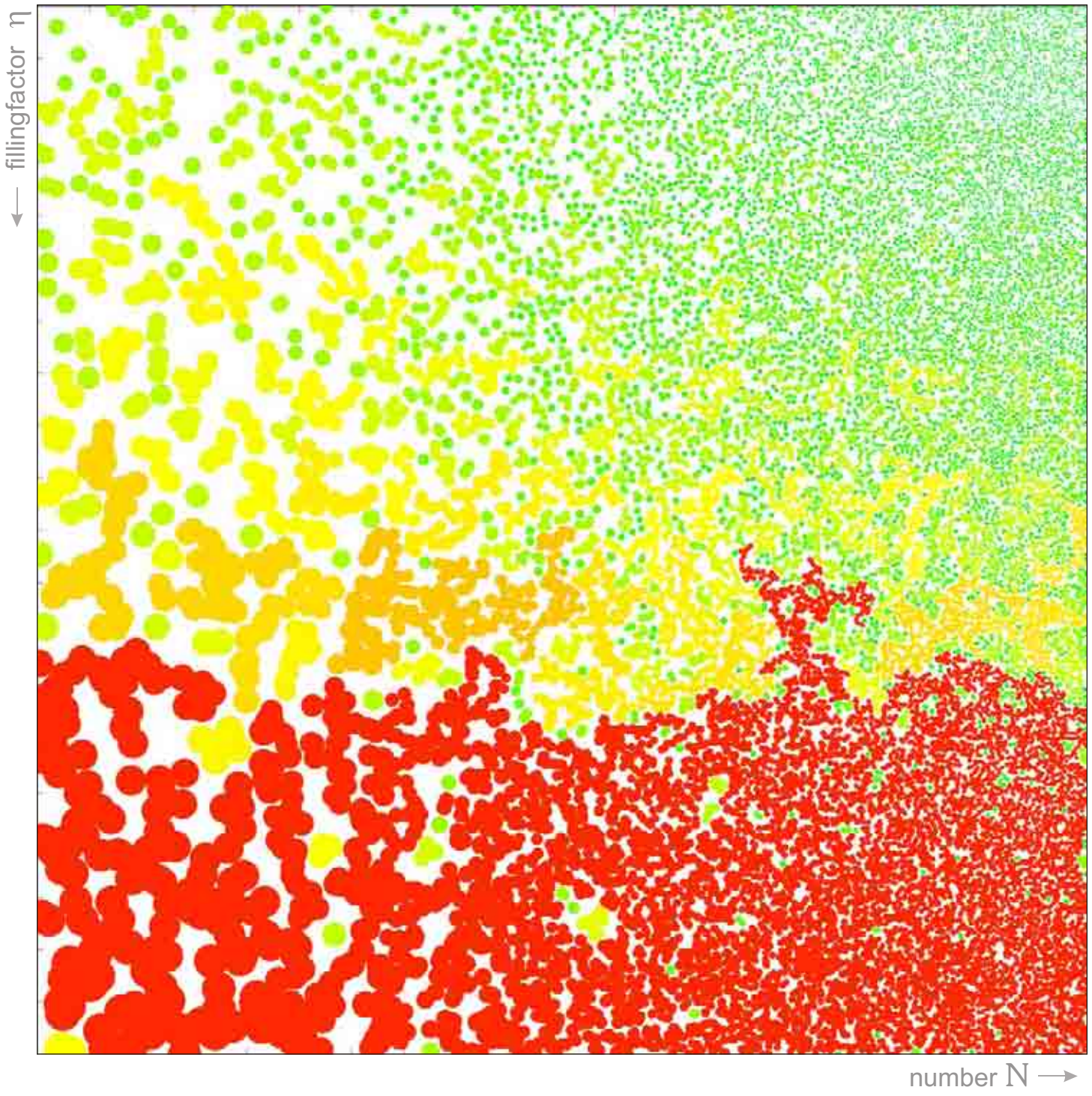
Andreas Krüger

Dimensionality in Continuum Percolation Thresholds



Diploma Thesis

Faculty of Physics, University of Bielefeld, Germany

Dimensionality in Continuum Percolation Thresholds
Diploma Thesis

Andreas Krüger
(AKrueger@Physik.Uni-Bielefeld.de, eMail@AndreasKrueger.de)
Physics Department, University of Bielefeld, Germany

April 8, 2003

Preface

When I became infected with *percolitis* during a lecture series of Helmut Satz and Philippe Blanchard, the whole concept of critical transitions soon started to grow clusters of associations on my own knowledge-map. In that time all spontaneous processes of the outside world seemed to be explicable by percolation.

It is a beautifully simple model - but it still provides us with astonishingly deep insights about collective behaviour of large systems in situations where we might without it only state:
Its more to the whole than only the sum of its parts.

Andreas Krueger, 4.12.2001

Questions

At my first contact with *continuum percolation*, a question was discussed between Philippe Blanchard and Daniel Gandolfo:

- Why are 4-, 5- and 6-clusters the building block of the critical cluster in continuum percolation?

