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

November 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
November 2018

Acknowledgements

And I would like to acknowledge ...

Abstract

This is where you write your abstract ...

Table of contents

List of figures

List of tables

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 physicits 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 π 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, ince 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 π 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 π theorem is one of

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

Buckingham π theorem describes dimensionless variables obtained from the power products of governing parameters denoted by $\Pi_1, \Pi_2, \ldots$ 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

$$a = f(a_1, a_2, \ldots, a_3) \tag{2.1}$$

where the quantities $a_1, a_2, \ldots, a_n$ 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 $a_{k+1}, \ldots, a_n$ have the independent dimensions and the dimensions of the arguments $a_1, a_2, \ldots, a_k$ can be expressed in terms of the dimensions of the governing independent parameters $a_{k+1}, \ldots, a_n$ in the following way

$$\begin{aligned} [a_1] &= [a_{k+1}]^{\alpha_1} \ldots [a_n]^{\gamma_1} \\ [a_2] &= [a_{k+1}]^{\alpha_2} \ldots [a_n]^{\gamma_2} \qquad (2.2) \\ &\vdots \qquad (2.3) \\ [a_k] &= [a_{k+1}]^{\alpha_k} \ldots [a_n]^{\gamma_k} \end{aligned}$$

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

$$[a] = [a_{k+1}]^{\alpha} \ldots [a_n]^{\gamma} \tag{2.4}$$

Thus, there exist number α, γ such that 2.4 holds. We have set of governing parameters.

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

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

$$\vdots \tag{2.7}$$

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

and a dimensionless governed parameter

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

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

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

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 Π. 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,

$$\Pi = \Phi(\Pi_1, \dots, \Pi_k) \tag{2.11}$$

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

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

equation 2.12 is known as the Buckingham Π 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 Π 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 θ. Assume that we are to measure the area S of the triangle. The area S can be written in the following form

$$S = S(a,b,c) \tag{2.13}$$

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

$$[a] \sim [c] \text{ and } [b] \sim [c] \tag{2.14}$$

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

$$\Pi_1 = \sin\theta = a/c \tag{2.15}$$

$$\Pi_2 = \cos\theta = b/c \tag{2.16}$$

and the dimensionless governed paremeter

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

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 Π_1, Π_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

$$\Pi = \phi(\Pi_1, \Pi_2) \tag{2.18}$$

However, Π_1 and Π_2 both depends on the dimensionless quantity θ and hence we can write

$$S = c^2 \phi(\theta) \tag{2.19}$$

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

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

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 θ 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

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

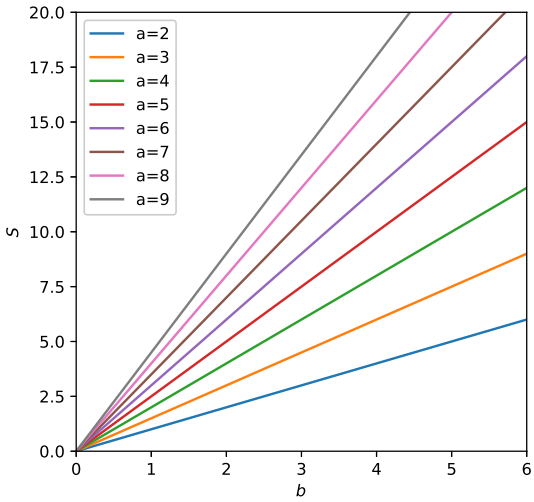
since the numerical value of the ratio of the area over the square of the hypotenuse depends on the angle θ. It implies that if we plot the ratio of the area over the square of the hypotenuse as a function θ all the data points should collapse onto a single curve regardless of the size of the hypotenuse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

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

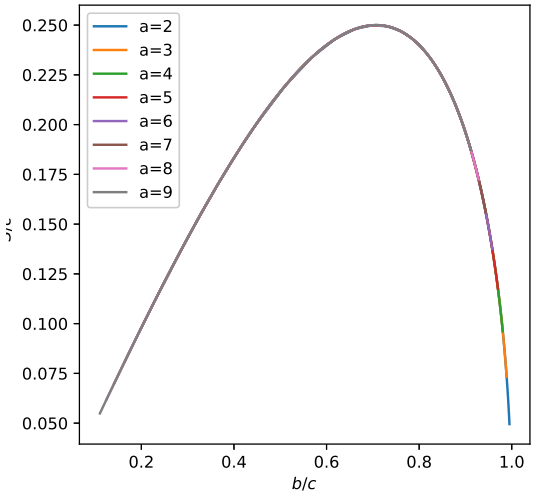
This data collapse implies that if two or more right triangles which have one of the acute angle identical then such triangles are similar. We shall see later that whenever we will find data collapse between two different systems of the same phenomenon then it would mean that the corresponding systems 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 param- eters, i.e. the lengths of the sides, while the dimensionless parameters, the angles at the vertices are identical for the two triangles. Analogously,



(a) S vs b graph. b is the base of the triangle which is varied for a single θ.



(b) S/c^2 vs b/c graph. This each graph contains information about all triangle with a certain acute angle, meaning the entire graph contains information about every right triangle.

Fig. 2.1 Application of Buckingham Π 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 $\Pi_1, \ldots, \Pi_k$ 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

$$\alpha = f(a_1, a_2, a_3, b_1, b_2) \tag{2.23}$$

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

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

and for model we have

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

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

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

and

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

where the function Φ 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.,

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

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

It also follows that the governed dimensionless parameter satisfies

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

Returning to dimensional variables, we get from the above equation

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

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 Π_1, Π_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

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

and

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

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.

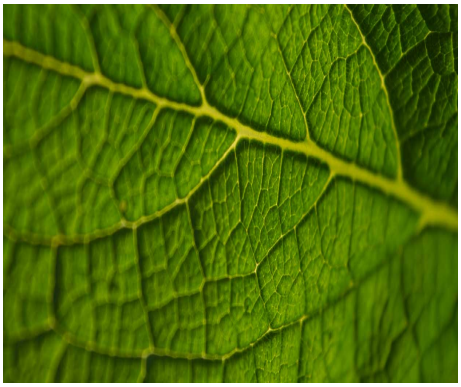
The cauliflower head contains branches or parts, which when removed and compared with the whole found to be very much the same except it is scaled down. These isolated branches can again be decomposed into smaller parts, which again look very similar to the whole as well as of the branches. Such self-similarity 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) Disection 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

$$f(x,t) \sim t^{\theta}\phi(x/t^{z}) \tag{2.34}$$

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

$$f(x,t) \sim x^{\omega}\phi(x/t^{z}) \tag{2.35}$$

Buckingham π-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

$$x \to \lambda^{z} x \tag{2.36}$$

$$t \to \lambda t \tag{2.37}$$

$$f \to \lambda^{\theta} \tag{2.38}$$

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

$$x \sim t^z \tag{2.39}$$

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 ξ 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^θ to obtain yet another dimensionless quantity $\phi = f(x,t)/t^\theta$ where the exponent θ is fixed by the dimensional requirement $[f] = \left[t^\theta\right]$. Since ϕ is a dimensionless quantity its numerical value can only depend on dimensionless quantity ξ 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

$$M \sim (T - T_c)^\beta \tag{2.40}$$

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

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

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

where, M, χ_M, C_V, ξ 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 $f(\varepsilon, L)$ with $\varepsilon = T - T_c$ is said to obey finite size scaling if it can be expressed as

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

However, using the Buckingham π-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 ξ in the limit $L \to \infty$ 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 ξ,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

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

and the corresponding dimensionless governing parameter is

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

Following the argument of the π-theorem we can immediately write that

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

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

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

Till to date neither of the two scaling forms obtained following π 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

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

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^{θ} which according to the prescription of π-theorem we have the following FSS scaling form

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

which is the same as the traditional scaling form given by 2.44 if we find θ negative and it is related to the exponent ν 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 $L \to \infty$. 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

$$f(\lambda x) = g(\lambda) f(x) \; \forall \lambda \tag{2.51}$$

where $g(\lambda)$ is yet unspecified function. That is, one is interested in the shape of $f(\lambda x)$ for some scale factor λ 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

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

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

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

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

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

This is a first order differential equation which has a solution

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

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 $\frac{f(\lambda r)}{f(x)}$ depends on λ alone. Thus the distribution does not need a characteristic scale. If we change the unit of measurement of x by a factor of λ, 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 λ the function $f(x,y)$ satisfies,

$$f(\lambda^a x, \lambda^b y) = \lambda f(x,y) \tag{2.56}$$

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,

$$f(\lambda^a x, \lambda^b y) = \lambda^p f(x,y) \tag{2.57}$$

choosing $p = 1$ in the above equation yields

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

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,

$$f(\lambda x, \lambda^b y) = \lambda^p f(x,y) \tag{2.59}$$

Similarly

$$f(\lambda^a x, \lambda y) = \lambda^p f(x,y) \tag{2.60}$$

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

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

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

This combining and hence the simplification of two variables x and y into a single term has far reaching consequence in Windom 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.

$$\Delta E = Q + W \tag{3.1}$$

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

$$dE = dQ - PdV \tag{3.2}$$

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

$$dQ = TdS\textbf{??} \tag{3.3}$$

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

$$dE = TdS - pdV \tag{3.4}$$

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

$$dE = TdS - pdV + \mu dN \tag{3.5}$$

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

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

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

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

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

3.2 Definition of Thermodynamic Quantities

3.2.1 Entropy

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 $1cm^3$, 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.

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.

$$C = \frac{Q}{dT} \tag{3.8}$$

and we know

$$Q = T\,dS \tag{3.9}$$

where S is the entropy. we immediately get

$$C = T\frac{dS}{dT} \tag{3.10}$$

3.2.4 Order Parameter

3.2.5 Susceptibility

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

$$dE = T\,dS - P\,dV \tag{3.11}$$

for a magnetic system $P \to h$ and $V \to -m$, 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

$$dE = TdS + hdm \tag{3.12}$$
$$= TdS + SdT - SdT + hdm \tag{3.13}$$
$$d(E - TS) = -SdT + hdm \tag{3.14}$$
$$dA = -SdT + hdm \tag{3.15}$$
$$dA = -SdT + hdm + mdh - mdh \tag{3.16}$$
$$d(A - mh) = -SdT - mdh \tag{3.17}$$
$$dG = -SdT - mdh \tag{3.18}$$

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 3.18 3.15 we can find the expression for entropy and magnetization in terms of free energy.

$$S = -\left(\frac{\partial G}{\partial T}\right)_h = -\left(\frac{\partial A}{\partial T}\right)_m \tag{3.19}$$

$$m = -\left(\frac{\partial G}{\partial h}\right)_T \tag{3.20}$$

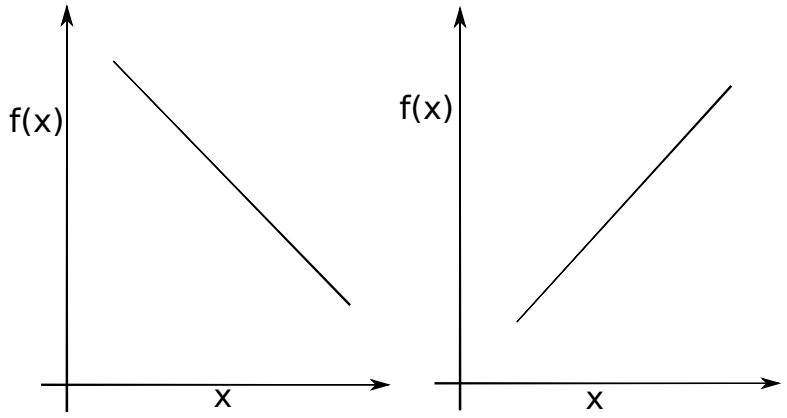
$$h = \left(\frac{\partial A}{\partial m}\right)_T \tag{3.21}$$

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

$$\chi = \left(\frac{\partial m}{\partial h}\right)_h = -\left(\frac{\partial^2 G}{\partial h^2}\right)_T \tag{3.22}$$

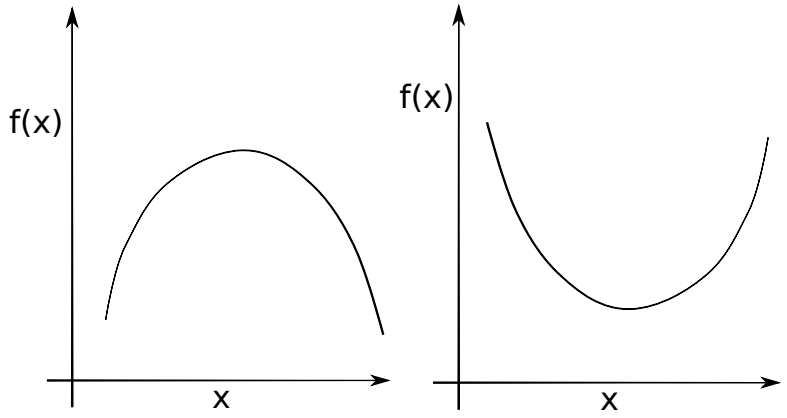
again we can write χ as

$$\chi = \frac{1}{\frac{\partial h}{\partial m}}$$
$$= \frac{1}{\frac{\partial^2 A}{\partial m^2}} \tag{3.23}$$



(a) negative slope (b) positive slope

Fig. 3.1 shape of $f(x)$ from $f'(x)$



(a) concave (b) convex

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

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 3.1.

If the second derivative of this function is negative (positive) the function is said to have concave (convex) shape 3.2. Thus if we know the sign of the first and second derivative we can approximate a shape of the function.

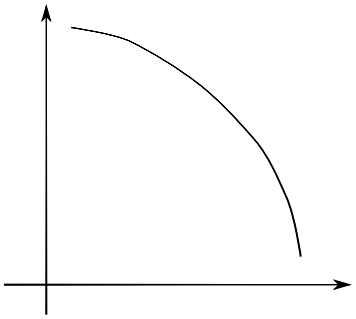


Fig. 3.3 Shape of the free energy

3.3.2 Free Energy

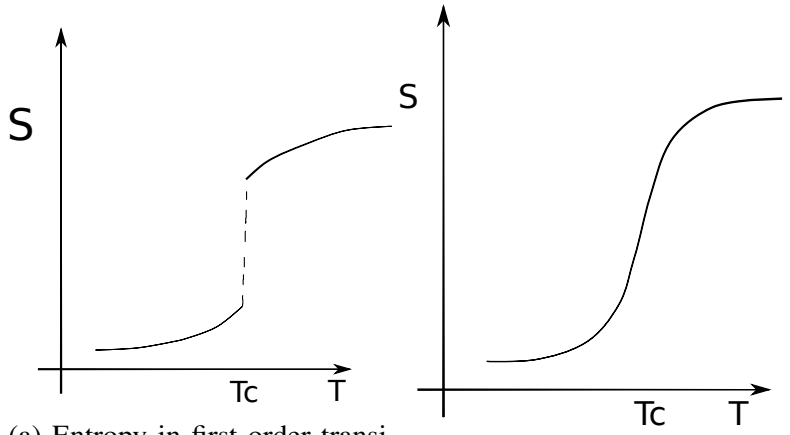
Now since we obtain from 3.7 that the entropy is the first derivative of the free energy 3.24 and from 3.29 specific heat is the second derivative of the free energy 3.25. Now since the entropy is a positive quantity and specific heat is also a positive quantity we get that the first and second derivative of the free energy is negative which implies from the laws of calculus that the shape of the free energy should be a decreasing concave curve 3.3 in other words concave shaped with negative slope.

$$S = -\frac{\partial G}{\partial T} \tag{3.24}$$

$$C = -\frac{\partial^2 G}{\partial T^2} \tag{3.25}$$

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 3.4. From the definition of the transition order we know that the first order transition is discontinuous and the second order transition continuous. 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



(a) Entropy in first order transition

(b) Entropy in 2nd order transition

Fig. 3.4 Shape of entropy

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 3.27. 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 3.22 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 3.20 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 3.23 we can say $\frac{\partial^2 A}{\partial m^2}$ is positive giving convex shape of the Helmholtz free energy. And 3.21 tells us that since h can ve 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

