary critical exponents  $\alpha, \beta, \gamma, \nu$  etc. are obtained numerically for each range of interactions. Like in its thermal counterpart, we find that the critical exponents of RSBD depend on the range of interactions and for a given range of interaction they obey the Rushbrooke inequality. Besides, we obtain the exponent  $\tau$  which characterizes the cluster size distribution ?? and the fractal dimension  $d_f$  that characterizes the spanning cluster at  $p_c$ . Our results suggest that the RSBD for each range of interaction belong to a new universality class which is in sharp contrast to earlier results of the only work that exist on RSBD.

We denote  $L0, L1, L2$  for expressing direct, first nearest neighbor and second nearest neighbor interaction respectively. Up until now we knew the exponents for  $L0$  for old definition of site percolation. We have found same exponents for the thermodynamically consistent new definition of site percolation and we also have investigated for short and long range interactions and we call this  $L1$  and  $L2$  respectively. Note that  $L1$  is the case where we can choose the first nearest neighbor and  $L2$  is the case where we can chose 2nd nearest neighbor in the direction of first nearest neighbor. We can use new feature of  $L1$  only if the feature of  $L0$  is unavailable, i.e., the selected site is already occupied. Similarly we can use new feature of  $L2$  only if the feature of  $L0$  and  $L1$  is unavailable. Using this in mind we perform simulation and we obtain the critical exponents which agree with the laws of thermodynamics and the Rushbrooke inequality is satisfied in all cases.

The combined exponents are listed below

The critical exponents found in all cases are listed below,

Here we notice that the critical point decreases as we increase the range of interaction. But the fractal dimension increases. This is reasonable since my occupying nearest and

Interaction	$p_c$	$1/v$	$\alpha/v$	$\beta/v$	$\gamma/v$
L0	0.5927	0.75	0.6799	0.103	0.64071
L1	0.5782	0.736	0.6712	0.1026	0.6287
L2	0.5701	0.721	0.6631	0.0982	0.6362

Table 6.1 List of combined exponents

Interaction	$p_c$	$1/v$	$\alpha$	$\beta$	$\gamma$	$\alpha + 2\beta + \gamma$	$d_f$
L0	0.5927	0.75	0.906	0.137	0.8543	2.0347	1.8939
L1	0.5782	0.736	0.911	0.139	0.8542	2.044	1.8994
L2	0.5701	0.721	0.919	0.136	0.882	2.07	1.90810

Table 6.2 List of exponents

second nearest neighbor we are increasing the chance of any individual cluster to grow faster. This is the reason for the  $p_c$  value to decrease. But it grows in area not in length average meaning when the spanning cluster appears it will contain more sites and bonds than in regular percolation which is evident from the fractal dimension  $d_f$ .

All other exponents changes a bit but their shape is not different. That's why change is not visible to the naked eye and it requires a thorough investigation.

# References

- [1]
- [2] Different kind of cubic lattice. URL <http://utkarshchemistry.blogspot.com/2012/09/solid-state-2.html>.
- [3] Fluid mechanics - theory. URL [https://www.ecourses.ou.edu/cgi-bin/ebook.cgi?doc=&topic=fl&chap\\_sec=06.1&page=theory](https://www.ecourses.ou.edu/cgi-bin/ebook.cgi?doc=&topic=fl&chap_sec=06.1&page=theory). Basic Dimensions of Common Parameters Table.
- [4] Honeycomb lattice. URL <https://www.quora.com/Why-are-there-only-14-bravais-lattices>.
- [5] Scale free network. URL <https://www.flickr.com/photos/sjcockell/8425835703>.
- [6] 286(5439):509–512, oct 1999. doi: 10.1126/science.286.5439.509. URL <https://doi.org/10.1126/science.286.5439.509>.
- [7] D. Achlioptas, R. M. D'Souza, and J. Spencer. Explosive percolation in random networks. *Science*, 323(5920):1453–1455, mar 2009. doi: 10.1126/science.1167782. URL <https://doi.org/10.1126/science.1167782>.
- [8] Joan Adler. Bootstrap percolation. *Physica A: Statistical Mechanics and its Applications*, 171(3):453–470, mar 1991. doi: 10.1016/0378-4371(91)90295-n. URL [https://doi.org/10.1016/0378-4371\(91\)90295-n](https://doi.org/10.1016/0378-4371(91)90295-n).
- [9] Didarul Alam. Entropy,specific heat,susceptibility and rushbrooke inequality: New insights in percolation, 2016. Master's thesis,.
- [10] Gerald Warnecke Andreas Greven, Gerhard Keller. *Entropy (Princeton Series in Applied Mathematics)*. Princeton University Press, 2003. ISBN 9780691113388.
- [11] Ralph Baierlein. *Thermal Physics*. Cambridge University Press, 1999. ISBN 0521658381.
- [12] Francisco Balibrea. On clausius, boltzmann and shannon notions of entropy. *Journal of Modern Physics*, 07(02):219–227, 2016. doi: 10.4236/jmp.2016.72022. URL <https://doi.org/10.4236/jmp.2016.72022>.
- [13] S. Bansal, B. T Grenfell, and L. A. Meyers. When individual behaviour matters: homogeneous and network models in epidemiology. *Journal of The Royal Society Interface*, 4(16):879–891, oct 2007. doi: 10.1098/rsif.2007.1100. URL <https://doi.org/10.1098/rsif.2007.1100>.

- [14] M. Bauer and O. Golinelli. Core percolation in random graphs: a critical phenomena analysis. *The European Physical Journal B*, 24(3):339–352, dec 2001. doi: 10.1007/s10051-001-8683-4. URL <https://doi.org/10.1007/s10051-001-8683-4>.
- [15] L Benguigui. The different paths to entropy. *European Journal of Physics*, 34(2):303–321, jan 2013. doi: 10.1088/0143-0807/34/2/303. URL <https://doi.org/10.1088/0143-0807/34/2/303>.
- [16] M. Blume. Theory of the first-order magnetic phase change in UO<sub>2</sub>. *Physical Review*, 141(2):517–524, jan 1966. doi: 10.1103/physrev.141.517. URL <https://doi.org/10.1103/physrev.141.517>.
- [17] S. Boccaletti, G. Bianconi, R. Criado, C.I. del Genio, J. Gómez-Gardeñes, M. Romance, I. Sendiña-Nadal, Z. Wang, and M. Zanin. The structure and dynamics of multilayer networks. *Physics Reports*, 544(1):1–122, nov 2014. doi: 10.1016/j.physrep.2014.07.001. URL <https://doi.org/10.1016/j.physrep.2014.07.001>.
- [18] Marián Boguñá, M. franceschetti, r. meester: Random networks for communication. from statistical physics to information systems. *Journal of Statistical Physics*, 135(3):585–586, apr 2009. doi: 10.1007/s10955-009-9740-2. URL <https://doi.org/10.1007/s10955-009-9740-2>.
- [19] Béla Bollobás and Oliver Riordan. Percolation on self-dual polygon configurations. In *Bolyai Society Mathematical Studies*, pages 131–217. Springer Berlin Heidelberg, 2010. doi: 10.1007/978-3-642-14444-8\_3. URL [https://doi.org/10.1007/978-3-642-14444-8\\_3](https://doi.org/10.1007/978-3-642-14444-8_3).
- [20] Ilan Breskin, Jordi Soriano, Elisha Moses, and Tsvi Tlusty. Percolation in living neural networks. *Physical Review Letters*, 97(18), oct 2006. doi: 10.1103/physrevlett.97.188102. URL <https://doi.org/10.1103/physrevlett.97.188102>.
- [21] S. R. Broadbent and J. M. Hammersley. Percolation processes. *Mathematical Proceedings of the Cambridge Philosophical Society*, 53(03):629, jul 1957. doi: 10.1017/s0305004100032680. URL <https://doi.org/10.1017/s0305004100032680>.
- [22] Duncan S. Callaway, M. E. J. Newman, Steven H. Strogatz, and Duncan J. Watts. Network robustness and fragility: Percolation on random graphs. *Physical Review Letters*, 85(25):5468–5471, dec 2000. doi: 10.1103/physrevlett.85.5468. URL <https://doi.org/10.1103/physrevlett.85.5468>.
- [23] John Cardy. *Scaling and Renormalization in Statistical Physics*. Cambridge University Press, 1996. doi: 10.1017/cbo9781316036440. URL <https://doi.org/10.1017/cbo9781316036440>.
- [24] Nicolas Léonard Sadi Carnot et al. *Reflections on the motive power of heat : and on machines fitted to develop that power*. 1890.
- [25] J Chalupa, P L Leath, and G R Reich. Bootstrap percolation on a bethe lattice. *Journal of Physics C: Solid State Physics*, 12(1):L31–L35, jan 1979. doi: 10.1088/0022-3719/12/1/008. URL <https://doi.org/10.1088/0022-3719/12/1/008>.

- [26] Barry A. Cipra. An introduction to the ising model. *The American Mathematical Monthly*, 94(10):937, dec 1987. doi: 10.2307/2322600. URL <https://doi.org/10.2307/2322600>.
- [27] Rudolf Clausius. *The Mechanical Theory Of Heat (1879)*. J. Van Voorst- Steam-engines, 1867.
- [28] Reuven Cohen, Daniel ben Avraham, and Shlomo Havlin. Percolation critical exponents in scale-free networks. *Physical Review E*, 66(3), sep 2002. doi: 10.1103/physreve.66.036113. URL <https://doi.org/10.1103/physreve.66.036113>.
- [29] Paul H. Coleman and Luciano Pietronero. The fractal structure of the universe. *Physics Reports*, 213(6):311–389, may 1992. doi: 10.1016/0370-1573(92)90112-d. URL [https://doi.org/10.1016/0370-1573\(92\)90112-d](https://doi.org/10.1016/0370-1573(92)90112-d).
- [30] Antonio Coniglio, Chiara Rosanna Nappi, Fulvio Peruggi, and Lucio Russo. Percolation and phase transitions in the ising model. *Communications in Mathematical Physics*, 51(3):315–323, oct 1976. doi: 10.1007/bf01617925. URL <https://doi.org/10.1007/bf01617925>.
- [31] P. Csermely, A. London, L.-Y. Wu, and B. Uzzi. Structure and dynamics of core/periphery networks. *Journal of Complex Networks*, 1(2):93–123, oct 2013. doi: 10.1093/comnet/cnt016. URL <https://doi.org/10.1093/comnet/cnt016>.
- [32] Imre Derényi, Gergely Palla, and Tamás Vicsek. Clique percolation in random networks. *Physical Review Letters*, 94(16), apr 2005. doi: 10.1103/physrevlett.94.160202. URL <https://doi.org/10.1103/physrevlett.94.160202>.
- [33] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes. k-core organization of complex networks. *Physical Review Letters*, 96(4), feb 2006. doi: 10.1103/physrevlett.96.040601. URL <https://doi.org/10.1103/physrevlett.96.040601>.
- [34] Tomasz Downarowicz. *Entropy in Dynamical Systems*. Cambridge University Press, 2009. doi: 10.1017/cbo9780511976155. URL <https://doi.org/10.1017/cbo9780511976155>.
- [35] P. Erdős and A. Rényi. On the strength of connectedness of a random graph. *Acta Mathematica Academiae Scientiarum Hungaricae*, 12(1-2):261–267, mar 1964. doi: 10.1007/bf02066689. URL <https://doi.org/10.1007/bf02066689>.
- [36] J W Essam. Percolation theory. *Reports on Progress in Physics*, 43(7):833–912, jul 1980. doi: 10.1088/0034-4885/43/7/001. URL <https://doi.org/10.1088/0034-4885/43/7/001>.
- [37] Michael E. Fisher. Critical probabilities for cluster size and percolation problems. *Journal of Mathematical Physics*, 2(4):620–627, jul 1961. doi: 10.1063/1.1703746. URL <https://doi.org/10.1063/1.1703746>.
- [38] Paul J. Flory. Molecular size distribution in three dimensional polymers. II. trifunctional branching units. *Journal of the American Chemical Society*, 63(11):3091–3096, nov 1941. doi: 10.1021/ja01856a062. URL <https://doi.org/10.1021/ja01856a062>.

- [39] C.M. Fortuin. On the random-cluster model II. the percolation model. *Physica*, 58(3):393–418, apr 1972. doi: 10.1016/0031-8914(72)90161-9. URL [https://doi.org/10.1016/0031-8914\(72\)90161-9](https://doi.org/10.1016/0031-8914(72)90161-9).
- [40] C.M. Fortuin. On the random-cluster model. *Physica*, 59(4):545–570, jun 1972. doi: 10.1016/0031-8914(72)90087-0. URL [https://doi.org/10.1016/0031-8914\(72\)90087-0](https://doi.org/10.1016/0031-8914(72)90087-0).
- [41] C.M. Fortuin and P.W. Kasteleyn. On the random-cluster model. *Physica*, 57(4):536–564, feb 1972. doi: 10.1016/0031-8914(72)90045-6. URL [https://doi.org/10.1016/0031-8914\(72\)90045-6](https://doi.org/10.1016/0031-8914(72)90045-6).
- [42] Jay D. Friedenberg and Gordon W. Silverman. *Cognitive Science: An Introduction to the Study of Mind*. SAGE Publications, Inc, 2011. ISBN 1412977614. URL <https://www.amazon.com/Cognitive-Science-Introduction-Study-Mind/dp/1412977614?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1412977614>.
- [43] Liv Furuberg, Knut Jørgen Måløy, and Jens Feder. Intermittent behavior in slow drainage. *Physical Review E*, 53(1):966–977, jan 1996. doi: 10.1103/physreve.53.966. URL <https://doi.org/10.1103/physreve.53.966>.
- [44] Peter Hall. On continuum percolation. *Institute of Mathematical Statistics*, 13(4):1250–1266, 1985. doi: 110.2307/2244176.
- [45] R. V. L. Hartley. Transmission of information1. *Bell System Technical Journal*, 7(3):535–563, jul 1928. doi: 10.1002/j.1538-7305.1928.tb01236.x. URL <https://doi.org/10.1002/j.1538-7305.1928.tb01236.x>.
- [46] M. K. Hassan. Lecture notes on statistical mechanics of non-equilibrium phenomena.
- [47] J. Hoshen and R. Kopelman. Percolation and cluster distribution. i. cluster multiple labeling technique and critical concentration algorithm. *Physical Review B*, 14(8):3438–3445, oct 1976. doi: 10.1103/physrevb.14.3438. URL <https://doi.org/10.1103/physrevb.14.3438>.
- [48] H. Hu, S. Myers, V. Colizza, and A. Vespignani. WiFi networks and malware epidemiology. *Proceedings of the National Academy of Sciences*, 106(5):1318–1323, jan 2009. doi: 10.1073/pnas.0811973106. URL <https://doi.org/10.1073/pnas.0811973106>.
- [49] John R. Hughes. Post-tetanic potentiation. *Physiological Reviews*, 38(1):91–113, jan 1958. doi: 10.1152/physrev.1958.38.1.91. URL <https://doi.org/10.1152/physrev.1958.38.1.91>.
- [50] Ernst Ising. Beitrag zur theorie des ferromagnetismus. *Zeitschrift für Physik*, 31(1):253–258, feb 1925. doi: 10.1007/bf02980577. URL <https://doi.org/10.1007/bf02980577>.
- [51] P. W. Kasteleyn and C. M. Fortuin. Phase transitions in lattice systems with random local properties. *Physical Society of Japan Journal Supplement, Vol. 26. Proceedings of the International Conference on Statistical Mechanics held 9-14 September, 1968 in Kyoto.*, p.11, 26:11, 1969. URL <http://adsabs.harvard.edu/abs/1969PSJJS..26...11K>. Provided by the SAO/NASA Astrophysics Data System.

- [52] Harry Kesten. Scaling relations for 2d-percolation. *Communications in Mathematical Physics*, 109(1):109–156, mar 1987. doi: 10.1007/bf01205674. URL <https://doi.org/10.1007/bf01205674>.
- [53] M. Kivela, A. Arenas, M. Barthelemy, J. P. Gleeson, Y. Moreno, and M. A. Porter. Multilayer networks. *Journal of Complex Networks*, 2(3):203–271, jul 2014. doi: 10.1093/comnet/cnu016. URL <https://doi.org/10.1093/comnet/cnu016>.
- [54] P. L. Leath. Cluster size and boundary distribution near percolation threshold. *Physical Review B*, 14(11):5046–5055, dec 1976. doi: 10.1103/physrevb.14.5046. URL <https://doi.org/10.1103/physrevb.14.5046>.
- [55] Roland Lenormand and Cesar Zarcone. Invasion percolation in an etched network: Measurement of a fractal dimension. *Physical Review Letters*, 54(20):2226–2229, may 1985. doi: 10.1103/physrevlett.54.2226. URL <https://doi.org/10.1103/physrevlett.54.2226>.
- [56] Eduardo López, Roni Parshani, Reuven Cohen, Shai Carmi, and Shlomo Havlin. Limited path percolation in complex networks. *Physical Review Letters*, 99(18), oct 2007. doi: 10.1103/physrevlett.99.188701. URL <https://doi.org/10.1103/physrevlett.99.188701>.
- [57] Christian D. Lorenz and Robert M. Ziff. Precise determination of the bond percolation thresholds and finite-size scaling corrections for the sc, fcc, and bcc lattices. *Physical Review E*, 57(1):230–236, jan 1998. doi: 10.1103/physreve.57.230. URL <https://doi.org/10.1103/physreve.57.230>.
- [58] Christian D. Lorenz, Raechelle May, and Robert M. Ziff. *Journal of Statistical Physics*, 98(3/4):961–970, 2000. doi: 10.1023/a:1018648130343. URL <https://doi.org/10.1023/a:1018648130343>.
- [59] Knut Jo?rgen Målo?y, Liv Furuberg, Jens Feder, and Torstein Jo?ssang. Dynamics of slow drainage in porous media. *Physical Review Letters*, 68(14):2161–2164, apr 1992. doi: 10.1103/physrevlett.68.2161. URL <https://doi.org/10.1103/physrevlett.68.2161>.
- [60] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman, 1983. ISBN 0716711869. URL <https://www.amazon.com/Fractal-Geometry-Nature-Benoit-Mandelbrot/dp/0910321647?SubscriptionId=AKIAIOBINVZYXQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0910321647>.
- [61] Lauren Ancel Meyers. Contact network epidemiology: Bond percolation applied to infectious disease prediction and control. *Bulletin of the American Mathematical Society*, 44(01):63–87, oct 2006. doi: 10.1090/s0273-0979-06-01148-7. URL <https://doi.org/10.1090/s0273-0979-06-01148-7>.
- [62] Elliott W. Montroll, Renfrey B. Potts, and John C. Ward. Correlations and spontaneous magnetization of the two-dimensional ising model. *Journal of Mathematical Physics*, 4(2):308–322, feb 1963. doi: 10.1063/1.1703955. URL <https://doi.org/10.1063/1.1703955>.

- [63] Cristopher Moore and M. E. J. Newman. Exact solution of site and bond percolation on small-world networks. *Physical Review E*, 62(5):7059–7064, nov 2000. doi: 10.1103/physreve.62.7059. URL <https://doi.org/10.1103/physreve.62.7059>.
- [64] M. E. J. Newman and R. M. Ziff. Efficient monte carlo algorithm and high-precision results for percolation. *Physical Review Letters*, 85(19):4104–4107, nov 2000. doi: 10.1103/physrevlett.85.4104. URL <https://doi.org/10.1103/physrevlett.85.4104>.
- [65] M. E. J. Newman and R. M. Ziff. Fast monte carlo algorithm for site or bond percolation. *Physical Review E*, 64(1), jun 2001. doi: 10.1103/physreve.64.016706. URL <https://doi.org/10.1103/physreve.64.016706>.
- [66] H. Nyquist. Certain factors affecting telegraph speed. *Transactions of the American Institute of Electrical Engineers*, XLIII:412–422, jan 1924. doi: 10.1109/t-aiee.1924.5060996. URL <https://doi.org/10.1109/t-aiee.1924.5060996>.
- [67] H. Nyquist. Certain topics in telegraph transmission theory. *Proceedings of the IEEE*, 90(2):280–305, 2002. doi: 10.1109/5.989875. URL <https://doi.org/10.1109/5.989875>.
- [68] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814–818, jun 2005. doi: 10.1038/nature03607. URL <https://doi.org/10.1038/nature03607>.
- [69] M. D. Penrose. CONTINUUM PERCOLATION (cambridge tracts in mathematics 119). *Bulletin of the London Mathematical Society*, 30(4):435–436, jul 1998. doi: 10.1112/s0024609397243808. URL <https://doi.org/10.1112/s0024609397243808>.
- [70] Mason Porter and James Gleeson. *Dynamical Systems on Networks*. Springer International Publishing, 2016. doi: 10.1007/978-3-319-26641-1. URL <https://doi.org/10.1007/978-3-319-26641-1>.
- [71] M. S. Rahman and M. K. Hassan. Redefinition of site percolation in light of entropy and the second law of thermodynamics, 2018.
- [72] G. R. Reich and P. L. Leath. High-density percolation: Exact solution on a bethe lattice. *Journal of Statistical Physics*, 19(6):611–622, dec 1978. doi: 10.1007/bf01011772. URL <https://doi.org/10.1007/bf01011772>.
- [73] M Sahini and M Sahimi. *Applications Of Percolation Theory*. CRC Press, 1994. ISBN 9780748400768. URL <https://www.amazon.com/Applications-Percolation-Theory-M-Sahini/dp/0748400761?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0748400761>.
- [74] L.M. Sander, C.P. Warren, I.M. Sokolov, C. Simon, and J. Koopman. Percolation on heterogeneous networks as a model for epidemics. *Mathematical Biosciences*, 180(1-2):293–305, nov 2002. doi: 10.1016/s0025-5564(02)00117-7. URL [https://doi.org/10.1016/s0025-5564\(02\)00117-7](https://doi.org/10.1016/s0025-5564(02)00117-7).