Moreover, an upper critical dimension for percolation was mentioned, so I wondered:

- What happens to the percolation process in higher dimensions?
- How can one at all get used to the concept of higher dimensions? They are frequently mentioned, but our geometric imagination fails, so I wanted to do research on the *properties* of higher dimensional spaces to get some grip at the topology.

Reading about the backgrounds and listening to experts, those important "critical exponents" obviously draw a lot of attention because of their (use in) universality. Might there be a chance for a universal systems theory by cataloguing critical behaviour?

- It is widely accepted that continuum percolation is in same universality class as random percolation on the lattice - but that question still seems to lack numerical confirmation for the higher dimensions.

When I was asked by friends, what this toy-model percolation could be used for (which is a whole question in itself), often the question of "detecting percolation before it happens" was asked:

- Applications of Percolation Theory? What can it be useful for in the "Real World"?
- Is it possible to find a criterion for an upcoming percolation before it actually happens?

During the work, when my computer program started to produce the first numerical data, I didn't succeed in fitting it - it actually turned out that while in the vicinity of the threshold the polynomial fits with critical exponents help to classify the singularity, this question, however, is far more difficult than expected:

- Can one find fit functions for the observables within the whole percolation interval, also in the off-critical domain?

And of course, such a diploma thesis had a lot of practical aspects:

- How to do Science?
  My unfortunate inclination of re-inventing the wheel instead of consulting the literature is founded in a playful curiosity about the things themselves. On the other hand, it helped a lot to identify the *crucial* problems of the research object.
- Scientific Programming, Object Oriented Programming
  Handling a source code of now almost 10,000 lines needs a thorough abstraction and capsuling of the functional "onion shells".

- Personally, the most challenging (and thus rewarding) aspect of the work during the first phase was to create an algorithm from the scratch that shows a good time complexity - and still works in arbitrary dimensions.
- To use the computer, this learning machine as a learning-machine.

Most of these questions could not be solved within the limited time, but actually one success of the work was that deeper insight itself - it provided the distinction of the important aspects of that field.

After you have finished reading this work, I would very much appreciate your opinion what to do further research on.

Roadmap - How to Read This Work

At first, a quick *Introduction* into the concepts of percolation is given (chapter 1). The *Theory*-chapter is a collection of background material on the subject; spheres and boxes in higher dimensional spaces are studied, the main parameter "filling-factor" of continuum percolation is explained, some statistical quantities and the Poisson distribution are introduced and the concepts of Universality and Complexity are mentioned (chapter 2). Then the used *Algorithm* is explained in a rather abstract and non-technical way (chapter 3) and the visualization program is presented (chapter 4).

The most important results can be found in the *Thresholds*-chapter 5, the long way from masses of data to one single table of results can be followed.

The *Technicalities*-chapter 6 gives more technical details about the algorithm and some background on programming, a short manual for the sourcecode and hints about compilers. The last chapter 7 *Outlook* shows what could not be done due to lack of time; naturally such a work tries to follow one - more or less consistent - path; so since the beginning many interesting sideways and ideas had to be postponed to later studies.

The Appendix gives links to several interesting algorithms in Mathematica© (mainly used for the graphical visualizations throughout this work) and to the C++ program that was programmed to obtain the main results of this thesis.

Summary

Continuum percolation was chosen as a simulation model for phase transitions. Interesting and yet unknown topics were identified while establishing knowledge about the necessary fundaments and the breadth of this subject.

A dimension independent algorithm for clustering overlapping objects was developed and then implemented in C++. It was run for about a year to examine the algorithm itself, the observables in the phase transition interval and the position of the critical transition. Throughout this time, the program was improved and enlarged greatly.

The position of the critical transition (threshold) was measured in dimensions 1-11. An approximate fit function is given.

Finally, further interests and research ideas are collected and presented. (The C++ program is capable of producing much more data than what could be examined up to now.)

URL of this work on the WWW

You find this thesis (and the sourcecode and binaries) at

`www.AndreasKrueger.de/thesis/`

for your convenience, there are versions in PS (PostScript) and PDF (Portable Document Format).

Contents

Chapter 1

Introduction to Percolation

The mathematical model "percolation" is introduced to the reader, first by an intuitive picture and by naming the fields where percolation was applied to - then by a short overview about "lattice percolation", the usually examined model, where the clustered objects are confined to lattice points.

The concept of criticality is introduced and the most important observables are shown.

Then continuum percolation is explained; in contrast to the lattice model, here the density of spheres must be given by a "filling factor" which can be related to the occupied volume.

Finally, the construction of "clusters" by overlap is explained.