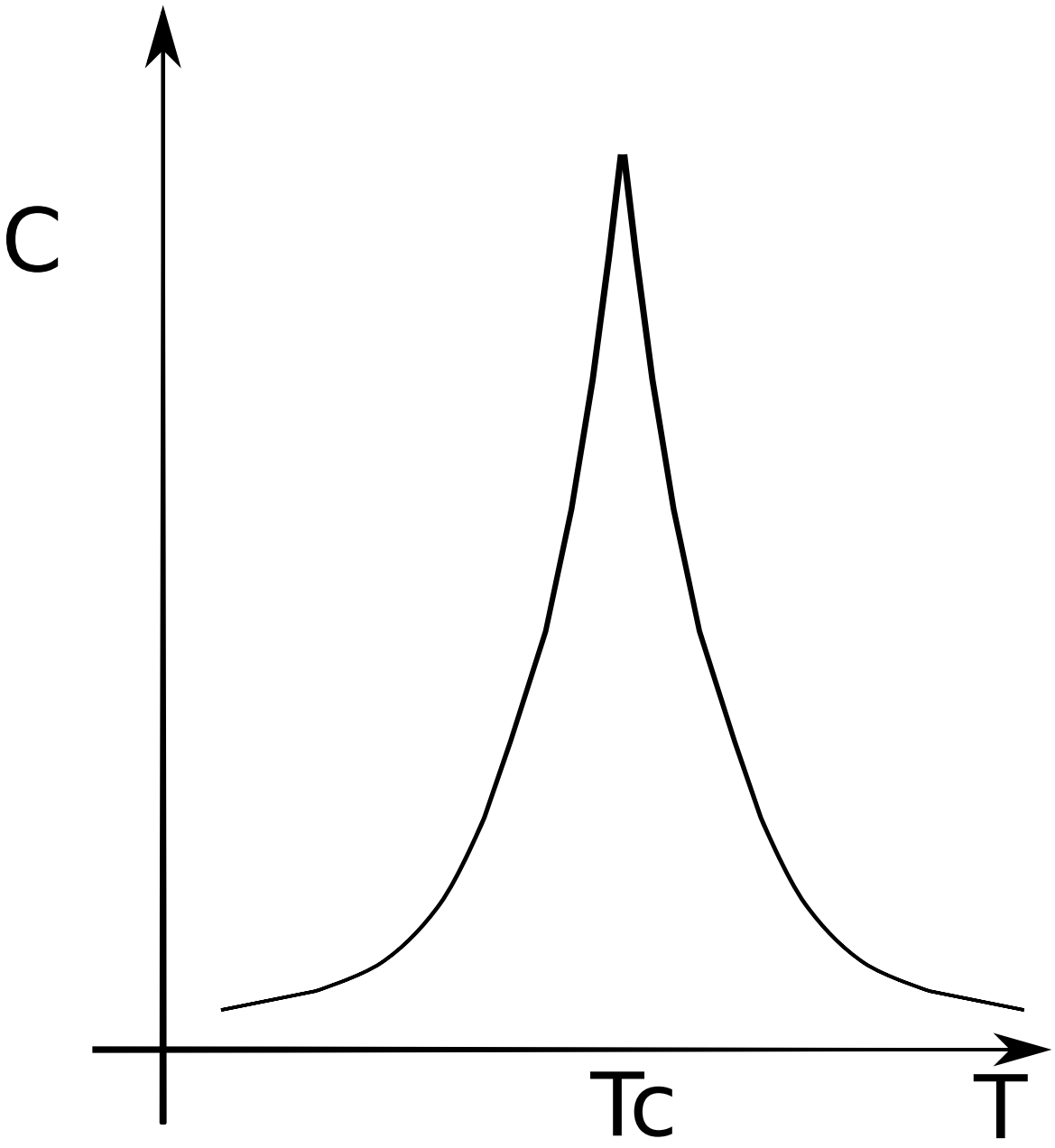


Fig. 3.5 Shape of Specific Heat

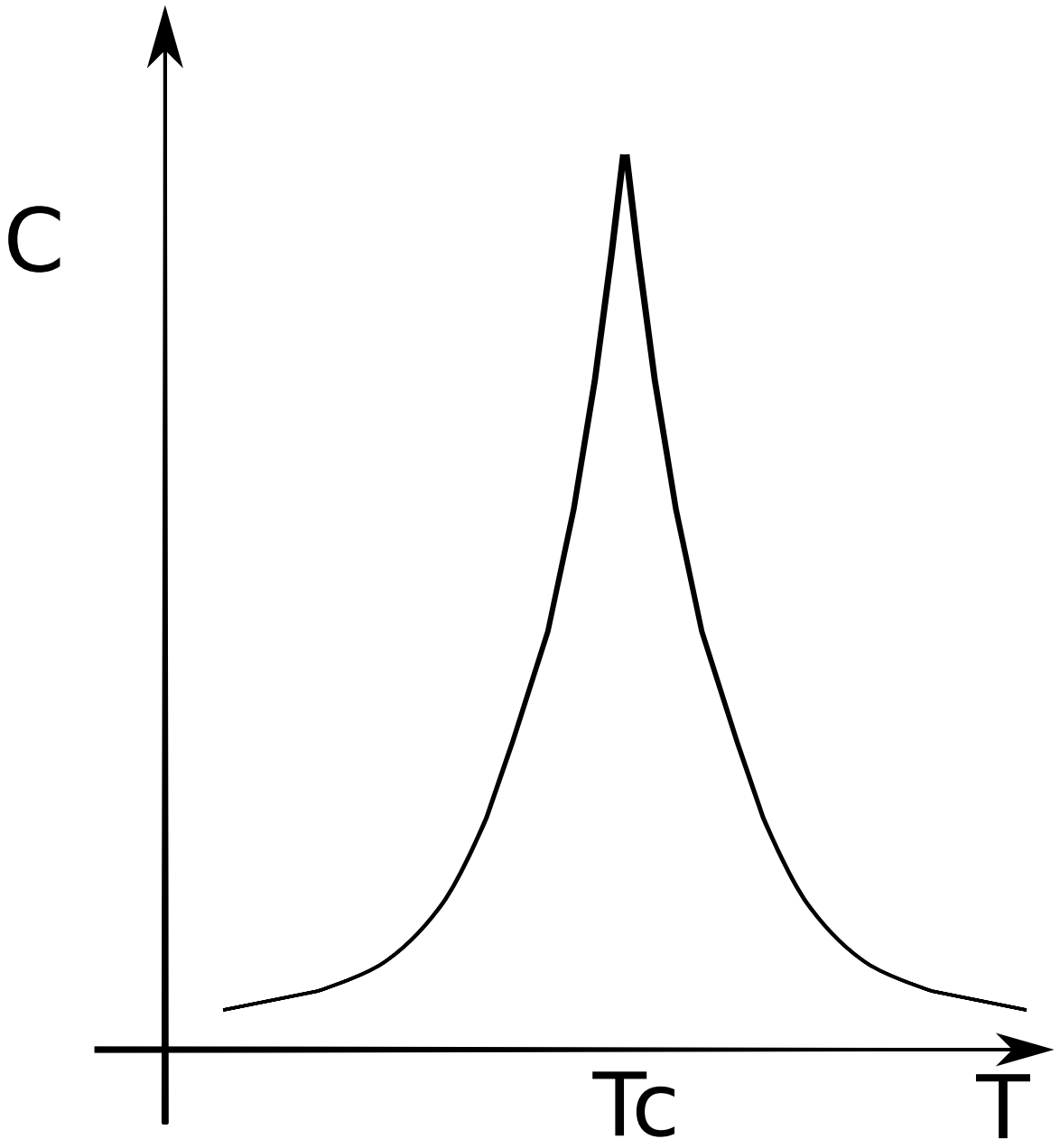


Fig. 3.6 figure required

respect to the temperature at constant pressure or volume respectively.

$$C_x = \left(\frac{dQ}{dT}\right)_x \tag{3.26}$$

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

$$C_x = T\left(\frac{dS}{dT}\right) \tag{3.27}$$

To give the following

$$C_v = -T\left(\frac{\partial^2 A}{\partial T^2}\right)_v \tag{3.28}$$

$$C_p = -T\left(\frac{\partial^2 G}{\partial T^2}\right)_p \tag{3.29}$$

Nearest neighbor interation 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

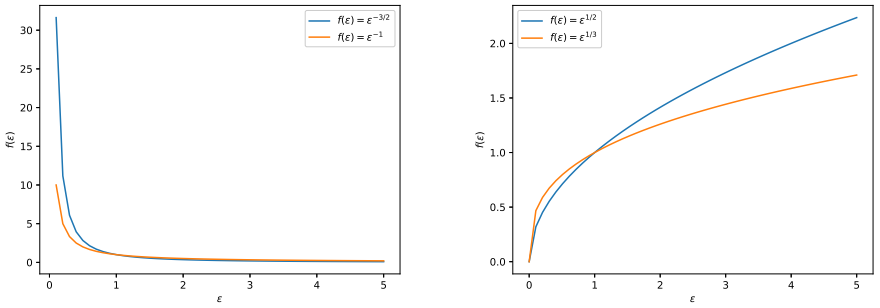
$$\varepsilon = \frac{T - T_c}{T_c} = \frac{T}{T_c} - 1 \tag{3.30}$$

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

$$f(\varepsilon) \sim \varepsilon^{\lambda} \tag{3.31}$$

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

$$f(\varepsilon) = A\varepsilon^{\lambda}(1 + B\varepsilon^{a} + C\varepsilon^{b} + ...) \tag{3.32}$$



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

Fig. 3.7 Function showing power law

Near T_c only the first term is dominating **??**.
We had the Gibbs free energy (for a magnetic system)

$$G = G(T,h) \equiv G(\varepsilon,h) \tag{3.33}$$

since ε is a better parameter. Assume that G is a generalized Homogeneous function. Therefore from **??** we can write

$$G(\lambda^{a_\varepsilon}\varepsilon, \lambda^{a_h}h) = \lambda G(\varepsilon,h) \tag{3.34}$$

differentiating equation 3.34 with respect to h

$$\begin{aligned}
\frac{\partial G(\lambda^{a_\varepsilon}\varepsilon, \lambda^{a_h}h)}{\partial h} &= \lambda \frac{\partial G(\varepsilon,h)}{\partial h} \\
\frac{\partial G(\lambda^{a_\varepsilon}\varepsilon, \lambda^{a_h}h)}{\partial \lambda^{a_h} h} \lambda^{a_h} &= \lambda \frac{\partial G(\varepsilon,h)}{\partial h} \\
\lambda^{a_h} G'(\lambda^{a_\varepsilon}\varepsilon, \lambda^{a_h}h) &= \lambda G'(\varepsilon,h) \\
-\lambda^{a_h} m(\lambda^{a_\varepsilon}\varepsilon, \lambda^{a_h}h) &= -\lambda m(\varepsilon,h) \\
m(\lambda^{a_\varepsilon}\varepsilon, \lambda^{a_h}h) &= \lambda^{1-a_h} m(\varepsilon,h) \qquad (3.35) \\
& \qquad (3.36)
\end{aligned}$$

Here m is the magnetization or order parameter.
Figure of order general parameter

Setting $h = 0$ in equation 3.36

$$\begin{aligned} m(\lambda^{a_\varepsilon}\varepsilon) &= \lambda^{1-a_h} m(\varepsilon) \\ m(1) &= \lambda^{1-a_h} m(\varepsilon) \\ m(1) &= \varepsilon^{-\frac{1-a_h}{a_\varepsilon}} m(\varepsilon) \\ m(\varepsilon) &= \varepsilon^{\beta} m(1) \end{aligned} \tag{3.37}$$

where

$$\beta = \frac{1-a_h}{a_\varepsilon} \tag{3.38}$$

Note that, setting $\lambda^{a_\varepsilon}\varepsilon = 1$ gives $\lambda = \varepsilon^{-1/a_\varepsilon}$.
Setting $\varepsilon = 0$ in equation 3.36

$$\begin{aligned} m(\lambda^{a_h} h) &= \lambda^{1-a_h} m(h) \\ m(1) &= h^{-\frac{1-a_h}{a_h}} m(h) \\ m(h) &= m(1) h^{\delta} \end{aligned} \tag{3.39}$$

where

$$\delta = \frac{a_h}{1-a_h} \tag{3.40}$$

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 3.34 twice with respect to ε we get the Specific Heat

$$\begin{aligned} \lambda^{2a_\varepsilon} G''(\lambda^{a_\varepsilon}\varepsilon, \lambda^{a_h} h) &= \lambda G''(\varepsilon, h) \\ \lambda^{2a_\varepsilon} \frac{\partial G(\lambda^{a_\varepsilon}\varepsilon, \lambda^{a_h} h)}{\left(\frac{1}{T_c}\right)^2 \partial T^2} &= \lambda G''(\varepsilon, h) \\ \lambda^{2a_\varepsilon} C(\lambda^{a_\varepsilon}\varepsilon, \lambda^{a_h} h) &= \lambda C(\varepsilon, h) \end{aligned} \tag{3.41}$$

We have used $\varepsilon = \frac{T}{T_c} - 1$ and $d\varepsilon = \frac{1}{T_c} dT$. Now setting $h = 0$ we get from 3.41

$$\begin{aligned} \lambda^{2a_\varepsilon} C(\lambda^{a_\varepsilon}\varepsilon) &= \lambda C(\varepsilon) \\ C(1) &= \lambda^{1-2a_\varepsilon} C(\varepsilon) \\ &= \varepsilon^{-\frac{1-2a_\varepsilon}{a_\varepsilon}} C(\varepsilon) \\ C(\varepsilon) &= \varepsilon^{-\alpha} C(1) \end{aligned} \tag{3.42}$$

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

$$\alpha = \frac{2a_\varepsilon - 1}{a_\varepsilon} \tag{3.43}$$

Again by differentiating 3.34 twice with respect to h we get the susceptibility. Then we set $h = 0$ to get only ε dependency.

$$\begin{aligned}
\lambda \frac{\partial^2 G(\varepsilon, h)}{\partial h^2} &= \lambda^{2a_h} \frac{\partial^2 G(\lambda^{a_\varepsilon}\varepsilon, \lambda^{a_h} h)}{\partial h^2} \\
\lambda \chi(\varepsilon, h) &= \lambda^{2a_h} \chi(\lambda^{a_\varepsilon}\varepsilon, \lambda^{a_h} h) \\
\chi(\varepsilon) &= \lambda^{2a_h - 1} \chi(\lambda^{a_\varepsilon}\varepsilon) \\
\chi(\varepsilon) &= \varepsilon^{-\gamma} \chi(1)
\end{aligned} \tag{3.44}$$

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

$$\gamma = \frac{2a_h - 1}{a_\varepsilon} \tag{3.45}$$

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 α

$$C \sim \varepsilon^{-\alpha} \tag{3.46}$$

The exponent that scales order-parameter is called β

$$m \sim \varepsilon^{\beta} \tag{3.47}$$

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

$$m \sim h^{\delta} \tag{3.48}$$

The exponent that scales susceptibility is called γ

$$\chi \sim \varepsilon^{-\gamma} \tag{3.49}$$

Note that these quantities only follows power law near T_c.

3.5.2 Rushbrooke Inequality

$$\alpha + 2\beta + \gamma \geq 2 \tag{3.50}$$

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

$$\alpha + \beta(1+\delta) = 2 \tag{3.51}$$

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

$$\begin{aligned}\alpha + \beta(1+\delta) &= \frac{2a_\varepsilon - 1}{a_\varepsilon} + \frac{1-a_h}{a_\varepsilon}\left(1 + \frac{a_h}{1-a_h}\right) \\ &= \frac{2a_\varepsilon - 1}{a_\varepsilon} + \frac{1}{a_\varepsilon} \\ &= \frac{2a_\varepsilon}{a_\varepsilon} \\ &= 2\end{aligned} \tag{3.52}$$

So the Griffiths Inequality is also satisfied.

3.6 Models

3.6.1 Ising model in 1D lattice

3.6.2 Ising model in 2D lattice

3.6.3 Bragg William Model

Chapter 4

Percolation Theory

Percolation theory potentially has been of great interest as it can describe many phenomena [48]. 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 disperate 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 artichect 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 $r <= p$ 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 [56, 39].

4.2 Historical Overview

In 1941, the idea of percolation was first conceived by Flory [25] 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 [14], 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 [23, 24]. 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 [30]. This work paved the way to many exact results in percolation, and also allowed the usage of powerful re-normalization group ideas [16]. Finding percolation threshold both exactly and by simulation has been an enduring subject of research in this field [49], as well as the development of algorithms such as those by Hoshen and Kopelman [29], by Leath [33], and by Newman and Ziff [42] (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 [31], Wierman [53], Bollobas and Riordan [13] etc.) Another infusion of interest in percolation came from the surgin 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, 22]. 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 [40]. 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 [46]. 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 [51]. Determining K-cores computationally is fast and they are insightful in many situations [21, 19]. 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 $c >= K$ 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 [45].

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 sits 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 [10], it is concluded that the effects like bootstrap percolation may occur in some real magnets [17].

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 [35]. 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" [43]. Various percolation processes have also been studied in several different types of multi-layer networks (e.g. multiplex networks and interdependent networks) [11, 32]. 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* [27, 44, 12], 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* [26, 38, 55]; 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 [34].

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.

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. There simulations provide very precise values of the critical thresholds [36]. They calculated Fisher exponent, τ, the finite-size correction exponent, Ω and the scaling function exponent, σ confirmed to be universal. They also did percolation on the HCP (hexagonal closed packed) lattice [37]. Before that in 1981 JC Wireman studied bond percolation on honeycomb and triangular lattices [54].

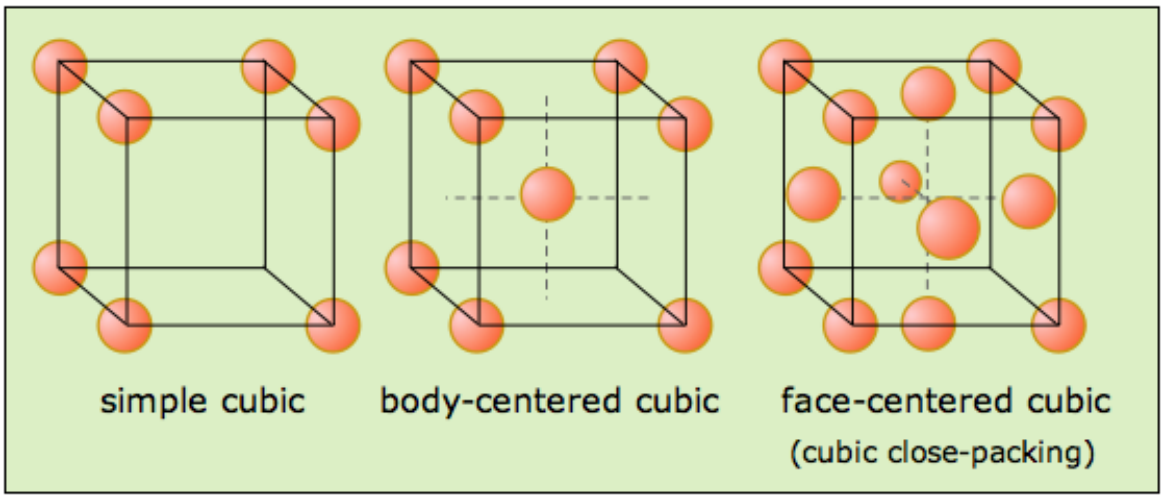
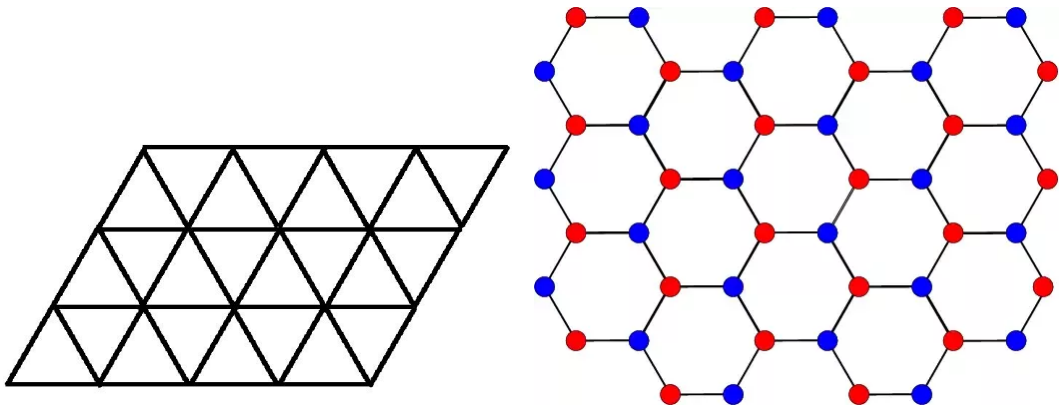


Fig. 4.2 Different types of cubic lattice [2]



(a) Triangular Lattice [2] (b) Honeycomb Lattice [4]

Fig. 4.3 Different types of cubic lattice

Furthermore, an exact solution on Bethe lattice for high density percolation was done by G. R. Reich and P. L. Leath in 1978 [47] and J. Chalupa et al. did bootstrap percolation on Bethe lattice in the following year [17].