- [75] John E.J. Schmitz. *The Second Law of Life: Energy, Technology, and the Future of Earth As We Know It.* William Andrew, 2007. ISBN 0815515375. URL <https://www.amazon.com/Second-Law-Life-Energy-Technology/dp/0815515375?SubscriptionId=AKIAIOBINVYZXQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0815515375>.
- [76] K. J. Schrenk, N. A. M. Araújo, and H. J. Herrmann. Stacked triangular lattice: Percolation properties. *Physical Review E*, 87(3), mar 2013. doi: 10.1103/physreve.87.032123. URL <https://doi.org/10.1103/physreve.87.032123>.
- [77] N. Schwartz, R. Cohen, D. ben Avraham, A.-L. Barabási, and S. Havlin. Percolation in directed scale-free networks. *Physical Review E*, 66(1), jul 2002. doi: 10.1103/physreve.66.015104. URL <https://doi.org/10.1103/physreve.66.015104>.
- [78] Philip E. Seiden and Lawrence S. Schulman. Percolation model of galactic structure. *Advances in Physics*, 39(1):1–54, feb 1990. doi: 10.1080/00018739000101461. URL <https://doi.org/10.1080/00018739000101461>.
- [79] Stephen B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, sep 1983. doi: 10.1016/0378-8733(83)90028-x. URL [https://doi.org/10.1016/0378-8733\(83\)90028-x](https://doi.org/10.1016/0378-8733(83)90028-x).
- [80] James P. Sethna. *Statistical Mechanics: Entropy, Order Parameters and Complexity (Oxford Master Series in Physics)*. Oxford University Press, 2006. ISBN 0198566778. URL <https://www.amazon.com/Statistical-Mechanics-Entropy-Parameters-Complexity/dp/0198566778?SubscriptionId=AKIAIOBINVYZXQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0198566778>.
- [81] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, jul 1948. doi: 10.1002/j.1538-7305.1948.tb01338.x. URL <https://doi.org/10.1002/j.1538-7305.1948.tb01338.x>.
- [82] J C Wierman. A bond percolation critical probability determination based on the star-triangle transformation. *Journal of Physics A: Mathematical and General*, 17(7):1525–1530, may 1984. doi: 10.1088/0305-4470/17/7/020. URL <https://doi.org/10.1088/0305-4470/17/7/020>.
- [83] John C. Wierman. Bond percolation on honeycomb and triangular lattices. *Advances in Applied Probability*, 13(02):298–313, jun 1981. doi: 10.1017/s0001867800036028. URL <https://doi.org/10.1017/s0001867800036028>.
- [84] D Wilkinson and J F Willemsen. Invasion percolation: a new form of percolation theory. *Journal of Physics A: Mathematical and General*, 16(14):3365–3376, oct 1983. doi: 10.1088/0305-4470/16/14/028. URL <https://doi.org/10.1088/0305-4470/16/14/028>.
- [85] C. N. Yang. The spontaneous magnetization of a two-dimensional ising model. *Physical Review*, 85(5):808–816, mar 1952. doi: 10.1103/physrev.85.808. URL <https://doi.org/10.1103/physrev.85.808>.
- [86] Bruno H. Zimm and Walter H. Stockmayer. The dimensions of chain molecules containing branches and rings. *The Journal of Chemical Physics*, 17(12):1301–1314, dec 1949. doi: 10.1063/1.1747157. URL <https://doi.org/10.1063/1.1747157>.



# Appendix A

## Percolation

### A.1 Algorithm

### A.2 Code

Each header file starts with a directive `#ifndef` and `#define`, which is necessary because one header file is needed multiple times and including it more than once results in error. These directive prevents it. Of course this directive must be closed by `#endif`.

#### A.2.1 Index

Here the notion of index of site and index of bond is defined. Site index has two element which determine row and column. Bond Index had three element describing bond type, row, column. Bond type can be horizontal or vertical.

the `src/index/index.h` file

```
1 #ifndef SITEPERCOLATION_INDEX_H
2 #define SITEPERCOLATION_INDEX_H
3
4 #include <ostream>
5 #include <iostream>
6 #include <sstream>
7 #include <vector>
8
9 #include " ../types.h"
10 #include " ../exception/exceptions.h"
11 #include " ../lattice/bond_type.h"
12
13
```

```
14 /* *
15 *
16 */
17 struct Index{
18     value_type row_{};
19     value_type column_{};

20
21     ~Index() = default;
22     Index() = default;

23
24     Index(value_type x, value_type y) : row_{x}, column_{y} {}

25
26 };
27
28
29 /**
30 * Template index
31 * @tparam T
32 */
33 class IndexRelative{
34 public:
35     int x_{};
36     int y_{};

37
38     ~IndexRelative() = default;
39     IndexRelative() = default;

40
41     IndexRelative(int x, int y) : x_{x}, y_{y} {}

42
43 };
44
45 /**
46 *
47 */
48 struct BondIndex{
49     BondType bondType;

50
51     value_type row_;
52     value_type column_;

53
54     ~BondIndex() = default;
55     BondIndex() = default;

56
57     BondIndex(BondType hv, value_type row, value_type column)
```

```

58 :    row_{row}, column_{column}
59 {
60 bondType = hv;
61 }
62
63 bool horizontal() const { return bondType == BondType::Horizontal; }
64 bool vertical() const { return bondType == BondType::Vertical; }
65
66 };
67
68
69 std::ostream& operator<<(std::ostream& os, const Index& index);
70 bool operator==(const Index& index1, const Index& index2);
71 bool operator<(const Index& index1, const Index& index2);
72
73 std::ostream& operator<<(std::ostream& os, const IndexRelative& index);
74
75 std::ostream& operator<<(std::ostream& os, const BondIndex& index);
76 bool operator==(BondIndex index1, BondIndex index2);
77 bool operator<(BondIndex index1, BondIndex index2);
78
79
80 /**
81 * Get 2nd nearest neighbor / sin the direction of 1st nearest neighbor
82 * , while @var center is the center
83 */
84 Index get_2nn_in_1nn_direction(Index center, Index nn_1, value_type
length);
85 std::vector<Index> get_2nn_s_in_1nn_s_direction(Index center, const std
::vector<Index> &nn_1, value_type length);
86 #endif /* SITEPERCOLATION_INDEX_H */

```

### The src/index/index.cpp file

```

1 //
2 // Created by shahnoor on 10/2/2017.
3 //
4
5 #include <iomanip>
6 #include "index.h"
7
8 using namespace std;
9
10

```

```

11 ostream& operator<<(ostream& os , const Index& index)
12 {
13     return os << '(' << index.row_ << ',' << index.column_ << ')';
14 }
15
16 ostream& operator<<(ostream& os , const IndexRelative& index)
17 {
18     return os << '(' << std::setw(3) << index.x_ << ',' << std::setw(3) <<
19         index.y_ << ')';
20 }
21
22 bool operator==(const Index& index1 , const Index& index2){
23     return (index1.row_ == index2.row_) && (index1.column_ == index2.column_);
24 }
25
26 bool operator<(const Index& index1 , const Index& index2){
27     if(index1.row_ < index2.row_)
28         return true;
29     if(index1.row_ == index2.row_){
30         return index1.column_ < index2.column_;
31     }
32     return false;
33 }
34
35 ostream& operator<<(ostream& os , const BondIndex& index){
36     if(index.horizontal()){
37         // horizontal bond
38         os << "<=>" ;
39     }
40     if (index.vertical()){
41         // vertical bond
42         os << "<|>" ;
43     }
44     return os << '(' << index.row_ << ',' << index.column_ << ')';
45 }
46
47 bool operator==(BondIndex index1 , BondIndex index2){
48     if(index1.horizontal() == index2.horizontal() || index1.vertical() ==
49         index2.vertical()){
50         // horizontal or vertical
51         return index1.row_ == index2.row_ && index1.column_ == index2.column_;
52     }

```

```
52 return false;
53 }
54
55
56 bool operator<(BondIndex index1, BondIndex index2){
57 cout << "not yet defined : line " << __LINE__ << endl;
58 return false;
59 }
60
61
62 /**
63 * Get the 2nd nearest nearest neighbor in the direction of 1st nearest
64 * neighbor.
65 * Periodicity is not considered here.
66 * @param center
67 * @param nn_1
68 * @param length
69 * @return
70 */
71 Index get_2nn_in_1nn_direction(Index center, Index nn_1, value_type
72 length){
73 int delta_c = int(nn_1.column_) - int(center.column_);
74 int delta_r = int(nn_1.row_) - int(center.row_);
75 if (delta_c == 0 && delta_r == 0){
76 cout << "Both indices are same : line " << __LINE__ << endl;
77 }
78 else if(delta_c > 1 || delta_r > 1){
79 // meaning, the sites are on the opposite edges
80 // cout << "2nd index is not the First nearest neighbor : line "
81 // << __LINE__ << " : file " << __FILE__ << endl;
82 }
83
84
85 /**
86 * Get all second nearest neighbors based on the first nearest neighbors.
87 * Periodicity is not considered here
88 * @param center
89 * @param nn_1
90 * @param length
91 * @return
```

```

92 */
93 vector<Index> get_2nn_s_in_1nn_s_direction(Index center, const vector<
94     Index> &nn_1, value_type length){
95     vector<Index> nn_2(nn_1.size());
96
97     for(size_t i{}; i != nn_1.size(); ++i){
98         int delta_c = int(nn_1[i].column_) - int(center.column_);
99         int delta_r = int(nn_1[i].row_) - int(center.row_);
100        if(delta_c == 0 && delta_r == 0){
101            cout << "Both indices are same : line " << __LINE__ << endl;
102        }
103        else if(delta_c > 1 || delta_r > 1){
104            // meaning, the sites are on the opposite edges
105            // cout << "center " << center << " nn " << nn_1 << endl;
106            // cout << "2nd index is not the First nearest neighbor :
107            // line " << __LINE__ << " : file " << __FILE__ << endl;
108        }
109        nn_2[i] = Index{(nn_1[i].row_ + delta_r + length) % length, (nn_1[i].
110                         column_ + delta_c + length) % length};
111    }
112    return nn_2;
113 }
```

## A.2.2 Site

The site class contains all information about a site, e.g., if it is active or not and if it is then what is its group id or relative index.

The `src/lattice/site.h` file

```

1 #ifndef SITEPERCOLATION_SITE_H
2 #define SITEPERCOLATION_SITE_H
3
4 #include <array>
5 #include <set>
6 #include <vector>
7 #include <iostream>
8 #include <memory>
9
10 #include "../index/index.h"
11 #include "../types.h"
12
13
```

```
14 /**
15 * single Site of a lattice
16 */
17 struct Site{
18 /**
19 * if true -> site is placed.
20 * if false -> the (empty) position is there but the site is not (
21 * required for site percolation)
22 */
23 bool _status{false};
24 int _group_id{-1};
25 Index _id{};
26
27 // relative distance from the root site. {0,0} if it is the root site
28 // for detecting wrapping
29 IndexRelative _relative_index{0,0};
30
31 public:
32
33 ~Site() = default;
34 Site() = default;
35 Site(const Site&) = default;
36 Site(Site&&) = default;
37 Site& operator=(const Site&) = default;
38 Site& operator=(Site&&) = default;
39
40 Site(Index id, value_type length){
41 // I have handle _neighbor or corner points and edge points carefully
42 if(id.row_ >= length || id.column_ >= length){
43 std::cout << "out of range : line " << __LINE__ << std::endl;
44 }
45 _id.row_ = id.row_;
46 _id.column_ = id.column_;
47 }
48
49
50 bool isActive() const { return _status; }
51 void activate(){ _status = true; }
52 void deactivate() {
53 _relative_index = {0,0};
54 _group_id = -1;
55 _status = false;
56 }
```

```

57 Index ID() const { return _id; }
58 /*
59 * Group get_ID is the set_ID of the cluster they are in
60 */
61 int     get_groupID() const { return _group_id; }
62 void    set_groupID(int g_id) {_group_id = g_id;}
63
64 std::stringstream getSite() const {
65 std::stringstream ss;
66 if(isActive())
67 ss << _id;
68 else
69 ss << "(*)";
70 return ss;
71 }
72
73
74 void relativeIndex(IndexRelative r){
75 _relative_index = r;
76 }
77
78 void relativeIndex(int x, int y){
79 _relative_index = {x,y};
80 }
81
82 IndexRelative relativeIndex() const {return _relative_index;}
83 };
84
85 std::ostream& operator<<(std::ostream& os, const Site& site);
86 bool operator==(Site& site1, Site& site2);
87 #endif

```

### The src/lattice/site.cpp file

```

1 #include <iomanip>
2 #include "site.h"
3
4 std::ostream& operator<<(std::ostream& os, const Site& site)
5 {
6 if(site.isActive())
7 return os << site._id;
8 else
9 return os << "(*)";
10 }
11

```

```

12
13 bool operator==(Site& site1 , Site& site2){
14     return (site1.ID().row_ == site2.ID().row_) && (site1.ID().column_ ==
15         site2.ID().column_);

```

### A.2.3 Bond

The bond class contains all information about a bond, e.g., if it is active or not and if it is then what is its group id.

The `src/lattice/bond.h` file

```

1 #ifndef SITEPERCOLATION_BOND_H
2 #define SITEPERCOLATION_BOND_H
3
4
5 #include <ostream>
6 #include <iostream>
7 #include <sstream>
8
9 #include "../index/index.h"
10 #include "../types.h"
11
12
13 /**
14 * A bond has two end
15 * say a 5x5 lattice bond between end1 (0,0) and end2 (0,1)
16 * if _status is false -> bond is not there
17 *
18 */
19 struct Bond{
20 // check if active or not
21 bool _status{false};
22 value_type _length;
23 int _group_id{-1};
24
25 BondType bondType;
26 // relative distance from the root site. {0,0} if it is the root site
27 IndexRelative _relative_index{0,0};
28 Index _end1;
29 Index _end2;
30 BondIndex _id;
31

```

```
32 ~Bond() = default;
33 Bond() = default;
34 Bond(Index end1, Index end2, value_type length){
35     _end1.row_ = end1.row_;
36     _end1.column_ = end1.column_;
37     _end2.row_ = end2.row_;
38     _end2.column_ = end2.column_;
39
40     _length = length;
41     // check if the bond is valid
42     if(_end1.row_ == _end2.row_){
43         bondType = BondType::Horizontal;
44         // means x_ values are equal
45         if(_end1.column_ > _end2.column_){
46             if(_end1.column_ == _length-1 && _end2.column_ ==0){
47                 // do nothing
48             }
49         } else {
50             // sort them out
51             _end1.column_ = end2.column_;
52             _end2.column_ = end1.column_;
53         }
54     }
55     else if(_end1.column_ < _end2.column_){
56         if(_end1.column_ == 0 && _end2.column_ == _length-1){
57             _end1.column_ = end2.column_;
58             _end2.column_ = end1.column_;
59         }
60     }
61 }
62 else if(_end1.column_ == _end2.column_){
63     bondType = BondType::Vertical;
64     // means y_ values are equal
65     if(_end1.row_ > _end2.row_){
66         if(_end1.row_ == _length-1 && _end2.row_ ==0){
67             }
68         } else {
69             // sort them out
70             _end1.row_ = end2.row_;
71             _end2.row_ = end1.row_;
72         }
73     }
74 else if(_end1.row_ < _end2.row_){
75     if(_end1.row_ == 0 && _end2.row_ == _length-1){
```

```
76 _end1.row_ = end2.row_;
77 _end2.row_ = end1.row_;
78 }
79 }
80 }
81 else{
82 std::cout << '(' << _end1.row_ << ',' << _end1.column_ << ')'
83 << '(' << _end2.row_ << ',' << _end2.column_ << ')'
84 << " is not a valid bond : line " << __LINE__ << std::endl;
85 }
86
87 _id = BondIndex(bondType, _end1.row_, _end1.column_); // unsigned long
88 }
89
90 std::vector<Index> getSites() const { return {_end1, _end2};}
91
92 Index id() const {
93 return _end1;
94 }
95
96 BondIndex ID() const {
97 return _id;
98 }
99
100 void activate() {_status = true;}
101 void deactivate() {
102 _relative_index = {0,0};
103 _group_id = -1;
104 _status = false;
105 }
106 bool isActive() const { return _status;}
107 /*
108 * Group get_ID is the set_ID of the cluster they are in
109 */
110 int get_groupID() const {return _group_id;}
111 void set_groupID(int g_id) {_group_id = g_id;}
112
113 std::stringstream getBondString() const {
114 std::stringstream ss;
115 if(isActive()) {
116 // place '-' for horizontal bond and '|' for vertical bond
117 if(bondType == BondType::Horizontal) {
118 ss << '(' << _end1 << "<->" << _end2 << ')';
119 }
```

```

120 else {
121 ss << '(' << _end1 << "<|>" << _end2 << ')';
122 }
123 }
124 else
125 ss << "(**)";
126 return ss;
127 }
128
129 bool isHorizontal() const { return bondType == BondType::Horizontal; }
130 bool isVertical() const { return bondType == BondType::Vertical; }
131
132 void relativeIndex(IndexRelative r){
133 _relative_index = r;
134 }
135
136 void relativeIndex(int x, int y){
137 _relative_index = {x,y};
138 }
139
140 IndexRelative relativeIndex() const {return _relative_index;}
141 };
142
143
144 std::ostream& operator<<(std::ostream& os, const Bond& bond);
145 bool operator==(Bond a, Bond b);
146 bool operator<(const Bond& bond1, const Bond& bond2);
147 bool operator>(const Bond& bond1, const Bond& bond2);
148
149 #endif // SITEPERCOLATION_BOND_H

```

### The src/lattice/bond.cpp file

```

1 #include "bond.h"
2
3 /**
4 * use '-' and '|' in between '<>' to indicate horizontal or vertical
5 * bond
6 * ((0,1)<->(0,0)) for horizontal bond
7 * ((1,1)<|>(0,1)) for vertical bond
8 * @param os
9 * @param bond
10 * @return
11 */
11 std::ostream& operator<<(std::ostream& os, const Bond& bond)

```

```
12 {
13 if(bond.isActive()) {
14 // place '-' for horizontal bond and '|' for vertical bond
15 if(bond.isHorizontal()) {
16 return os << '(' << bond._end1 << "<->" << bond._end2 << ')';
17 }
18 return os << '(' << bond._end1 << "<|>" << bond._end2 << ')';
19 }
20 else
21 return os << "(**)";
22 }
23
24 bool operator==(Bond a, Bond b)
25 {
26 if(a.isHorizontal() && b.isHorizontal())
27 {
28 return (a.id().row_ == b.id().row_) && (a.id().column_ == b.id().column_);
29 }
30 if(a.isVertical() && b.isVertical()){
31 return (a.id().row_ == b.id().row_) && (a.id().column_ == b.id().column_);
32 }
33
34 return false;
35 }
36
37 bool operator<(const Bond& bond1, const Bond& bond2){
38 if(bond1.isHorizontal() && bond2.isHorizontal()){
39 return bond1._end1.column_ < bond2._end1.column_;
40 }
41 if(bond1.isVertical() && bond2.isVertical()){
42 return bond1._end1.row_ < bond2._end1.row_;
43 }
44 return bond1.isHorizontal();
45 }
46
47
48 bool operator>(const Bond& bond1, const Bond& bond2){
49 if(bond1.isHorizontal() && bond2.isHorizontal()){
50 return bond1._end1.column_ > bond2._end1.column_;
51 }
52 if(bond1.isVertical() && bond2.isVertical()){
53 return bond1._end1.row_ > bond2._end1.row_;
```

```

54 }
55 return bond1.isVertical();
56 }
```

The `src/lattice/bond_type.cpp` file

```

1 #ifndef PERCOLATION_BOND_V2_H
2 #define PERCOLATION_BOND_V2_H
3 /**
4 * Only two type bonds in 2D lattice
5 */
6 enum class BondType{
7 Horizontal,
8 Vertical
9 };
10#endif //PERCOLATION_BOND_V2_H
```

## A.2.4 Lattice

The lattice class consists of array of sites and bonds. This class contains information about lattice size. And contains functions to view the lattice differently in the console.

the `src/lattice/lattice.h` file

```

1 #ifndef SITEPERCOLATION_LATTICE_H
2 #define SITEPERCOLATION_LATTICE_H
3
4 #include <vector>
5 #include <cmath>
6
7 #include "../percolation/cluster.h"
8 #include "../types.h"
9 #include "site.h"
10 #include "bond.h"
11
12
13 /**
14 * The square Lattice
15 * Site and Bonds are always present But they will not be counted unless
16 * they are activated
17 * always return by references , so that values in the class itself is
18 * modified
19 */
20 class SqLattice {
21 std::vector<std::vector<Site>> _sites; // holds all the sites
```

```
20 std :: vector<std :: vector<Bond>> _h_bonds; // holds all horizontal bonds
21 std :: vector<std :: vector<Bond>> _v_bonds; // holds all vertical bonds
22
23 bool _bond_resetting_flag=true; // so that we can reset all bonds
24 bool _site_resetting_flag=true; // and all sites
25
26 value_type _length {};
27
28 private:
29 void reset_sites();
30 void reset_bonds();
31 public:
32 ~SqLattice() = default;
33 SqLattice() = default;
34 SqLattice(SqLattice&) = default;
35 SqLattice(SqLattice&&) = default;
36 SqLattice& operator=(const SqLattice&) = default;
37 SqLattice& operator=(SqLattice&&) = default;
38
39 SqLattice(value_type length, bool activate_bonds, bool activate_sites,
        bool bond_reset, bool site_reset);
40
41 void reset(bool reset_all=false);
42
43
44 /* **** */
45 * I/O functions
46 ****
47 void view_sites();
48 void view_sites_extended();
49 void view_sites_by_id();
50 void view_sites_by_relative_index();
51 void view_bonds_by_relative_index_v4();
52 void view_by_relative_index();
53 void view(); // view lattice bonds and sites together
54
55 void view_h_bonds();
56 void view_v_bonds();
57
58 void view_bonds(){
59 view_h_bonds();
60 view_v_bonds();
61 }
62
```

```

63
64 void view_bonds_by_id();
65
66 /* **** Activation functions **** */
67 * Activation functions
68 ****
69 void activate_site(Index index);
70 void activateBond(BondIndex bond);
71
72 void deactivate_site(Index index);
73 void deactivate_bond(Bond bond);
74
75 value_type length() const { return _length; }
76
77 Site& getSite(Index index);
78 Bond& getBond(BondIndex);
79
80 const Site& getSite(Index index) const ;
81 const Bond& getBond(BondIndex index) const ;
82
83 void setGroupID(Index index, int group_id);
84 void setGroupID(BondIndex index, int group_id);
85 const int getGroupID(Index index) const;
86 const int getGroupID(BondIndex index) const;
87
88 /*
89 * Get Neighbor from given index
90 ****
91 */
92 std::vector<Index> get_neighbor_site_indices(Index site); // site
93 std::vector<BondIndex> get_neighbor_bond_indices(BondIndex site); // bond
94 std::vector<Index> get_neighbor_indices(BondIndex bond); // two site
95 neighbor of bond.
96 static std::vector<Index> get_neighbor_site_indices(size_t length, Index
97 site); // 4 site neighbor of site
98 static std::vector<BondIndex> get_neighbor_bond_indices(size_t length,
99 BondIndex site); // 6 bond neighbor of bond