1.1 Percolation

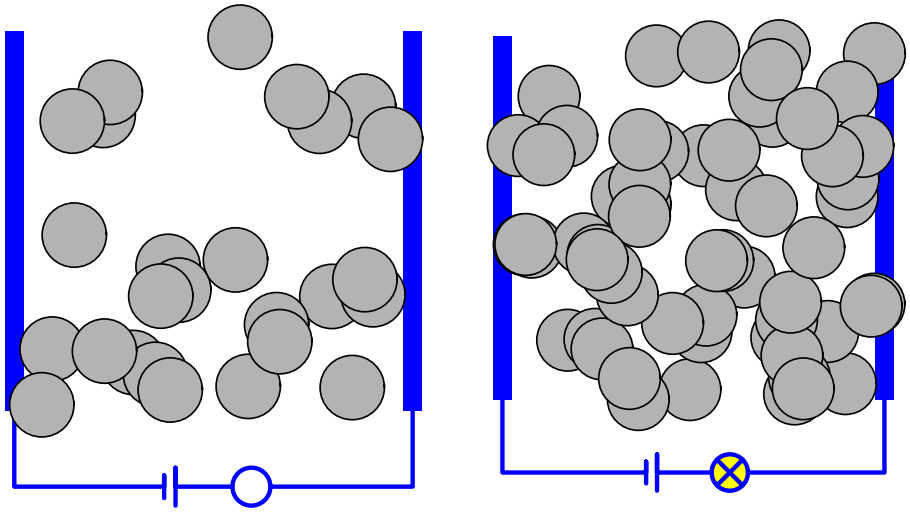


Figure 1.1: When a cluster of coins touches both electrodes, the light goes ON. Shown are 25 and 60 coins, the theoretical critical threshold in this setting would be 50 coins.

Imagine a table with two electrodes on opposite sides. If you connect the electrodes, a light is switched on. Now throw coins (of metal) randomly on that table. Nothing happens in the beginning, but suddenly, with the throw of one pivotal coin, the light goes on.
This happens when a *cluster* of overlapping coins spans across the table - when *percolation* appears.

This scheme is a simple model to study *phase transitions*, where a macroscopic property ("the light is ON or OFF") changes its state by means of microscopic influence ("the clustering of coins"). See figure 1.1 for illustration.

1.1.1 Applications

Percolation is one of the simplest mathematical models to generate non-trivial phase transitions geometrically. Because of its simplification and abstraction it could be adapted to a wide range of phenomena.

Some examples are epidemics like the spreading of HIV in the body and among people [5], galaxy creation, forest fires, boiling eggs [53, 6], the quark-deconfinement transition in QCD [45, 15, 49], Gaussian Primes [59], Fiber networks in paper-making [39], social models for crashes on the stock market [54] or the success of blockbuster-films [52] and many more. It helps to formulate a theory for the gelation of polymers under a critical dilution [56], in gelation science the principal thoughts of percolation (without naming it so) are even reaching back to 1941 [16].

1.1.2 Lattice Percolation

The system can be modelled in an easier way (and computed faster) on a grid, a lattice with definite neighbours and only one or no disc per lattice vertex.

It has been studied since Broadbent and Hammersley coined the term "percolation" for these models in 1957 [10, 23], and Sykes and Essam tried to find exact solutions to the critical density and exponents for 2 dimensions in 1964 [57]. Only a short overview will be given here, for a deeper treatment see [26], [53], [22], [32] and chapter 2.5.1 of this work.

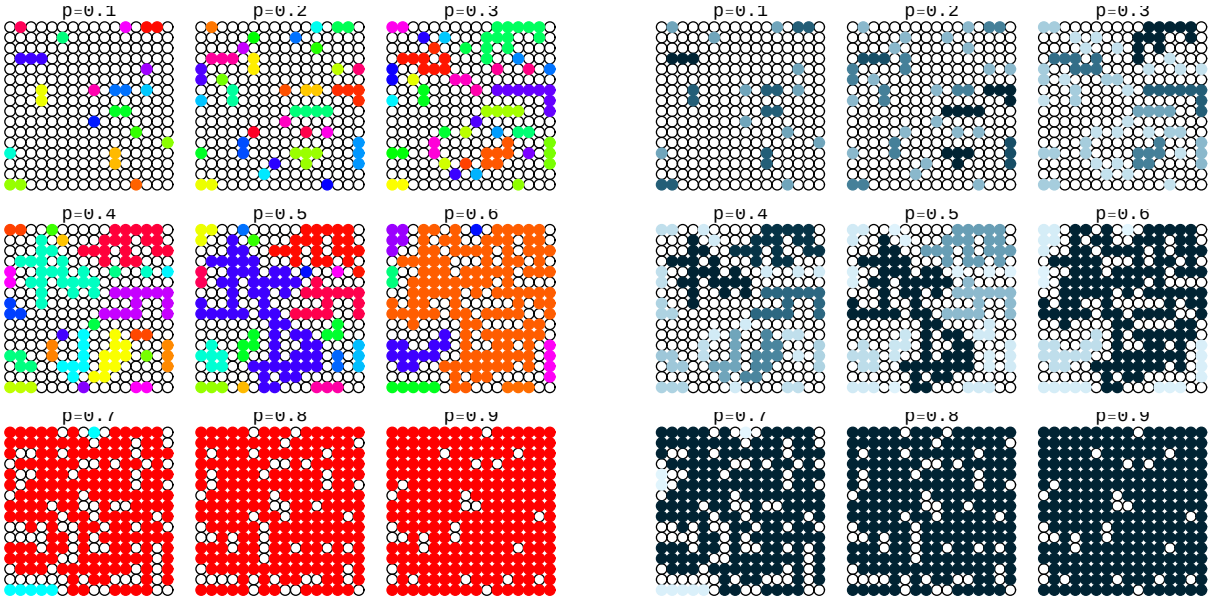


Figure 1.2: Site percolation in 2 dimensions. In a hypercube $\mathcal{HC}^{int}_{d=2}(l = 16)$ (a square of length 16 with integer coordinates) there are occupied sites with probability $p \in \{0.1, 0.2, \ldots, 0.9\}$. The left colouring depends on the *cluster numbers* (different clusters have different random colours), the right one on the *cluster sizes* (the bigger the cluster the darker the colour). Clearly you see undercritical and overcritical realizations: the critical (percolation) threshold is characterized by the first path that crosses the square from left to right.