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 [15]. Clique percolation is also studied on random networks [20]. A critical phenomenon analysis on random networks for core percolation is done by Bauer [9]. 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 [18]. N. Schwartz et al. worked on percolation problem in directed scale free networks [50]. 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

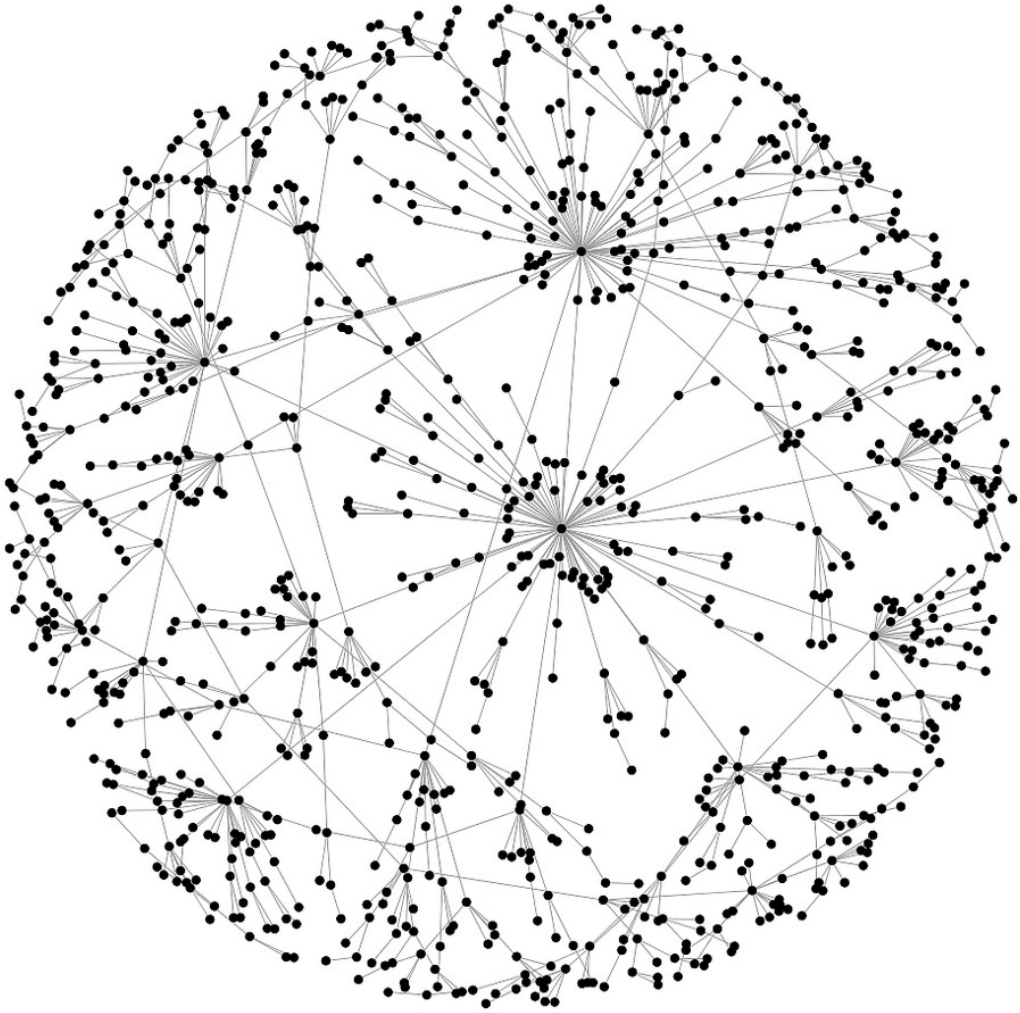


Fig. 4.4 Scale Free Network [5]

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 [41, 42].

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 [46]. 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.

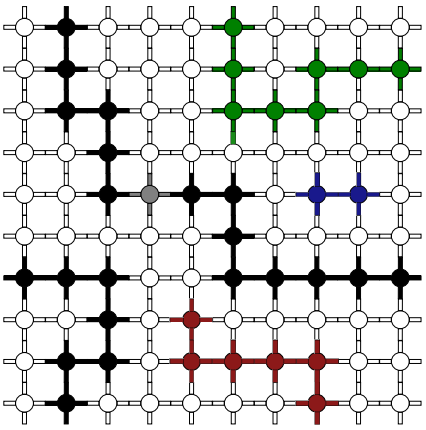
4.4.3 Spanning cluster

4.5 Observable Quantities

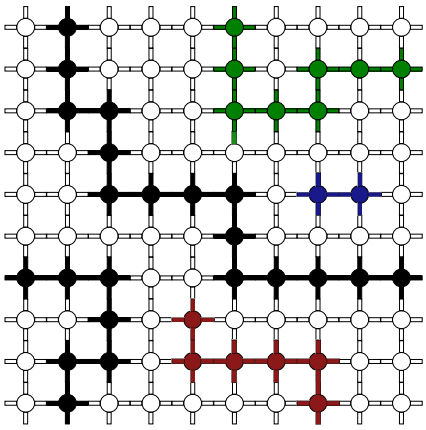
In percolation we study formation of clusters, their properties such as how they are distributed as a function of control parameter p. A cluster is a group of occupied bonds (sites) in site (bond) percolation [46] With no gap or unoccupied object between them. In a network cluster is defined as a group 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.



(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

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

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

$$p_{c_avg} = \lim_{M\to\infty} \frac{1}{M} \sum_i p_{c_i} \tag{4.1}$$

But performing infinite number of experiment is not possible. So we use another approach which is described in section 4.5.2 and 5.3.1

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

The best quantity for finding the critical exponent ν 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 $p = p_{c_i}$ then it means that there exists a spanning cluster for all $p_{c_i} \leq p \leq 1$. 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 Δp. 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 Δp chosen as per convenience [46]. 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 $L \to \infty$ 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 $(p - p_c)L^{1/\nu}$ we would get a perfect data collapse for a specific $1/nu$ that is our exponent. This process is discussed further in section 5.3.1 and 5.3.2.

4.5.3 Entropy, $H(p,L)$

Entropy in one of the key feature of a phase transition model. But thermodynamic entropy cannot be calculated for any model 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 information entropy was introduced by Claude Shannon in his 1948 paper "A Mathematical Theory of Communication" [52]. The Defition is

$$H = -\sum_i \mu_i \log \mu_i \tag{4.2}$$

where, μ_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 $n_1, n_2, \ldots, n_{12}$ 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_1' 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

$$\Omega = \frac{M!}{(M_{\mu_1})!\,(M_{\mu_2})!\ldots(M_{\mu_m})!} \tag{4.3}$$

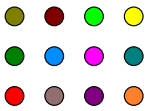
since $n_i' \approx M_{\mu_i}$. Taking log on both side of the above equation and using the String's approximation $\log(M!) = M\log M - M$ for very large M we get

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

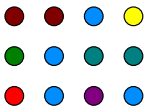
Finally we have,

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

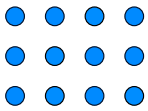
where $H(\mu)$ is nothing but the Shannon entropy $H(p)$ and clearly it is the degree of uncertainty per attempt. Total entropy or information is therefore equal to NH if we consider $M = N$. In the case when each site has distinct color then each site will have the equal chance of being picked. It means $\mu_i = 1/N \forall i$ 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 $S = 0$.



(a) Entropy of this system is in minimum



(b) Entropy of this system is in between maximum and minimum



(c) All distinct cluster. A system of maximum entropy

Fig. 4.6 A system of 12 cluster

Thus percolation is indeed an order-disorder transition where disorder is equivalent to degree of confusion [46].

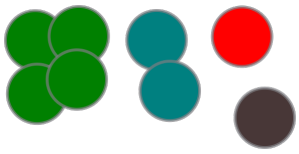
Say we have a coin with a head and a tail. If the coin is unbiased then the probability of getting the head or the tail is 50% or 0.5. Now what's the entropy of this system? The Shanon entropy is the one that can be used here. For convenience $\log_2$ will be used for evaluating logarithms here, after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

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

(a) Free particles. $H = 3$



(b) Some Free and Some Bound Particles. $0 < H < 3$

(c) Bount particles. $H = 0$

Fig. 4.7 entropy of a system of 8 particles

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 3.10 we can find specific heat if we know the temperature and entropy. In percolation theory we measure the Shannon entropy, $H(p,L)$. And we use $(1-p)$ as the analogue of temperature which gives

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

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

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

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

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

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

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

Both definition, though looks a bit different when plotted, gives the same critical exponent β. How to find this exponent is shown in section 5.3.4.

Mathematically

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

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

$$\sum_i k_i = L^2 \tag{4.11}$$

and without periodic boundary condition

$$\sum_i k_i = L(L-1) \tag{4.12}$$

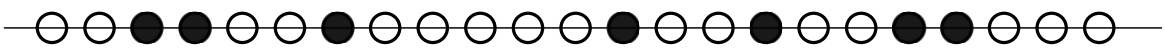


Fig. 4.8 One Dimensional Lattice. Empty ones are white and filled ones are black.

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

4.5.7 Mean Cluster Size, S

4.5.8 Cluster Size Distribution Function, n_s

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

4.5.9 Correlation Function, $g(r)$

4.5.10 Correlation length, ξ

4.5.11 Fractal Dimension, d_f

4.6 Exact Solutions

Percolation problem can be solved exactly in 1 and ∞ 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 ∞ 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 ∞ 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.

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

For $p = 1$ all sites of the lattice are occupied and a percolating cluster spans from $-\infty$ to ∞ so that each and every and every sites of the lattice belong to the percolating cluster.For $p < 1$ we will have on the average $(1-p)^L$ empty sites. So if $L \to \infty$, we have $(1-p)^L \to const.$ 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

$$p_c = 1 \tag{4.15}$$

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 are 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

$$n_s = p^s(1-p)^s \tag{4.16}$$

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

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

where ξ is the correlation length and defined as

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

in the limit $p \to p_c$ 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.

$$\sum_{s=1}^{\infty} s n_s = \frac{number of total occupied sites}{number of total lattice sites} = p \tag{4.19}$$

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

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

Here we have used the series sum 4.21.

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

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

$$w_s = \frac{n_s s}{\sum_s n_s s} \tag{4.22}$$

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

$$S = \sum_s w_s s \tag{4.23}$$

This equation is very much similar to

$$\bar{x} = \int x p(x) dx \tag{4.24}$$

using equation 4.22 we get

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

we can write

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

using the fact that $p_c = 1$ in 1D lattice. This equation reveals that the mean cluster size diverges for $p \to p_c$ 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 [28]

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

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

$$g(r) = p^r \tag{4.28}$$

It can also be expressed in terms of correlation length ξ

$$g(r) = \exp(\ln(p^r)) = \exp(-r/\xi) \tag{4.29}$$

where ξ is the correlation length 4.18.
Now that we have correlation function, we can define mean cluster size in terms of it

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

At this point it is evident that the cutoff cluster size s_ξ , mean cluster size $S(p)$, and correlation length ξ 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

$$A \propto V^{1-1/d} \tag{4.31}$$

now as $d \to \infty$ we have

$$A \propto V \tag{4.32}$$

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

$$1 + 3 \times (1 + 2 + \ldots + 2^{r-1}) = 3.2^r - 2 \tag{4.33}$$

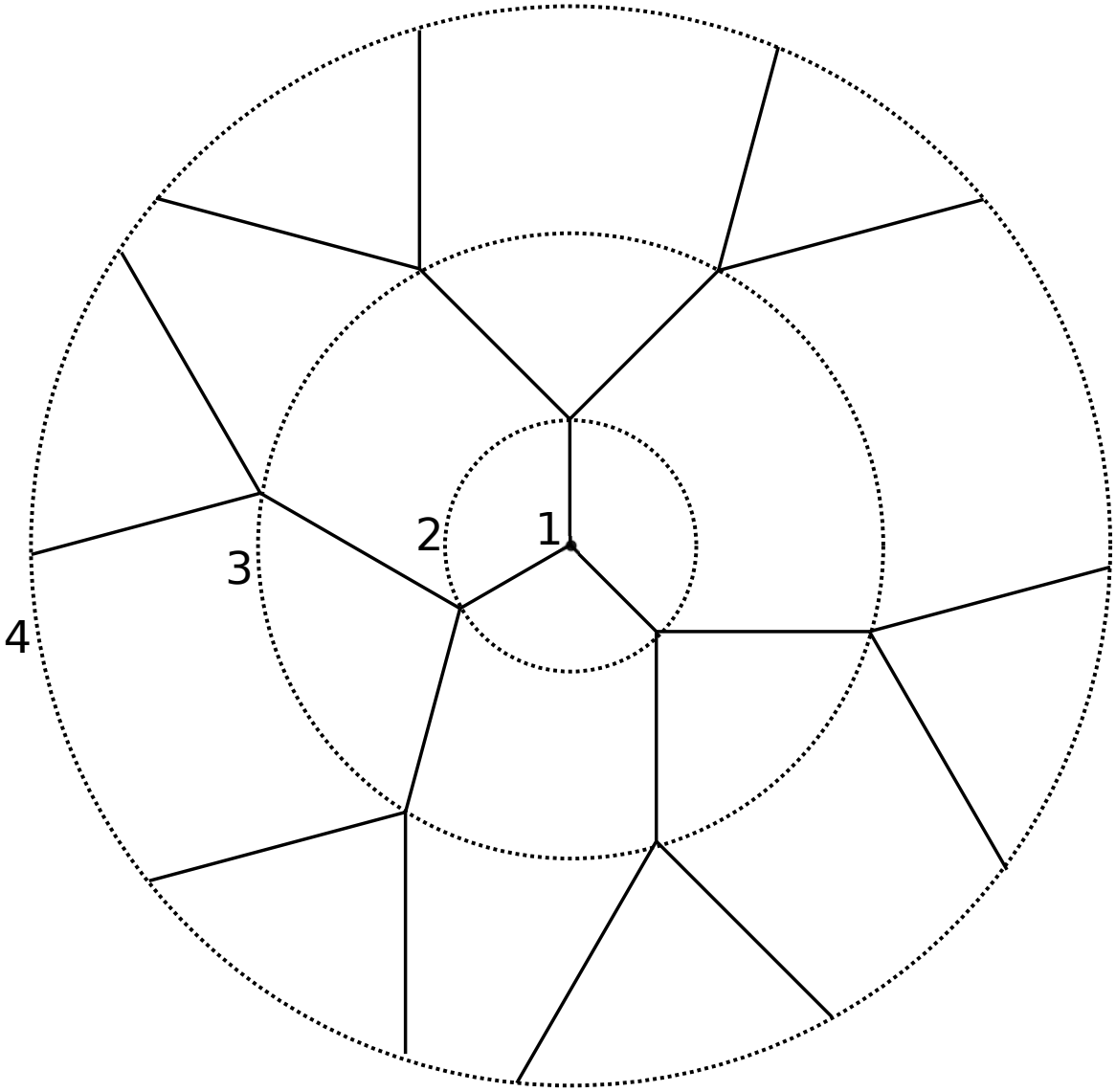


Fig. 4.9 Bethe Lattice for $z = 3$

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

$$1+2+2^2+2^3+\ldots+2^r = 2^{r+1}-1 \tag{4.34}$$

And if we measure the surface to volume ratio we get

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

as $r \to \infty$ we get

$$\frac{A}{V} \sim \frac{3 \times 2^{r-1}}{3 \times 2^r} = \frac{1}{2} = constant \tag{4.36}$$

Therefore Bethe lattice is indeed an infinite dimensional lattice.

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 $r \to \infty$ 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

$$\begin{aligned} (z-1)p_c &= 1 \\ p_c &= \frac{1}{z-1} \end{aligned} \tag{4.37}$$

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

$$P(p) = p(1-Q^3) \tag{4.38}$$

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,

$$Q = (1-p) + pQ^2 \tag{4.39}$$

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

$$Q = 1, \frac{1-p}{p} \tag{4.40}$$

For z neighbors, in general we have

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

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 $p = p_c = 1/2$

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

Let,

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

Then

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

$$P(p) = P(p_c) + (p - p_c)P'(p_c) + \dots \tag{4.45}$$
$$= 0 + (p - p_c)\left(1 - f(p_c) - pf'(p_c)\right) + \dots \tag{4.46}$$
$$= 6(p - p_c) + \dots \tag{4.47}$$
$$\tag{4.48}$$

Therefore we get

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

the critical exponent β is defined by

$$P(p) \propto (p - p_c)^\beta \tag{4.50}$$

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,

$$T = (1 - p) \times 0 + p(1 + 2T) \tag{4.51}$$

Solving this we get

$$T = \frac{p}{1 - 2p} \tag{4.52}$$

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

$$S(p) = 1 + 3T \tag{4.53}$$
$$= \frac{1 + p}{1 - 2p}$$
$$= \frac{1 + p}{2(p_c - p}$$
$$= \frac{1 + p}{2}(p_c - p)^{-1} \tag{4.54}$$

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 $p \to p_c$.

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

$$g(r) = z(z-1)^{r-1}p^r \tag{4.55}$$
$$= \frac{z}{z-1}\left(p(z-1)\right)^r \tag{4.56}$$
$$= \frac{z}{z-1}\exp\left[\log\left[p(z-1)\right]\right] \tag{4.57}$$

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 $\to \infty$. 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

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

as p approaches p_c, that is $\nu = 1$.
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 [29], 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 very in each realization. However, the expected or ensemble average over M experiments will give $n = pM$ in the limit $M \rightarrow$. Thus the number of occupied bonds or sites is also a measure of p. Using this idea Ziff and Newman [42] 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

Definitions of some quantities used in percolation. Basic Algorithm

4.9.1 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 $L = \{L_1, L_2, \ldots L_n\}$, where n is a finite number and $L_1 < L_2 < \ldots < L_n$), and extrapolate the results for infinite lattice. The visual structure of the square lattice is as follows 4.10. This is and empty lattice structure. Filled circles are for occupied site and filled bonds are for occupied bonds 4.11.

4.9.2 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.