```

```

98 static std :: vector<Index> get_neighbor_indices(size_t length , BondIndex
99 bond); // 2 site neighbor of bond.
100 };
101
102
103 #endif // SITEPERCOLATION_LATTICE_H

```

### The src/lattice/lattice.cpp file

```

1 //
2 // Created by shahnoor on 10/2/2017.
3 //
4
5 #include <iomanip>
6 #include "lattice.h"
7 #include "../util/printer.h"
8
9 using namespace std;
10
11
12 /**
13 *
14 * @param length      -> length of the lattice
15 * @param activate_bonds -> if true all bonds are activated by default
16 *                           and will not be deactivated as long as the
17 *                           object exists ,
18 *                           even if SLattice::reset function is called.
19 * @param activate_sites -> if true all sites are activated by default
20 *                           and will not be deactivated as long as the
21 *                           object exists ,
22 *                           even if SLattice::reset function is called.
23 */
24 SqLattice::SqLattice(
25     value_type length ,
26     bool activate_bonds , bool activate_sites ,
27     bool bond_reset , bool site_reset )
28 : _length{length} , _bond_resetting_flag{bond_reset} ,
29   _site_resetting_flag{site_reset}
30 {
31     cout << "Constructing Lattice object : line " << __LINE__ << endl;
32     _sites = std :: vector<std :: vector<Site>>(_length);
33     _h_bonds = std :: vector<std :: vector<Bond>>(_length);
34     _v_bonds = std :: vector<std :: vector<Bond>>(_length);

```

```

32 if (!activate_bonds && !activate_sites) {      // both are deactivated by
33     default
34     for (value_type i{}; i != _length; ++i) {
35         _sites[i] = std::vector<Site>(_length);
36         _h_bonds[i] = std::vector<Bond>(_length);
37         _v_bonds[i] = std::vector<Bond>(_length);
38         for (value_type j{}; j != _length; ++j) {
39             _sites[i][j] = Site(Index(i, j), _length);
40             _h_bonds[i][j] = {Index(i, j), Index(i, (j + 1) % _length), _length};
41             _v_bonds[i][j] = {Index(i, j), Index((i + 1) % _length, j), _length};
42         }
43     }
44 } else if (activate_bonds && !activate_sites) {      // all bonds are
45     activated by default
46     for (value_type i{}; i != _length; ++i) {
47         _sites[i] = std::vector<Site>(_length);
48         _h_bonds[i] = std::vector<Bond>(_length);
49         _v_bonds[i] = std::vector<Bond>(_length);
50         for (value_type j{}; j != _length; ++j) {
51             _sites[i][j] = Site(Index(i, j), _length);
52             _h_bonds[i][j] = {Index(i, j), Index(i, (j + 1) % _length), _length};
53             _v_bonds[i][j] = {Index(i, j), Index((i + 1) % _length, j), _length};
54             _h_bonds[i][j].activate();
55             _v_bonds[i][j].activate();
56         }
57     }
58 } else if (!activate_bonds && activate_sites) {      // all sites are
59     activated by default
60     for (value_type i{}; i != _length; ++i) {
61         _sites[i] = std::vector<Site>(_length);
62         _h_bonds[i] = std::vector<Bond>(_length);
63         _v_bonds[i] = std::vector<Bond>(_length);
64         for (value_type j{}; j != _length; ++j) {
65             _sites[i][j] = Site(Index(i, j), _length);
66             _sites[i][j].activate();
67             _h_bonds[i][j] = {Index(i, j), Index(i, (j + 1) % _length), _length};
68             _v_bonds[i][j] = {Index(i, j), Index((i + 1) % _length, j), _length};
69         }
70     }
71 } else {

```

```
72 for (value_type i{}; i != _length; ++i) { // all bonds and sites are
    activated by default
73 _sites[i] = std::vector<Site>(_length);
74 _h_bonds[i] = std::vector<Bond>(_length);
75 _v_bonds[i] = std::vector<Bond>(_length);
76 for (value_type j{}; j != _length; ++j) {
77 _sites[i][j] = Site(Index(i, j), _length);
78 _sites[i][j].activate();
79 _h_bonds[i][j] = {Index(i, j), Index(i, (j + 1) % _length), _length};
80 _v_bonds[i][j] = {Index(i, j), Index((i + 1) % _length, j), _length};
81 _h_bonds[i][j].activate();
82 _v_bonds[i][j].activate();
83 }
84 }
85 }
86
87 }
88
89 /* *****
90 * Activation and Deactivation
91 *****/
92
93
94 void SqLattice::activate_site(Index index) {
95 _sites[index.row_][index.column_].activate();
96 }
97
98
99 void SqLattice::activateBond(BondIndex bond) {
100 // check if the bond is vertical or horizontal
101 // then call appropriate function to activate _h_bond or _v_bond
102 if(bond.horizontal()){ // horizontal
103 if(_h_bonds[bond.row_][bond.column_].isActive()){
104 cout << "Bond is already activated : line " << __LINE__ << endl;
105 }
106 _h_bonds[bond.row_][bond.column_].activate();
107 }
108 else if(bond.vertical()) // vertical
109 {
110 if(_v_bonds[bond.row_][bond.column_].isActive()){
111 cout << "Bond is already activated : line " << __LINE__ << endl;
112 }
113 _v_bonds[bond.row_][bond.column_].activate();
114 }
```

```

115 else {
116 cout << bond << " is not a valid bond : line " << __LINE__ << endl;
117 }
118 }
119 }
120
121
122 void SqLattice::deactivate_site(Index index){
123 _sites[index.row_][index.column_].deactivate();
124 }
125
126
127 void SqLattice::deactivate_bond(Bond bond) {
128 // check if the bond is vertical or horizontal
129 // then call appropriate function to activate _h_bond or _v_bond
130 if(bond.isHorizontal()){
131 if(_h_bonds[bond.id().row_][bond.id().column_].isActive()){
132 cout << "Bond is already activated : line " << __LINE__ << endl;
133 }
134 _h_bonds[bond.id().row_][bond.id().column_].deactivate();
135 }
136 else if(bond.isVertical())
137 {
138 if(_v_bonds[bond.id().row_][bond.id().column_].isActive()){
139 cout << "Bond is already activated : line " << __LINE__ << endl;
140 }
141 _v_bonds[bond.id().row_][bond.id().column_].deactivate();
142 }
143 else{
144 bond.activate();
145 cout << bond << " is not a valid bond : line " << __LINE__ << endl;
146 }
147 }
148
149
150 /* ****
151 * Viewing methods
152 **** */
153 /**
154 * View the sites of the lattice
155 * place (*) if the site is not active
156 */
157 void SqLattice::view_sites()
158 {

```

```

159 std::cout << "view sites" << std::endl;
160 std::cout << '{';
161 for(value_type i{} ; i != _length ; ++i) {
162 if(i!=0) std::cout << " ";
163 else std::cout << '{';
164 for (value_type j{}; j != _length; ++j) {
165 if(_sites[i][j].isActive()){
166 std::cout << _sites[i][j] ;
167 }
168 else{
169 std::cout << "(*";
170 }
171 if(j != _length-1)
172 std::cout << ',';
173 }
174 std::cout << '}';
175 if(i != _length-1)
176 std::cout << std::endl;
177 }
178 std::cout << '}';
179 std::cout << std::endl;
180 }

181 /**
182 * View the sites of the lattice
183 * place (*) if the site is not active
184 * Shows the group_id along with sites
185 *
186 *
187 * Very good output format. Up to lattice size < 100
188 */
189 void SqLattice::view_sites_extended()
190 {
191 std::cout << "view sites" << std::endl;
192 std::cout << '{';
193 for(value_type i{} ; i != _length ; ++i) {
194 if(i!=0) std::cout << " ";
195 else std::cout << '{';
196 for (value_type j{}; j != _length; ++j) {
197 std::cout << std::setw(3) << _sites[i][j].get_groupID() << ":";
198 if(_sites[i][j].isActive()) {
199 cout << '(' << std::setw(2) << _sites[i][j]._id.row_ << ','
200 << std::setw(2) << _sites[i][j]._id.column_ << ')';
201 }
}

```

```

203 else {
204 cout << std::setw(7) << "(*)" ;
205 }
206
207 if(j != _length -1)
208 std::cout << ',' ;
209 }
210 std::cout << '}' ;
211 if(i != _length -1)
212 std::cout << std::endl ;
213 }
214 std::cout << '}' ;
215 std::cout << std::endl ;
216 }
217
218
219
220 /**
221 * Displays group ids of sites in a matrix form
222 */
223 void SqLattice::view_sites_by_id() {
224 std::cout << "Sites by id : line " << __LINE__ << endl ;
225 cout << "    ";
226 for(value_type j{}; j != _length; ++ j){
227 cout << " " << setw(3) << j ;
228 }
229 cout << endl << " __|";
230 for(value_type j{}; j != _length; ++ j){
231 cout << " __ ";
232 }
233 cout << endl ;
234 for(value_type i{} ; i != _length; ++i){
235 cout << setw(3) << i << "|";
236 for(value_type j{} ; j != _length ; ++ j){
237 cout << setw(3) << _sites[i][j].get_groupID() << ' ' ;
238 }
239 cout << endl ;
240 }
241 }
242
243
244 /**
245 *
246 */

```

```

247 void SqLattice::view_sites_by_relative_index(){
248 std::cout << "Relative index : line " << __LINE__ << endl;
249 cout << "Format: \"id(x,y)\" " << endl;
250 cout << "    |";
251 for(value_type j{}; j != _length; ++ j){
252 cout << setw(4) << j << "        |";
253 }
254 cout << endl;
255 print_h_barrier(_length , "___|__" , "-----|__");
256 for(value_type i{} ; i != _length; ++i){
257 cout << setw(3) << i << " |";
258 for(value_type j{} ; j != _length ; ++ j){
259 if(_sites[i][j].get_groupID() == -1){
// left blank
260 cout << setw(4) << _sites[i][j].get_groupID() << "        |";
261 continue;
262 }
263 cout << setw(4) << _sites[i][j].get_groupID() << _sites[i][j].
relativeIndex() << " |";
264 }
265 cout << endl;
266 print_h_barrier(_length , "___|__" , "-----|__");
267 }
268 //    print_h_barrier(_length , "___|__" , "-----|_");
269 }
270 }

271
272
273
274
275
276 /**
277 * View bonds in the lattice by relative index. id of the site is showed
278 * format : id(relative index) for site and only id for bond
279 * if any site is isolated relative index is not shown
280 */
281 void SqLattice::view_bonds_by_relative_index_v4() {
282
283 std::cout << "Bonds by id : line " << __LINE__ << endl;
284 // cout << "site id -1 means isolated site and 0 means connected site in
285 // bond percolation(definition)" << endl;
286 print_h_barrier(15 , "_" , "___" , "\n");
287 cout << "|(site id) (horizontal bond id(relative index))|" << endl;
288 cout << "|(vertical bond id(relative index))" << endl;
289 print_h_barrier(15 , "_" , "___" , "\n");

```

```

289 // printing indices for columns
290 std::cout << "    | ";
291 for(value_type i{}; i != _length; ++i){
292     std::cout << i << "           | ";
293 }
294 std::cout << std::endl;
295
296 // printing H,V label
297
298 print_h_barrier(_length, "      | ", " V           H | ");
299 print_h_barrier(_length, "_____|__", "-----|__");
300
301 // for each row there will be two columns
302 for(value_type i{}; i != _length; ++i){
303     std::cout << i << ' ';
304     std::cout << "H | ";
305     for(value_type j1{}; j1 != _length; ++j1){
306         int id = _sites[i][j1].get_groupID();
307         std::cout << std::setw(3) << id << " ";
308         if(id != -1){
309             cout << _sites[i][j1].relativeIndex();
310         } else {
311             cout << "(-,-)   ";
312         }
313         std::cout << "    " << std::setw(3) << _h_bonds[i][j1].get_groupID() << " ";
314     }
315     std::cout << std::endl;
316     print_h_barrier(_length, "      | ", "           | "); // just to
317     // see a better view
318     std::cout << "    " << "V | ";
319     for(value_type j2{}; j2 != _length; ++j2){
320         std::cout << std::setw(3) << _v_bonds[i][j2].get_groupID() << " ";
321     }
322     std::cout << std::endl;
323
324 // printing horizontal separator
325 print_h_barrier(_length, "_____|__", "-----|__");
326
327 std::cout << std::endl;
328 }
```

```
330
331
332 /**
333 * View lattice (sites and bonds) by relative index.
334 * format : id(relative_index)
335 */
336 void SqLattice::view_by_relative_index() {
337
338 std::cout << "Bonds by id : line " << __LINE__ << endl;
339 // printing indices for columns
340 std::cout << "      | ";
341 for(value_type i{}; i != _length; ++i){
342 std::cout << i << "           | ";
343 }
344 std::cout << std::endl;
345
346 // printing H,V label
347
348 print_h_barrier(_length, "      | ", " V           H   |   ");
349 print_h_barrier(_length, "_____|__", "_____|__");
350
351 // for each row there will be two columns
352 for(value_type i{}; i != _length; ++i){
353 std::cout << i << ' ';
354 std::cout << "H | ";
355 for(value_type j1{}; j1 != _length; ++j1){
356 std::cout << std::setw(3) << _sites[i][j1].get_groupID() << _sites[i][j1]
357 .relativeIndex();
358 std::cout << "    " << std::setw(3) << _h_bonds[i][j1].get_groupID()
359 << _h_bonds[i][j1].relativeIndex() << " | ";
360 }
361 std::cout << std::endl;
362 print_h_barrier(_length, "      | ", "           |   "); // just to see a better view
363 std::cout << "    " << "V | ";
364 for(value_type j2{}; j2 != _length; ++j2){
365 std::cout << std::setw(3) << _v_bonds[i][j2].get_groupID()
366 << _v_bonds[i][j2].relativeIndex() << "           | ";
367 }
368 std::cout << std::endl;
369
370 // printing horizontal separator
371 print_h_barrier(_length, "_____|__", "_____|__");
372 }
```

```

372 //           view_h_bonds_extended();
373 //           view_v_bonds_extended();
374 std::cout << std::endl;
375
376 }
377
378 /**
379 * View lattice (sites and bonds) by relative index.
380 * format : id(relative_index)
381 */
382
383 void SqLattice::view() {
384
385 std::cout << "Bonds by id : line " << __LINE__ << endl;
386 cout << "Structure " << endl;
387 print_h_barrier(10, " ", "___", "\n");
388 cout << "|(site id) (horizontal bond id)|" << endl;
389 cout << "|(vertical bond id)           |" << endl;
390 print_h_barrier(10, " ", "___", "\n");
391 // printing indices for columns
392 std::cout << "    | ";
393 for(value_type i{}; i != _length; ++i){
394 std::cout << i << "    | ";
395 }
396 std::cout << std::endl;
397
398 // bringing H,V label
399
400 print_h_barrier(_length, "      | ", "V      H|   ");
401 print_h_barrier(_length, "_____|__", "_____|__");
402
403 // for each row there will be two columns
404 for(value_type i{}; i != _length; ++i){
405 std::cout << i << ' ';
406 std::cout << "H | ";
407 for(value_type j1{}; j1 != _length; ++j1){
408 std::cout << std::setw(3) << _sites[i][j1].get_groupID() ;
409 std::cout << " " << std::setw(3) << _h_bonds[i][j1].get_groupID() << " |
410     ";
411 }
412 std::cout << std::endl;
413 print_h_barrier(_length, "      | ", "      | "); // just to see a
414     better view
415 std::cout << " " << "V | ";

```

```
414 for(value_type j2{}; j2 != _length; ++j2){  
415     std::cout << std::setw(3) << _v_bonds[i][j2].get_groupID() << "      |";  
416 }  
417 std::cout << std::endl;  
418  
419 // printing horizontal separator  
420 print_h_barrier(_length, "_____|__", "_____|__");  
421 }  
422  
423 //      view_h_bonds_extended();  
424 //      view_v_bonds_extended();  
425 std::cout << std::endl;  
426  
427 }  
428  
429  
430 /**  
431 *  
432 */  
433 void SqLattice::view_h_bonds()  
434 {  
435     std::cout << "view horizontal bonds" << std::endl;  
436     std::cout << '{';  
437     for(value_type i{}; i != _length; ++i) {  
438         if(i!=0) std::cout << "  ";  
439         else std::cout << '{';  
440         for (value_type j{}; j != _length; ++j) {  
441             std::cout << _h_bonds[i][j] ;  
442             if(j != _length-1)  
443                 std::cout << ',';  
444         }  
445         std::cout << '}';  
446         if(i != _length-1)  
447             std::cout << std::endl;  
448     }  
449     std::cout << '}' ;  
450     std::cout << std::endl;  
451 }  
452  
453 /**  
454 *  
455 */  
456 void SqLattice::view_v_bonds()  
457 {
```

```

458 std::cout << "view vertical bonds" << std::endl;
459 std::cout << '{';
460 for(value_type i{} ; i != _length ; ++i) {
461 if(i!=0) std::cout << " ";
462 else std::cout << '{';
463 for (value_type j{}; j != _length; ++j) {
464 std::cout << _v_bonds[i][j] ;
465 if(j != _length-1)
466 std::cout << ',';
467 }
468 std::cout << '}';
469 if(i != _length-1)
470 std::cout << std::endl;
471 }
472 std::cout << '}';
473 std::cout << std::endl;
474 }

475

476

477 /**
478 *
479 */
480 void SqLattice::view_bonds_by_id(){
481 std::cout << "Bonds by id : line " << __LINE__ << endl;
482 cout << "Structure " << endl;
483 print_h_barrier(8, "___", "___", "\n");
484 cout << "|      (horizontal bond id)|" << endl;
485 cout << "|(vertical bond id)      |" << endl;
486 print_h_barrier(8, "___", "___", "-\n");
487 // printing indices for columns
488 std::cout << "      |";
489 for(value_type i{}; i != _length; ++i){
490 std::cout << i << "      |";
491 }
492 std::cout << std::endl;
493

494 // bringing H,V label
495

496 print_h_barrier(_length, "      | ", "V H| ");
497 print_h_barrier(_length, "_____|__", "_____|__");
498

499 // for each row there will be two columns
500 for(value_type i{}; i != _length; ++i){
501 std::cout << i << ' ';

```

```
502 std::cout << "H | ";
503 for(value_type j1{}; j1 != _length; ++j1){
504     std::cout << "    " << std::setw(3) << _h_bonds[i][j1].get_groupID() << "|";
505 }
506 std::cout << std::endl;
507 std::cout << "    " << "V | ";
508 for(value_type j2{}; j2 != _length; ++j2){
509     std::cout << std::setw(3) << _v_bonds[i][j2].get_groupID() << "    |";
510 }
511 std::cout << std::endl;
512
513 // printing horizontal separator
514 print_h_barrier(_length, "_____|__", "_____|__");
515 }
516
517 //      view_h_bonds_extended();
518 //      view_v_bonds_extended();
519 std::cout << std::endl;
520 }
521
522
523 Site& SqLattice::getSite(Index index) {
524     return _sites[index.row_][index.column_];
525 }
526
527
528 const Site& SqLattice::getSite(Index index) const {
529     return _sites[index.row_][index.column_];
530 }
531
532 const Bond& SqLattice::getBond(BondIndex index) const {
533     if (index.horizontal()) {
534         return _h_bonds[index.row_][index.column_];
535     }
536     if (index.vertical()) {
537         return _v_bonds[index.row_][index.column_];
538     }
539     throw InvalidBond{"Invalid bond : line " + to_string(__LINE__));
540 }
541
542 void SqLattice::setGroupID(Index index, int group_id){
543     _sites[index.row_][index.column_].set_groupID(group_id);
544 }
```

```
545 void SqLattice::setGroupID(BondIndex index, int group_id){  
546     if(index.horizontal()){  
547         _h_bonds[index.row_][index.column_].set_groupID(group_id);  
548     }  
549     if(index.vertical()){  
550         _v_bonds[index.row_][index.column_].set_groupID(group_id);  
551     }  
552 }  
553 }  
554  
555 const int SqLattice::getGroupID(Index index) const{  
556     return _sites[index.row_][index.column_].get_groupID();  
557 }  
558  
559 const int SqLattice::getGroupID(BondIndex index) const{  
560     if(index.horizontal()){  
561         return _h_bonds[index.row_][index.column_].get_groupID();  
562     }  
563     if(index.vertical()){  
564         return _v_bonds[index.row_][index.column_].get_groupID();  
565     }  
566     return -1;  
567 }  
568  
569  
570 Bond& SqLattice::getBond(BondIndex index) {  
571     if(index.horizontal())  
572         return _h_bonds[index.row_][index.column_];  
573     if(index.vertical())  
574         return _v_bonds[index.row_][index.column_];  
575     throw InvalidBond{"Invalid bond : line " + to_string(__LINE__)};  
576 }  
577  
578 void SqLattice::reset(bool reset_all) {  
579     if(reset_all){  
580         reset_sites();  
581         reset_bonds();  
582         return;  
583     }  
584     // setting all group id to -1  
585     if(_site_resetting_flag) {  
586         reset_sites();  
587     }  
588     // cout << "Bond resetting is disabled : line " << __LINE__ << endl;
```

```
589 if(_bond_resetting_flag) {  
590     reset_bonds();  
591 }  
592 }  
593 }  
594  
595 /**/  
596 *  
597 */  
598 void SqLattice::reset_bonds() {  
599     for(value_type i{}; i != _h_bonds.size(); ++i){  
600         for (int j{}; j != _h_bonds[i].size(); ++j) {  
601             // deactivating. automatically set group id == -1 and relative index ==  
602             // (0,0)  
603             // setting group id = -1 and deactivating the bond  
604             _h_bonds[i][j].deactivate();  
605             _v_bonds[i][j].deactivate();  
606         }  
607     }  
608 }  
609 }  
610  
611 /**/  
612 *  
613 */  
614 void SqLattice::reset_sites() {  
615     for(value_type i{}; i != _sites.size(); ++i){  
616         for(value_type j{}; j != _sites[i].size(); ++j) {  
617             // deactivating. automatically set group id == -1 and relative index ==  
618             // (0,0)  
619             // setting group id = -1 and deactivating the site  
620             _sites[i][j].deactivate();  
621         }  
622     }  
623 }  
624  
625 /*  
*****  
626 * Get Neighbor from given index  
627 *****  
*/
```

```

628 /**
629 * Periodic case only.
630 * Each site has four neighbor sites.
631 * @param site
632 * @return
633 */
634 std::vector<Index> SqLattice::get_neighbor_site_indices(Index site){
635 std::vector<Index> sites(4);
636 sites[0] = {(site.row_ + 1) % _length, site.column_};
637 sites[1] = {((site.row_ - 1 + _length) % _length, site.column_};
638 sites[2] = {site.row_, (site.column_ + 1) % _length};
639 sites[3] = {site.row_, (site.column_ - 1 + _length) % _length};
640 return sites;
641 }
642
643 /**
644 * Periodic case only.
645 * Each bond has six neighbor bonds.
646 * @param site
647 * @return
648 */
649 std::vector<BondIndex> SqLattice::get_neighbor_bond_indices(BondIndex
   bond) {
650 value_type next_column = (bond.column_ + 1) % _length;
651 value_type prev_column = (bond.column_ - 1 + _length) % _length;
652 value_type prev_row = (bond.row_ - 1 + _length) % _length;
653 value_type next_row = (bond.row_ + 1) % _length;
654
655 vector<BondIndex> bonds(6);
656
657 // horizontal bond case
658 if (bond.horizontal()) {
659 // increase column index for the right neighbor
660
661 // left end of bond
662 bonds[0] = {BondType::Vertical, bond.row_, bond.column_};
663 bonds[1] = {BondType::Vertical, prev_row, bond.column_};
664 bonds[2] = {BondType::Horizontal, bond.row_, prev_column};
665
666 // right end bond
667 bonds[3] = {BondType::Vertical, prev_row, next_column};
668 bonds[4] = {BondType::Vertical, bond.row_, next_column};
669 bonds[5] = {BondType::Horizontal, bond.row_, next_column};
670