Definition

For an intuitive understanding, first please look at the example realization of 2dimensional site percolation in figure 1.2.

Consider the hypercubic lattice $\mathbb{Z}^d$ in d dimensions. *Site percolation* is defined by choosing each site of $\mathbb{Z}^d$ to be *occupied* with probability p and *vacant* with probability $1 - p$, independent of all other sites. Two directly occupied neighbours are said to belong to the same (occupied) cluster. *Bond percolation* is defined accordingly, by choosing each bond that is connecting two sites of $\mathbb{Z}^d$ at Euclidean distance 1 (i.e. no diagonal connection) to be occupied with probability p and vacant with probability $1-p$, independent of all other bonds. Two occupied bonds ending in the same site are said to belong to the same (occupied)

cluster.
A *path* $A \leftrightarrow B$ is a set of occupied sites (or bonds) in a cluster connecting sites A and B.

Critical Transition

There is a distinct point (a certain density of occupation) at which a cluster *spans* any area from one side to the other. While the density is increased, this *critical transition* takes places suddenly:

1) There exists a *critical probability* p_c for which an *infinite path* exists with probability $P_\infty(p) > 0$ if $p > p_c$ and no infinite path exists with probability 1 if $p < p_c$.
2) Or let $\mathcal{HC}_d(l) \cap \mathbb{Z}^d$ be a hypercube in d dimensions with edge-length l, containing l^d vertices at integer coordinates. For growing hypercube size $l \to \infty$ there exists a *critical spanning probability* p_{Sp} for which a *spanning cluster* occurs with probability $P_{Sp}(p) > 0$ if $p > p_{Sp}$. A spanning cluster contains a path of occupied sites (bonds) connecting two opposite faces of the hypercube.
Properties of these critical probabilities are [32]:

$$p_{Sp} = p_c \tag{1.1}$$
$$0 < p_c < 1 \quad \text{for} \quad d \geq 2 \tag{1.2}$$
$$p_c = 1 \quad \text{for} \quad d = 1 \tag{1.3}$$

So the one dimensional case is a triviality, the system doesn't percolate if only one site (bond) is not occupied. For higher dimensions, there is a non-trivial critical occupancy probability p_c marking the transition point from an undercritical to an overcritical phase.
In computer simulations, definition 2 is more practical, because we will always look at *finite* hypercubes, rarely at the whole $\mathbb{Z}^d$.

Critical Exponents and Universality

The measured observables $\Omega(p)$, like the percolation probability $P_\infty(p)$, the correlation length $\xi(p)$, and the mean size of the finite clusters $\chi(p)$ have singularities (or singular derivatives) at the critical point; they behave like power laws in the vicinity of the critical threshold:

$$\Omega(p) \quad \sim \quad |p - p_c|^\omega \tag{1.4}$$

The set of exponents $\{\omega\}$ classify the phase transition; in the concept of *Universality*, two systems with identical exponents are in one *universality class*. There exist *scaling relations* that exhibit the relations of those exponents to each other. For further treatment please consult chapter 2.5.1

Off-critical Behaviour of Observables

Not only at the critical threshold, percolation models are interesting to study. Models for "real-life"-phase transitions have to take into account, that the system might not be in critical state yet - and still one likes to extrapolate to criticality to compare with other systems; e.g. the use of epidemic models is motivated by the hope to be able to predict the threshold without trespassing it (this would mean global infection). Moreover, the *saturation threshold* might be more important in some situations.
The whole transition from single-site clusters (unconnected points) to one single remaining cluster only ("Full-Connect") can be studied by measuring the relevant observables. In figure 1.3 simulation results are shown for 2dimensional site percolation in a square of length $l \in \{4, 16, 64, 256\}$, so for a number of sites $l^2 \in \{16, 256, 4096, 65536\}$. At the threshold $p_c \sim 0.593$ you see the (finite-size) signature of a sharp jump of the spanning probability $P_\infty(p)$ and a divergence of the mean size of the finite clusters $\chi(p)$. The size of the biggest cluster behaves very similar to the spanning probability, because the biggest cluster is likely to be the spanning one. Especially interesting for transition-prediction might be the **number of clusters** - at p_c it has already lowered again to a low number, the maximum being at $p_{maxNOC} \sim 0.272$.

An extensive study of percolation literature didn't yield functional expressions for these observables over the whole range of occupancy $p \in [0, 1]$ - so one of the most interesting questions has no answer yet: What is the functional form of the observables $\Omega(p)$?