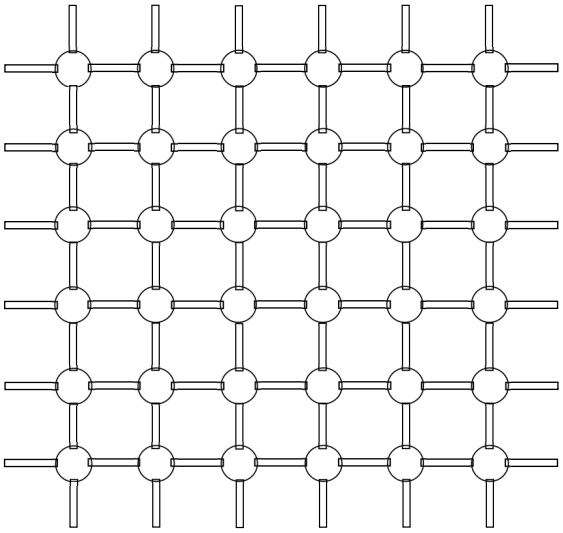
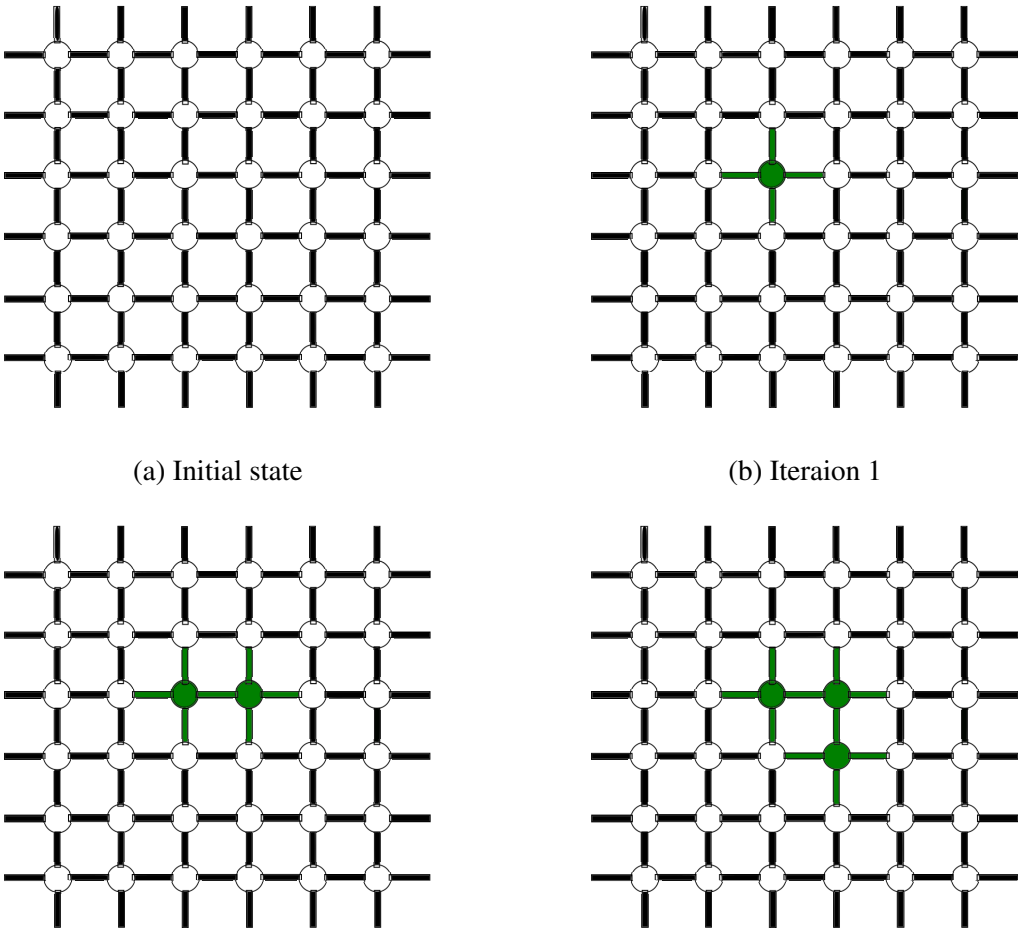


Fig. 4.10 Square Lattie (empty) of length 6

Fig. 4.11 Site and Bond symbol (empty and occupied).



(a) Initial state

(b) Iteraion 1

(c) Iteraion 2

(d) Iteraion 3

Fig. 4.12 Growth of a Cluster in Site Percolation on square Lattice

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

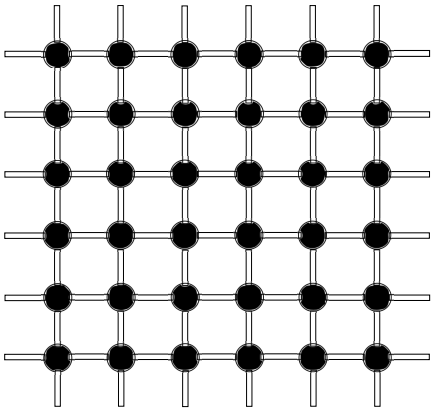
The formation of cluster is visualized in the figure 4.12.

4.9.3 Bond Percolation

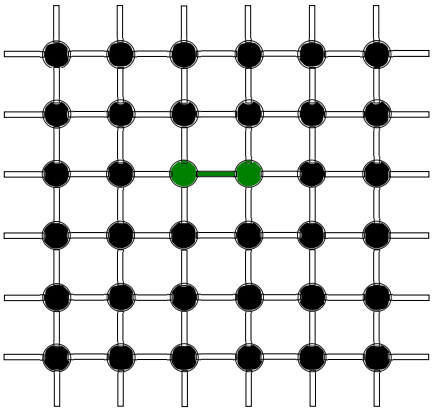
The algorithm for bond percolation is as follows,

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.
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

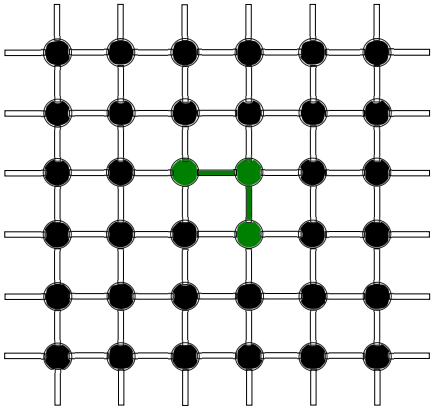
The formation of cluster is visualized in the figure 4.13.



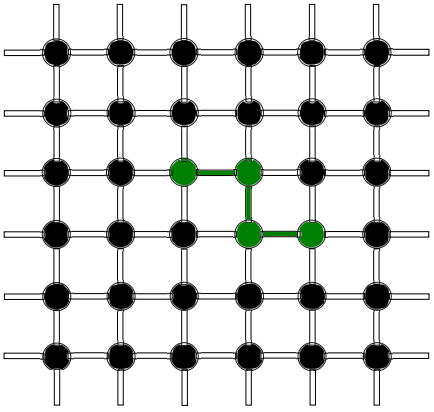
(a) Initial state



(b) Iteraion 1



(c) Iteraion 2



(d) Iteraion 3

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

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 α, β, γ, ν 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 τ 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 exhist 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 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 neighour site along the same line to make room for itself. However, the trial attempt of pushing the neighbour is successful if the next neibhouring 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 $i = 1, 2, ..., L$, the second row again from left to right as $i = L+1, L+2, ..., 2L$ and we continue this till we reach the bottom row which we label as $i = (L-1)L+1, ..., L^2$. Then at each step we pick a discrete random number R from $1, 2, ..., L^2-1, L^2$ 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 successul 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 ocupied too but the next nearest neighbour site is empty then the neighbour move to the empty site to make space for the incoming perticle to be deposited there.

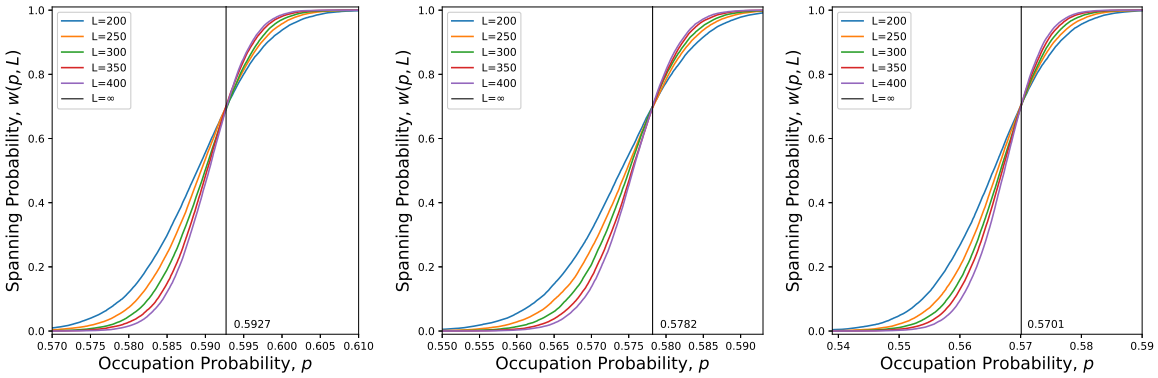
1. take a square lattice of length L.
2. choose a site randomly
3. if the chosen site is empty then occupy it else choose one of the four neighbor randomly
4. if the chosen neighbor is empty then occupy it else choose second nearest neighbor in that direction
5. if the second nearest neighbor is empty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

5.2 Finite Size Scaling in Percolation Theory

5.3 Finding Numerical Values

5.3.1 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 $(L0,L1,L2)$ for different lengths. We have found that p_c is $0.5927, 0.5782, 0.5701$ for $L0,L1,L2$ respectively.Thus for long range interaction p_c is smaller than for short range interaction. The quantity $w(p,L)$ is called 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,



(a) L0 (b) L1 (c) L2

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

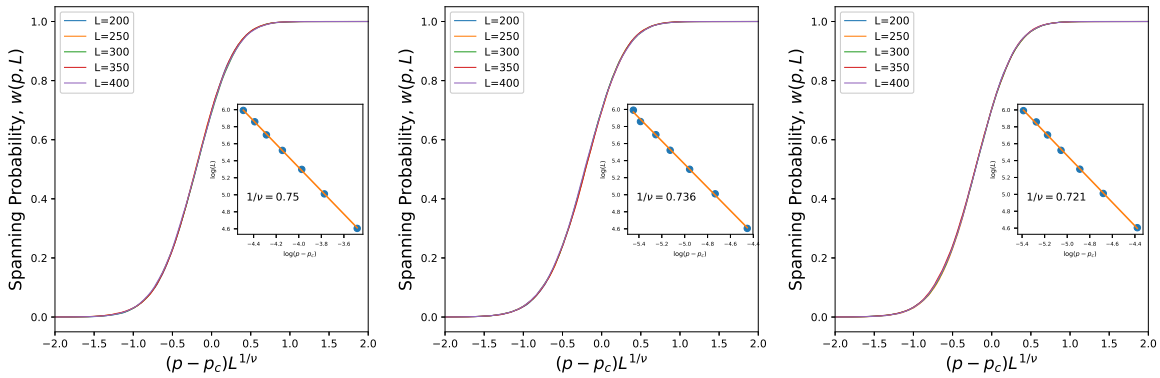
instead it is a range of values that contains the p_c. For example, say we have a lattice of length 200 and for that we will have different p_c at each realization. After an infinite number of experimentation we will have to take an average and that will give the exact value of p_c. But since it is practically 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.

5.3.2 Spanning Probability and finding $1/\nu$

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

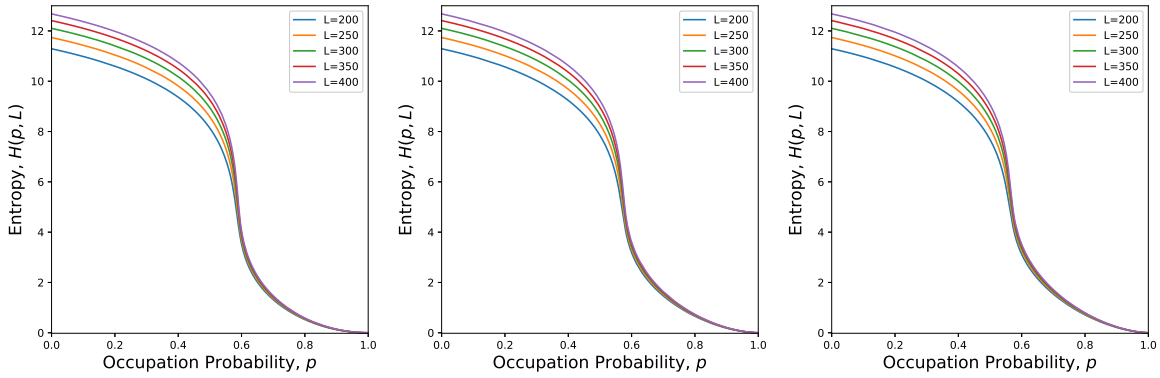
$$w = (p - p_c)L^{1/\nu} \tag{5.1}$$

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 **??**.



(a) L0 (b) L1 (c) L2

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

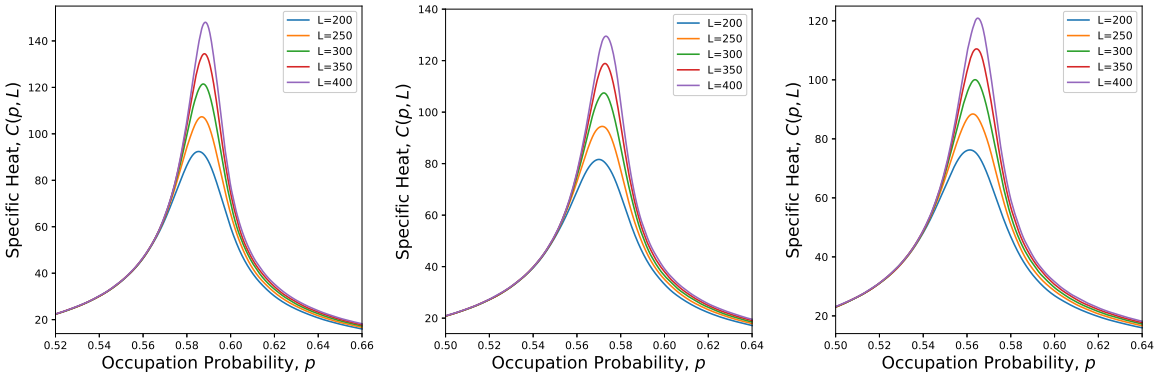


(a) L0 (b) L1 (c) L2

Fig. 5.3 Entropy, $H(p,L)$ vs Occupation Probability, p

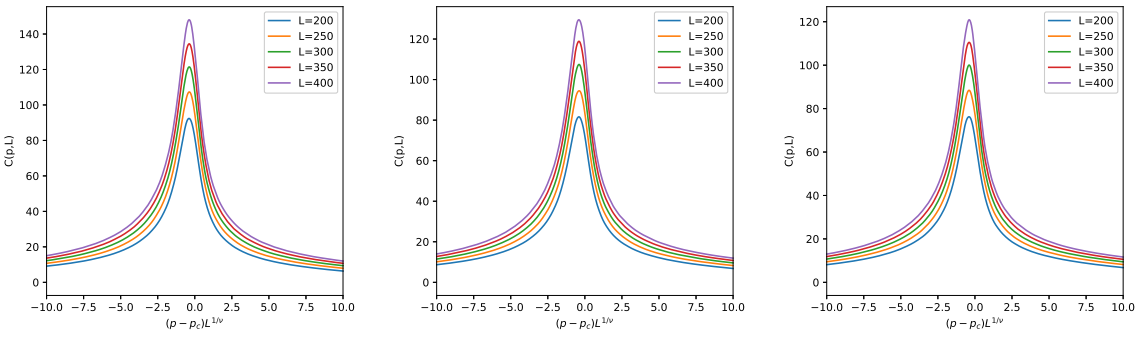
5.3.3 Entropy, Specific Heat and finding α

For any phase transition model the entropy is crucial. In percolation theory we use Shannon Entropy [52]. 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 α. To do this first we need to scale the x-values of the specific heat data using the exponent $1/\nu$ obtained from 5.3.2 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 absolute value of the graph will give α/ν and from that we can find the exponent α simply by dividing α/ν by $1/\nu$. We get α 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



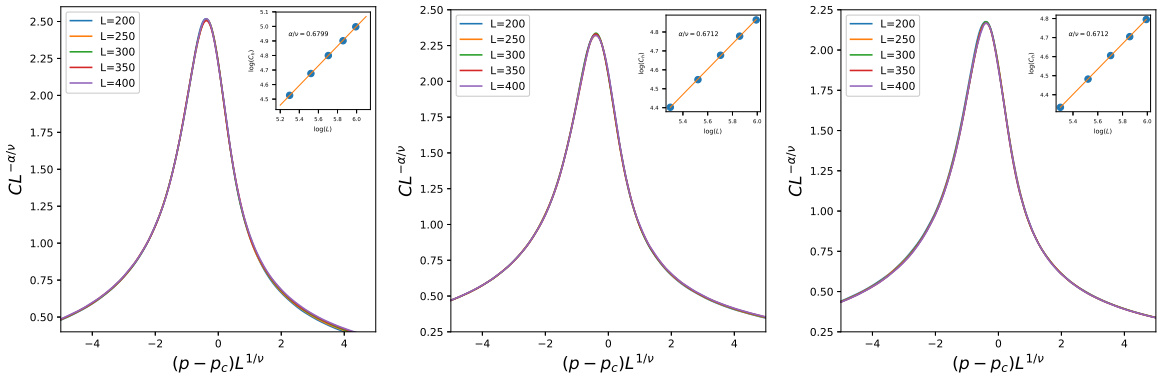
(a) L0 (b) L1 (c) L2

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



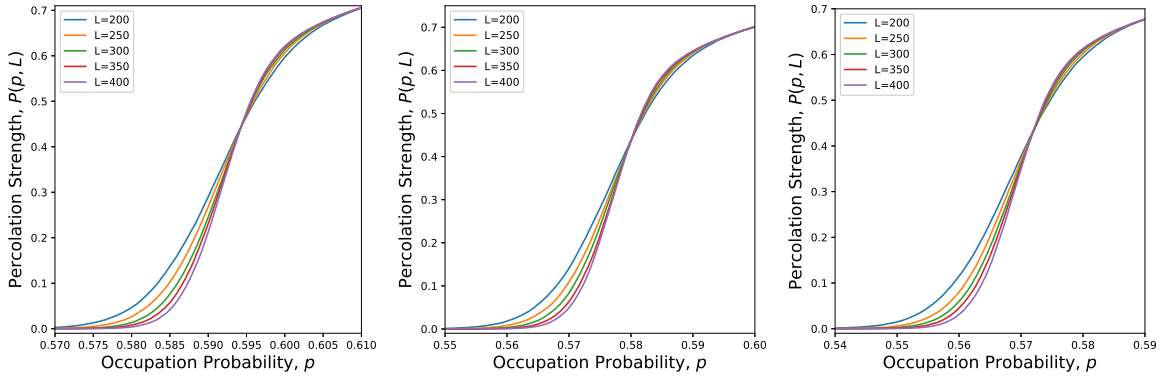
(a) L0 (b) L1 (c) L2

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



(a) L0 (b) L1 (c) L2

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



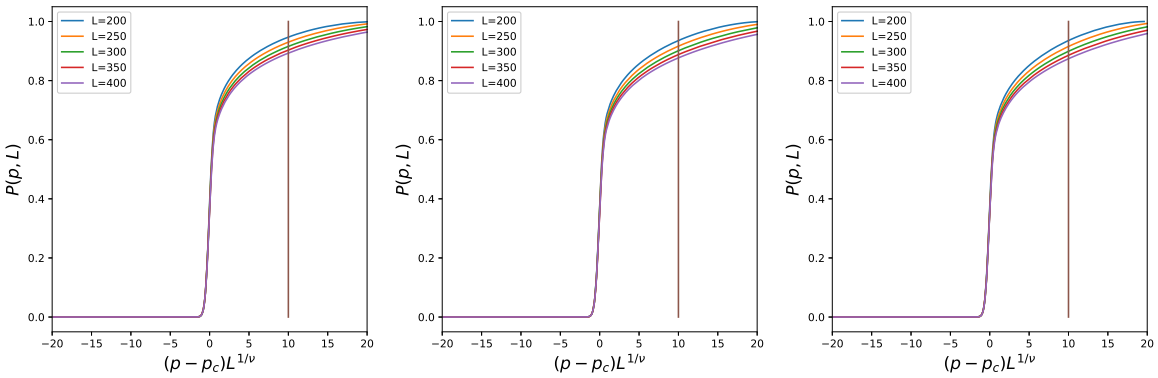
(a) L0 (b) L1 (c) L2

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

$(p-p_c)L^{1/\nu}$ we get perfect data collapse for $L0, L1, L2$ and it is shown in figure 5.6