```

```
671 }
672 // vertical bond case
673 else if (bond.vertical()) {
674 // increase row index
675
676 // top end of bond
677 bonds[0] = {BondType::Horizontal, bond.row_, bond.column_};
678 bonds[1] = {BondType::Horizontal, bond.row_, prev_column_};
679 bonds[2] = {BondType::Vertical, prev_row_, bond.column_};
680
681 // bottom end of bond
682 bonds[3] = {BondType::Horizontal, next_row_, bond.column_};
683 bonds[4] = {BondType::Horizontal, next_row_, prev_column_};
684 bonds[5] = {BondType::Vertical, next_row_, bond.column_};
685
686 }
687
688
689 return bonds;
690 }
691
692 std::vector<Index> SqLattice::get_neighbor_indices(BondIndex bond) {
693 value_type r = bond.row_;
694 value_type c = bond.column_;
695 vector<Index> sites(2);
696 sites[0] = {r, c};
697 if(bond.horizontal()){
698 sites[1] = {r, (c+1) % _length};
699 } else{
700 sites[1] = {((r+1) % _length, c};
701 }
702 return sites;
703 }
704
705 /* **** */
706 * Static methods
707 */
708 std::vector<Index> SqLattice::get_neighbor_site_indices(size_t length,
    Index site){
709 std::vector<Index> sites(4);
710 sites[0] = {(site.row_ + 1) % length, site.column_};
711 sites[1] = {((site.row_ - 1 + length) % length, site.column_};
712 sites[2] = {site.row_, (site.column_ + 1) % length};
713 sites[3] = {site.row_, (site.column_ - 1 + length) % length};
```

```
714 return sites;
715 }
716
717 /**
718 * Periodic case only.
719 * Each bond has six neighbor bonds.
720 * @param site
721 * @return
722 */
723 std::vector<BondIndex> SqLattice::get_neighbor_bond_indices(size_t
    length, BondIndex bond) {
724     value_type next_column = (bond.column_ + 1) % length;
725     value_type prev_column = (bond.column_ - 1 + length) % length;
726     value_type prev_row = (bond.row_ - 1 + length) % length;
727     value_type next_row = (bond.row_ + 1) % length;
728
729     vector<BondIndex> bonds(6);
730
731     // horizontal bond case
732     if (bond.horizontal()) {
733         // increase column index for the right neighbor
734
735         // left end of bond
736         bonds[0] = {BondType::Vertical, bond.row_, bond.column_};
737         bonds[1] = {BondType::Vertical, prev_row, bond.column_};
738         bonds[2] = {BondType::Horizontal, bond.row_, prev_column};
739
740         // right end bond
741         bonds[3] = {BondType::Vertical, prev_row, next_column};
742         bonds[4] = {BondType::Vertical, bond.row_, next_column};
743         bonds[5] = {BondType::Horizontal, bond.row_, next_column};
744
745     }
746     // vertical bond case
747     else if (bond.vertical()) {
748         // increase row index
749
750         // top end of bond
751         bonds[0] = {BondType::Horizontal, bond.row_, bond.column_};
752         bonds[1] = {BondType::Horizontal, bond.row_, prev_column};
753         bonds[2] = {BondType::Vertical, prev_row, bond.column_};
754
755         // bottom end of bond
756         bonds[3] = {BondType::Horizontal, next_row, bond.column_};
```

```

757 bonds[4] = {BondType::Horizontal, next_row, prev_column};
758 bonds[5] = {BondType::Vertical, next_row, bond.column_};
759 }
760
761
762
763 return bonds;
764 }
765
766 std::vector<Index> SqLattice::get_neighbor_indices(size_t length,
    BondIndex bond) {
767 value_type r = bond.row_;
768 value_type c = bond.column_;
769 vector<Index> sites(2);
770 sites[0] = {r, c};
771 if(bond.horizontal()){
772 sites[1] = {r, (c+1) % length};
773 } else{
774 sites[1] = {(r+1) % length, c};
775 }
776 return sites;
777 }
```

### A.2.5 Exception

Different types of exceptions.

The `src/exception/exception.h` file

```

1 #ifndef PERCOLATION_EXCEPTIONS_H
2 #define PERCOLATION_EXCEPTIONS_H
3
4 #include <string>
5 #include <iostream>
6
7 /**
8 *
9 */
10 struct Mismatch{
11     std::string msg_;
12     size_t line_;
13     Mismatch(size_t line, std::string msg="")
14         :line_{line}, msg_{msg} {}
```

```
16     void what() const {
17         std::cerr << msg_ << "\nId and index mismatch at line " << line_
18         << std::endl;
19     }
20 };
21 /**
22 *
23 */
24 struct InvalidIndex{
25     std::string msg_;
26     InvalidIndex( std::string msg) :msg_{msg} {}
27
28     void what() const {
29         std::cerr << msg_ << std::endl;
30     }
31 };
32 /**
33 *
34 */
35 /**
36 struct InvalidBond{
37     std::string msg_;
38     InvalidBond( std::string msg) :msg_{msg} {}
39
40     void what() const {
41         std::cout << msg_ << std::endl;
42     }
43 };
44
45 /**
46 * When any neighbor is occupied and no suitable neighbor is found , throw
47 * this exception
48 */
49 struct OccupiedNeighbor{
50     std::string msg_;
51     OccupiedNeighbor( std::string msg): msg_{msg}{}
52
53     void what() const {
54         std::cout << msg_ << std::endl;
55     }
56 };
57 }
```

```

58 /**
59 * If the 1st or the 2nd nearest neighbor is not valid
60 */
61 struct InvalidNeighbor{
62     std::string msg_;
63     InvalidNeighbor(std::string msg) :msg_{msg} {}
64
65     void what() const {
66         std::cout << msg_ << std::endl;
67     }
68 };
69
70
71 #endif //PERCOLATION_EXCEPTIONS_H

```

## A.2.6 Cluster

The cluster class contains all information about a cluster. Number of sites and bonds in a cluster and the id of the cluster is contained in a cluster.

The **src/percolation/cluster.h** file

```

1 #ifndef SITEPERCOLATION_CLUSTER_H
2 #define SITEPERCOLATION_CLUSTER_H
3
4 #include <vector>
5 #include <set>
6 #include "../lattice/bond.h"
7 #include "../types.h"
8 #include "../lattice/site.h"
9
10
11 /**
12 * Cluster of bonds and sites
13 * version 3
14 * final goal -> make a template cluster. so that we can use it for Bond
15 * cluster or Site cluster
16 * root site (bond) is the first site (bond) of the cluster. needed for (wrapping)
17 * site percolation
18 */
19 class Cluster{
20     // contains bond and site
21     std::vector<BondIndex> _bond_index; // BondIndex for indexing bonds
22     std::vector<Index> _site_index; // Site index

```

```

21
22 int _creation_time{-1}; // holds the creation birthTime of a cluster
23 object
24 int _id{-1};
25 public:
26 //    using iterator = std::vector<Bond>::iterator;
27
28 ~Cluster() = default;
29 Cluster() = default;
30 Cluster(Cluster&) = default;
31 Cluster(Cluster&&) = default;
32 Cluster& operator=(const Cluster&) = default;
33 Cluster& operator=(Cluster&&) = default;
34
35 explicit Cluster(int id){
36     _id = id; // may be modified in the program
37
38 /**
39 * Only readable, not modifiable.
40 * when time = 0 => only lattice exists and bonds in site percolation,
41 *      not any sites
42 * When id = 0, time = 1 => we have placed the first site, hence created
43 *      a cluster with size greater than 1
44 *      Only then Cluster constructor is called.
45 */
46     _creation_time = id + 1; // only readable, not modifiable
47
48
49 void addSiteIndex(Index );
50 void addBondIndex(BondIndex );
51
52 Index lastAddedSite(){return _site_index.back();}
53 BondIndex lastAddedBond(){return _bond_index.back();}
54
55
56 void insert(const std::vector<BondIndex>& bonds);
57 void insert(const std::vector<Index>& sites);
58
59 void insert(const Cluster& cluster);
60 void insert_v2(const Cluster& cluster);
61 void insert_with_id_v2(const Cluster& cluster, int id);

```

```

62
63
64 friend std :: ostream& operator <<(std :: ostream& os, const Cluster& cluster
   );
65
66 const std :: vector<BondIndex>& getBondIndices() { return _bond_index
   ;}
67 const std :: vector<Index>& getSiteIndices() { return _site_index
   ;}
68
69 const std :: vector<BondIndex>& getBondIndices() const { return
   _bond_index; }
70 const std :: vector<Index>& getSiteIndices() const { return
   _site_index; }
71
72 value_type numberOfBonds() const { return _bond_index.size(); }
73 value_type numberOfSites() const { return _site_index.size(); }
74 int get_ID() const { return _id; }
75 void set_ID(int id) { _id = id; }
76
77 int birthTime() const { return _creation_time; }
78
79 Index getRootSite() const { return _site_index[0]; } // for site percolation
80 BondIndex getRootBond() const { return _bond_index[0]; } // for bond
   percolation
81 bool empty() const { return _bond_index.empty() && _site_index.empty(); }
82 void clear() { _bond_index.clear(); _site_index.clear(); }
83 };
84
85 #endif // SITEPERCOLATION_CLUSTER_H

```

The `src/percolation/cluster.cpp` file

```

1
2 #include "cluster.h"
3
4 using namespace std;
5
6 // add Site index
7 void Cluster::addSiteIndex(Index index) {
8 _site_index.push_back(index);
9 }
10
11 void Cluster::addBondIndex(BondIndex bondIndex) {
12 _bond_index.push_back(bondIndex);

```

```

13 }
14
15
16
17 void Cluster::insert(const std::vector<BondIndex>& bonds){
18 _bond_index.reserve(bonds.size());
19 for(value_type i{} ; i != bonds.size() ; ++i){
20 _bond_index.push_back(bonds[i]);
21 }
22 }
23
24 void Cluster::insert(const std::vector<Index>& sites){
25 _site_index.reserve(sites.size());
26 for(value_type i{} ; i != sites.size() ; ++i){
27 _site_index.push_back(sites[i]);
28 }
29 }
30
31 /**
32 * Merge two cluster as one
33 * All intrinsic property should be considered , e.g., creation time of a
34 * cluster must be recalculated
35 * @param cluster
36 */
37 void Cluster::insert(const Cluster &cluster) {
38 if(_id > cluster._id){
39 cout << "_id > cluster._id : line " << __LINE__ << endl;
40 _id = cluster._id;
41 }
42 // older time or smaller time is the creation birthTime of the cluster
43 // cout << "Comparing " << _creation_time << " and " << cluster.
44 // _creation_time;
45 _creation_time = _creation_time < cluster._creation_time ?
46     _creation_time : cluster._creation_time;
47 // cout << " Keeping " << _creation_time << endl;
48 _bond_index.insert(_bond_index.end() , cluster._bond_index.begin() ,
49                   cluster._bond_index.end());
50 _site_index.insert(_site_index.end() , cluster._site_index.begin() ,
51                   cluster._site_index.end());
52 }
53
54
55 /**
56 * Merge two cluster as one
57 */

```

```
52 * All intrinsic property should be considered , e.g., creation time of a
53 * cluster must be recalculated
54 */
55 void Cluster::insert_v2(const Cluster &cluster) {
56 // older time or smaller time is the creation birthTime of the cluster
57 // cout << "Comparing " << _creation_time << " and " << cluster.
58 // _creation_time;
59 _creation_time = _creation_time < cluster._creation_time ?
60     _creation_time : cluster._creation_time;
61 // cout << " Keeping " << _creation_time << endl;
62 _bond_index.insert(_bond_index.end(), cluster._bond_index.begin(),
63     cluster._bond_index.end());
64 _site_index.insert(_site_index.end(), cluster._site_index.begin(),
65     cluster._site_index.end());
66 }
67
68 void Cluster::insert_with_id_v2(const Cluster &cluster, int id) {
69 _id = id;
70 // older time or smaller time is the creation birthTime of the cluster
71 // cout << "Comparing " << _creation_time << " and " << cluster.
72 // _creation_time;
73 _creation_time = _creation_time < cluster._creation_time ?
74     _creation_time : cluster._creation_time;
75 // cout << " Keeping " << _creation_time << endl;
76 _bond_index.insert(_bond_index.end(), cluster._bond_index.begin(),
77     cluster._bond_index.end());
78 _site_index.insert(_site_index.end(), cluster._site_index.begin(),
79     cluster._site_index.end());
80 }
81
82 std :: ostream &operator<<(std :: ostream &os, const Cluster &cluster) {
83 os << "Sites : size (" << cluster._site_index.size() << ") : ";
84 os << '{';
85 for(auto a: cluster._site_index){
86 os << a << ',' ;
87 }
88 os << '}' << endl;
89
90 os << "Bonds : size (" << cluster._bond_index.size() <<") : ";
91 os << '{';
92 for(auto a: cluster._bond_index){
```

```

87 os << a << ',';
88 }
89 os << '}';
90
91 return os << endl;
92 }
```

### A.2.7 Percolation

The `src/percolation/percolation.h` file

```

1 #ifndef SITEPERCOLATION_PERCOLATION_H
2 #define SITEPERCOLATION_PERCOLATION_H
3
4 #include <vector>
5 #include <set>
6 #include <unordered_set>
7 #include <map>
8 #include <climits>
9 #include <fstream>
10
11
12 #include "../types.h"
13 #include "../lattice/lattice.h"
14 #include "../index/index.h"
15
16 #include <random>
17
18
19 /**
20 * The Square Lattice Percolation class
21 */
22 class SqLatticePercolation{
23 // constants
24 value_type _length;
25 value_type _max_number_of_bonds;
26 value_type _max_number_of_sites;
27 char type{'0'}; // percolation type. 's' -> site percolation. 'b' ->
28     bond percolation
29 protected:
30 // structural variables of lattice
31 SqLattice _lattice;
32 }
```

```
33 value_type _index_sequence_position {};
34
35 std :: vector<Cluster> _clusters; // check and remove reapted index
36 // manually
37 // every birthTime we create a cluster we assign an set_ID for them
38
39 double _occupation_probability {};
40 // entropy
41 double _entropy {};
42 double _entropy_current {};
43 size_t _cluster_count {};
44 value_type _bonds_in_cluster_with_size_two_or_more {0}; // total number
45 // of bonds in the clusters. all cluster has bonds > 1
46 bool _reached_critical = false; // true if the system has reached
47 // critical value
48
49
50
51
52
53 void set_type(char t){type = t;} // setting percolation type
54 public:
55 static constexpr const char* signature = "SqLatticePercolation";
56
57 virtual ~SqLatticePercolation() = default;
58 SqLatticePercolation(value_type length);
59 void reset();
60
61
62 bool occupy();
63 value_type length() const { return _length; }
64 value_type maxSites() const { return _max_number_of_sites; }
65 value_type maxBonds() const { return _max_number_of_bonds; }
66
67 /**
68 * I/O functions
69 */
70 virtual void viewCluster();
71 virtual void viewClusterExtended();
72 virtual void view_bonds() {
73 _lattice.view_bonds();
```

```
74 }
75 virtual void viewLattice(){
76     _lattice.view_sites();
77 }
78 /**
79 * Also shows the cluster index of the sites
80 */
81 virtual void viewLatticeExtended(){
82     _lattice.view_sites_extended();
83 }
84 /**
85 * Displays group ids of sites in a matrix form
86 */
87 virtual void viewLatticeByID(){
88     _lattice.view_sites_by_id();
89     _lattice.view_bonds_by_id();
90 }
91 /**
92 * Displays group ids of sites in a matrix form
93 */
94 virtual void viewSiteByID(){
95     _lattice.view_sites_by_id();
96 }
97 /**
98 * Displays group ids of sites in a matrix form
99 */
100 virtual void viewBondByID(){
101     _lattice.view_bonds_by_id();
102 }
103 /**
104 * Displays group ids of sites in a matrix form
105 */
106 virtual void viewSiteByRelativeIndex(){
107     _lattice.view_sites_by_relative_index();
108 }
109 /**
110 * Displays group ids of sites in a matrix form
111 */
112 virtual void viewBondByRelativeIndex(){
113     _lattice.view_bonds_by_relative_index_v4();
114 }
115 /**
116 * Displays group ids of sites in a matrix form
117 */
118 virtual void viewByRelativeIndex(){
119     _lattice.view_by_relative_index();
120 }
121 /**
122 * Displays group ids of sites in a matrix form
123 */
124 virtual void view(){
125     _lattice.view();
126 }
```

```

118 virtual double occupationProbability() const { return
119     _occupation_probability;}
120 virtual double entropy() { return _entropy_current;}
121 double entropy_by_site(); // for future convenience. // the shannon
122     entropy. the full calculations. time consuming
123 double entropy_by_bond(); // for future convenience. // the shannon
124     entropy. the full calculations. time consuming
125 size_t numberOfcluster() const {return _cluster_count;}
126
127 void get_cluster_info(
128 std :: vector<value_type> &site ,
129 std :: vector<value_type> &bond
130 );
131
132 char get_type() const {return type;} // get percolation type
133 virtual value_type maxIterationLimit() {return _max_iteration_limit;}
134
135
136
137 /**
138 * Site Percolation by Placing Sites
139 *
140 * version 9
141 *
142 * First it randomizes the site index list then use it.
143 * Paradigm Shift:
144 * Does not delete cluster only makes it empty so that index and id
145 * remains the same.
146 * This way Searching for index of the cluster using id can be omitted.
147 *
148 * Feature :
149 * 1. Can turn on and off both horizontal and boundary condition
150 *
151 * 2. Uses class Cluster_v2 for storing clusters
152 *
153 * 3. Uses Group_ID for Bonds and Sites to identify that they are in the
154 * same cluster
155 *
156 * 4. Occupation probability is calculated by sites ,
157 *      i.e., number of active sites divided by total number of sites
158 */

```

```

157 * 5. Spanning is calculated by number of bonds in a spanning clusters
158 *      with periodicity turned off,
159 *
160 *      i.e., number of bonds in the spanning clusters divided by total
161 *      number of bonds
162 *
163 *
164 * 6. Unweighted relabeling is ommited in this version ??
165 *
166 * 7. Runtime is significantly improved. For example , if L=200 program
167 *      will take ~1 min to place all sites .
168 *
169 *
170 *
171 */
172 class SitePercolation_ps_v9 : public SqLatticePercolation{
173 protected:
174 // flags to manipulate method
175 bool _periodicity{false};
176
177 value_type min_index; // minimum index = 0
178 value_type max_index; // maximum index = length - 1
179
180 // index sequence
181 std::vector<Index> index_sequence; // initialized once
182 std::vector<value_type> randomized_index;
183
184 // every birthTime we create a cluster we assign an set_ID for them
185 int _cluster_id{};
186 value_type _index_last_modified_cluster{}; // id of the last modified
187     cluster
188
189 // order parameter calculation ingradients
190 // id of the cluster which has maximum number of bonds. used to
191 // calculate order parameter
192 value_type _number_of_bonds_in_the_largest_cluster{};
193 value_type _number_of_sites_in_the_largest_cluster{}; // might be
194     useful later
195
196 Index _last_placed_site; // keeps track of last placed site