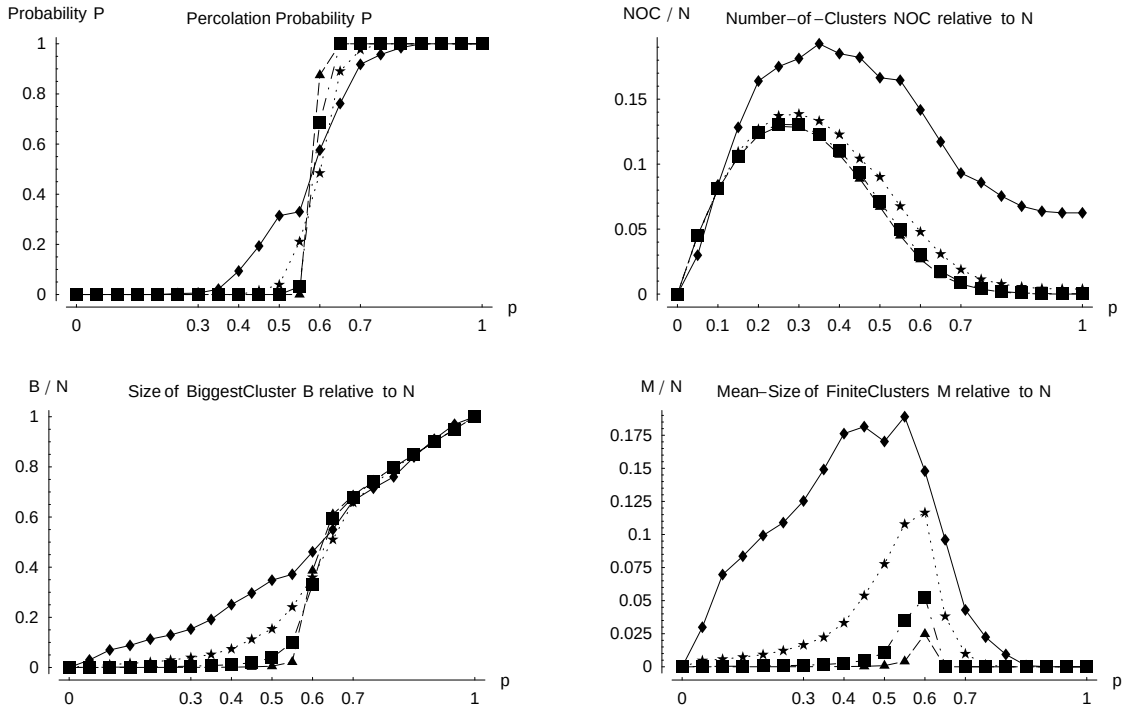


Figure 1.3: Observables of site percolation simulations in 2 dimensions for system sizes $N = l^2 \in \{16(diamonds), 256(stars), 4096(squares), 65536(triangles)\}$ for the whole intervall of occupation densities $p \in [0; 1]$: The spanning probability $P_\infty(p)$, and the relative (normalized with l^2) number-of-clusters, size-of-biggest-cluster and mean-size-of-finite-clusters. The $l \to \infty$ limit of the percolation threshold at $p_c = 0.593$ can be seen in the first plot, the $l \to \infty$ limit of the maxima of the number of clusters is at $p_{maxNOC} \sim 0.272$ (second plot).

1.1.3 Continuum Percolation (the Boolean Model, the Inverted Swiss-Cheese Model)

In contrast to the "classical" lattice model described above, a variation lies in the focus of this work. In *continuum percolation*, the discrete lattice vertices are replaced by d-dimensional *continuous coordinates* within a hypercube $\mathcal{HC}_d(l)$ with length l. Each coordinate is generated independently by a random point process and serves as the *center* of a decorating **(hyper)sphere** with radius R.
For a visualization in 2 dimensions take figure 1.1

The Filling Factor

To have a measure for the occupied volume density of the system that can replace the occupancy p of lattice models, a new parameter is introduced: The **filling factor** η in d-dimensions (see chapter 2.2)

$$\eta = \frac{\pi_d R^d N}{l^d} = \pi_d R^d \rho \tag{1.5}$$

$$= \pi R^2 \frac{N}{l^2} \quad \text{for the 2-dim case} \tag{1.6}$$

with sphere-radii R, number of spheres N, volume of d-dimensional hypercube l^d, point density ρ and the *d-dim unitsphere volume* π_d (see chapter 2.1.1 for more information about π_d):

$$\pi_d = \frac{\pi^{d/2}}{\Gamma(1 + \frac{d}{2})} \tag{1.7}$$

The role of the critical occupancy probability p_c is now taken by the critical filling factor $p_c \to \eta_c$. As an example, the numerical result for the 2-dimensional case is:

$$\eta_{dim=2}^{critical}(N \to \infty) = 1.128$$

Thus in 2 dimensions, in the "thermodynamical limit" $N \to \infty$, the sum of all volumes of discs thrown into a l^2-square has to be $1.128 \cdot l^2$ to get the critical state that a spanning-cluster becomes possible, i.e. a cluster that spans the entire domain for the first time.