5.3.4 Order Parameter and finding β

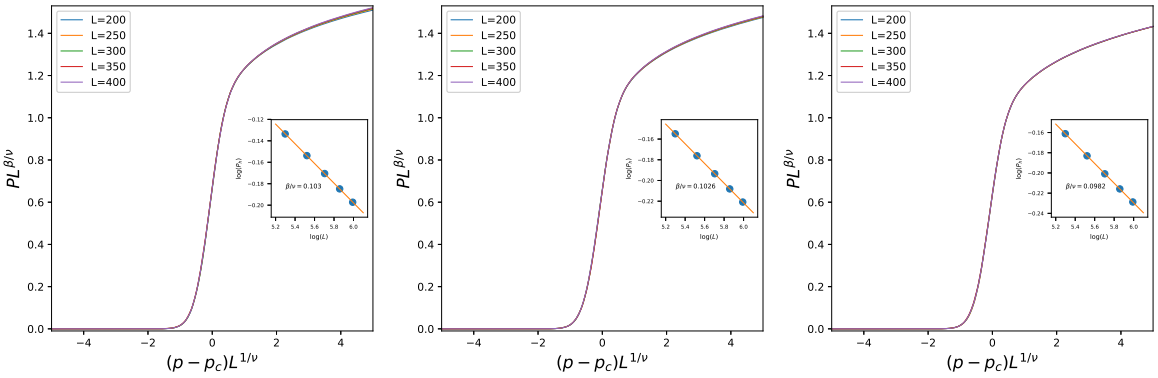
Order parameter, also knows as the percolation strength, is ,along with entropy, an important quantity in the study of phase transition. It is denoted as $P(p,L)$. Using the definition **??** we obtain the order parameter for our system and it is shown in the figure 5.7. Since using spanning cluster 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 section 5.3.2 we scale the x-values as $(p-p_c)L^{1/\nu}$ 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



(a) L0 (b) L1 (c) L2

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

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



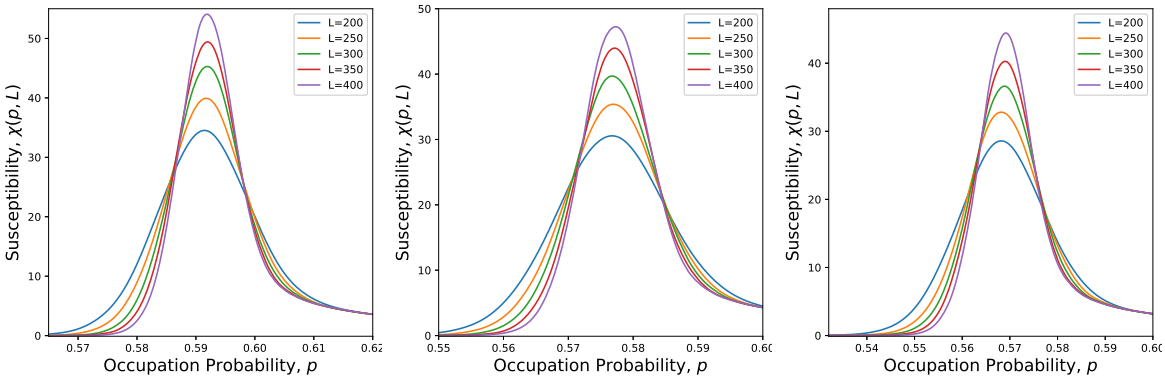
(a) L0 (b) L1 (c) L2

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

get the exponent β/ν from it's slope and obtain exponent β by dividing β/ν by $1/\nu$. Using the FSS hypothesis 2.4.2 we plot $PL^{\beta/\nu}$ versus $(p-p_c)L^1/\nu$ and get a good data collapse which is shown in figure 5.10.

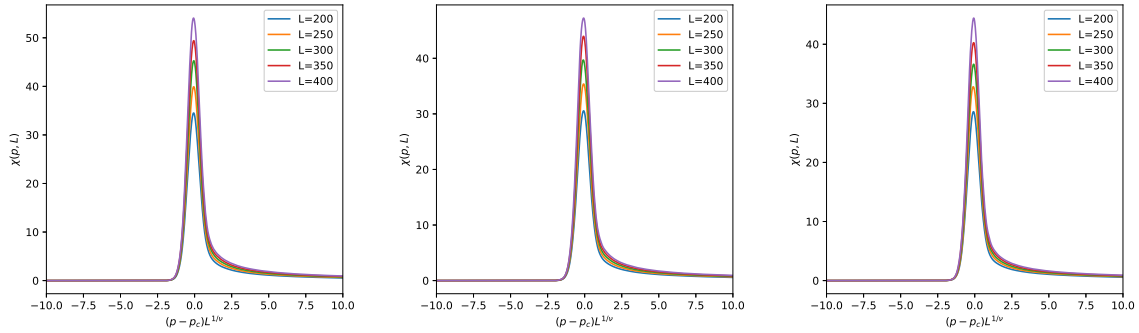
5.3.5 Susceptibility and finding γ

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 χ vs $(p-p_c)L^{1/\nu}$ we get all the peak point aligned (figure 5.12). Note that the value of $1/\nu$ is known from section 5.3.2. Then we take the reading of the height of each line and call it χ_h. Since each line represents a different lattice size, plotting $\log(\chi_h)$ vs $\log(L)$ gives the slope γ/ν. And using the FSS hypothesis



(a) L0 (b) L1 (c) L2

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



(a) L0 (b) L1 (c) L2

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

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 γ to be $0.8543, 0.8542, 0.882$.

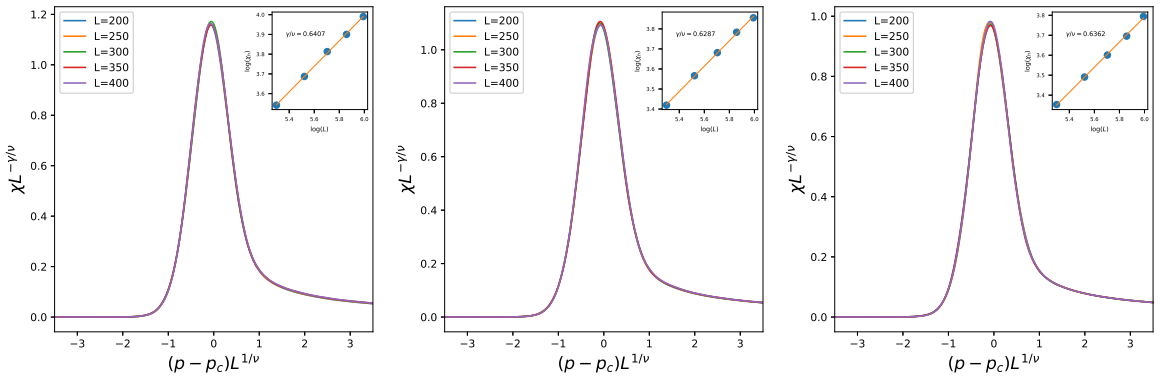
5.3.6 Cluster Size Distribution

n_s vs s graph

5.3.7 Cluster Rank Size Distribution

5.3.8 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



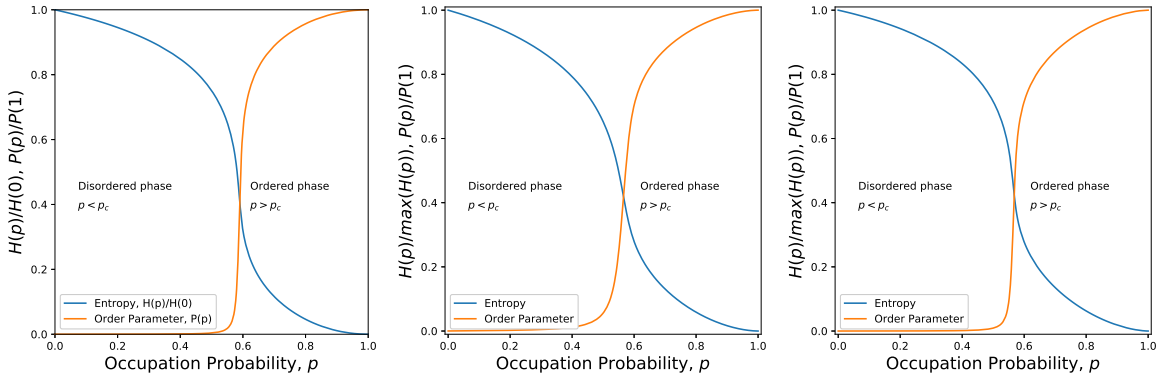
(a) L0 (b) L1 (c) L2

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

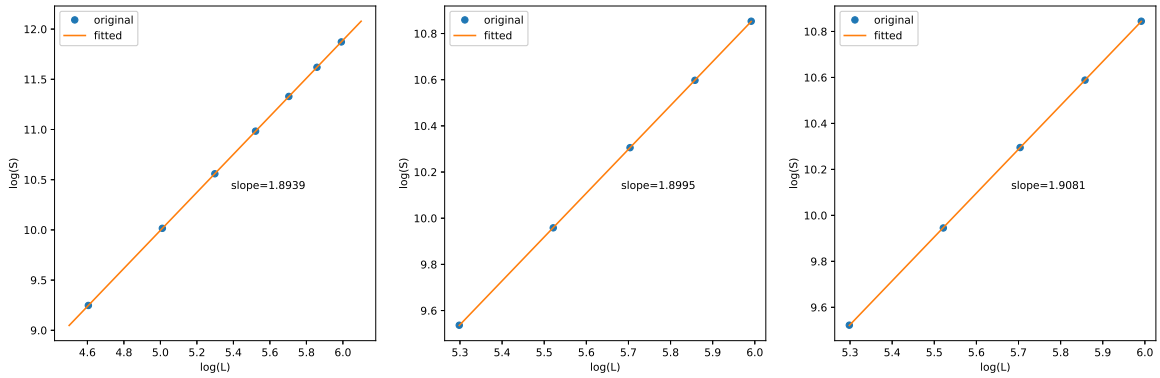
are completely different. It's astonishing how the behavior changes. In percolation theory, this order disorder transition is different than in thermodynamics. Here disordered means uncertainty, since we are dealing with a system where probability is the control parameter (the occupation probability p). When p is minimum all clusters are disconnected and have size of unity. This means that we can to 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 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).

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



(a) L0 (b) L1 (c) L2

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



(a) L0 (b) L1 (c) L2

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

5.3.9 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

$$S \sim L^{d_f} \tag{5.2}$$

taking log we get

$$\log(S) = d_f \log(L) \tag{5.3}$$

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.

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 α, β, γ, ν 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 τ 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 exhist 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	$1/\nu$	α/ν	β/ν	γ/ν
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	$1/\nu$	α	β	γ	$\alpha+2\beta+\gamma$	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 change 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. 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.

[9] 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.

[10] M. Blume. Theory of the first-order magnetic phase change in UO2. *Physical Review*, 141(2):517–524, jan 1966. doi: 10.1103/physrev.141.517. URL https://doi.org/10.1103/physrev.141.517.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] John Cardy. *Scaling and Renormalization in Statistical Physics*. Cambridge University Press, 1996. doi: 10.1017/cbo9781316036440. URL https://doi.org/10.1017/cbo9781316036440.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Peter Hall. On continuum percolation. *Institute of Mathematical Statistics*, 13(4): 1250–1266, 1985. doi: 110.2307/2244176.

[28] M. K. Hassan. Lecture notes on statistical mechanics of non-equilibrium phenomena.

[29] 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.

[30] 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 Koyto., p.11*, 26:11, 1969. URL http://adsabs.harvard.edu/abs/1969PSJJS..26...11K. Provided by the SAO/NASA Astrophysics Data System.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0910321647.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] M. S. Rahman and M. K. Hassan. Redefinition of site percolation in light of entropy and the second law of thermodynamics, 2018.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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

A.2.1 Index

The **index.h** file

```
#ifndef SITEPERCOLATION_INDEX_H
#define SITEPERCOLATION_INDEX_H

#include <ostream>
#include <iostream>
#include <sstream>
#include <vector>

#include "../types.h"
#include "../exception/exceptions.h"
#include "../lattice/bond_type.h"


struct Index{
value_type row_{};
value_type column_{};

~Index()                        = default;
Index()                         = default;

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

```
22
23  };
24
25
26  class IndexRelative{
27  public:
28  int x_{};
29  int y_{};
30
31  ~IndexRelative()                        = default;
32  IndexRelative()                         = default;
33
34  IndexRelative(int x, int y) : x_{x}, y_{y} {}
35
36  };
37
38
39  struct BondIndex{
40  BondType bondType;
41
42  value_type row_;
43  value_type column_;
44
45  ~BondIndex()                           = default;
46  BondIndex()                            = default;
47
48  BondIndex(BondType hv, value_type row, value_type column)
49  :  row_{row}, column_{column}
50  {
51  bondType = hv;
52  }
53
54  bool horizontal() const { return bondType == BondType::Horizontal;}
55  bool vertical() const { return bondType == BondType::Vertical;}
56
57  };
58
59  std::ostream& operator<<(std::ostream& os, const Index& index);
60  bool operator==(const Index& index1, const Index& index2);
61  bool operator<(const Index& index1, const Index& index2);
62
63  std::ostream& operator<<(std::ostream& os, const IndexRelative& index)
      ;
64
```

```
65  std::ostream& operator<<(std::ostream& os, const BondIndex& index);
66  bool operator==(BondIndex index1, BondIndex index2);
67  bool operator<(BondIndex index1, BondIndex index2);
68
69
70  /**
71  *  Get 2nd nearest neightbor / sin the direction of 1st nearest
     neighbor, while @var center is the center
72  */
73  Index get_2nn_in_1nn_direction(Index center, Index nn_1, value_type
     length);
74  std::vector<Index> get_2nn_s_in_1nn_s_direction(Index center, const
     std::vector<Index> &nn_1, value_type length);
75
76  #endif //SITEPERCOLATION_INDEX_H
77
78
```

The **index.cpp** file

```
1   #include <iomanip>
2   #include "index.h"
3
4   using namespace std;
5
6   ostream& operator<<(ostream& os, const Index& index)
7   {
8   return os <<'(' << index.row_ << ',' << index.column_ << ')';
9   }
10
11  ostream& operator<<(ostream& os, const IndexRelative& index)
12  {
13  return os << '(' << std::setw(3) << index.x_ << ',' << std::setw(3) <<
      index.y_ << ')';
14  }
15
16  bool operator==(const Index& index1, const Index& index2){
17  return (index1.row_ == index2.row_) && (index1.column_ == index2.
     column_);
18  }
19
20  bool operator<(const Index& index1, const Index& index2){
21  if(index1.row_ < index2.row_)
22  return true;
23  if(index1.row_ == index2.row_){
```

```
24  return index1.column_ < index2.column_;
25  }
26  return false;
27  }
28
29
30  ostream& operator<<(ostream& os, const BondIndex& index){
31  if(index.horizontal()){
32  // horizontal bond
33  os << "<->" ;
34  }
35  if (index.vertical()){
36  // vertical bond
37  os << "<|>" ;
38  }
39  return os << '(' << index.row_ << ',' << index.column_ << ')';
40  }
41
42  bool operator==(BondIndex index1, BondIndex index2){
43  if(index1.horizontal() ==  index2.horizontal() || index1.vertical() ==
        index2.vertical()){
44  // horizontal or vertical
45  return index1.row_ == index2.row_ && index1.column_ == index2.column_;
46  }
47  return false;
48  }
49
50
51  bool operator<(BondIndex index1, BondIndex index2){
52  cout << "not yet defined : line " << __LINE__ << endl;
53  return false;
54  }
55
56
57  /**
58  * Get the 2nd nearest nearest neighbor in the direction of 1st nearest
       neighbor.
59  * Periodicity is not considered here.
60  * @param center
61  * @param nn_1
62  * @param length
63  * @return
64  */
```

```
65  Index get_2nn_in_1nn_direction(Index center, Index nn_1, value_type
     length){
66  int delta_c = int(nn_1.column_) - int(center.column_);
67  int delta_r = int(nn_1.row_) - int(center.row_);
68  if (delta_c == 0 && delta_r == 0){
69  cout << "Both indices are same : line " << __LINE__ << endl;
70  }
71  else if(delta_c > 1 || delta_r > 1){
72  // meaning, the sites are on the opposite edges
73  //        cout << "2nd index is not the First nearest neighbor : line
     " << __LINE__ << " : file " << __FILE__ << endl;
74  }
75
76  return Index{(nn_1.row_ + delta_r + length) % length, (nn_1.column_ +
     delta_c + length) % length};
77  }
78
79
80  /**
81  * Get all second nearest neighbors based on the first nearest
     neighbors.
82  * Periodicity is not considered here
83  * @param center
84  * @param nn_1
85  * @param length
86  * @return
87  */
88  vector<Index> get_2nn_s_in_1nn_s_direction(Index center, const vector<
     Index> &nn_1, value_type length){
89  vector<Index> nn_2(nn_1.size());
90
91  for(size_t i{}; i != nn_1.size() ; ++i){
92  int delta_c = int(nn_1[i].column_) - int(center.column_);
93  int delta_r = int(nn_1[i].row_) - int(center.row_);
94  if (delta_c == 0 && delta_r == 0){
95  cout << "Both indices are same : line " << __LINE__ << endl;
96  }
97  else if(delta_c > 1 || delta_r > 1){
98  // meaning, the sites are on the opposite edges
99  //            cout << "center " << center << " nn " << nn_1 << endl;
100 //            cout << "2nd index is not the First nearest neighbor :
     line " << __LINE__ << " : file " << __FILE__ << endl;
101 }
102
```

```
103 nn_2[i] =  Index{(nn_1[i].row_ + delta_r + length) % length, (nn_1[i].
     column_ + delta_c + length) % length};
104 }
105 return nn_2;
106 }
107
108
```

A.2.2 Site and Bond

The **site.h** file

```
1
2  #ifndef SITEPERCOLATION_SITE_H
3  #define SITEPERCOLATION_SITE_H
4
5  #include <array>
6  #include <set>
7  #include <vector>
8  #include <iostream>
9  #include <memory>
10
11 #include "../index/index.h"
12 #include "../types.h"
13
14
15 /**
16 * single Site of a lattice
17 */
18 struct Site{
19
20 /**
21 * if true -> site is placed.
22 * if false -> the (empty) position is there but the site is not (
     required for site percolation)
23 */
24 bool _status{false};
25 int _group_id{-1};
26 Index _id{};
27 value_type _weight{};    // when one site is occupied multiple times
     only weight increases by occupation
28
29 //relative distance from the root site. {0,0} if it is the root site
30 //very useful for detecting wrapping // todo
```