```

```
194
195 /* *****
196 * Spanning variables
197 ***** */
198 /* Holds indices on the edges */
199 std :: vector<Index> _top_edge , _bottom_edge , _left_edge , _right_edge ;
200
201 std :: vector<Index> _spanning_sites ;
202 std :: vector<Index> _wrapping_sites ;
203 std :: vector<value_type> number_of_sites_to_span ;
204 std :: vector<value_type> number_of_bonds_to_span ;
205
206 value_type _total_relabeling {};
207
208 /* *****
209 * Private Methods
210 ***** */
211 void relabel_sites(const std :: vector<Index> &sites , int id_a , int
212 delta_x_ab , int delta_y_ab ) ;
213
214 double time_relabel {};
215 public:
216 static constexpr const char* signature = "SitePercolation_ps_v8";
217
218 ~SitePercolation_ps_v9() = default;
219 SitePercolation_ps_v9() = default;
220 SitePercolation_ps_v9(SitePercolation_ps_v9 &) = default;
221 SitePercolation_ps_v9(SitePercolation_ps_v9 &&) = default;
222 explicit SitePercolation_ps_v9(value_type length , bool periodicity=true)
223 ;
224
225 SitePercolation_ps_v9& operator=(SitePercolation_ps_v9 &) = default;
226 //      SitePercolation_ps_v8&& operator=(SitePercolation_ps_v8 &&) =
227 //      default;
228 double get_relabeling_time() {return time_relabel;}
229 value_type relabeling_count() const {return _total_relabeling;}
230
231 virtual void reset();
232
233 bool periodicity() const {return _periodicity;}
234 std :: string getSignature();
```

```

235 void add_entropy_for_bond(value_type index);
236 void subtract_entropy_for_bond(const std::set<value_type> &
237     found_index_set, int base=-1);
238
239 /* **** Site placing methods ****
240 **** */
241 virtual bool occupy();
242 value_type placeSite_weighted(Index site); // uses weighted relabeling
243         by first identifying the largest cluster
244 value_type placeSite_weighted(Index site,
245     std::vector<Index>& neighbor_sites,
246     std::vector<BondIndex>& neighbor_bonds);
247
248 Index selectSite(); // selecting site
249
250 void connection_v2(Index site, std::vector<Index> &site_neighbor, std::vector<BondIndex> &bond_neighbor);
251
252 // applicable to weighted relabeling
253 void relabel_sites_v5(Index root_a, const Cluster& clstr_b); // relative
254         index is set accordingly
255
256 /* **** Information about current state of Class ****
257 **** */
258 double numberOccupiedSite() const { return _number_of_occupied_sites; }
259 double occupationProbability() const { return double(
260         _number_of_occupied_sites)/maxSites(); }
261 double entropy(); // the shannon entropy
262
263 value_type numberOfBondsInTheLargestCluster_v2();
264 value_type numberOfSitesInTheLargestCluster();
265
266 value_type numberOfSitesInTheSpanningClusters_v2(); ;
267 value_type numberOfBondsInTheSpanningClusters_v2(); ;
268
269 value_type numberOfSitesInTheWrappingClusters(); ;
270 value_type numberOfBondsInTheWrappingClusters(); ;
271
272 /* **** Spanning Detection ****
273 **** */
274 bool detectSpanning_v6(const Index& site);

```

```
274 bool check_if_id_matches(Index site, const std::vector<Index> &edge);  
275  
276 bool detectWrapping();  
277  
278 /* *****  
279 * Tracker  
280 * Must be called each time a site is placed  
281 * *****  
282 void track_numberOfBondsInLargestCluster();  
283 void track_numberOfSitesInLargestCluster();  
284  
285 /* *****  
286 * I/O functions  
287 * Printing Status  
288 * *****  
289 Index lastPlacedSite() const { return _last_placed_site; }  
290  
291 void spanningIndices() const;  
292 void wrappingIndices() const;  
293  
294 /* *****  
295 * Visual data for plotting  
296 * *****  
297 // lattice visual data for python  
298 void writeVisualLatticeData(const std::string& filename, bool  
299 only_spanning=true);  
300  
301 protected:  
302 void initialize();  
303 void initialize_index_sequence();  
304 void randomize_v2(); // better random number generator  
305  
306 int find_cluster_index_for_placing_new_bonds(const std::vector<Index> &  
307 neighbors, std::set<value_type> &found_indices);  
308  
309 value_type manage_clusters(  
310 const std::set<value_type> &found_index_set,  
311 std::vector<BondIndex> &hv_bonds,  
312 Index &site,  
313 int base_id // since id and index is same  
314 );  
315  
316 public:
```

```

316 // on test
317 IndexRelative getRelativeIndex(Index root, Index site_new);
318 };
319
320 /*
321 * Site Percolation Ballistic Deposition
322 * Extended from SitePercolation_ps_v9
323 */
324 class SitePercolationBallisticDeposition_v2 : public
325     SitePercolation_ps_v9{
326 protected:
327 // elements of @indices_tmp will be erased if needed but not of @indices
328 std::vector<value_type> indices;
329 std::vector<value_type> indices_tmp;
330 public:
331 static constexpr const char* signature =
332     "SitePercolation_BallisticDeposition_v2";
333 virtual ~SitePercolationBallisticDeposition_v2() {
334 indices.clear();
335 indices_tmp.clear();
336 };
337 SitePercolationBallisticDeposition_v2(value_type length, bool
338     periodicity);
339
340 virtual bool occupy();
341 /*
342 * Site selection methods
343 */
344 Index select_site(std::vector<Index> &sites, std::vector<BondIndex> &
345     bonds);
346 Index select_site_up_to_1nn(std::vector<Index> &sites, std::vector<
347     BondIndex> &bonds);
348 Index select_site_up_to_2nn(std::vector<Index> &sites, std::vector<
349     BondIndex> &bonds);
350
351 void reset();
352 void initialize_indices();
353
354 virtual std::string getSignature() {
355 std::string s = "sq_lattice_site_percolation_ballistic_deposition_";
356 if(_periodicity)

```

```
352 s += "_periodic_";
353 else
354 s += "_non_periodic_";
355 return s;
356 }
357
358
359 /* *****
360 * occupy upto 1st nearest neighbor.
361 * If the randomly selected site is occupied then select one of the
362 * nearest neighbor randomly
363 * If it is also occupied skip the rest steps and start next iteration
364 * Else occupy it
365 */
366 value_type placeSite_1nn_v2();
367 /* *****
368 * occupy upto 2nd nearest neighbor.
369 * If the randomly selected site is occupied then select one of the
370 * nearest neighbor randomly
371 * If it is also occupied, select the next neighbor in the direction of
372 * motion Else occupy it.
373 * If the 2nd nearest neighbor in the direction of motion is also
374 * occupied then skip the rest of the steps
375 * and start the next iteration
376 */
377 value_type placeSite_2nn_v1();
378 };
379
380 /* *****
381 * Only L1
382 */
383 class SitePercolationBallisticDeposition_L1_v2: public
384     SitePercolationBallisticDeposition_v2{
385 public:
386     ~SitePercolationBallisticDeposition_L1_v2() = default;
387     SitePercolationBallisticDeposition_L1_v2(value_type length, bool
388         periodicity)
389     : SitePercolationBallisticDeposition_v2(length, periodicity){}
390
391     bool occupy() {
392         // if no site is available then return false
393         if(_number_of_occupied_sites == maxSites()){
394             return false;
```

```

389 }
390 try {
391     value_type v = placeSite_1nn_v2();
392     _occupation_probability = occupationProbability(); // for super class
393     return v != ULONG_MAX;
394 } catch (OccupiedNeighbor& on) {
395     //          on.what();
396     return false;
397 }
398
399 }
400
401 std::string getSignature() {
402     std::string s = "sq_lattice_site_percolation_ballistic_deposition_L1";
403     if (_periodicity)
404         s += "_periodic_";
405     else
406         s += "_non_periodic_";
407     return s;
408 }
409
410 };
411
412 /* *****
413 *
414 */
415 class SitePercolationBallisticDeposition_L2_v2: public
416     SitePercolationBallisticDeposition_v2{
417 public:
418     ~SitePercolationBallisticDeposition_L2_v2() = default;
419     SitePercolationBallisticDeposition_L2_v2(value_type length, bool
420         periodicity)
421     : SitePercolationBallisticDeposition_v2(length, periodicity){}
422
423     bool occupy() {
424         // if no site is available then return false
425         if (_number_of_occupied_sites == maxSites())
426             return false;
427     }
428
429     try {
430         //          value_type v = placeSite_2nn_v0();

```

```

431 value_type v = placeSite_2nn_v1();
432 _occupation_probability = occupationProbability(); // for super class
433
434 return v != ULLONG_MAX;
435 } catch (OccupiedNeighbor& on){
436     on.what();
437     cout << "line : " << __LINE__ << endl;
438     return false;
439 }
440
441 }
442
443 std::string getSignature() {
444 std::string s = "sq_lattice_site_percolation_ballistic_deposition_L2";
445 if(_periodicity)
446 s += "_periodic_";
447 else
448 s += "_non_periodic_";
449 return s;
450 }
451
452 };
453
454 #endif // SITEPERCOLATION_PERCOLATION_H

```

### The src/percolation/percolation.cpp file

```

1 #include "percolation.h"
2
3 using namespace std;
4
5 /**
6 *
7 * @param length
8 */
9 SqLatticePercolation::SqLatticePercolation(value_type length) {
10 if (length <= 2) {
11 /**
12 * Because if _length=2
13 * there are total of 4 distinct bond. But it should have been 8, i.e., (2
14 * _length * _length = 8)
15 */
16 cerr << "_length <= 2 does not satisfy _lattice properties for "
17     "percolation : line" << __LINE__ << endl;
18 exit(1);

```

```

17 }
18 _length = length;
19 value_type _length_squared = length * length;
20 _max_number_of_bonds = 2*_length_squared;
21 _max_number_of_sites = _length_squared;
22 _clusters = vector<Cluster>();
23
24 //     size_t seed = 0;
25 //     cerr << "automatic seeding is commented : line " << __LINE__ <<
26 //         endl;
26 auto seed = _random_device();
27 _random_generator.seed(seed); // seeding
28 cout << "seeding with " << seed << endl;
29 }
30
31
32 /**
33 *
34 */
35 void SqLatticePercolation::viewCluster() {
36 cout << "clusters with numberOfBonds greater than 1" << endl;
37 value_type total_bonds{}, total_sites{};
38
39 for (value_type i{}; i != _clusters.size(); ++i) {
40 if(_clusters[i].empty()){
41 //             cout << "Empty cluster : line " << endl;
42 continue;
43 }
44 cout << "cluster [" << i << "] : " << '{' << endl;
45 cout << _clusters[i];
46 total_bonds += _clusters[i].numberOfBonds();
47 total_sites += _clusters[i].numberOfSites();
48
49 cout << '}' << endl;
50 }
51 cout << "Total bonds " << total_bonds << endl;
52 cout << "Total sites " << total_sites << endl;
53 }
54
55
56
57 /**
58 * Extended version of view_cluster
59 */

```

```
60 void SqLatticePercolation::viewClusterExtended() {
61 cout << "clusters with numberOfBonds greater than 1" << endl;
62 value_type total_bonds {}, total_sites {};
63
64 std::vector<Index> sites;
65 std::vector<BondIndex> bonds;
66 for (value_type i{}; i != _clusters.size(); ++i) {
67 if (_clusters[i].empty()) {
68 // cout << "Empty cluster : line " << endl;
69 continue;
70 }
71 cout << "cluster [" << i << "] : ID (" << _clusters[i].get_ID() << ")";
72 // printing sites
73 sites = _clusters[i].getSiteIndices();
74 cout << "Sites : size (" << sites.size() << ") : ";
75 cout << '{';
76 for (auto a: sites) {
77 cout << a << ',';
78 }
79 cout << '}' << endl;
80
81 bonds = _clusters[i].getBondIndices();
82 cout << "Bonds : size (" << bonds.size() << ") : ";
83 cout << '{';
84 for (auto a: bonds) {
85 if (a.horizontal()) {
86 // horizontal bond
87 cout << _lattice.getBond({BondType::Horizontal, a.row_, a.column_}) << ',';
88 } else if (a.vertical()) {
89 // vertical bond
90 cout << _lattice.getBond({BondType::Vertical, a.row_, a.column_}) << ',';
91 } else {
92 cout << '!' << a << '!' << ','; // bond is not valid
93 }
94 }
95 cout << '}';
96
97 cout << endl;
98
99 total_bonds += _clusters[i].numberOfBonds();
100 total_sites += _clusters[i].numberOfSites();
```

```
101 cout << '}' << endl;
102 }
103 cout << "Total bonds " << total_bonds << endl;
104 cout << "Total sites " << total_sites << endl;
105 }
106
107 /**
108 *
109 * @param site
110 * @param bond
111 * @param total_site
112 * @param total_bond
113 */
114
115 void
116 SqLatticePercolation::get_cluster_info(
117     vector<value_type> &site,
118     vector<value_type> &bond
119 ) {
120     value_type total_site {}, total_bond {};
121     site.clear();
122     bond.clear();
123
124     unsigned long size = _clusters.size();
125     site.reserve(size);
126     bond.reserve(size);
127
128     value_type s, b;
129
130     for(value_type i{}; i < size; ++i){
131         if(_clusters[i].empty()){
132             // cout << "Empty cluster : line " << endl;
133             continue;
134         }
135         s = _clusters[i].numberOfSites();
136         b = _clusters[i].numberOfBonds();
137         site.push_back(s);
138         bond.push_back(b);
139         total_site += s;
140         total_bond += b;
141     }
142     if(site.size() != bond.size()){
143         cout << "Size mismatched : line " << __LINE__ << endl;
144 }
```

```

145 //     cout << "total bonds " << total_bond << endl;
146 //     cout << "total sites " << total_site << endl;
147 if(type == 's'){
148 for(value_type j{total_bond}; j < maxBonds(); ++j){
149 bond.push_back(1); // cluster of length 1
150 total_bond += 1;
151 }
152 }
153 if(type == 'b'){
154 for(value_type j{total_site}; j < maxSites(); ++j){
155 total_site += 1;
156 site.push_back(1); // cluster of length 1
157 }
158 }
159 }

160

161 void SqLatticePercolation::reset() {
162 _lattice.reset();
163 _clusters.clear();
164 _index_sequence_position = 0;
165
166 _occupation_probability = 0;
167 // entropy
168 _entropy=0;
169 _entropy_current=0;
170 _total_relabeling = 0;
171 time_relabel = 0;
172 _cluster_count = 0;
173 _reached_critical = false;
174 }

175

176
177 /**
178 * Entropy calculation is performed here. The fastest method possible.
179 * Cluster size is measured by site.
180 * @return current entropy of the lattice
181 */
182 double SqLatticePercolation::entropy_by_site() {
183 double H{}, mu ;
184
185 for(size_t i{}; i < _clusters.size(); ++i){
186 if(!_clusters[i].empty()){
187 mu = _clusters[i].numberOfSites() / double(_number_of_occupied_sites);
188 H += mu*log(mu);

```

```

189 }
190 }
191
192 return -H;
193 }
194
195 /**
196 * Entropy calculation is performed here. The fastest method possible.
197 * Cluster size is measured by site.
198 * @return current entropy of the lattice
199 */
200 double SqLatticePercolation::entropy_by_bond() {
201     double H{}, mu ;
202
203     for(size_t i{}; i < _clusters.size(); ++i){
204         if(!_clusters[i].empty()){
205             mu = _clusters[i].numberOfBonds() / double(maxBonds());
206             H += mu*log(mu);
207         }
208     }
209
210     double number_of_cluster_with_size_one = maxBonds() -
211         _bonds_in_cluster_with_size_two_or_more;
212     // cout << " _bonds_in_cluster_with_size_two_or_more " <<
213     // _bonds_in_cluster_with_size_two_or_more << " : line " << __LINE__ <<
214     endl;
215     mu = 1.0/double(maxBonds());
216     H += number_of_cluster_with_size_one * log(mu) * mu;
217
218     return -H;
219 }
```

The **src/percolation/percolation\_site\_v9.cpp** file

```

1 #include <cstdlib>
2 #include <climits>
3 #include <unordered_set>
4 #include <mutex>
5
6 #include "percolation.h"
7
8 #include "../util/printer.h"
9 #include <omp.h>
10 #include <thread>
11 #include <algorithm>
```

```
12
13 #include " ../ util/time_tracking.h"
14
15 using namespace std;
16
17
18
19 /**
20 *
21 * @param length      : length of the lattice
22 * @param impure_sites : number of impure sites. cannot be greater than
23 *                      length*length
24 */
25 SitePercolation_ps_v9::SitePercolation_ps_v9(value_type length, bool
26       periodicity)
27 : SqLatticePercolation(length)
28 {
29     std::cout << "Constructing SitePercolation_ps_v9 object : line " <<
30     __LINE__ << endl;
31     SqLatticePercolation::set_type('s');
32
33     _periodicity = periodicity;
34     _index_sequence_position = 0;
35     _lattice = SqLattice(length, true, false, false, true); // since it is
36                 // a site percolation all bonds will be activated by default
37
38     min_index = 0;
39     max_index = length - 1;
40
41     index_sequence.resize(maxSites());
42     randomized_index.resize(maxSites());
43     _max_iteration_limit = maxSites();
44
45
46
47 /**
48 *
49 */
50 void SitePercolation_ps_v9::initialize() {
```

```

52 // to improve performance
53 number_of_sites_to_span.reserve(maxSites());
54 number_of_bonds_to_span.reserve(maxSites());
55
56 _top_edge.reserve(length());
57 _bottom_edge.reserve(length());
58 _left_edge.reserve(length());
59 _right_edge.reserve(length());
60
61 // randomized_index_sequence = index_sequence;
62 }
63
64
65 /**
66 * Called only once when the object is constructed for the first time
67 */
68 void SitePercolation_ps_v9::initialize_index_sequence() {
69 value_type m{}, n{};
70 for (value_type i{}; i != index_sequence.size(); ++i) {
71 randomized_index[i] = i;
72 index_sequence[i] = Index(m, n);
73 ++n;
74 if (n == length()) {
75 n = 0;
76 ++m;
77 }
78 }
79 // for (value_type i{}; i != index_sequence.size(); ++i) {cout <<
80 //     index_sequence[i] << endl;}
81 }
82
83 /**
84 * Reset all calculated values and then call initiate()
85 * to initialize for reuse
86 *
87 * caution -> it does not erase _calculation_flags , for it will be used
88 * for calculation purposes
89 */
90 void SitePercolation_ps_v9::reset() {
91 SqLatticePercolation::reset();
92 // variables
93 _number_of_occupied_sites = 0;
94 _index_sequence_position = 0;

```

```
94 _cluster_id = 0;
95
96 // containers
97 number_of_sites_to_span.clear();
98 number_of_bonds_to_span.clear();
99 _spanning_sites.clear();
100 _wrapping_sites.clear();
101 _bonds_in_cluster_with_size_two_or_more = 0;
102 _index_last_modified_cluster = 0; // id of the last modified cluster
103 _number_of_bonds_in_the_largest_cluster = 0;
104 _number_of_sites_in_the_largest_cluster = 0;
105 // clearing edges
106 _top_edge.clear();
107 _bottom_edge.clear();
108 _left_edge.clear();
109 _right_edge.clear();
110 initialize();
111 randomize_v2();
112 time_relabel = 0;
113 _total_relabeling = 0;
114 }
115
116
117 /**
118 * Randomize the indices
119 */
120 void SitePercolation_ps_v9::randomize_v2(){
121
122 std::shuffle(randomized_index.begin(), randomized_index.end(),
123 _random_generator);
124 // cout << "Index sequence : " << randomized_index_sequence << endl;
125
126
127 /* *****/
128 * Calculation methods
129 *
130 *****/
131
132 /*
133 * Instead of calculating entropy for 1000s of cluster in every iteration
134 * just keep track of entropy change, i.e.,
135 * how much to subtract and how much to add.
136 */
```

```
137 /**
138 * Must be called before merging the clusters
139 * @param found_index_set
140 */
141 void SitePercolation_ps_v9::subtract_entropy_for_bond(const set<
142     value_type> &found_index, int base){
143     double nob, mu_bond, H{};
144     if(base >= 0){
145         nob = _clusters[base].numberOfBonds();
146         mu_bond = nob / maxBonds();
147         H += log(mu_bond) * mu_bond;
148     }
149     for(auto x : found_index){
150         nob = _clusters[x].numberOfBonds();
151         mu_bond = nob / maxBonds();
152         H += log(mu_bond) * mu_bond;
153     }
154     _entropy -= -H;
155 }
156
157
158 /**
159 * Must be called after merging the clusters
160 * Cluster length is measured by bonds
161 * @param index
162 */
163 void SitePercolation_ps_v9::add_entropy_for_bond(value_type index){
164     double nob = _clusters[index].numberOfBonds();
165     double mu_bond = nob / maxBonds();
166     double H = log(mu_bond) * mu_bond;
167     _entropy += -H;
168 }
169
170
171
172 /**
173 * Condition: must be called each time a site is placed
174 */
175 void SitePercolation_ps_v9::track_numberOfBondsInLargestCluster() {
176
177 // calculating number of bonds in the largest cluster // by cluster
178 // index
179 // checking number of bonds
```

```
179 if(_clusters[_index_last_modified_cluster].numberOfBonds() >
    _number_of_bonds_in_the_largest_cluster){
180     _number_of_bonds_in_the_largest_cluster = _clusters[
        _index_last_modified_cluster].numberOfBonds();
181 }
182 }
183 }
184
185 /**
186 *
187 */
188 void SitePercolation_ps_v9::track_numberOfSitesInLargestCluster(){
189
190 // calculating number of bonds in the largest cluster // by cluster
191 // index
192 // checking number of bonds
193 if(_clusters[_index_last_modified_cluster].numberOfSites() >
    _number_of_sites_in_the_largest_cluster){
194     _number_of_sites_in_the_largest_cluster = _clusters[
        _index_last_modified_cluster].numberOfSites();
195 }
196
197 /**
198 *
199 * @param neighbors          :
200 * @param found_index_set    : index of the clusters that will be merged
201 *                             together.
202 *                                         Does not contain the base cluster index or
203 *                                         id.
204 * @return                   : id of the base cluster
205 */
206 int
207 SitePercolation_ps_v9::find_cluster_index_for_placing_new_bonds(
208     const vector<Index> &neighbors, std::set<value_type> &found_index_set
209 ){
210     found_index_set.clear();
211     value_type size{}, tmp{}, index, base{ULONG_MAX};
212     int base_id{-1};
213     int id;
214     for (auto n: neighbors) {
215         id = _lattice.getGroupID(n);
216         if(id >=0) {
```

```

216 index = value_type(id);
217 tmp = _clusters[index].numberOfSites();
218 if(tmp > size){
219 size = tmp;
220 base_id = id;
221 base = index;
222 }
223
224 found_index_set.insert(index);
225
226 }
227 }
228 found_index_set.erase(base);
229 return base_id;
230 }
231
232
233 /**
234 * Last placed site is added to a cluster. If this connects other
235 * clusters then merge all
236 * cluster together to get one big cluster. All sites that are part of
237 * the other clusters
238 * are relabeled according to the id of the base cluster.
239 * @param found_index_set : index of the clusters that are neighbors of
240 * the last placed site
241 * @param hv_bonds : bonds that connects the last placed site and
242 * its neighbors
243 * and which are not part of any cluster of size
244 * larger than one
245 * @param site : last placed site
246 * @param base_id : id of the base cluster
247 * @return
248 */
249 value_type SitePercolation_ps_v9::manage_clusters(
250 const set<value_type> &found_index_set,
251 vector<BondIndex> &hv_bonds,
252 Index &site,
253 int base_id
254 )
255 {
256
257 if (base_id != -1) {
258 value_type base = value_type(base_id); // converting here