The Occupied Volume Fraction

The filling factor sums all thrown volumes, so where ever there is overlap, occupied volumes are counted several times (That is the reason why η_c can be > 1). If one wants to know the actually occupied volume, a property of the Poisson point process (see chapter 2.3) gives a conversion:
The *occupied volume fraction* (without multiple overlap) is

$$\phi_c = 1 - \text{empty-volume-fraction} = 1 - \exp(-\eta_c) \tag{1.8}$$

so that for the 2-dimensional case we get a "occupancy" of about 68%:

$$\phi_c^{2dim} = 1 - \exp(-1.128) = 0.676$$

Clusters are Formed by Overlap

A *cluster* is herein defined as a set of overlapping hyperspheres. Two hyperspheres S_1 and S_2 with coordinates x and y and radii R_1 and R_2 *overlap* if the relative distance of their centers is smaller than the sum of their radii:

$$||x - y|| = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2} \quad \leq \quad R_1 + R_2 \tag{1.9}$$

Other conditions could be introduced, e.g. only spheres at overlapping distance with the *same colour* are to be connected; or other objects like ellipsoids to model easier overlap in certain dimensional directions, etc. The basic shape for all these variations, however, should be the hypersphere, because of its simplicity and symmetry.

Chapter 2

Theoretical Backgrounds

Percolation is such a simple model, so the computer program bases on very few theoretical facts only. This chapter is designed to shed light on some of the underlying theoretical facts of what is necessary to understand to do the simulations.

First of all, the clustered objects, the d-dimensional (hyper)spheres, are described by the volume they take in d-dim space. Therefore, a generalization of π is explained and examined, which poses a strange but interesting interpretatory question.
Then, after these "round" objects, the simplest "square" objects are studied: Hypercubes. They are d-dim generalizations of the regular 3-dim "dice"-cube, and while the construction stays the same (parallel and orthogonal lines between corner-vertices), their composition changes a lot with rising dimension, which is treated in detail.
This first section is completely devoted to a deeper understanding of the properties of higher dimensional spaces, because to imagine them geometrically seems to be a difficult task for us humans.

Then in section 2.2 the already mentioned *filling factor* is explained and related to other measure that are sometimes used in publications. All of them give the density of thrown objects, which is the main parameter that will be changed to drive the system from a non-percolating state to the critical state (= critical density) - and then above into the overcritical domain.

When coordinates are thrown independently at random, the resulting distribution is Poissonian, so section 2.3 gives some information about the Poisson distribution, which wasn't used throughout the simulation, but is a useful tool for analytical proofs and to calculate the mean number of neighbours per sphere at some filling factor.

Section 2.4 mentions backgrounds of statistics, especially the four scalars set (mean, variance, skewness, kurtosis) that can be used to characterize distributions.

In section 2.5, the concept of Universality is introduced and lattice results are given. It is essential for understanding what should be done next with the existing computer program.

Finally, section 2.6 gives an overview about computational complexity - a framework in which it becomes clear why it was such an important task to improve the algorithm (see next chapter 3).

2.1 Properties of d-dimensional Euclidean Spaces

We are embedded in a 3-dimensional space and our brains are optimized to its properties after aeons of evolution. For most of us this leads to unsurmountable difficulties to imagine higher-than-three dimensional spaces geometrically. They can be treated formally in an easy way - as seen in equation 1.9 or the algorithm part of this thesis - but one likes to have a more intuitive understanding of the properties. Especially "square" and "round" objects are to be understood - so let's look at boxes and spheres.

2.1.1 Hyperspheres

A hypersphere of radius R contains all points within distance R from the center c:

$$B_d(R) := \{x \in \mathbb{R}^d : ||x|| \leq R\} \tag{2.1}$$
$$B_d(c,R) := c + B_d(R) = \{x \in \mathbb{R}^d : ||x - c|| \leq R\} \tag{2.2}$$

The volume of this d-dimensional hyperspheres is proportional to the d-th power of its radius: $V \sim R^d$. The proportionality factor is the volume of a hypersphere with radius 1:

$$V_d(R) = V_d(1) \cdot R^d \tag{2.3}$$

Let's call it π_d.

The Unitsphere-Volume π_d

This volume of a unitsphere (2.4) is found by d-fold integration of the sphere indicator function (see [17]). Of course, this approach is only possible for non-zero, positive integer dimensions; however, the resulting formula can be analyzed for non-integer (and even negative or imaginary) dimensions.
As you see in figure 2.1, it exhibits a strange dimension-dependence resulting from the antagonism of a exponential and a Gamma-Function.

$$\pi_d = \frac{\pi^{d/2}}{\Gamma(1+\frac{d}{2})} \tag{2.4}$$

We all know the π_d for 1, 2 and 3 dimensions, they are the prefactors of the volume formulas $V_1(R) = 2 \cdot R$, $V_2(R) = \pi \cdot R^2$ and $V_3(R) = \frac{4\pi}{3} \cdot R^3$, but the π_d-curve keeps rising for the 4- and 5-dimensional hypersphere, while it curves down towards zero for dimensions higher than 6.
The exact maximum is found at the root of

$$\frac{\Gamma'(1+\frac{d}{2})}{\Gamma(1+\frac{d}{2})} - \log(\pi) \stackrel{!}{=} 0 \tag{2.5}$$

which evaluates to $d_{maxvol} = 5.256946$ with volume $\pi_{5.256946} = 5.277768$.