```
31  IndexRelative _relative_index{0,0};
32
33  //     std::vector<Index> _neighbor{};
34  // _connection[0] -> _connection between "set_ID" and "_neighbor[0]"
35
36  /**
37  * using set gives following advantages
38  * 1. Keeps elements sorted
39  * 2. automatically manage repeated value
40  */
41  //     std::set<int> _connection{};
42
43  public:
44
45  ~Site()                 = default;
46  Site()                  = default;
47  Site(const Site&)             = default;
48  Site(Site&&)            = default;
49  Site& operator=(const Site&)  = default;
50  Site& operator=(Site&&) = default;
51
52  Site(Index id, value_type length){
53  // I have handle _neighbor or corner points and edge points carefully
54  if(id.row_ >= length || id.column_ >= length){
55  std::cout << "out of range : line " << __LINE__ << std::endl;
56  }
57  _id.row_ = id.row_;
58  _id.column_ = id.column_;
59  //         _neighbor = Lattice_Helper::get_neighbor(id, length);
60  }
61
62
63  bool isActive() const { return _status;}
64  void activate(){ _status = true;}
65  void deactivate() {
66  _relative_index = {0,0};
67  _group_id = -1;
68  _status = false;
69  }
70  Index ID() const { return  _id;}
71  /*
72  * Group get_ID is the set_ID of the cluster they are in
73  */
74  int      get_groupID() const {return _group_id;}
```

```
75  void     set_groupID(int g_id) {_group_id = g_id;}
76
77  std::stringstream getSite() const {
78  std::stringstream ss;
79  if(isActive())
80  ss << _id;
81  else
82  ss << "(*)";
83  return ss;
84  }
85
86  value_type weight() const {return _weight;}
87  void increaseWeight(){_weight += 1;}
88
89  void relativeIndex(IndexRelative r){
90  _relative_index = r;
91  }
92
93  void relativeIndex(int x, int y){
94  _relative_index = {x,y};
95  }
96
97  IndexRelative relativeIndex() const {return _relative_index;}
98
99  };
100
101 std::ostream& operator<<(std::ostream& os, const Site& site);
102 bool operator==(Site& site1, Site& site2);
103
104 #endif //SITEPERCOLATION_SITE_H
105
```

file : **site.cpp**

```
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
14    bool operator==(Site& site1, Site& site2){
15    return (site1.ID().row_ == site2.ID().row_) && (site1.ID().column_
      == site2.ID().column_);
16    }
17
18
```

file : **bond_type.h**

```
1     #ifndef PERCOLATION_BOND_V2_H
2     #define PERCOLATION_BOND_V2_H
3
4     /**
5     * Only two type bonds in 2D lattice
6     */
7     enum class BondType{
8     Horizontal,
9     Vertical
10    };
11
12    #endif //PERCOLATION_BOND_V2_H
13
```

file : **bond.h**

```
1   #ifndef SITEPERCOLATION_BOND_H
2   #define SITEPERCOLATION_BOND_H
3
4   #include <ostream>
5   #include <iostream>
6   #include <sstream>
7
8   #include "../index/index.h"
9   #include "../types.h"
10
11  /**
12  * A bond has two end
13  * say a 5x5 lattice bond between end1 (0,0) and end2 (0,1)
14  * if _status is false -> bond is not there
15  *
16  */
17  struct Bond{
18  // check if active or not
19  bool _status{false};
```

```
20  value_type _length;
21  int _group_id{-1};
22
23  BondType bondType;
24  //relative distance from the root site. {0,0} if it is the root site
25  //very useful for detecting wrapping // todo
26  IndexRelative _relative_index{0,0};
27
28
29  Index _end1;
30  Index _end2;
31
32  BondIndex _id;
33
34  ~Bond() = default;
35  Bond() = default;
36  Bond(Index end1, Index end2, value_type length){
37  _end1.row_ = end1.row_;
38  _end1.column_ = end1.column_;
39  _end2.row_ = end2.row_;
40  _end2.column_ = end2.column_;
41
42  // correct the bond here
43  _length = length;
44  // check if the bond is valid??
45  if(_end1.row_ == _end2.row_){
46  bondType = BondType::Horizontal;
47  //            horizontal = true;
48  // means x_ values are equal
49  if(_end1.column_ > _end2.column_){
50  // case (0,0)<->(0,4) for L=5 and
51  // case (0,4)<->(0,0) are equal for bond like this end1=(0,4) end2
     =(0,0)
52  // for all other cases lower index is end1
53  if(_end1.column_ == _length-1 && _end2.column_ ==0){
54  // do nothing
55  //                    std::cout << "_end1.y_ == _length-1 && _end2.y_
     ==0 : line " << __LINE__ << std::endl;
56  }
57  else{
58  // sort them out
59  _end1.column_ = end2.column_;
60  _end2.column_ = end1.column_;
61  }
```

```
62  }
63  else if(_end1.column_ < _end2.column_){
64  if(_end1.column_ == 0 && _end2.column_ == _length-1){
65  _end1.column_ = end2.column_;
66  _end2.column_ = end1.column_;
67  }
68  }
69  //            _id = set_ID<3>({0, _end1.x_, _end1.y_});
70  }
71  else if(_end1.column_ == _end2.column_){
72  bondType = BondType::Vertical;
73  //            vertical = true;
74  // means y_ values are equal
75  if(_end1.row_ > _end2.row_){
76  // case (0,0)<->(4,0) for L=5 and
77  // case (4,0)<->(0,0) are equal for bond like this end1=(4,0) end2
     =(0,0)
78  // for all other cases lower index is end1
79  if(_end1.row_ == _length-1 && _end2.row_ ==0){
80  // do nothind
81  //                    std::cout << "_end1.x_ == _length-1 && _end2.x_
     ==0 : line " << __LINE__ << std::endl;
82  }
83  else{
84  // sort them out
85  _end1.row_ = end2.row_;
86  _end2.row_ = end1.row_;
87  }
88  }
89  else if(_end1.row_ < _end2.row_){
90  if(_end1.row_ == 0 && _end2.row_ == _length-1){
91  _end1.row_ = end2.row_;
92  _end2.row_ = end1.row_;
93  }
94  }
95  //            _id = set_ID<3>({1, _end1.x_, _end1.y_});
96  }
97  else{
98  std::cout << '(' << _end1.row_ << ',' << _end1.column_ << ')' << "<->"
99  << '(' << _end2.row_ << ',' << _end2.column_ << ')'
100 << " is not a valid bond : line " << __LINE__ << std::endl;
101 //            _id = set_ID<3>({-1, -1, -1});
102 }
103
```

```
104 //         _id = {(horizontal) ? 0ul : 1ul, _end1.x_, end1.y_};  //
     unsigned long
105 _id = BondIndex(bondType, _end1.row_, _end1.column_);  // unsigned
     long
106
107 }
108
109 std::vector<Index> getSites() const { return {_end1, _end2};}
110
111 Index id() const {
112 return _end1;
113 }
114
115 BondIndex ID() const {
116 return _id;
117 }
118
119 void activate() {_status = true;}
120 void deactivate() {
121 _relative_index = {0,0};
122 _group_id = -1;
123 _status = false;
124 }
125 bool isActive() const { return _status;}
126 /*
127 * Group get_ID is the set_ID of the cluster they are in
128 */
129 int get_groupID() const {return _group_id;}
130 void set_groupID(int g_id) {_group_id = g_id;}
131
132
133 std::stringstream getBondString() const {
134 std::stringstream ss;
135 if(isActive()) {
136 // place '-' for horizontal bond and '|' for vertical bong
137 if(bondType == BondType::Horizontal) {
138 ss << '(' << _end1 << "<->" << _end2 << ')';
139 }
140 else {
141 ss << '(' << _end1 << "<|>" << _end2 << ')';
142 }
143 }
144 else
145 ss << "(**)";
```

```
146   return ss;
147   }
148
149   bool isHorizontal() const { return bondType == BondType::Horizontal;}
150   bool isVertical()   const { return bondType == BondType::Vertical;}
151
152   void relativeIndex(IndexRelative r){
153   _relative_index = r;
154   }
155
156   void relativeIndex(int x, int y){
157   _relative_index = {x,y};
158   }
159
160   IndexRelative relativeIndex() const {return _relative_index;}
161   };
162
163
164   std::ostream&   operator<<(std::ostream& os, const Bond& bond);
165   bool            operator==(Bond a, Bond b);
166   bool            operator<(const Bond& bond1, const Bond& bond2);
167   bool            operator>(const Bond& bond1, const Bond& bond2);
168
169
170   #endif //SITEPERCOLATION_BOND_H
171
172
```

file : **bond.cpp**

```
1    #include "bond.h"
2
3    /**
4    * use '-' and '|' in between '<>' to indicate horizontal or vertical
     bond
5    * ((0,1)<->(0,0)) for horizontal bond
6    * ((1,1)<|>(0,1)) for vertical bond
7    * @param os
8    * @param bond
9    * @return
10   */
11   std::ostream& operator<<(std::ostream& os, const Bond& bond)
12   {
13   if(bond.isActive()) {
14   // place '-' for horizontal bond and '|' for vertical bong
```

```
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      }
```

```
57
```

A.2.3 Lattice

file: **lattice.h**

```
1
2  #ifndef SITEPERCOLATION_LATTICE_H
3  #define SITEPERCOLATION_LATTICE_H
4
5  #include <vector>
6  #include <cmath>
7
8  #include "../percolation/cluster.h"
9  #include "../types.h"
10 #include "site.h"
11 #include "bond.h"
12
13
14 /**
15 * The square Lattice
16 * Site and Bonds are always present But they will not be counted
     unless they are activated
17 * always return by references, so that values in the class itself is
     modified
18 */
19 class SqLattice {
20 //     std::vector<std::vector<Index>> _clusters;  // only store index
     in the cluster
21 std::vector<std::vector<Site>> _sites;  // holds all the sites
22 std::vector<std::vector<Bond>> _h_bonds;  // holds all horizontal
     bonds
23 std::vector<std::vector<Bond>> _v_bonds;  // holds all vertical bonds
24
25 bool _bond_resetting_flag=true; // so that we can reset all bonds
26 bool _site_resetting_flag=true; // and all sites
27
28 value_type _length{};
29
30 private:
31 void reset_sites();
32 void reset_bonds();
33 public:
34 ~SqLattice() = default;
```

```
35  SqLattice() = default;
36  SqLattice(SqLattice&) = default;
37  SqLattice(SqLattice&&) = default;
38  SqLattice& operator=(const SqLattice&) = default;
39  SqLattice& operator=(SqLattice&&) = default;
40
41  SqLattice(value_type length, bool activate_bonds, bool activate_sites,
      bool bond_reset, bool site_reset);
42
43  void reset(bool reset_all=false);
44
45
46  /*************************************
47  * I/O functions
48  *************************************/
49  void view_sites();
50  void view_sites_extended();
51  void view_sites_by_id();
52  void view_sites_by_relative_index();
53  void view_bonds_by_relative_index();
54  void view_bonds_by_relative_index_v2();
55  void view_bonds_by_relative_index_v3();
56  void view_bonds_by_relative_index_v4();
57  void view_by_relative_index();
58  void view(); // view lattice bonds and sites together
59
60  void view_h_bonds();
61  void view_v_bonds();
62
63  void view_bonds(){
64  view_h_bonds();
65  view_v_bonds();
66  }
67
68  void view_h_bonds_extended();
69  void view_v_bonds_extended();
70
71  void view_bonds_by_id();
72
73  /*************************************
74  * Activation functions
75  *************************************/
76  void activateAllSite();
77  void activateAllBond();
```

```
78  void activate_site(Index index);
79  void activateBond(BondIndex bond);
80
81  void deactivate_site(Index index);
82  void deactivate_bond(Bond bond);
83
84
85  value_type length() const { return  _length;}
86
87  //     Site getSite(Index index);
88  //     Bond get_h_bond(Index set_ID);
89  //     Bond get_v_bond(Index set_ID);
90
91  Site& getSite(Index index);
92  //     Site&& getSiteR(Index index);
93  Bond& get_h_bond(Index id);
94  Bond& get_v_bond(Index id);
95  Bond& getBond(BondIndex);
96
97  const Site& getSite(Index index) const ;
98
99  void setGroupID(Index index, int group_id);
100 void setGroupID(BondIndex index, int group_id);
101 int getGroupID(Index index);
102 int getGroupID(BondIndex index);
103
104
105 /*
     ************************************************************************

106 * Get Neighbor from given index
107
     ************************************************************************
     */
108 std::vector<Index> get_neighbor_site_indices(Index site);   // site
     neighbor of site
109 std::vector<BondIndex> get_neighbor_bond_indices(BondIndex site); //
     bond neighbor of bond
110 std::vector<Index> get_neighbor_indices(BondIndex bond);   // two site
      neighbor of bond.
111
112 static std::vector<Index> get_neighbor_site_indices(size_t length,
     Index site);   // 4 site neighbor of site
```

```
113  static std::vector<BondIndex> get_neighbor_bond_indices(size_t length,
       BondIndex site); // 6 bond neighbor of bond
114  static std::vector<Index> get_neighbor_indices(size_t length,
      BondIndex bond);    // 2 site neighbor of bond.
115
116
117  };
118
119
120  #endif //SITEPERCOLATION_LATTICE_H
121
122
```

file: **lattice.cpp**

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

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

```
68  for (value_type i{}; i != _length; ++i) {    // all bonds and sites are
      activated by default
69  _sites[i] = std::vector<Site>(_length);
70  _h_bonds[i] = std::vector<Bond>(_length);
71  _v_bonds[i] = std::vector<Bond>(_length);
72  for (value_type j{}; j != _length; ++j) {
73  _sites[i][j] = Site(Index(i, j), _length);
74  _sites[i][j].activate();
75  _h_bonds[i][j] = {Index(i, j), Index(i, (j + 1) % _length), _length};
76  _v_bonds[i][j] = {Index(i, j), Index((i + 1) % _length, j), _length};
77  _h_bonds[i][j].activate();
78  _v_bonds[i][j].activate();
79  }
80  }
81  }
82
83  }
84
85  /*************************************************************************
86  * Activation and Deactivation
87  *************************************************************************/
88  void SqLattice::activateAllSite()
89  {
90  for(value_type i{} ; i != _length ; ++i) {
91  for (value_type j{}; j != _length; ++j) {
92  _sites[i][j].activate();
93  }
94  }
95  }
96
97  /**
98  *
99  */
100 void SqLattice::activateAllBond()
101 {
102 for(value_type i{} ; i != _length ; ++i) {
103 for (value_type j{}; j != _length; ++j) {
104 _h_bonds[i][j].activate();
105 _v_bonds[i][j].activate();
106 }
107 }
108 }
109
110 void SqLattice::activate_site(Index index) {
```

```
111 //     cout << "activating site " << index << endl;
112 _sites[index.row_][index.column_].activate();
113 }
114
115
116 void SqLattice::activateBond(BondIndex bond) {
117 // check if the bond is vertical or horizontal
118 // then call appropriate function to activate _h_bond or _v_bond
119 if(bond.horizontal()){ // horizontal
120 if(_h_bonds[bond.row_][bond.column_].isActive()){
121 cout << "Bond is already activated : line " << __LINE__ << endl;
122 }
123 _h_bonds[bond.row_][bond.column_].activate();
124 }
125 else if(bond.vertical()) // vertical
126 {
127 if(_v_bonds[bond.row_][bond.column_].isActive()){
128 cout << "Bond is already activated : line " << __LINE__ << endl;
129 }
130 _v_bonds[bond.row_][bond.column_].activate();
131 }
132 else{
133 cout << bond << " is not a valid bond : line " << __LINE__ << endl;
134 }
135
136 }
137
138
139 void SqLattice::deactivate_site(Index index){
140 _sites[index.row_][index.column_].deactivate();
141 }
142
143
144 void SqLattice::deactivate_bond(Bond bond) {
145 // check if the bond is vertical or horizontal
146 // then call appropriate function to activate _h_bond or _v_bond
147 if(bond.isHorizontal()){
148 if(_h_bonds[bond.id().row_][bond.id().column_].isActive()){
149 cout << "Bond is already activated : line " << __LINE__ << endl;
150 }
151 _h_bonds[bond.id().row_][bond.id().column_].deactivate();
152 }
153 else if(bond.isVertical())
154 {
```

```
155 if(_v_bonds[bond.id().row_][bond.id().column_].isActive()){
156 cout << "Bond is already activated : line " << __LINE__ << endl;
157 }
158 _v_bonds[bond.id().row_][bond.id().column_].deactivate();
159 }
160 else{
161 bond.activate();
162 cout << bond << " is not a valid bond : line " << __LINE__ << endl;
163 }
164 }
165
166
167 /*********************************************************
168 * Viewing methods
169 *********************************************************/
170 /**
171 *  View the sites of the lattice
172 *  place (*) if the site is not active
173 */
174 void SqLattice::view_sites()
175 {
176 std::cout << "view sites" << std::endl;
177 std::cout << '{';
178 for(value_type i{} ; i != _length ; ++i) {
179 if(i!=0) std::cout << "  ";
180 else std::cout << '{';
181 for (value_type j{}; j != _length; ++j) {
182 if(_sites[i][j].isActive()){
183 std::cout << _sites[i][j] ;
184 }
185 else{
186 std::cout << "(*)";
187 }
188 if(j != _length-1)
189 std::cout << ',';
190 }
191 std::cout << '}';
192 if(i != _length-1)
193 std::cout << std::endl;
194 }
195 std::cout << '}';
196 std::cout << std::endl;
197 }
198
```

```
199 /**
200 *  View the sites of the lattice
201 *  place (*) if the site is not active
202 *  Shows the group_id along with sites
203 *
204 *  Very good output format. Up to lattice size < 100
205 */
206 void SqLattice::view_sites_extended()
207 {
208 std::cout << "view sites" << std::endl;
209 std::cout << '{';
210 for(value_type i{} ; i != _length ; ++i) {
211 if(i!=0) std::cout << "  ";
212 else std::cout << '{';
213 for (value_type j{}; j != _length; ++j) {
214 std::cout << std::setw(3) << _sites[i][j].get_groupID() << ":";
215
216 if(_sites[i][j].isActive()) {
217 cout << '(' << std::setw(2) << _sites[i][j]._id.row_ << ','
218 << std::setw(2) << _sites[i][j]._id.column_ << ')';
219 }
220 else{
221 cout << std::setw(7) << "(*)";
222 }
223
224 if(j != _length-1)
225 std::cout << ',';
226 }
227 std::cout << '}';
228 if(i != _length-1)
229 std::cout << std::endl;
230 }
231 std::cout << '}';
232 std::cout << std::endl;
233 }
234
235
236
237 /**
238 * Displays group ids of sites in a matrix form
239 */
240 void SqLattice::view_sites_by_id() {
241 std::cout << "Sites by id : line " << __LINE__ << endl;
242 cout << "   ";
```