```

```
255 _clusters[base].addSiteIndex(site);
256 int id_base = _clusters[base].get_ID();
257 vector<Index> neibhgors = _lattice.get_neighbor_site_indices(site);
258 // find which of the neighbors are of id_base as the base cluster
259 IndexRelative r;
260 for(auto n: neibhgors){
261 if(_lattice.getGroupID(n) == id_base){
262 // find relative index with respect to this site
263 r = getRelativeIndex(n, site);
264 break; // since first time r is set running loop is doing no good
265 }
266 }
267
268 // put_values_to_the_cluster new values in the 0-th found index
269 _clusters[base].insert(hv_bonds);
270 _lattice.getSite(site).relativeIndex(r);
271 _lattice.setGroupID(site, id_base); // relabeling for 1 site
272
273 // merge clusters with common values from all other cluster      //
274 // merge clusters with common values from all other cluster
275
276 for(value_type ers: found_index_set){
277
278 _total_relabeling += _clusters[ers].numberOfSites(); // only for
279 // debugging purposes
280 // perform relabeling on the sites
281 relabel_sites_v5(site, _clusters[ers]);
282
283 // store values of other found indices to the cluster
284 _clusters[base].insert_v2(_clusters[ers]);
285 _cluster_count--; // reducing number of clusters
286 _clusters[ers].clear(); // emptying the cluster
287 }
288 _index_last_modified_cluster = base;
289
290
291 } else {
292 // create new element for the cluster
293 _clusters.push_back(Cluster(_cluster_id));
294 value_type _this_cluster_index = _clusters.size() -1;
295 _lattice.setGroupID(site, _cluster_id); // relabeling for 1 site
296 _cluster_count++; // increasing number of clusters
```

```
297 _cluster_id++;
298 _clusters.back().insert(hv_bonds);
299 _clusters[_this_cluster_index].addSiteIndex(site);
300 _index_last_modified_cluster = _this_cluster_index; // last cluster is
   the place where new bonds are placed
301 }
302 }
303 return _index_last_modified_cluster;
304 }
305
306
307
308
309
310 /**
311 * Relative index of site_new with respect to root
312 * @param root
313 * @param site_new
314 * @return
315 */
316 IndexRelative SitePercolation_ps_v9::getRelativeIndex(Index root, Index
   site_new){
317 //   cout << "Entry \"SitePercolation_ps_v9::getRelativeIndex\" : line
   " << __LINE__ << endl;
318 int delta_x = -int(root.column_) + int(site_new.column_); // if +1 then
   root is on the right ??
319 int delta_y = int(root.row_) - int(site_new.row_); // if +1 then root is
   on the top ??
320
321
322 // normalizing delta_x
323 if(delta_x > 1){
324 delta_x /= -delta_x;
325 }
326 else if(delta_x < -1){
327 delta_x /= delta_x;
328 }
329
330 // normalizing delta_y
331 if(delta_y > 1){
332 delta_y /= -delta_y;
333 } else if(delta_y < -1){
334 delta_y /= delta_y;
335 }
```

```
336
337 IndexRelative indexRelative_root = _lattice.getSite(root).relativeIndex()
338 //    cout << "Relative index of root " << indexRelative_root << endl;
339 //    cout << "Delta x,y " << delta_x << ", " << delta_y << endl;
340 IndexRelative r = {indexRelative_root.x_ + delta_x, indexRelative_root.
341     y_ + delta_y};
342 //    cout << "Relative index of site_new " << r << endl;
343 return r;
344
345
346
347 /**
348 * Take a bond index only if the corresponding site is active
349 * takes longer? time than version 1?, i.e., connection()
350 * @param site
351 * @param site_neighbor
352 * @param bond_neighbor
353 */
354 void SitePercolation_ps_v9::connection_v2(Index site, vector<Index> &
355     site_neighbor, vector<BondIndex> &bond_neighbor)
356 {
357     value_type prev_column = (site.column_ + length() - 1) % length();
358     value_type prev_row = (site.row_ + length() - 1) % length();
359     value_type next_row = (site.row_ + 1) % length();
360     value_type next_column = (site.column_ + 1) % length();
361
362     if(!_periodicity){
363         // without periodicity
364         if (site.row_ == min_index) { // top edge including corners
365             if(site.column_ == min_index){
366                 // upper left corner
367
368                 site_neighbor.resize(2);
369                 site_neighbor[0] = {site.row_, next_column};
370                 site_neighbor[1] = {next_row, site.column_};
371
372                 bond_neighbor.reserve(2);
373                 if(!_lattice.getSite(site_neighbor[0]).isActive()){
374                     bond_neighbor.push_back({BondType::Horizontal, site.row_, site.column_})
375                     ;
376                 }
377             }
378         }
379     }
380 }
```

```
376 if (! _lattice .getSite (site_neighbo r [1]) .isA ctive ()) {
377 bond_neighbo r.push_back ({BondType :: Vertical , site .row_ , site .column_ });
378 }
379
380 return ;
381 }
382 }
383 else if (site .column_ == max_index) {
384 // upper right corner
385
386 site_neighbo r.resize (2);
387 site_neighbo r[0] = {site .row_ , prev_column };
388 site_neighbo r[1] = {next_row , site .column_ };
389
390 bond_neighbo r.reserve (2);
391 if (! _lattice .getSite (site_neighbo r [0]) .isA ctive ()) {
392 bond_neighbo r.push_back ({BondType :: Horizontal , site .row_ , prev_column });
393 }
394 if (! _lattice .getSite (site_neighbo r [1]) .isA ctive ()) {
395 bond_neighbo r.push_back ({BondType :: Vertical , site .row_ , site .column_ });
396 }
397
398 return ;
399 }
400 else {
401 // top edge excluding corners
402 site_neighbo r.resize (3);
403 site_neighbo r[0] = {site .row_ , next_column };
404 site_neighbo r[1] = {site .row_ , prev_column };
405 site_neighbo r[2] = {next_row , site .column_ };
406
407 bond_neighbo r.reserve (4);
408 if (! _lattice .getSite (site_neighbo r [0]) .isA ctive ()) {
409 bond_neighbo r.push_back ({BondType :: Horizontal , site .row_ , site .column_ })
410 ;
411 }
412 if (! _lattice .getSite (site_neighbo r [1]) .isA ctive ()) {
413 bond_neighbo r.push_back ({BondType :: Horizontal , site .row_ , prev_column });
414 }
415 if (! _lattice .getSite (site_neighbo r [2]) .isA ctive ()) {
416 bond_neighbo r.push_back ({BondType :: Vertical , site .row_ , site .column_
417 });
418 }
```

```
418 return ;
419 }
420 }
421 }
422 else if (site.row_ == max_index) { // bottom edge including corners
423 if (site.column_ == min_index) {
424 // lower left corner
425 site_neighbor.resize(2);
426 site_neighbor[0] = {site.row_, next_column};
427 site_neighbor[1] = {prev_row, site.column_};
428
429 bond_neighbor.reserve(2);
430 if(!_lattice.getSite(site_neighbor[0]).isActive()){
431 bond_neighbor.push_back({BondType::Horizontal, site.row_, site.column_})
432 ;
433 }
434 if(!_lattice.getSite(site_neighbor[1]).isActive()){
435 bond_neighbor.push_back({BondType::Vertical, prev_row, site.column_});
436 }
437
438 return ;
439
440 } else if (site.column_ == max_index) {
441 // lower right corner
442 site_neighbor.resize(2);
443 site_neighbor[0] = {site.row_, prev_column};
444 site_neighbor[1] = {prev_row, site.column_};
445
446 bond_neighbor.reserve(2);
447 if(!_lattice.getSite(site_neighbor[0]).isActive()){
448 bond_neighbor.push_back({BondType::Horizontal, site.row_, prev_column});
449 }
450 if(!_lattice.getSite(site_neighbor[1]).isActive()){
451 bond_neighbor.push_back({BondType::Vertical, prev_row, site.column_});
452 }
453
454 return ;
455
456 } else {
457 // bottom edge excluding corners
458 // bottom edge
459 site_neighbor.resize(3);
460 site_neighbor[0] = {site.row_, next_column};
```

```

461 site_neighbor[1] = {site.row_, prev_column};
462 site_neighbor[2] = {prev_row, site.column_};
463
464 bond_neighbor.reserve(3);
465 if(!_lattice.getSite(site_neighbor[0]).isActive()) {
466 bond_neighbor.push_back({BondType::Horizontal, site.row_, site.column_})
467 ;
468 }
469 if(!_lattice.getSite(site_neighbor[1]).isActive()){
470 bond_neighbor.push_back({BondType::Horizontal, site.row_, prev_column});
471 }
472 if(!_lattice.getSite(site_neighbor[2]).isActive()){
473 bond_neighbor.push_back({BondType::Vertical, prev_row, site.column_});
474 }
475
476 }
477 }
478 /* site.x_ > min_index && site.x_ < max_index && is not possible
   anymore */
479 else if (site.column_ == min_index) { // left edge not in the corners
480 site_neighbor.resize(3);
481 site_neighbor[0] = {site.row_, next_column};
482 site_neighbor[1] = {next_row, site.column_};
483 site_neighbor[2] = {prev_row, site.column_};
484
485 bond_neighbor.reserve(3);
486 if(!_lattice.getSite(site_neighbor[0]).isActive()) {
487 bond_neighbor.push_back({BondType::Horizontal, site.row_, site.column_})
488 ;
489 }
490 if(!_lattice.getSite(site_neighbor[1]).isActive()){
491 bond_neighbor.push_back({BondType::Vertical, site.row_, site.column_});
492 }
493 if(!_lattice.getSite(site_neighbor[2]).isActive()){
494 bond_neighbor.push_back({BondType::Vertical, prev_row, site.column_});
495 }
496
497 }
498 else if (site.column_ == max_index) {
499 // right edge no corners
500

```

```
501 site_neighbor.resize(3);
502 site_neighbor[0] = {site.row_, prev_column};
503 site_neighbor[1] = {next_row, site.column_};
504 site_neighbor[2] = {prev_row, site.column_};
505
506 bond_neighbor.reserve(3);
507 if(!_lattice.getSite(site_neighbor[0]).isActive()){
508 bond_neighbor.push_back({BondType::Horizontal, site.row_, prev_column});
509 }
510 if(!_lattice.getSite(site_neighbor[1]).isActive()){
511 bond_neighbor.push_back({BondType::Vertical, site.row_, site.column_});
512 }
513 if(!_lattice.getSite(site_neighbor[2]).isActive()){
514 bond_neighbor.push_back({BondType::Vertical, prev_row, site.column_});
515 }
516
517 return;
518 }
519 }
520 }
521 // 1 level inside the lattice
522 // not in any the boundary
523 site_neighbor.resize(4);
524 site_neighbor[0] = {site.row_, next_column};
525 site_neighbor[1] = {site.row_, prev_column};
526 site_neighbor[2] = {next_row, site.column_};
527 site_neighbor[3] = {prev_row, site.column_};
528
529 bond_neighbor.reserve(4);
530 if(!_lattice.getSite(site_neighbor[0]).isActive()) {
531 bond_neighbor.push_back({BondType::Horizontal, site.row_, site.column_})
532 ;
533 }
534 if(!_lattice.getSite(site_neighbor[1]).isActive()){
535 bond_neighbor.push_back({BondType::Horizontal, site.row_, prev_column});
536 }
537 if(!_lattice.getSite(site_neighbor[2]).isActive()){
538 bond_neighbor.push_back({BondType::Vertical, site.row_, site.column_});
539 }
540 if(!_lattice.getSite(site_neighbor[3]).isActive()) {
541 bond_neighbor.push_back({BondType::Vertical, prev_row, site.column_});
542 }
```

```
542 }
543
544
545
546
547 /**
548 *
549 * @param site
550 * @param edge
551 * @return
552 */
553 bool SitePercolation_ps_v9 :: check_if_id_matches(Index site, const vector
554 <Index> &edge){
555     for(auto s : edge){
556         if(_lattice.getGroupID(site) == _lattice.getGroupID(s)){
557             // no need to put the site here
558             return true;
559         }
560     }
561     return false;
562 }
563
564
565 /**
566 * Placing sites
567 *
568 ****
569
570 /**
571 * All site placing method in one place
572 *
573 * @return true if operation is successfull
574 */
575 bool SitePercolation_ps_v9 :: occupy() {
576     if(_index_sequence_position >= maxSites()){
577         return false;
578     }
579     Index site = selectSite();
580     placeSite_weighted(site);
581     _occupation_probability = occupationProbability(); // for super class
582     return true;
583 }
584
```

```
585 /**
586 * Index of the selected site must be provided with the argument
587 *
588 * Wrapping and spanning index arrangement is enabled.
589 * Entropy is calculated smoothly.
590 * Entropy is measured by site and bond both.
591 * @param current_site
592 * @return
593 */
594 value_type SitePercolation_ps_v9::placeSite_weighted(Index current_site)
595 {
596 // randomly choose a site
597 if (_number_of_occupied_sites == maxSites()) {
598 return ULONG_MAX; // unsigned long int maximum value
599 }
600
601 _last_placed_site = current_site;
602 _lattice.activate_site(current_site);
603 ++_number_of_occupied_sites;
604 // find the bonds for this site
605 vector<BondIndex> bonds;
606 vector<Index> sites;
607 connection_v2(current_site, sites, bonds);
608 _bonds_in_cluster_with_size_two_or_more += bonds.size();
609
610 // find one of hv_bonds in _clusters and add ever other value to that
611 // place. then erase other position
612 set<value_type> found_index_set;
613 int base_id = find_cluster_index_for_placing_new_bonds(sites,
614 found_index_set);
615
616 subtract_entropy_for_bond(found_index_set, base_id); // tracking
617 // entropy change
618 value_type merged_cluster_index = manage_clusters(
619 found_index_set, bonds, current_site, base_id
620 );
621 add_entropy_for_bond(merged_cluster_index); // tracking entropy change
622 // running tracker
623 track_numberOfBondsInLargestCluster(); // tracking number of bonds in
624 // the largest cluster
625 track_numberOfSitesInLargestCluster();
626 return merged_cluster_index;
627 }
```

```

624 /**
625 * Index of the selected site must be provided with the argument
626 *
627 * Wrapping and spanning index arrangement is enabled.
628 * Entropy is calculated smoothly.
629 * Entropy is measured by site and bond both.
630 * @param current_site
631 * @return
632 */
633 value_type SitePercolation_ps_v9::placeSite_weighted(
634 Index current_site ,
635 vector<Index>& neighbor_sites ,
636 vector<BondIndex>& neighbor_bonds
637 ) {
638 // randomly choose a site
639 if (_number_of_occupied_sites == maxSites()) {
640 return ULONG_MAX; // unsigned long int maximum value
641 }
642 _bonds_in_cluster_with_size_two_or_more += neighbor_bonds.size();
643 _last_placed_site = current_site;
644 _lattice.activate_site(current_site);
645 ++_number_of_occupied_sites;
646 // find one of hv_bonds in _clusters and add ever other value to that
647 // place. then erase other position
648 set<value_type> found_index_set;
649 int base_id = find_cluster_index_for_placing_new_bonds(neighbor_sites ,
650 found_index_set);
651 subtract_entropy_for_bond(found_index_set , base_id); // tracking
652 // entropy change
653 value_type merged_cluster_index = manage_clusters(
654 found_index_set , neighbor_bonds , current_site , base_id
655 );
656 add_entropy_for_bond(merged_cluster_index); // tracking entropy change
657 // running tracker
658 track_numberOfBondsInLargestCluster(); // tracking number of bonds in
659 // the largest cluster
660 track_numberOfSitesInLargestCluster();
661 return merged_cluster_index;
662 }
663 /**
664 *

```

```
664 * @return
665 */
666 Index SitePercolation_ps_v9::selectSite(){
667     Index current_site = randomized_index_sequence[
668         _index_sequence_position]; // old
669     value_type index = randomized_index[_index_sequence_position];
670     Index current_site = index_sequence[index]; // new process
671     ++_index_sequence_position;
672     return current_site;
673 }
674
675 /* *****
676 * View methods
677 *****/
678
679
680 /**
681 *
682 */
683 void SitePercolation_ps_v9::spanningIndices() const {
684     cout << "Spanning Index : id" << endl;
685     for(Index i: _spanning_sites){
686         cout << "Index " << i << " : id " << _lattice.getGroupID(i) << endl;
687     }
688 }
689
690 void SitePercolation_ps_v9::wrappingIndices() const {
691     cout << "Wrapping Index : id : relative index" << endl;
692     for(auto i: _wrapping_sites){
693         cout << "Index " << i << " : id "
694             << _lattice.getGroupID(i)
695             << " relative index : " << _lattice.getSite(i).relativeIndex() << endl;
696     }
697 }
698
699
700 /* *****
701 * Spanning Detection
702 *****/
703
704
705 /**
706 * success : gives correct result
```

```

707 * length          time
708 * 200            7.859000 sec
709 * 500            2 min 18.874000 sec
710 * only check for the cluster id of the recently placed site
711 * @param site : Check spanning for this argument
712 * @return
713 */
714 bool SitePercolation_ps_v9::detectSpanning_v6(const Index& site) {
715 //    cout << "Entry -> detectSpanning_v4() : line " << __LINE__ << endl
716 //    ;
717 if(_periodicity) {
718 cout << "Cannot detect spanning if _periodicity if ON: line " <<
719     __LINE__ << endl;
720 return false;
721 }
722 if(_reached_critical ){
723 return true; // we have already reached critical point
724 }
725 // first check if the site with a cluster id is already a spanning site
726 for(const Index& ss: _spanning_sites){
727 if(_lattice.getSite(ss).get_groupID() == _lattice.getSite(site).
728     get_groupID()){
729 //            cout << "Already a spanning site : line " << __LINE__ <<
730 //            endl;
731 return true;
732 }
733 // only check for the newest site placed
734 if(site.row_ == min_index){ // top index
735 if(!check_if_id_matches(site, _top_edge)) {
736 _top_edge.push_back(site);
737 }
738 else if(site.row_ == max_index){
739 if(!check_if_id_matches(site, _bottom_edge)){
740 _bottom_edge.push_back(site);
741 }
742 }
743 // checking column indices for Left-Right boundary
744 if(site.column_ == min_index){ // left edge
745 if(!check_if_id_matches(site, _left_edge)) {

```

```
747 _left_edge.push_back(site);
748 }
749 }
750 else if(site.column_ == max_index){
751 if(!check_if_id_matches(site, _right_edge)) {
752 _right_edge.push_back(site);
753 }
754 }
755
756 if(_number_of_occupied_sites < length()){
757 // cout << "Not enough site to span : line " << __LINE__ << endl;
758 return false;
759 }
760
761
762 vector<Index>::iterator it_top = _top_edge.begin();
763 vector<Index>::iterator it_bot = _bottom_edge.begin();
764 bool found_spanning_site = false;
765 int id = _lattice.getGroupID(site);
766
767 if(_top_edge.size() < _bottom_edge.size()){
768 // if matched found on the smaller edge look for match in the larger
769 // edge
770 for(; it_top < _top_edge.end(); ++it_top){
771 if(id == _lattice.getGroupID(*it_top)){
772 for(; it_bot < _bottom_edge.end(); ++it_bot){
773 if(id == _lattice.getGroupID(*it_bot)){
774 // match found !
775 if(!check_if_id_matches(*it_top, _spanning_sites)) {
776 _reached_critical = true;
777 _spanning_sites.push_back(*it_top);
778 }
779 found_spanning_site = true;
780 _bottom_edge.erase(it_bot);
781 }
782 }
783 if(found_spanning_site){
784 found_spanning_site = false;
785 _top_edge.erase(it_top);
786 }
787 }
788 }
789 }
```

```

790 } else {
791 for (; it_bot < _bottom_edge.end(); ++it_bot) {
792 if (id == _lattice.getGroupID(*it_bot)) {
793 for (; it_top < _top_edge.end(); ++it_top) {
794 if (id == _lattice.getGroupID(*it_top)) {
795 // match found !
796 if (!check_if_id_matches(*it_top, _spanning_sites)) {
797 _reached_critical = true;
798 _spanning_sites.push_back(*it_top);
799 }
800 found_spanning_site = true;
801 _top_edge.erase(it_top);
802 }
803 }
804 if (found_spanning_site){
805 found_spanning_site = false;
806 _bottom_edge.erase(it_top);
807 }
808 }
809 }
810 }
811 }
812
813 found_spanning_site = false;
814 vector<Index>::iterator it_lft = _left_edge.begin();
815 vector<Index>::iterator it_rht = _right_edge.begin();
816
817 if (_left_edge.size() < _right_edge.size()){
818 for (; it_lft < _left_edge.end(); ++it_lft) {
819 if (id == _lattice.getGroupID(*it_lft)) {
820 for (; it_rht < _right_edge.end(); ++it_rht) {
821 if (id == _lattice.getGroupID(*it_rht)) {
822 if (!check_if_id_matches(*it_lft, _spanning_sites)) {
823 _spanning_sites.push_back(*it_lft);
824 _reached_critical = true;
825 }
826 found_spanning_site = true;
827 _right_edge.erase(it_rht);
828 }
829 }
830 if (found_spanning_site) {
831 found_spanning_site = false;
832 _left_edge.erase(it_lft);
833 }