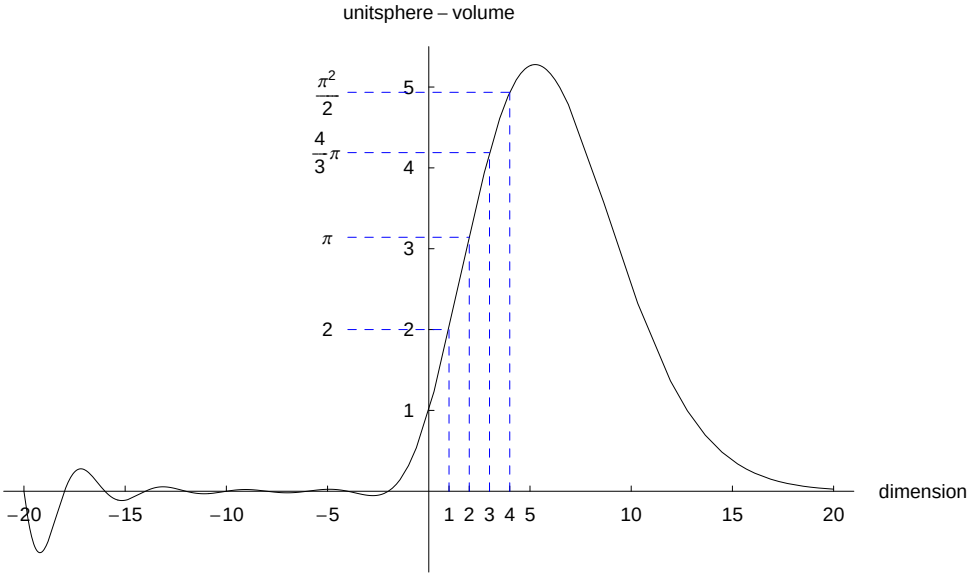


Figure 2.1: The volume of a unitsphere in d dimensions (equation 2.4) - only the non-negative part makes sense geometrically (values for negative d are obtained by continuation of the complex Γ-function). The one-, two-, and three-dimensional volume are the well known 2, π and $\frac{4\pi}{3}$ prefactors of the line-, disc-, sphere-volume formulas. The maximum of the curve is at "dimension" $d_{maxvol} = 5.256946$.

A Hypercube and the Inscribed Hypersphere

Another question is the "volume-ratio of round to square objects" in d-dimensions. An (oriented) hyper*cube* of length $2R$ and center at origin contains all points with each absolute coordinate below R:

$$\mathcal{HC}_d(2R) \quad := \quad \{x \in \mathbb{R}^d : |x_i| \leq R, \forall i \quad \text{with} \quad 1 < i \leq d\} \tag{2.6}$$

The volumes of the hypercube with length $2R$ and the inscribed hypersphere with radius R have the ratio

$$\begin{aligned} \frac{V_d^{sphere}(R)}{V_d^{cube}(2R)} &= \frac{\pi_d \cdot R^d}{(2R)^d} \\ &= \frac{\pi_d}{2^d} \end{aligned} \tag{2.7}$$

In one dimension a "unitsphere" of volume (=length) 2 is a line (= 1dim "hypercube") of length 2, so the ratio is naturally 1. A zero-dimensional "cube" is also the same as a "sphere" (both are points), so the ratio must be 1, either! For 2 and 3 dimensions, the ratio is $\pi/4 = 78.54\%$ and $\pi/6 = 52.36\%$.
For rising positive dimensions, (2.7) is a falling number above $d_{maxratio} = 0.476583$, where the "sphere" is actually 1.03869 times bigger than the "cube".
Please have a look at figure 2.2 to see this surprising behaviour; again the warning, (2.4) was derived only for non-zero, positive integer dimensions - how can those continuous properties be interpreted?