```
243  for(value_type j{}; j != _length; ++ j){
244  cout << " " << setw(3) << j;
245  }
246  cout << endl << " __|";
247  for(value_type j{}; j != _length; ++ j){
248  cout << "_ _ ";
249  }
250  cout << endl;
251  for(value_type i{} ; i != _length; ++i){
252  cout << setw(3) << i << "|";
253  for(value_type j{} ; j != _length ; ++ j){
254  cout << setw(3) << _sites[i][j].get_groupID() << ' ';
255  }
256  cout << endl;
257  }
258  }
259
260
261  /**
262  *
263  */
264  void SqLattice::view_sites_by_relative_index(){
265  std::cout << "Relative index : line " << __LINE__ << endl;
266  cout << "Format: \"id(x,y)\"" << endl;
267  cout << "   |";
268  for(value_type j{}; j != _length; ++ j){
269  cout << setw(4) << j << "          |";
270  }
271  cout << endl;
272  print_h_barrier(_length, "___|__", "___________|__");
273  for(value_type i{} ; i != _length; ++i){
274  cout << setw(3) << i << "|";
275  for(value_type j{} ; j != _length ; ++ j){
276  if(_sites[i][j].get_groupID() == -1){
277  // left blank
278  cout << setw(4) << _sites[i][j].get_groupID() << "          |";
279  continue;
280  }
281  cout << setw(4) << _sites[i][j].get_groupID() << _sites[i][j].
      relativeIndex() << "|";
282  }
283  cout << endl;
284  print_h_barrier(_length, "___|__", "___________|__");
285  }
```

```
286 //     print_h_barrier(_length, "___|__", "___________|_");
287 }
288
289 /**
290 * View bonds in the lattice by relative index.
291 * format : id(relative_index)
292 */
293 void SqLattice::view_bonds_by_relative_index() {
294
295 std::cout << "Bonds by id : line " << __LINE__ << endl;
296 // printing indices for columns
297 std::cout << "    | ";
298 for(value_type i{}; i != _length; ++i){
299 std::cout << "   " << setw(4) << i << "         | ";
300 }
301 std::cout << std::endl;
302
303 // pringing H,V label
304
305 print_h_barrier(_length, "    |  ", "V          H  |  ");
306 print_h_barrier(_length, "____|__", "_______________|__");
307
308 // for each row there will be two columns
309 for(value_type i{}; i != _length; ++i){
310 std::cout << i << ' ';
311 std::cout << "H |";
312 for(value_type j1{}; j1 != _length; ++j1){
313 std::cout << "     " << std::setw(3) << _h_bonds[i][j1].get_groupID()
314 << _h_bonds[i][j1].relativeIndex() << "|";
315 }
316 std::cout << std::endl;
317 print_h_barrier(_length, "    |  ", "               |  "); // just for
    better viewing
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 << _v_bonds[i][j2].relativeIndex() << "     |";
322 }
323 std::cout << std::endl;
324
325 // printing horizontal separator
326 print_h_barrier(_length, "____|__", "_______________|__");
327 }
328 std::cout << std::endl;
```

```
329 }
330
331
332 /**
333 * View bonds in the lattice by relative index. id of the site is
     showed
334 * format : id for site or id(relative_index) for bond
335 */
336 void SqLattice::view_bonds_by_relative_index_v2() {
337
338 std::cout << "Bonds by id : line " << __LINE__ << endl;
339 cout << "site id -1 means isolated site and 0 means connected site in
     bond percolation(definition)" << endl;
340 print_h_barrier(15, "_", "___", "_\n");
341 cout << "|(site id) (horizontal bond id(relative index))|" << endl;
342 cout << "|(vertical bond id(relative index))            |" << endl;
343 print_h_barrier(15, "-", "___", "-\n");
344 // printing indices for columns
345 std::cout << "    | ";
346 for(value_type i{}; i != _length; ++i){
347 std::cout << i << "                  | ";
348 }
349 std::cout << std::endl;
350
351 // pringing H,V label
352
353 print_h_barrier(_length, "    |  ", " V           H   |  ");
354 print_h_barrier(_length, "____|__", "__________________|__");
355
356 // for each row there will be two columns
357 for(value_type i{}; i != _length; ++i){
358 std::cout << i << ' ';
359 std::cout << "H |";
360 for(value_type j1{}; j1 != _length; ++j1){
361 std::cout << std::setw(3) << _sites[i][j1].get_groupID() ;
362 std::cout << "     " << std::setw(3) << _h_bonds[i][j1].get_groupID()
363 << _h_bonds[i][j1].relativeIndex() << "|";
364 }
365 std::cout << std::endl;
366 print_h_barrier(_length, "    |  ", "                  |  "); // just
     to see a better view
367 std::cout << "  " << "V |";
368 for(value_type j2{}; j2 != _length; ++j2){
369 std::cout << std::setw(3) << _v_bonds[i][j2].get_groupID()
```

```
370 << _v_bonds[i][j2].relativeIndex() << "        |";
371 }
372 std::cout << std::endl;
373
374 // printing horizontal separator
375 print_h_barrier(_length, "____|__", "___________________|__");
376 }
377
378 std::cout << std::endl;
379 }
380
381
382 /**
383 * View bonds in the lattice by relative index. id of the site is
     showed
384 * format : id(relative index) for site and only id for bond
385 */
386 void SqLattice::view_bonds_by_relative_index_v3() {
387
388 std::cout << "Bonds by id : line " << __LINE__ << endl;
389 //cout << "site id -1 means isolated site and 0 means connected site
     in bond percolation(definition)" << endl;
390 print_h_barrier(15, "_", "___", "_\n");
391 cout << "|(site id) (horizontal bond id(relative index))|" << endl;
392 cout << "|(vertical bond id(relative index))            |" << endl;
393 print_h_barrier(15, "-", "___", "-\n");
394 // printing indices for columns
395 std::cout << "    | ";
396 for(value_type i{}; i != _length; ++i){
397 std::cout << i << "                | ";
398 }
399 std::cout << std::endl;
400
401 // pringing H,V label
402
403 print_h_barrier(_length, "    |  ", " V          H  |  ");
404 print_h_barrier(_length, "____|__", "_________________|__");
405
406 // for each row there will be two columns
407 for(value_type i{}; i != _length; ++i){
408 std::cout << i << ' ';
409 std::cout << "H |";
410 for(value_type j1{}; j1 != _length; ++j1){
```

```
411 std::cout << std::setw(3) << _sites[i][j1].get_groupID() << _sites[i][
     j1].relativeIndex() ;
412 std::cout << "    " << std::setw(3) << _h_bonds[i][j1].get_groupID() <<
      "|";
413 }
414 std::cout << std::endl;
415 print_h_barrier(_length, "    |  ", "                |  "); // just to
      see a better view
416 std::cout << "  " << "V |";
417 for(value_type j2{}; j2 != _length; ++j2){
418 std::cout << std::setw(3) << _v_bonds[i][j2].get_groupID() << "
          |";
419 }
420 std::cout << std::endl;
421
422 // printing horizontal separator
423 print_h_barrier(_length, "____|__", "_________________|__");
424 }
425
426 std::cout << std::endl;
427 }
428
429 /**
430 * View bonds in the lattice by relative index. id of the site is
     showed
431 * format : id(relative index) for site and only id for bond
432 * if any site is isolated relative index is not shown
433 */
434 void SqLattice::view_bonds_by_relative_index_v4() {
435
436 std::cout << "Bonds by id : line " << __LINE__ << endl;
437 //cout << "site id -1 means isolated site and 0 means connected site
     in bond percolation(definition)" << endl;
438 print_h_barrier(15, "_", "___", "_\n");
439 cout << "|(site id) (horizontal bond id(relative index))|" << endl;
440 cout << "|(vertical bond id(relative index))           |" << endl;
441 print_h_barrier(15, "-", "___", "-\n");
442 // printing indices for columns
443 std::cout << "    | ";
444 for(value_type i{}; i != _length; ++i){
445 std::cout << i << "                | ";
446 }
447 std::cout << std::endl;
448
```

```
449  // pringing H,V label
450
451  print_h_barrier(_length, "    |  ", " V          H  |  ");
452  print_h_barrier(_length, "____|__", "_________________|__");
453
454  // for each row there will be two columns
455  for(value_type i{}; i != _length; ++i){
456  std::cout << i << ' ';
457  std::cout << "H |";
458  for(value_type j1{}; j1 != _length; ++j1){
459  int id = _sites[i][j1].get_groupID();
460  std::cout << std::setw(3) << id  ;
461  if(id != -1){
462  cout << _sites[i][j1].relativeIndex();
463  }else{
464  cout << "(-,-)    ";
465  }
466  std::cout << "   " << std::setw(3) << _h_bonds[i][j1].get_groupID() <<
       "|";
467  }
468  std::cout << std::endl;
469  print_h_barrier(_length, "    |  ", "                |  "); // just to
       see a better view
470  std::cout << "  " << "V |";
471  for(value_type j2{}; j2 != _length; ++j2){
472  std::cout << std::setw(3) << _v_bonds[i][j2].get_groupID() << "
           |";
473  }
474  std::cout << std::endl;
475
476  // printing horizontal separator
477  print_h_barrier(_length, "____|__", "_________________|__");
478  }
479
480  std::cout << std::endl;
481  }
482
483
484
485  /**
486  * View lattice (sites and bonds) by relative index.
487  * format : id(relative_index)
488  */
489  void SqLattice::view_by_relative_index() {
```

```
490
491  std::cout << "Bonds by id : line " << __LINE__ << endl;
492  // printing indices for columns
493  std::cout << "    | ";
494  for(value_type i{}; i != _length; ++i){
495  std::cout << i << "                          | ";
496  }
497  std::cout << std::endl;
498
499  // pringing H,V label
500
501  print_h_barrier(_length, "    |  ", " V                    H   |  ");
502  print_h_barrier(_length, "____|__", "_____________________________|__");
503
504  // for each row there will be two columns
505  for(value_type i{}; i != _length; ++i){
506  std::cout << i << ' ';
507  std::cout << "H |";
508  for(value_type j1{}; j1 != _length; ++j1){
509  std::cout << std::setw(3) << _sites[i][j1].get_groupID() << _sites[i][
     j1].relativeIndex();
510  std::cout << "   " << std::setw(3) << _h_bonds[i][j1].get_groupID()
511  << _h_bonds[i][j1].relativeIndex() << "|";
512  }
513  std::cout << std::endl;
514  print_h_barrier(_length, "    |  ", "                          |  ");
     // just to see a better view
515  std::cout << "  " << "V |";
516  for(value_type j2{}; j2 != _length; ++j2){
517  std::cout << std::setw(3) << _v_bonds[i][j2].get_groupID()
518  << _v_bonds[i][j2].relativeIndex() << "                |";
519  }
520  std::cout << std::endl;
521
522  // printing horizontal separator
523  print_h_barrier(_length, "____|__", "_____________________________|__");
524  }
525
526  //        view_h_bonds_extended();
527  //        view_v_bonds_extended();
528  std::cout << std::endl;
529
530  }
531
```

```
532 /**
533 * View lattice (sites and bonds) by relative index.
534 * format : id(relative_index)
535 */
536 void SqLattice::view() {
537
538 std::cout << "Bonds by id : line " << __LINE__ << endl;
539 cout << "Structure " << endl;
540 print_h_barrier(10, "_", "___", "_\n");
541 cout << "|(site id) (horizontal bond id)|" << endl;
542 cout << "|(vertical bond id)            |" << endl;
543 print_h_barrier(10, "-", "___", "-\n");
544 // printing indices for columns
545 std::cout << "    | ";
546 for(value_type i{}; i != _length; ++i){
547 std::cout << i << "      | ";
548 }
549 std::cout << std::endl;
550
551 // pringing H,V label
552
553 print_h_barrier(_length, "    |  ", "V    H|  ");
554 print_h_barrier(_length, "____|__", "______|__");
555
556 // for each row there will be two columns
557 for(value_type i{}; i != _length; ++i){
558 std::cout << i << ' ';
559 std::cout << "H |";
560 for(value_type j1{}; j1 != _length; ++j1){
561 std::cout << std::setw(3) << _sites[i][j1].get_groupID() ;
562 std::cout << "  " << std::setw(3) << _h_bonds[i][j1].get_groupID() <<
    "|";
563 }
564 std::cout << std::endl;
565 print_h_barrier(_length, "    |  ", "      |  "); // just to see a
    better view
566 std::cout << "  " << "V |";
567 for(value_type j2{}; j2 != _length; ++j2){
568 std::cout << std::setw(3) << _v_bonds[i][j2].get_groupID() << "     |"
    ;
569 }
570 std::cout << std::endl;
571
572 // printing horizontal separator
```

```
573 print_h_barrier(_length, "____|__", "______|__");
574 }
575
576 //          view_h_bonds_extended();
577 //          view_v_bonds_extended();
578 std::cout << std::endl;
579
580 }
581
582
583 /**
584 *
585 */
586 void SqLattice::view_h_bonds()
587 {
588 std::cout << "view horizontal bonds" << std::endl;
589 std::cout << '{';
590 for(value_type i{} ; i != _length ; ++i) {
591 if(i!=0) std::cout << "  ";
592 else std::cout << '{';
593 for (value_type j{}; j != _length; ++j) {
594 std::cout << _h_bonds[i][j] ;
595 if(j != _length-1)
596 std::cout << ',';
597 }
598 std::cout << '}';
599 if(i != _length-1)
600 std::cout << std::endl;
601 }
602 std::cout << '}';
603 std::cout << std::endl;
604 }
605
606 /**
607 *
608 */
609 void SqLattice::view_v_bonds()
610 {
611 std::cout << "view vertical bonds" << std::endl;
612 std::cout << '{';
613 for(value_type i{} ; i != _length ; ++i) {
614 if(i!=0) std::cout << "  ";
615 else std::cout << '{';
616 for (value_type j{}; j != _length; ++j) {
```

```
617   std::cout << _v_bonds[i][j] ;
618   if(j != _length-1)
619   std::cout << ',';
620   }
621   std::cout << '}';
622   if(i != _length-1)
623   std::cout << std::endl;
624   }
625   std::cout << '}';
626   std::cout << std::endl;
627   }
628
629
630   /**
631   *
632   */
633   void SqLattice::view_h_bonds_extended(){
634   std::cout << "view horizontal bonds" << std::endl;
635   std::cout << '{';
636   for(value_type i{} ; i != _length ; ++i) {
637   if(i!=0) std::cout << "  ";
638   else std::cout << '{';
639   for (value_type j{}; j != _length; ++j) {
640   std::cout << "(" << _h_bonds[i][j].get_groupID() << ":" << _h_bonds[i
       ][j] << ")" ;
641   if(j != _length-1)
642   std::cout << ',';
643   }
644   std::cout << '}';
645   if(i != _length-1)
646   std::cout << std::endl;
647   }
648   std::cout << '}';
649   std::cout << std::endl;
650   }
651
652   /**
653   *
654   */
655   void SqLattice::view_v_bonds_extended(){
656   std::cout << "view vertical bonds" << std::endl;
657   std::cout << '{';
658   for(value_type i{} ; i != _length ; ++i) {
659   if(i!=0) std::cout << "  ";
```

```
660   else std::cout << '{';
661   for (value_type j{}; j != _length; ++j) {
662   std::cout << "(" << _v_bonds[i][j].get_groupID() << ":" << _v_bonds[i
       ][j] << ")" ;
663   if(j != _length-1)
664   std::cout << ',';
665   }
666   std::cout << '}';
667   if(i != _length-1)
668   std::cout << std::endl;
669   }
670   std::cout << '}';
671   std::cout << std::endl;
672   }
673
674
675   /**
676   *
677   */
678   void SqLattice::view_bonds_by_id(){
679   std::cout << "Bonds by id : line " << __LINE__ << endl;
680   cout << "Structure " << endl;
681   print_h_barrier(8, "__", "___", "_\n");
682   cout << "|      (horizontal bond id)|" << endl;
683   cout << "|(vertical bond id)        |" << endl;
684   print_h_barrier(8, "__", "___", "-\n");
685   // printing indices for columns
686   std::cout << "    | ";
687   for(value_type i{}; i != _length; ++i){
688   std::cout << i << "    | ";
689   }
690   std::cout << std::endl;
691
692   // pringing H,V label
693
694   print_h_barrier(_length, "    |  ", "V  H|  ");
695   print_h_barrier(_length, "____|__", "____|__");
696
697   // for each row there will be two columns
698   for(value_type i{}; i != _length; ++i){
699   std::cout << i << ' ';
700   std::cout << "H |";
701   for(value_type j1{}; j1 != _length; ++j1){
```

```
702   std::cout << "    " << std::setw(3) << _h_bonds[i][j1].get_groupID() <<
        "|";
703   }
704   std::cout << std::endl;
705   std::cout << "  " << "V |";
706   for(value_type j2{}; j2 != _length; ++j2){
707   std::cout << std::setw(3) << _v_bonds[i][j2].get_groupID() << "   |";
708   }
709   std::cout << std::endl;
710
711   // printing horizontal separator
712   print_h_barrier(_length, "____|__", "____|__");
713   }
714
715   //        view_h_bonds_extended();
716   //        view_v_bonds_extended();
717   std::cout << std::endl;
718   }
719
720
721   Site& SqLattice::getSite(Index index) {
722   return _sites[index.row_][index.column_];
723   }
724
725   //Site&& SqLattice::getSiteR(Index index) {
726   //    Site a = _sites[index.x_][index.y_];
727   //    return std::move(a);
728   //}
729
730   const Site& SqLattice::getSite(Index index) const {
731   return _sites[index.row_][index.column_];
732   }
733
734   void SqLattice::setGroupID(Index index, int group_id){
735   _sites[index.row_][index.column_].set_groupID(group_id);
736   }
737
738   void SqLattice::setGroupID(BondIndex index, int group_id){
739   if(index.horizontal()){
740   _h_bonds[index.row_][index.column_].set_groupID(group_id);
741   }
742   if(index.vertical()){
743   _v_bonds[index.row_][index.column_].set_groupID(group_id);
744   }
```

```
745 }
746
747 int SqLattice::getGroupID(Index index){
748 return _sites[index.row_][index.column_].get_groupID();
749 }
750
751 int SqLattice::getGroupID(BondIndex index){
752 if(index.horizontal()){
753 return _h_bonds[index.row_][index.column_].get_groupID();
754 }
755 if(index.vertical()){
756 return  _v_bonds[index.row_][index.column_].get_groupID();
757 }
758 return -1;
759 }
760
761 //Bond Lattice::get_h_bond(Index set_ID) {
762 //     return _h_bonds[id.x_][set_ID.y_];
763 //}
764 //
765 //Bond Lattice::get_v_bond(Index set_ID) {
766 //     return _v_bonds[id.x_][set_ID.y_];
767 //}
768
769
770 Bond& SqLattice::get_h_bond(Index id) {
771 return _h_bonds[id.row_][id.column_];
772 }
773
774 Bond& SqLattice::get_v_bond(Index id) {
775 return _v_bonds[id.row_][id.column_];
776 }
777
778
779 Bond& SqLattice::getBond(BondIndex index) {
780 if(index.horizontal())
781 return _h_bonds[index.row_][index.column_];
782 if(index.vertical())
783 return _v_bonds[index.row_][index.column_];
784 // todo throw exception
785 throw InvalidBond{"Invalid bond : line " + to_string(__LINE__)};
786 }
787
788 void SqLattice::reset(bool reset_all) {
```