```

```
834 }
835 }
836 } else{
837     for (; it_rht < _right_edge.end(); ++it_rht) {
838         if (id == _lattice.getGroupID(*it_rht)) {
839             for(; it_lft < _left_edge.end(); ++it_lft) {
840                 if (id == _lattice.getGroupID(*it_lft)) {
841                     if (!check_if_id_matches(*it_lft , _spanning_sites)) {
842                         _spanning_sites.push_back(*it_lft);
843                         _reached_critical = true;
844                     }
845                     found_spanning_site = true;
846                     _left_edge.erase(it_lft);
847                 }
848             }
849             if (found_spanning_site) {
850                 found_spanning_site = false;
851                 _right_edge.erase(it_rht);
852             }
853         }
854     }
855 }
856
857
858 // now do the matching with left and right for horizontal spanning
859 // meaning new site is added to _spanning_sites so remove them from top
860 // and bottom edges
861
862
863 // filter spanning ids
864
865
866 return !_spanning_sites.empty();
867 }
868
869
870
871
872 /* **** */
873 * Wrapping Detection
874 **** */
875 /**
876 * Wrapping is detected here using the last placed site
```

```

877 * @return bool. True if wrapping occurred.
878 */
879 bool SitePercolation_ps_v9::detectWrapping() {
880 Index site = lastPlacedSite();
881 // only possible if the cluster containing 'site' has sites >= length of
882 // the lattice
883 if (_number_of_occupied_sites < length()){
884 return false;
885 }
886
887 if(_reached_critical){
888 return true; // reached critical in previous step
889 }
890 // check if it is already a wrapping site
891 int id = _lattice.getGroupID(site);
892 int tmp_id{ };
893 for (auto i: _wrapping_sites){
894 tmp_id = _lattice.getGroupID(i);
895 if(id == tmp_id ){
896 return true;
897 }
898 }
899 // get four neighbors of site always. since wrapping is valid if
900 // periodicity is implied
901 vector<Index> sites = _lattice.get_neighbor_site_indices(site);
902
903 if(sites.size() < 2){ // at least two neighbor of site is required
904 return false;
905 } else{
906 IndexRelative irel = _lattice.getSite(site).relativeIndex();
907 // cout << "pivot's " << site << " relative " << irel << endl;
908 IndexRelative b;
909 for (auto a:sites){
910 if(_lattice.getGroupID(a) != _lattice.getGroupID(site)){
911 // different cluster
912 continue;
913 }
914 // belongs to the same cluster
915 b = _lattice.getSite(a).relativeIndex();
916 // cout << "neibhbor " << a << " relative " << b << endl;
917 if(abs(irel.x_- - b.x_-) > 1 || abs(irel.y_- - b.y_-) > 1){
918 // cout << "Wrapping : line " << __LINE__ << endl;
919 _wrapping_sites.push_back(site);

```

```
919 _reached_critical = true;
920 return true;
921 }
922 }
923 }
924 // if %_wrapping_indices is not empty but wrapping is not detected for
// the current site (%site)
925 // that means there is wrapping but not for the %site
926 return !_wrapping_sites.empty();
927 }
928
929 /* **** */
930 * Relabeling
931 *
932 ****
933 /**
934 * Relabels site and also reassign relative index to the relabeled sites
935 *
936 * @param site_a : last added site index of the base cluster
937 * @param clstr_b : 2nd cluster, which to be merged with the root
938 */
939 void SitePercolation_ps_v9::relabel_sites_v5(Index site_a, const Cluster
& clstr_b) {
940 const vector<Index> sites = clstr_b.getSiteIndices();
941 int id_a = _lattice.getGroupID(site_a);
942 int id_b = clstr_b.get_ID();
943 Index b = clstr_b.getRootSite();
944
945 // get four site_b of site_a
946 vector<Index> sites_neighbor_a = _lattice.get_neighbor_site_indices(
site_a);
947 Index site_b;
948 IndexRelative relative_index_b_after;
949 bool flag{false};
950 // find which site_b has id_a of clstr_b
951 for(auto n: sites_neighbor_a){
952 if(id_b == _lattice.getGroupID(n)){
953 // checking id_a equality is enough. since id_a is the id_a of the
// active site already.
954 relative_index_b_after = getRelativeIndex(site_a, n);
955 site_b = n;
956 flag = true;
957 break;
958 }
```

```

959 }
960 if(!flag){
961 cout << "No neibhgor found! : line " << __LINE__ << endl;
962 }
963
964 IndexRelative relative_site_a = _lattice.getSite(site_a).relativeIndex()
965     ;
966 // with this delta_a and delta_y find the relative index of site_b while
967 // relative index of site_a is known
968 IndexRelative relative_site_b_before = _lattice.getSite(site_b).
969     relativeIndex();
970 int delta_x_ab = relative_index_b_after.x_ - relative_site_b_before.x_;
971 int delta_y_ab = relative_index_b_after.y_ - relative_site_b_before.y_;
972 relabel_sites(sites, id_a, delta_x_ab, delta_y_ab);
973 }

974 void SitePercolation_ps_v9::relabel_sites(const vector<Index> &sites,
975     int id_a, int delta_x_ab, int delta_y_ab) {
976     int x, y;
977     Index a;
978     IndexRelative relative_site_a;
979     for (value_type i = 0; i < sites.size(); ++i) {
980         a = sites[i];
981         _lattice.setGroupID(a, id_a);
982         relative_site_a = _lattice.getSite(a).relativeIndex();
983         x = relative_site_a.x_ + delta_x_ab;
984         y = relative_site_a.y_ + delta_y_ab;
985         _lattice.getSite(a).relativeIndex(x, y);
986     }
987
988
989
990 /* ****
991 * Information about current state of Class
992 **** */
993
994 /**
995 * Entropy calculation is performed here. The fastest method possible.
996 * Cluster size is measured by bond.
997 * @return current entropy of the lattice
998 */

```

```
999 double SitePercolation_ps_v9::entropy() {
1000     double H{ };
1001     double number_of_cluster_with_size_one = maxBonds() -
1002         _bonds_in_cluster_with_size_two_or_more;
1003     //    cout << " _bonds_in_cluster_with_size_two_or_more " <<
1004         _bonds_in_cluster_with_size_two_or_more << " : line " << __LINE__ <<
1005         endl;
1006     double mu = 1.0 / double(maxBonds());
1007     H += number_of_cluster_with_size_one * log(mu) * mu;
1008     H *= -1;
1009     _entropy_current = _entropy + H;
1010     return _entropy_current;
1011 }
1012 /**
1013 * Only applicable if the number of bonds in the largest cluster is
1014 * calculated when occupying the lattice.
1015 * Significantly efficient than the previous version
1016 *      numberOfBondsInTheLargestCluster()
1017 * @return
1018 */
1019 value_type SitePercolation_ps_v9::numberOfBondsInTheLargestCluster_v2()
1020 {
1021     //    return _clusters[_index_largest_cluster].numberOfBonds();
1022     return _number_of_bonds_in_the_largest_cluster;
1023 }
1024 /**
1025 *
1026 * @return
1027 */
1028 value_type SitePercolation_ps_v9::numberOfSitesInTheLargestCluster() {
1029     value_type len{}, nob{};
1030     for (auto c: _clusters){
1031         nob = c.numberOfSites();
1032         if (len < nob){
1033             len = nob;
1034         }
1035     }
1036     _number_of_sites_in_the_largest_cluster = len;
```

```
1037     return len;
1038 }
1039
1040
1041 /* ****
1042 * Spanning methods
1043 **** */
1044
1045 /**
1046 *
1047 * @return
1048 */
1049 value_type SitePercolation_ps_v9::numberOfSitesInTheSpanningClusters_v2
1050     () {
1051
1052     if (! _spanning_sites.empty()) {
1053         int id = _lattice.getGroupID(_spanning_sites.front());
1054         if (id >= 0) {
1055             return _clusters[id].numberOfSites();
1056         }
1057     }
1058
1059
1060
1061 /**
1062 *
1063 * @return
1064 */
1065 value_type SitePercolation_ps_v9::numberOfBondsInTheSpanningClusters_v2
1066     () {
1067     if (!_spanning_sites.empty()) {
1068         // cout << "number of spanning sites " << _spanning_sites.size()
1069         // << " : line " << __LINE__ << endl;
1070         int id = _lattice.getGroupID(_spanning_sites.front());
1071         if (id >= 0) {
1072             return _clusters[id].numberOfBonds();
1073         }
1074     }
1075
1076 /**
1077 *
```

```
1078 * @return
1079 */
1080 value_type SitePercolation_ps_v9::numberOfSitesInTheWrappingClusters() {
1081     value_type nos{ };
1082     int id{ };
1083     for(auto i: _wrapping_sites){
1084         id = _lattice.getGroupID(i);
1085         if(id >= 0) {
1086             nos += _clusters[id].numberOfSites();
1087         }
1088     }
1089     return nos;
1090 }
1091
1092 /**
1093 *
1094 * @return
1095 */
1096 value_type SitePercolation_ps_v9::numberOfBondsInTheWrappingClusters() {
1097     value_type nob{ };
1098     int id{ };
1099     for(auto i: _wrapping_sites){
1100         id = _lattice.getGroupID(i);
1101         if(id >= 0) {
1102             nob += _clusters[id].numberOfBonds();
1103         }
1104     }
1105     return nob;
1106 }
1107
1108
1109 std::string SitePercolation_ps_v9::getSignature() {
1110     string s = "sq_lattice_site_percolation";
1111     if(_periodicity)
1112         s += "_periodic_";
1113     else
1114         s += "_non_periodic_";
1115     return s;
1116 }
1117
1118 /**
1119 *
1120 * @param filename
1121 * @param only_spanning
```

```

1122 /*
1123 void SitePercolation_ps_v9::writeVisualLatticeData(const string &
1124   filename, bool only_spanning) {
1125 std::ofstream fout(filename);
1126 ostringstream header_info;
1127 header_info << "{"
1128 << "\\" length \":\\" << length()
1129 << "\\", \"signature \":\\" " << getSignature() << "\\" "
1130 << "\\", \"x \":\\" " << lastPlacedSite().column_ << "\\" "
1131 << "\\", \"y \":\\" " << lastPlacedSite().row_ << "\\" "
1132 << "}" ;
1133
1134 fout << "#" << header_info.str() << endl;
1135 fout << "#<x>,<y>,<color>" << endl;
1136 fout << "# color=0 -means-> unoccupied site" << endl;
1137 int id{-1};
1138 if(!_spanning_sites.empty()){
1139   id = _lattice.getGroupID(_spanning_sites.front());
1140 } else if(!_wrapping_sites.empty()){
1141   id = _lattice.getGroupID(_wrapping_sites.front());
1142 }
1143
1144 if(only_spanning){
1145   if(id < 0){
1146     cerr << "id < 0 : line " << __LINE__ << endl;
1147   }
1148   vector<Index> sites = _clusters[id].getSiteIndices();
1149   for(auto s: sites){
1150     fout << s.column_ << ',' << s.row_ << ',' << id << endl;
1151   }
1152 }
1153 else {
1154   for (value_type y{}; y != length(); ++y) {
1155     for (value_type x{}; x != length(); ++x) {
1156       id = _lattice.getGroupID({y, x});
1157       if(id != -1) {
1158         fout << x << ',' << y << ',' << id << endl;
1159       }
1160     }
1161   }
1162 }
1163 fout.close();
1164 }
```

**The src/percolation/percolation\_site\_ballistic\_deps\_v2.cpp file**

```
1 #include <cstdlib>
2 #include <climits>
3
4 #include "percolation.h"
5
6 using namespace std;
7
8 /**
9 *
10 * @param length
11 */
12 SitePercolationBallisticDeposition_v2::
13     SitePercolationBallisticDeposition_v2(value_type length, bool
14     periodicity)
15 : SitePercolation_ps_v9(length, periodicity)
16 {
17
18     std::cout << "Constructing SitePercolationBallisticDeposition_v2 object
19         : line " << __LINE__ << endl;
20
21     initialize_indices();
22     indices_tmp = indices;
23     //    randomize_index();
24 }
25
26 /**
27 *
28 */
29 void SitePercolationBallisticDeposition_v2::reset() {
30     SitePercolation_ps_v9::reset();
31     indices_tmp = indices;
32 }
33
34 /**
35 * Called only once when the object is constructed for the first time
36 */
37 void SitePercolationBallisticDeposition_v2::initialize_indices() {
38     indices = vector<value_type>(maxSites());
39     for(value_type i{}; i != indices.size(); ++i){
40         indices[i] = i; // assign index first
41     }
42 }
```

```
40
41
42
43 /* *****
44 * Site selection methods
45 */
46
47 /**
48 *
49 * @param sites
50 * @param bonds
51 * @return
52 */
53 Index SitePercolationBallisticDeposition_v2::select_site(vector<Index> &
54   sites, vector<BondIndex> &bonds) {
55 // randomly choose a site
56 value_type r = std::rand() % (indices_tmp.size());
57
58 Index current_site = index_sequence[indices_tmp[r]];
59 cout << "current site " << current_site << endl;
60 // find the bonds for this site
61
62 if (_lattice.getSite(current_site).isActive()){
63 indices_tmp.erase(indices_tmp.begin()+r);
64
65 throw OccupiedNeighbor{"all of the 1nd neighbors are occupied : line " +
66   std::to_string(__LINE__)};
67 }
68
69 cout << "choosing " << current_site << " out of the neighbors : line "
70   << __LINE__ << endl;
71 sites.clear();
72 bonds.clear();
73 connection_v2(current_site, sites, bonds);
74 return current_site;
75 }
76 /**
77 *
78 * @param sites
79 * @param bonds
80 * @return
81 */
82 Index SitePercolationBallisticDeposition_v2::select_site_up_to_1nn(
```

```
81 vector<Index> &sites , vector<BondIndex> &bonds
82 ) {
83 // randomly choose a site
84 value_type r = _random_generator() % (indices_tmp.size());
85
86 Index current_site = index_sequence[indices_tmp[r]];
87 // cout << "current site " << current_site << endl;
88 // find the bonds for this site
89
90 // connection_v1(current_site , sites , bonds);
91 connection_v2(current_site , sites , bonds);
92
93 if (_lattice.getSite(current_site).isActive()){ // if the current site
94     is occupied or active
95 value_type r2 = _random_generator() % (sites.size());
96 current_site = sites[r2]; // select one of the neighbor randomly
97
98 if(_lattice.getSite(current_site).isActive()){
99 // if the neighbor is also occupied cancel current step
100 bool flag = true;
101 // cout << "if one of the neighbor is inactive. it's enough
102 // to go on" << endl;
103 for(auto s : sites){
104 // cout << s << "->";
105 if(!_lattice.getSite(s).isActive()){
106 // if one of the neighbor is unoccupied then
107 flag = false;
108 // cout << " inactive" << endl;
109 break;
110 }
111 }
112 if(flag){
113 // erase the index, since its four neighbors are occupied
114 indices_tmp.erase(indices_tmp.begin()+r);
115 throw OccupiedNeighbor{"all of the 1st neighbors are occupied : line " +
116 std::to_string(__LINE__));
117 }
118 }
```

```

120 //     cout << "choosing " << current_site << " out of the neighbors :
121     line " << __LINE__ << endl;
122 sites.clear();
123 bonds.clear();
124 connection_v2(current_site, sites, bonds);
125 return current_site;
126
127
128
129 /**
130 * Select neighbor upto 2nd nearest neighbor
131 * uses direction of motion when selecting 2nd nearest neighbor
132 * @param r : index of sites in the randomized array
133 * @param sites
134 * @param bonds
135 * @return
136 */
137 Index SitePercolationBallisticDeposition_v2::select_site_up_to_2nn(
138 vector<Index> &sites, vector<BondIndex> &bonds
139 ){
140 value_type r = _random_generator() % (indices_tmp.size());
141
142 Index central_site = index_sequence[indices_tmp[r]];
143 Index selected_site;
144 // find the bonds for this site
145
146
147 connection_v2(central_site, sites, bonds);
148
149 if (_lattice.getSite(central_site).isActive()){
150 bool flag_nn1 = true; // true means all 1st nearest neighbors are
151 // occupied
152 bool flag_nn2 = true; // true means all 2nd nearest neighbors are
153 // occupied
154 //     cout << "if one of the neighbor is inactive. it's enough to go
155 //     on" << endl;
156 for(auto s : sites){
157 //         cout << s << "->";
158 if(!_lattice.getSite(s).isActive()){
159 // if one of the neighbor is unoccupied then
160 flag_nn1 = false;
161 //         cout << " inactive" << endl;
162 break;

```

```
160 }
161 //           cout << " active" << endl;
162 }
163
164 value_type r2 = _random_generator() % (sites.size());
165 Index nn1 = sites[r2]; // select one of the neighbor randomly
166 //           cout << "nn1 " << nn1 << " : line " << __LINE__ << endl;
167 Index nn2;
168 if(_lattice.getSite(nn1).isActive()){
169 // if the neighbor is also occupied then choose the 2nd nearest neighbor
170 // in the direction of motion
171 nn2 = get_2nn_in_1nn_direction(central_site, nn1, length());
172 if(!_periodicity){
173 // if periodic boundary condition is not enabled then sites on the
174 // opposite edges will not contribute
175 vector<Index> tmp_sites;
176 vector<BondIndex> tmp_bonds;
177 // will find all possible neighbors of the selected first nearest
178 // neighbor
179 connection_v2(nn1, tmp_sites, tmp_bonds);
180 bool valid{false};
181 for(auto s: tmp_sites){
182 if(nn2 == s){
183 //           cout << "valid 2nd nearest neighbor : line "
184 //           << __LINE__ << endl;
185 valid = true;
186 break;
187 }
188 }
189 //           cout << "nn2 " << nn2 << " : line " << __LINE__ << endl;
190 // if it is also occupied the skip the step
191 if(_lattice.getSite(nn2).isActive()) {
192 flag_nn2 = true;
193 vector<Index> nn2_sites = get_2nn_s_in_1nn_s_direction(central_site,
194 sites, length());
195 for(auto x: nn2_sites){
196 if(!_lattice.getSite(x).isActive()){
197 flag_nn2 = false;
```

```
198 // cout << "inactive";
199 break;
200 }
201 }
202
203 if(flag_nn1 && flag_nn2){
204 // erase the index, since its 1st nearest neighbors are occupied
205 // and 2nd nearest neighbors are also occupied
206 indices_tmp.erase(indices_tmp.begin() + r);
207 }
208
209 throw OccupiedNeighbor{"2nd neighbor is also occupied : line " + std::
210 to_string(__LINE__);
211 } else {
212 selected_site = nn2;
213 }
214 } else {
215 selected_site = nn1;
216 }
217 sites.clear();
218 bonds.clear();
219
220 connection_v2(selected_site, sites, bonds);
221 } else {
222 selected_site = central_site;
223 }
224 return selected_site;
225 }
226
227
228
229 /* *****
230 * SitePercolationBallisticDeposition_v2
231 * select upto 1st nearest neighbor
232 */
233
234 /**
235 *
236 * @return
237 */
238 bool SitePercolationBallisticDeposition_v2::occupy() {
239 // if no site is available then return false
240 }
```

```
241 if (_number_of_occupied_sites == maxSites()) {
242     return false;
243 }
244
245 try {
246
247     value_type v = placeSite_1nn_v2();
248     _occupation_probability = occupationProbability(); // for super class
249
250
251     return v != ULLONG_MAX;
252 } catch (OccupiedNeighbor& on) {
253     //      on.what();
254     //      cout << "line : " << __LINE__ << endl;
255     return false;
256 }
257
258 }
259
260
261 /**
262 *
263 * 1. Randomly select a site from all sites
264 * 2. If it is not occupied occupy it.
265 * 3. If it is occupied select one of the 4 neighbor to occupy
266 * 4. If the selected neighbor is also occupied cancel current step
267 * 4. form cluster and track all informations
268 * 5. go to step 1
269 * 6. untill spanning cluster appears or no unoccupied site
270 */
271 value_type SitePercolationBallisticDeposition_v2::placeSite_1nn_v2() {
272
273     vector<BondIndex> bonds;
274     vector<Index>      sites;
275
276     _last_placed_site = select_site_up_to_1nn(sites, bonds);
277
278     return placeSite_weighted(_last_placed_site, sites, bonds);
279 }
280
281 /**
282 *
283 * @return
284 */
```

```

285 value_type SitePercolationBallisticDeposition_v2 :: placeSite_2nn_v1() {
286     vector<BondIndex> bonds;
287     vector<Index> sites;
288
289     try {
290         _last_placed_site = select_site_up_to_2nn(sites, bonds);
291         return placeSite_weighted(_last_placed_site, sites, bonds);
292         //    return placeSite_v11(_last_placed_site);
293     } catch (OccupiedNeighbor& e) {
294         cout << "Exception !!!!!!!!!!!!!!!" << endl;
295         e.what();
296     } catch (InvalidNeighbor& b) {
297         cout << "Exception !!!!!!!!!!!!!!!" << endl;
298         b.what();
299     }
300     return ULONG_MAX;
301 }
```

## A.2.8 Utilities

The `src/util/print.h` file

```

1 #ifndef PERCOLATION_PRINTER_H
2 #define PERCOLATION_PRINTER_H
3
4 #include <iostream>
5 #include <unordered_map>
6 #include <set>
7 #include <unordered_set>
8 #include <vector>
9 #include <map>
10 #include <initializer_list>
11
12 template <typename T>
13 std::ostream& operator<<(std::ostream& os, const std::vector<T> & vec) {
14     os << '{';
15     for (auto a: vec) {
16         os << a << ',';
17     }
18     return os << '}';
19 }
20
21 template <typename T>
22 std::ostream& operator<<(std::ostream& os, const std::set<T> & vec) {
```

```
23 os << '{' ;
24 for( auto a: vec){
25 os << a << ',' ;
26 }
27 return os << '}';
28 }
29
30 template <typename T>
31 std :: ostream& operator<<( std :: ostream& os , const std :: unordered_set<T> &
32 vec ){
33 os << '{' ;
34 for( auto a: vec){
35 os << a << ',' ;
36 }
37 return os << '}';
38 }
39 template <typename K, typename V>
40 std :: ostream& operator<<( std :: ostream& os , const std :: map<K, V> & m) {
41 os << '{' ;
42 for( auto a: m){
43 os << '(' << a.first << "->" << a.second << ")" , ",";
44 }
45 return os << '}';
46 };
47
48 template <typename K, typename V>
49 std :: ostream& operator<<( std :: ostream& os , const std :: unordered_map<K, V
50 > & m) {
51 os << '{' ;
52 for( auto a: m){
53 os << '(' << a.first << "->" << a.second << ")" , ",";
54 }
55 return os << '}';
56 };
57 /**
58 *
59 * Prints a horizontal barrier in the console.
60 * @param n : how many time the middle string is repeated.
61 * @param initial : string that is printed initially .
62 * @param middles : middle string .
63 * @param end : string that is printed at the end .
64 */
```

```

65 void print_h_barrier(size_t n, const std::string& initial, const std::
66   string& middles, const std::string& end="\n");
67 #endif //PERCOLATION_PRINTER_H

```

### The src/util/printer.cpp file

```

1 #include "printer.h"
2 #include <iostream>
3
4 using namespace std;
5
6 void print_h_barrier(size_t n, const string& initial, const string&
7   middles, const string& end){
8   cout << initial;
9   for(size_t i{}; i < n ; ++i){
10     cout << middles;
11   }
12   cout << end; // end of barrier
13 }

```

### The src/util/time\_tracking.h file

```

1 #ifndef PERCOLATION_TIME_TRACKING_H
2 #define PERCOLATION_TIME_TRACKING_H
3
4 #include <string>
5
6 std::string getFormattedTime(double t);
7 std::string currentTime();
8 std::string getCurrentTime();
9 #endif //PERCOLATION_TIME_TRACKING_H

```

### The src/util/time\_tracking.cpp file

```

1 #include "time_tracking.h"
2 #include <iostream>
3 #include <sstream>
4
5 using namespace std;
6
7 /**
8 *
9 * @param t -> Time in seconds
10 * @return
11 */
12 std::string getFormattedTime(double t){

```

```
13 std::string s;
14
15 size_t hr{}, min{};
16 double sec{}; // hour, min, sec
17 size_t integer_sec = size_t(t);
18
19 if(integer_sec >= 3600)
20 hr = integer_sec / 3600;
21 if(integer_sec >= 60)
22 min = (integer_sec - hr * 3600) / 60;
23 sec = t - min * 60 - hr * 3600;
24
25 s += to_string(hr);
26 s += " hr ";
27 s += to_string(min);
28 s += " min ";
29 s += to_string(sec);
30 s += " sec ";
31 return s;
32 }
33
34 std::string currentTime() {
35 time_t t = time(0); // get birthTime now
36 struct tm * now = localtime( & t );
37 stringstream ss;
38 ss << (now->tm_year + 1900) << '.'
39 << (now->tm_mon + 1) << '.'
40 << now->tm_mday << '_'
41 << now->tm_hour << '.' << now->tm_min << '.' << now->tm_sec;
42
43 return ss.str();
44 }
45
46 /**
47 formated date
48 24 hour formated time
49 */
50 string getCurrentTime(){
51
52 time_t rawtime;
53 struct tm * timeinfo;
54 char buffer [80];
55
56 time (&rawtime);
```

```

57 timeinfo = localtime (&rawtime);
58
59 strftime (buffer,80,"%F_%H%M%S",timeinfo);
60 return buffer;
61 }
```

### The `src/types.h` file

```

1 #ifndef SITEPERCOLATION_TYPES_H
2 #define SITEPERCOLATION_TYPES_H
3
4 using value_type = unsigned long;
5 using signed_value_type = long;
6
7 #endif // SITEPERCOLATION_TYPES_H
```

## A.2.9 Test

### The `src/test/test_percolation.h` file

```

1 #ifndef SQLATTICEPERCOLATION_TEST_PERCOLATION_H
2 #define SQLATTICEPERCOLATION_TEST_PERCOLATION_H
3
4 #include <iostream>
5 #include <string>
6 #include <chrono>
7 #include "../types.h"
8 #include "../percolation/percolation.h"
9 #include "../util/time_tracking.h"
10
11
12 /**
13 *
14 * @tparam PType : Template type of percolation class
15 * @param argc : argc from commandline
16 * @param argv : argv from commandline
17 */
18 template<class PType>
19 void simulate_site_percolation_T(value_type length, value_type
20 ensemble_size) {
21 std::cout << "length " << length << " ensemble_size " << ensemble_size
22 << std::endl;
23 value_type length_squared = length*length;
```

```
24 value_type twice_length_squared = 2 * length_squared;
25
26 PType lattice_percolation(length, true);
27
28 std::ostringstream header_info;
29 header_info << "{"
30 << "\\" length \":\"" << length
31 << "\\" ensemble_size \":\"" << ensemble_size
32 << "\\" signature \":\\" " << lattice_percolation.getSignature() << "\\" "
33 << "}" ;
34
35 std::string tm = getCurrentTime();
36
37 std::string filename_s = lattice_percolation.getSignature() + "
38   _cluster_by_site_" + std::to_string(length) + '_' + tm;
39 std::string filename_b = lattice_percolation.getSignature() + "
40   _cluster_by_bond_" + std::to_string(length) + '_' + tm;
41 std::string filename_critical = lattice_percolation.getSignature() + "
42   _critical_" + std::to_string(length) + '_' + tm;
43 std::string filename_entropy_order_parameter = lattice_percolation.
44   getSignature() + std::to_string(length) + '_' + tm;
45
46
47
48 std::ofstream fout_s(filename_s);
49 // JSON formated header
50 fout_s << '#' << header_info.str() << std::endl;
51 fout_s << "#each line is an independent realization" << std::endl;
52 fout_s << "#each line contains information about all clusters at
53   critical point" << std::endl;
54 fout_s << "#cluster size is measured by number of sites in it" << std::
55   endl;
56
57 std::ofstream fout_b(filename_b);
58 // JSON formated header
59 fout_b << '#' << header_info.str() << std::endl;
60 fout_b << "#each line is an independent realization" << std::endl;
61 fout_b << "#each line contains information about all clusters at
62   critical point" << std::endl;
```



```
97 for(value_type j{}; j != site.size(); ++j){  
98     fout_s << site[j] << ',';  
99 }  
100 for(value_type j{}; j != bond.size(); ++j){  
101     fout_b << bond[j] << ',';  
102 }  
103  
104  
105     fout_s << std::endl;  
106     fout_b << std::endl;  
107     wrapping_written = true;  
108 }  
109  
110  
111     ++counter;  
112 }  
113 if(counter >= lattice_percolation.maxIterationLimit()){ //  
    twice_length_squared is the number of bonds  
114     break;  
115 }  
116 }  
117  
118 auto t_end = std::chrono::system_clock::now();  
119 std::cout << "Iteration " << i << ". Elapsed time " << std::chrono::  
    duration<double>(t_end - t_start).count() << " sec" << std::endl;  
120 }  
121 }  
122  
123 fout_b.close();  
124 fout_s.close();  
125 fout_critical.close();  
126  
127  
128  
129 std::ofstream fout(filename_entropy_order_parameter);  
130 fout << '#' << header_info.str() << std::endl;  
131 fout << "#<p>,<H(p,L)>,<P1(p,L)>,<P2(p,L)>" << std::endl;  
132 fout << "#p = occupation probability" << std::endl;  
133 fout << "#H(p,L) = Entropy = sum( - u_i * log(u_i))" << std::endl;  
134 fout << "#P1(p,L) = Order parameter = (number of bonds in largest  
    cluster) / (total number of bonds)" << std::endl;  
135 fout << "#P2(p,L) = Order parameter = (number of bonds in spanning or  
    wrapping cluster) / (total number of bonds)" << std::endl;  
136 fout << "#C(p,L) = Specific heat = -T dH/dT" << std::endl;
```

```

137 fout << "#X(p,L) = Susceptibility = dP/dp" << std::endl;
138 fout << "#u_i = (number of bonds in the i-th cluster) / (total number of
   bonds)" << std::endl;
139 for(size_t i{}; i < lattice_percolation.maxIterationLimit(); ++i){
140     fout << (i+1) / double(lattice_percolation.maxIterationLimit()) << ",";
141     fout << entropy[i] / double(ensemble_size) << ",";
142     fout << nob_largest[i] / double(ensemble_size /* lattice_percolation.
       maxBonds() */) << ",";
143     fout << nob_wraping[i] / double(ensemble_size /* lattice_percolation.
       maxBonds() */) ;
144     fout << std::endl;
145 }
146 fout.close();
147 }
148
149 #endif //SQLATTICEPERCOLATION_TEST_PERCOLATION_H

```

### A.2.10 Main

The main function receives 3 additional command line argument. First one is an integer  $l \in 0, 1, 2$  which determine the range of interaction. Second one is the length of the lattice. And third one is the size of the ensemble. For example, 12005000 will run the program for  $l = 1$ ,  $L = 200$  for ensemble size of 5000

The `src/main.cpp` file

```

1 #include <iostream>
2 #include <fstream>
3 #include <ctime>
4 #include <chrono>
5 #include <thread>
6 #include <mutex>
7
8 #include "lattice/lattice.h"
9 #include "percolation/percolation.h"
10 #include "util/time_tracking.h"
11 #include "util/printer.h"
12
13 #include "tests/test_percolation.h"
14
15
16 using namespace std;
17
18

```

```
19 /* ***
20 *   All the function that is run in main
21 * @param argc
22 * @param argv
23 */
24 void run_in_main(int argc, char** argv){
25
26 int l = atoi(argv[1]);
27 value_type length = atoi(argv[2]);
28 value_type ensemble_size = atoi(argv[3]);
29
30 if(l==1) {
31 cout << "Simulating site percolation for l=1" << endl;
32 simulate_site_percolation_T<SitePercolationBallisticDeposition_L1_v2>(
33     length, ensemble_size); // 2018.11.03
34 }
35 else if(l==2) {
36 cout << "Simulating site percolation for l=2" << endl;
37 simulate_site_percolation_T<SitePercolationBallisticDeposition_L2_v2>(
38     length, ensemble_size); // 2018.11.03
39 } else {
40 cout << "Simulating site percolation for l=0" << endl;
41 simulate_site_percolation_T<SitePercolation_ps_v9>(length, ensemble_size
42 );
43 }
44
45
46 /* *****/
47 * The main function
48 *
49 *****/
50 int main(int argc, char** argv) {
51
52 cout << "Running started at : " << currentTime() << endl;
53 cout << "Compiled on           : " << __DATE__ << "\t at " << __TIME__ <<
54 endl;
55 std::cout << "Percolation in a Square Lattice" << std::endl;
56 auto t_start = std::chrono::system_clock::now();
57 time_t seed = time(NULL);
58 srand(seed); // seeding
```

```

59
60 run_in_main(argc, argv);
61
62 auto t_end= std::chrono::system_clock::now();
63 std::chrono::duration<double> drtion = t_end - t_start;
64 std::time_t end_time = std::chrono::system_clock::to_time_t(t_end);
65 cout << "Program finished at " << std::ctime(&end_time) << endl;
66 std::cout << "Time elapsed " << getFormattedTime(drtion.count()) <<
67 std::endl;
68 return 0;
69 }
```

## A.2.11 CMakeLists

### The CMakeLists.txt file

```

1 cmake_minimum_required(VERSION 3.0)
2 project(SqLatticePercolation)
3
4 set(CMAKE_CXX_STANDARD 11)
5
6 #set (CMAKE_C_COMPILER           /usr/bin/gcc)
7 #set (CMAKE_CXX_COMPILER         /home/shahnoor/software/pgi/linux86
8 #set (CMAKE_MAKE_PROGRAM        /usr/bin/make)
9 SET( CMAKE_CXX_FLAGS   "-pthread -fopenmp")
10 #SET( CMAKE_EXE_LINKER_FLAGS  "${CMAKE_EXE_LINKER_FLAGS} ${
11   GCC_COVERAGE_LINK_FLAGS}" )
12
13 set(SOURCE_FILES
14   src/main.cpp
15   src/types.h
16   src/exception/exceptions.h
17   src/index/index.cpp
18   src/index/index.h
19   src/lattice/bond.cpp
20   src/lattice/bond.h
21   src/lattice/bond_type.h
22   src/lattice/lattice.cpp
23   src/lattice/lattice.h
24   src/lattice/site.cpp
25   src/lattice/site.h
26   src/percolation/cluster.cpp
27   src/percolation/cluster.h
```

```
27 src/percolation/percolation.cpp
28 src/percolation/percolation.h
29 src/percolation/percolation_site_ballistic_deps_v2.cpp
30 src/util/printer.h
31 src/util/time_tracking.cpp
32 src/util/time_tracking.h
33 src/util/printer.cpp
34 src/percolation/percolation_site_v9.cpp
35 src/tests/test_percolation.h)
36
37 add_executable(SqLatticePercolation ${SOURCE_FILES})
```

### A.2.12 complete code

Complete code for RSBD model on square lattice is available at [https://github.com/sha314/SqLattice\\_RSBD](https://github.com/sha314/SqLattice_RSBD) or use the git link to clone the repository [https://github.com/sha314/SqLattice\\_RSBD.git](https://github.com/sha314/SqLattice_RSBD.git)

Detailed version of the same program with other extensions are available at <https://github.com/sha314/SqLatticePercolation> or the git link <https://github.com/sha314/SqLatticePercolation.git>



# Appendix B

## Convolution

### B.1 Algorithm

One further slightly tricky point in the implementation of our scheme is the performance of the convolution, Eq. (2), of the results of the algorithm with the binomial distribution. Since the number of sites or bonds on the lattice can easily be a million or more, direct evaluation of the binomial coefficients using factorials is not possible. And for high-precision studies, such as the calculations presented in Section III, a Gaussian approximation to the binomial is not sufficiently accurate. Instead, therefore, we recommend the following method of evaluation. The binomial distribution, Eq. (1), has its largest value for given  $N$  and  $p$  when  $n = n_{\max} = pN$ . We arbitrarily set this value to 1. (We will fix it in a moment.) Now we calculate  $B(N, n, p)$  iteratively for all other  $n$  from

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

Then we calculate the normalization coefficient  $C = \sum_n B(N, n, p)$  and divide all the  $B(N, n, p)$  by it, to correctly normalize the distribution. [site to Ziff paper](#)

### B.2 Code

Complete code for convolution is available at <https://github.com/sha314/Convolution>



# Appendix C

## Finding Exponents

### C.1 Algorithm

#### C.1.1 Specific Heat and Susceptibility

##### exponent for scaling the y-values

To find the best exponent for scaling the  $y$ -values of specific heat and susceptibility the following approach is pretty helpful.

##### Finding approximate value of the exponent

1. get the  $x$  and  $y$  values of the convoluted data for all lengths ( $L$ )
2. get the maximum value or the peak value of  $y$  as  $y_{max} = \max(y)$  for each length
3. now plot  $\log(y_{max})$  vs  $\log(L)$
4. slope of this graph is the approximate value of the exponent

##### Finding the best exponent

Now that we know the approximate value of the exponent,  $ex_{approx}$ , we can find the best exponent in the following way

1. take a list of all the exponent in the range  $E = [ex_{approx} - \epsilon, ex_{approx} + \epsilon]$  where  $\epsilon$  is a small number (usually  $\sim 0.05$ )
2. for each value of the exponent in the range above do the following
  - (a) get the  $x$  and  $y$  values of the convoluted data for all lengths ( $L$ )
  - (b) scale the  $y$  value with the exponent and call it  $y' = y * L^{-ex}$

- (c) get the maximum value or the peak value of  $y'$  as  $y'_{max} = \max(y')$  for each length
  - (d) if the  $y'_{max}$  values for different exponents does not lie in the order of 10 then scale  $y'_{max} = y'_{max}/\max(y'_{max})$  to normalize  $y'_{max}$ . By order of 10 I mean that if the order  $y'_{max}$  for  $ex_i$  is  $10^{-1}$  and for  $ex_j$  is  $10^{-2}$  then when comparing between  $std_{ex_i}$  and  $std_{ex_j}$  we will get the  $std_{ex_j}$  is lower always. Thus just because the standard deviation is lower we cannot say it is the best exponent if the order of  $y'_{max}$  is different.
  - (e) find the standard deviation of all the  $y'_{max}$ 's and call it  $std_{ex}$  where the subscript denotes the current selected exponent
3. out of a number of selected exponent find the one with the minimum standard deviation and the exponent corresponding to this deviation is the best exponent denoted as  $ex^*$ . Symbolically

$$ex^* = \operatorname{argmin}_{ex \in E} std_{ex} \quad (\text{C.1})$$

### **exponent for scaling the $x$ -values**

Usually the exponent that scales the  $x$ -values is called  $1/\nu$ . The critical point is denoted as  $x_c$ . To find an estimate from the data that looks like the graph of specific heat or susceptibility, the following algorithm is very helpful

1. get the  $x$  and  $y$  values of the convoluted data for all lengths ( $L$ )
2. get the  $x$  value at a specific height,  $h$ , and call it  $x_h$
3. plot a graph of  $\log(|x_h - x_c|)$  vs  $\log(L)$  where  $||$  denotes the absolute value.
4. slope of this graph is the estimate for the exponent  $1/\nu$

Now to find the exponent that best collapses the data is the main goal. To do this the following algorithm can be followed.

Finding the standard deviation of the points at height  $h$  after scaling  $x$ -values with an approximate value of the exponent  $(1/\nu)_{approx}$  that is obtained from the graph of  $\log(|x_h - x_c|)$  vs  $\log(L)$ .

1. write a function that takes  $h$ ,  $x_{scaler}$ ,  $y_{scaler}$ ,  $lr$  as argument where,  $h$  is the height at which we will be taking  $x$ -values,  $x_{scaler}$  is the exponent that scales the  $x$ -values,  $y_{scaler}$  is the exponent that scales the  $y$ -values and  $lr$  is the argument that tells if the left or right side of the critical point should be taken under consideration. call this function *find\_x\_deviation*

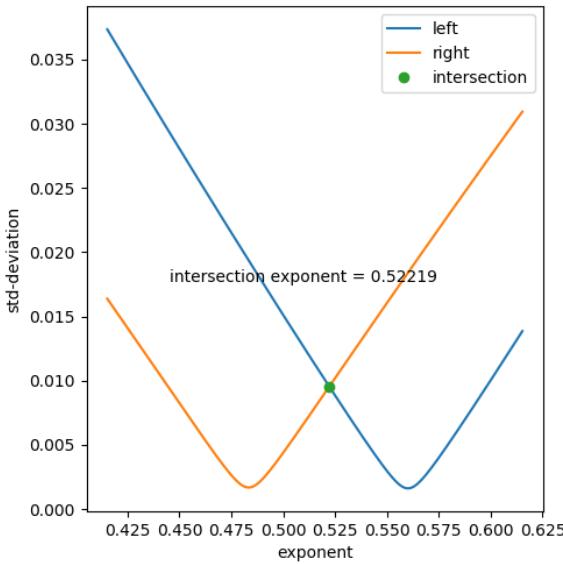


Fig. C.1 Best collapsing on either right or left  
??

2. take  $x' = (x - x_c)L^{x_{scaler}}$  and  $y' = yL^{-y_{scaler}}$ . Note that there is a minus sign used for scaling  $y$  values because the  $y_{\max}$  increases as  $L$  increases in our present case which is observed when finding  $y_{scaler}$  previously.
3. at height  $h$  we draw a horizontal straight line and the intersection of this line with the curve gives corresponding  $x$  value at  $h$ . For each length  $L$  we obtain the  $x$  value and denote it with  $x_L$ .
4. after that we find the standard deviation of all  $x_L$ 's that we have found and this function returns the standard deviation
5. if  $lr$  is 0 then the left points of the critical point is considered and if  $lr$  is 1 then the right points of the critical point is considered for getting  $x_L$ 's.

Note that at a specific height there are two points on the left of the critical point and another is on the right. So if we find the exponent that best collapses the points on the left, it might not best collapse the points on the right ?? To resolve this problem we take following approach

1. take a range  $ex = [(1/nu)_{approx} - \varepsilon, (1/nu)_{approx} + \varepsilon]$ , where  $\varepsilon$  is usually  $\sim 0.05$ .
2. for each value in this range find the standard deviation for left and right points of the critical point and call it  $d_{left}$  and  $d_{right}$  respectively

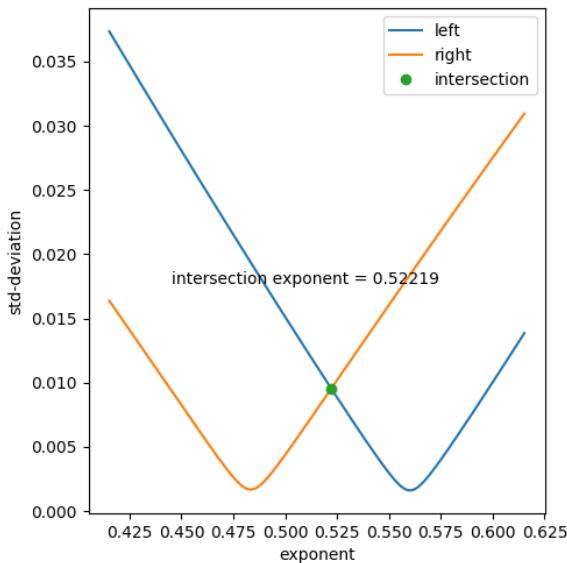


Fig. C.2 Minimizing exponent for scaling  $x$ -values

3. plot  $d_{left}$  vs  $ex$  and  $d_{right}$  vs  $ex$  on the same graph
4. the minima of line corresponding to  $d_{left}$  vs  $ex$  graph gives the exponent that collapses left points of the critical point at best.
5. the minima of line corresponding to  $d_{right}$  vs  $ex$  graph gives the exponent that collapses right points of the critical point at best.
6. the intersection of the graph is the value where both left and right points of the critical point fits better.

The figure C.2 gives the visual of the above process.