```
789  if(reset_all){
790  reset_sites();
791  reset_bonds();
792  return;
793  }
794  // setting all group id to -1
795  if(_site_resetting_flag) {
796  reset_sites();
797  }
798  //    cout << "Bond resetting is disabled : line " << __LINE__ << endl
       ;
799  if(_bond_resetting_flag) {
800  reset_bonds();
801  }
802
803  }
804
805  /**
806  *
807  */
808  void SqLattice::reset_bonds() {
809  for(value_type i{}; i != _h_bonds.size(); ++i){
810  for (int j{}; j != _h_bonds[i].size(); ++j) {
811  // deactivating. automatically set group id == - and relative index ==
       (0,0)
812  // setting group id = -1 and deactivating the bond
813  _h_bonds[i][j].deactivate();
814  _v_bonds[i][j].deactivate();
815
816
817  }
818  }
819  }
820
821  /**
822  *
823  */
824  void SqLattice::reset_sites() {
825  for(value_type i{}; i != _sites.size(); ++i){
826  for(value_type j{}; j != _sites[i].size(); ++j) {
827  // deactivating. automatically set group id == - and relative index ==
       (0,0)
828  // setting group id = -1 and deactivating the site
829  _sites[i][j].deactivate();
```

```
830   }
831   }
832   }
833
834
835
836   /*
       ***************************************************************************
837   * Get Neighbor from given index
838
       ***************************************************************************
       */
839   /**
840   * Periodic case only.
841   * Each site has four neighbor sites.
842   * @param site
843   * @return
844   */
845   std::vector<Index> SqLattice::get_neighbor_site_indices(Index site){
846   std::vector<Index> sites(4);
847   sites[0] = {(site.row_ + 1) % _length, site.column_};
848   sites[1] = {(site.row_ - 1 + _length) % _length, site.column_};
849   sites[2] = {site.row_, (site.column_ + 1) % _length};
850   sites[3] = {site.row_, (site.column_ - 1 + _length) % _length};
851   return sites;
852   }
853
854   /**
855   * Periodic case only.
856   * Each bond has six neibhbor bonds.
857   * @param site
858   * @return
859   */
860   std::vector<BondIndex> SqLattice::get_neighbor_bond_indices(BondIndex
       bond) {
861   value_type next_column = (bond.column_ + 1) % _length;
862   value_type prev_column = (bond.column_ - 1 + _length) % _length;
863   value_type prev_row = (bond.row_ - 1 + _length) % _length;
864   value_type next_row = (bond.row_ + 1) % _length;
865
866   vector<BondIndex> bonds(6);
867
868   // horizontal bond case
```

```
869  if (bond.horizontal()) {
870  // increase column index for the right neighbor
871
872  // left end of bond
873  bonds[0] = {BondType::Vertical, bond.row_, bond.column_};
874  bonds[1] = {BondType::Vertical, prev_row, bond.column_};
875  bonds[2] = {BondType::Horizontal, bond.row_, prev_column};
876
877  // right end bond
878  bonds[3] = {BondType::Vertical, prev_row, next_column};
879  bonds[4] = {BondType::Vertical, bond.row_, next_column};
880  bonds[5] = {BondType::Horizontal, bond.row_, next_column};
881
882  }
883  // vertical bond case
884  else if (bond.vertical()) {
885  // increase row index
886
887  // top end of bond
888  bonds[0] = {BondType::Horizontal, bond.row_, bond.column_};
889  bonds[1] = {BondType::Horizontal, bond.row_, prev_column};
890  bonds[2] = {BondType::Vertical, prev_row, bond.column_};
891
892  // bottom end of bond
893  bonds[3] = {BondType::Horizontal, next_row, bond.column_};
894  bonds[4] = {BondType::Horizontal, next_row, prev_column};
895  bonds[5] = {BondType::Vertical, next_row, bond.column_};
896
897  }
898
899
900  return bonds;
901  }
902
903  std::vector<Index> SqLattice::get_neighbor_indices(BondIndex bond) {
904  value_type r = bond.row_;
905  value_type c = bond.column_;
906  vector<Index> sites(2);
907  sites[0] = {r, c};
908  if(bond.horizontal()){
909  sites[1] = {r, (c+1) % _length};
910  }else{
911  sites[1] = {(r+1) % _length, c};
912  }
```

```
913  return sites;
914  }
915
916  /*************************************************************
917  * Static methods
918  */
919  std::vector<Index> SqLattice::get_neighbor_site_indices(size_t length,
       Index site){
920  std::vector<Index> sites(4);
921  sites[0] = {(site.row_ + 1) % length, site.column_};
922  sites[1] = {(site.row_ - 1 + length) % length, site.column_};
923  sites[2] = {site.row_, (site.column_ + 1) % length};
924  sites[3] = {site.row_, (site.column_ - 1 + length) % length};
925  return sites;
926  }
927
928  /**
929  * Periodic case only.
930  * Each bond has six neibhbor bonds.
931  * @param site
932  * @return
933  */
934  std::vector<BondIndex> SqLattice::get_neighbor_bond_indices(size_t
       length, BondIndex bond) {
935  value_type next_column = (bond.column_ + 1) % length;
936  value_type prev_column = (bond.column_ - 1 + length) % length;
937  value_type prev_row = (bond.row_ - 1 + length) % length;
938  value_type next_row = (bond.row_ + 1) % length;
939
940  vector<BondIndex> bonds(6);
941
942  // horizontal bond case
943  if (bond.horizontal()) {
944  // increase column index for the right neighbor
945
946  // left end of bond
947  bonds[0] = {BondType::Vertical, bond.row_, bond.column_};
948  bonds[1] = {BondType::Vertical, prev_row, bond.column_};
949  bonds[2] = {BondType::Horizontal, bond.row_, prev_column};
950
951  // right end bond
952  bonds[3] = {BondType::Vertical, prev_row, next_column};
953  bonds[4] = {BondType::Vertical, bond.row_, next_column};
954  bonds[5] = {BondType::Horizontal, bond.row_, next_column};
```

```
955
956   }
957   // vertical bond case
958   else if (bond.vertical()) {
959   // increase row index
960
961   // top end of bond
962   bonds[0] = {BondType::Horizontal, bond.row_, bond.column_};
963   bonds[1] = {BondType::Horizontal, bond.row_, prev_column};
964   bonds[2] = {BondType::Vertical, prev_row, bond.column_};
965
966   // bottom end of bond
967   bonds[3] = {BondType::Horizontal, next_row, bond.column_};
968   bonds[4] = {BondType::Horizontal, next_row, prev_column};
969   bonds[5] = {BondType::Vertical, next_row, bond.column_};
970
971   }
972
973
974   return bonds;
975   }
976
977   std::vector<Index> SqLattice::get_neighbor_indices(size_t length,
       BondIndex bond) {
978   value_type r = bond.row_;
979   value_type c = bond.column_;
980   vector<Index> sites(2);
981   sites[0] = {r, c};
982   if(bond.horizontal()){
983   sites[1] = {r, (c+1) % length};
984   }else{
985   sites[1] = {(r+1) % length, c};
986   }
987   return sites;
988   }
989
990
```

A.2.4 Cluster

file: **cluster.h**

```
1   //
2   // Created by shahnoor on 10/3/2017.
```

```
3  //
4
5  #ifndef SITEPERCOLATION_CLUSTER_H
6  #define SITEPERCOLATION_CLUSTER_H
7
8  #include <vector>
9  #include <set>
10 #include "../lattice/bond.h"
11 #include "../types.h"
12 #include "../lattice/site.h"
13
14
15 /**
16 * Cluster of bonds and sites
17 * version 3
18 * final goal -> make a template cluster. so that we can use it for
    Bond cluster or Site cluster
19 * root site (bond) is the first site (bond) of the cluster. nedeed for
     (wrapping) site percolation
20 */
21 class Cluster{
22 // contains bond and site
23 std::vector<BondIndex>  _bond_index; // BondIndex for indexing bonds
24 std::vector<Index>      _site_index; // Site index
25
26
27 int _creation_time;  // holds the creation birthTime of a cluster
    object
28 int _id;
29 public:
30 //    using iterator = std::vector<Bond>::iterator;
31
32 ~Cluster()                              = default;
33 Cluster()                               = default;
34 Cluster(Cluster&)                   = default;
35 Cluster(Cluster&&)                  = default;
36 Cluster& operator=(const Cluster&)      = default;
37 Cluster& operator=(Cluster&&)     = default;
38
39 explicit Cluster(int id){
40
41 _id = id;        // may be modified in the program
42
43 /*
```

```
44  * Only readable, not modifiable.
45  * when time = 0 => only lattice exists and bonds in site percolation,
      not any sites
46  * When id = 0, time = 1 => we have placed the first site, hence
      created a cluster with size greater than 1
47  *      Only then Cluster constructor is called.
48  *
49  */
50  _creation_time = id + 1;        // only readable, not modifiable
51  }
52
53
54  void addSiteIndex(Index );
55  void addBondIndex(BondIndex );
56
57  Index lastAddedSite(){return _site_index.back();}
58  BondIndex lastAddedBond(){return _bond_index.back();}
59
60  bool isPresent(BondIndex bond) const ;
61  bool isPresent(Index site) const ;
62
63  bool checkPresenceAndErase(BondIndex bond);
64  bool checkPresenceAndErase(Index bond);
65
66  bool checkPresenceAndEraseIf(BondIndex bond, bool flag);
67  bool checkPresenceAndEraseIf(Index bond, bool flag);
68
69  void eraseSite(value_type index);
70  void eraseBond(value_type index);
71
72
73  void insert(const std::vector<BondIndex>& bonds);
74  void insert(const std::vector<Index>& sites);
75
76  void insert(const Cluster& cluster);
77  void insert_v2(const Cluster& cluster);
78  void insert_with_id_v2(const Cluster& cluster, int id);
79
80
81  friend std::ostream& operator<<(std::ostream& os, const Cluster&
      cluster);
82
83  const std::vector<BondIndex>&   getBondIndices()    {return
      _bond_index;}
```

```
84  const std::vector<Index>&        getSiteIndices()     {return
     _site_index;}
85
86  const std::vector<BondIndex>&    getBondIndices()  const  {return
     _bond_index;}
87  const std::vector<Index>&        getSiteIndices()  const  {return
     _site_index;}
88
89  value_type numberOfBonds() const { return _bond_index.size();}
90  value_type numberOfSites() const { return _site_index.size();}
91  int get_ID() const { return _id;}
92  void set_ID(int id) { _id = id;}
93
94  int birthTime() const {return _creation_time;}
95
96  Index getRootSite()const{return _site_index[0];} // for site
     percolation
97  BondIndex getRootBond()const{return _bond_index[0];} // for bond
     percolation
98  bool empty() const { return _bond_index.empty() && _site_index.empty()
     ;}
99  void clear() {_bond_index.clear(); _site_index.clear(); }
100 };
101
102
103
104 #endif //SITEPERCOLATION_CLUSTER_H
105
106
```

file: **cluster.cpp**

```
1  //
2  // Created by shahnoor on 10/2/2017.
3  //
4
5
6  //
7  // Created by shahnoor on 10/11/2017.
8  //
9
10
11 #include "cluster.h"
12
13 using namespace std;
```

```
14
15  // add Site index
16  void Cluster::addSiteIndex(Index index) {
17  _site_index.push_back(index);
18  }
19
20  void Cluster::addBondIndex(BondIndex bondIndex) {
21  _bond_index.push_back(bondIndex);
22  }
23
24
25  bool Cluster::isPresent(BondIndex bond) const {
26  for (auto a: _bond_index) {
27  if (a == bond)
28  return true;
29  }
30  return false;
31  }
32
33  bool Cluster::isPresent(Index site) const {
34  for (auto a: _site_index) {
35  if (a == site)
36  return true;
37  }
38  return false;
39  }
40
41  bool Cluster::checkPresenceAndErase(BondIndex bond) {
42  for (auto it = _bond_index.begin(); it != _bond_index.end(); ++it) {
43  if (*it == bond) {
44  _bond_index.erase(it);
45  return true;
46  }
47  }
48  return false;
49  }
50
51  bool Cluster::checkPresenceAndEraseIf(BondIndex bond, bool flag) {
52  if(!flag)
53  return false;
54  for (auto it = _bond_index.begin(); it != _bond_index.end(); ++it) {
55  if (*it == bond) {
56  _bond_index.erase(it);
57  return true;
```

```
58  }
59  }
60  return false;
61  }
62
63  bool Cluster::checkPresenceAndEraseIf(Index bond, bool flag) {
64  if(!flag)
65  return false;
66  for (auto it = _site_index.begin(); it != _site_index.end(); ++it) {
67  if (*it == bond) {
68  _site_index.erase(it);
69  return true;
70  }
71  }
72  return false;
73  }
74
75
76  bool Cluster::checkPresenceAndErase(Index bond) {
77  for (auto it = _site_index.begin(); it != _site_index.end(); ++it) {
78  if (*it == bond) {
79  _site_index.erase(it);
80  return true;
81  }
82  }
83  return false;
84  }
85
86
87  void Cluster::eraseSite(value_type index) {
88  auto it = _site_index.begin();
89  it += index;
90  _site_index.erase(it);
91  }
92
93  void Cluster::eraseBond(value_type index) {
94  auto it = _bond_index.begin();
95  it += index;
96  _bond_index.erase(it);
97  }
98
99
100 void Cluster::insert(const std::vector<BondIndex>& bonds){
101 _bond_index.reserve(bonds.size());
```

```
102 for(value_type i{} ; i != bonds.size() ; ++i){
103 _bond_index.push_back(bonds[i]);
104 }
105 }
106
107 void Cluster::insert(const std::vector<Index>& sites){
108 _site_index.reserve(sites.size());
109 for(value_type i{} ; i != sites.size() ; ++i){
110 _site_index.push_back(sites[i]);
111 }
112 }
113
114 /**
115 * Merge two cluster as one
116 * All intrinsic property should be considered, e.g., creation time of
     a cluster must be recalculated
117 * @param cluster
118 */
119 void Cluster::insert(const Cluster &cluster) {
120 if(_id > cluster._id){
121 cout << "_id > cluster._id : line " << __LINE__ << endl;
122 _id = cluster._id;
123 }
124 // older time or smaller time is the creation birthTime of the cluster
125 //     cout << "Comparing " << _creation_time << " and " << cluster.
     _creation_time;
126 _creation_time = _creation_time < cluster._creation_time ?
     _creation_time : cluster._creation_time;
127 //     cout << " Keeping " << _creation_time << endl;
128 _bond_index.insert(_bond_index.end(), cluster._bond_index.begin(),
     cluster._bond_index.end());
129 _site_index.insert(_site_index.end(), cluster._site_index.begin(),
     cluster._site_index.end());
130 }
131
132
133 /**
134 * Merge two cluster as one
135 * All intrinsic property should be considered, e.g., creation time of
     a cluster must be recalculated
136 * @param cluster
137 */
138 void Cluster::insert_v2(const Cluster &cluster) {
139 // older time or smaller time is the creation birthTime of the cluster
```

```
140 //     cout << "Comparing " << _creation_time << " and " << cluster.
     _creation_time;
141 _creation_time = _creation_time < cluster._creation_time ?
     _creation_time : cluster._creation_time;
142 //     cout << " Keeping " << _creation_time << endl;
143 _bond_index.insert(_bond_index.end(), cluster._bond_index.begin(),
     cluster._bond_index.end());
144 _site_index.insert(_site_index.end(), cluster._site_index.begin(),
     cluster._site_index.end());
145 }
146
147
148 void Cluster::insert_with_id_v2(const Cluster &cluster, int id) {
149 _id = id;
150 // older time or smaller time is the creation birthTime of the cluster
151 //     cout << "Comparing " << _creation_time << " and " << cluster.
     _creation_time;
152 _creation_time = _creation_time < cluster._creation_time ?
     _creation_time : cluster._creation_time;
153 //     cout << " Keeping " << _creation_time << endl;
154 _bond_index.insert(_bond_index.end(), cluster._bond_index.begin(),
     cluster._bond_index.end());
155 _site_index.insert(_site_index.end(), cluster._site_index.begin(),
     cluster._site_index.end());
156 }
157
158
159 std::ostream &operator<<(std::ostream &os, const Cluster &cluster) {
160 os << "Sites : size (" << cluster._site_index.size() << ") : ";
161 os << '{';
162 for(auto a: cluster._site_index){
163 os << a << ',';
164 }
165 os << '}' << endl;
166
167 os << "Bonds : size (" << cluster._bond_index.size() <<") : ";
168 os << '{';
169 for(auto a: cluster._bond_index){
170 os << a << ',';
171 }
172 os << '}';
173
174 return os << endl;
175 }
```

```
176
177
```

A.2.5 Percolation

file: **cluster.h**

```
1  a = -1
2
```

A.2.6 Utility

A.2.7 Tests

A.2.8 Main

Complete code for RSBD model on square lattice is available at https://github.com/sha314/SqLattice_RSBD or use the git link to clone the repository 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 $E = [ex_{approx} - \varepsilon, ex_{approx} + \varepsilon]$ where ε is a small number (usually ~ 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

$$ex^* = argmin_{ex \in E}\ std_{ex} \tag{C.1}$$

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*

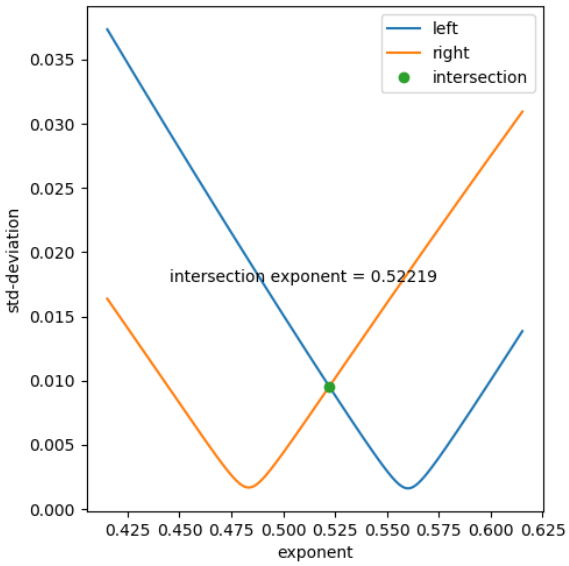


Fig. C.1 Best collapsing on either right or left
??

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 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 ε is usually ~ 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

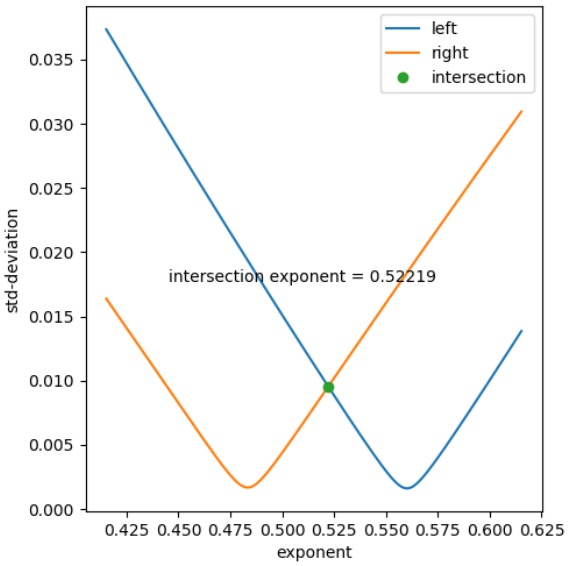


Fig. C.2 Minimizing exponent for scaling x-values

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.