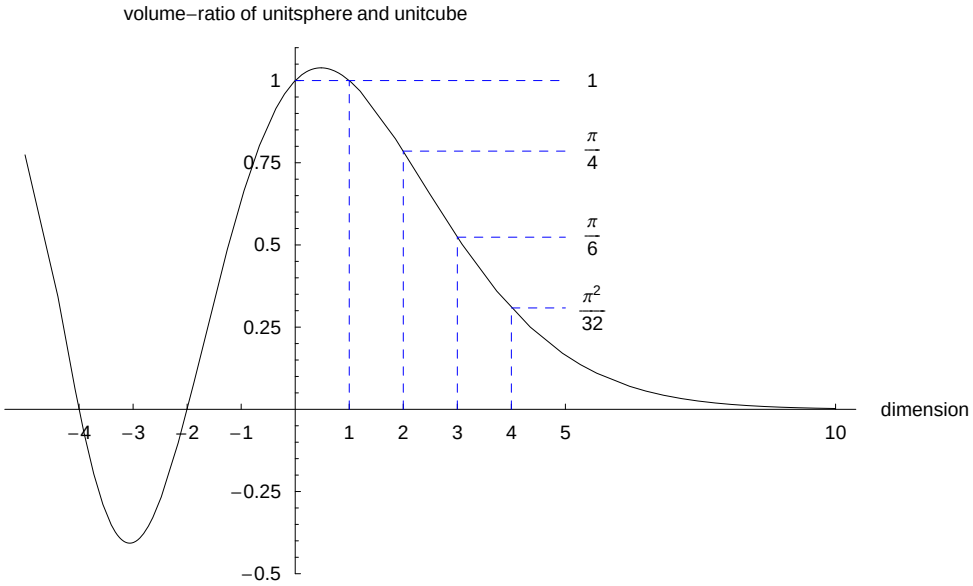


Figure 2.2: The volume-ratio of hypersphere to circumscribed hypercube in d-dimensions. For 0 and 1 dimensions, "cube" and "sphere" are identical, a 2 dimensional circle takes only 78% of the circumscribing square. For rising dimension, the hypersphere quickly "disappears" inside the hypercube - so it takes less and less d-dimensional space, e.g. for 10 dimensions only 0.25%.

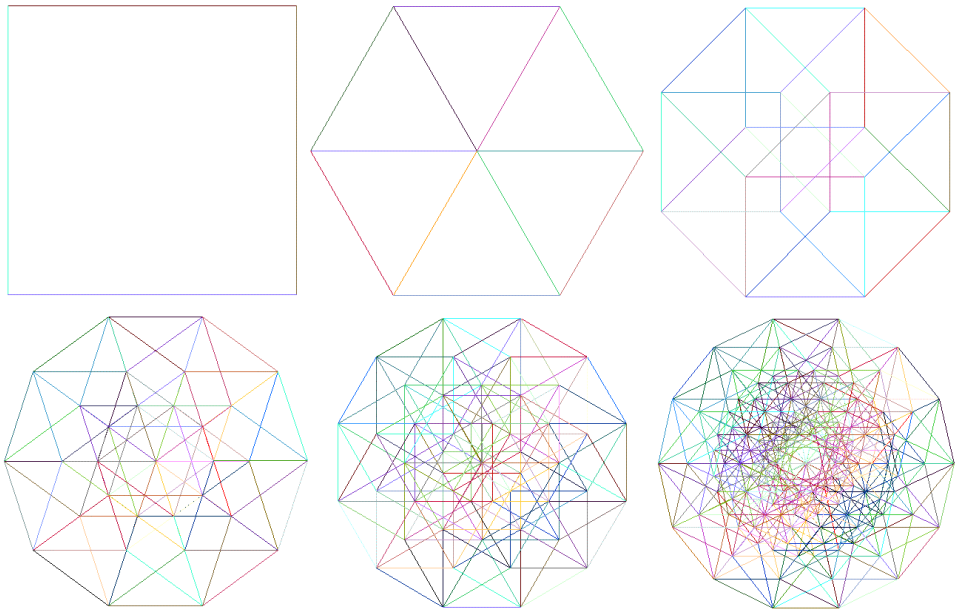


Figure 2.3: Hypercubes in 2-7 dimensions. Thanks for these images to Jonathan Bowen [8].

2.1.2 Hypercubes and Boxes - and the Number of Subobjects

Another way to imagine higher-dimensional spaces is to have a closer look at hypercubes, because they are very simple objects and still contain a lot of the features of their embedding space.

The (2dim) square table, on which the coins were thrown on in figure 1.1, in 3 dimensions becomes a cube that contains balls, in 4 dimensions a 4-dim hypercube with 4-dim hyperspheres, etc.

There is an excellent visualization in Java by Mark Newbold at [34], which projects a 4-dimensional hypercube onto 3-dimensional space, using red-blue glasses or stareo-projection.

A d-dimensional box $[0, a_1] \times [0, a_2] \times \ldots \times [0, a_d]$ is a *(hyper)cube*, if all edges have the same length: $a_1 = \cdots = a_d \equiv a$. The volume is $V = \prod a_i \equiv a^d$ with a being the length of one one-dimensional *edge* of the cube. Please have a look at figure 2.3 as a visualization of the following descriptions.

Boxes in d dimensions

A *d-dimensional box* is limited in the $2d$ directions by (d-1)-dimensional boxes, call them the $2d$ *sides*. All d edges that come together in one *vertex* (or *corner*) are mutually perpendicular.
A possible construction rule is given by:

1. take a (d-1) dimensional box
2. "pull" it in a new direction (perpendicular to the $d-1$ existing directions) for a length of a
3. The d-dimensional box contains the whole d-dimensional volume that was touched by the pulled (d-1)dim box

This description allows us to tell how many corners, edges, faces and volumes a 3-dimensional box has: $8, 12, 6, 1$. The box was generated by pulling a point (1) to a line (2,1), the line to a square (4,4,1) and the square to a box (8,12,6,1).

A Box is Made of Lower Dimensional Subboxes

Such a series of numbers of *sub-(hyper)cubes in d-dimensional hypercubes* should help to classify them; the above construction rule gives a recursive formula for these numbers.
The number of D-dimensional subcubes of a d-dimensional hypercube is:

- 2 times the number of the (D)-dimensional subcube of a d-1 dimensional hypercube plus the number of the (D-1)-dimensional subcube of a d-1 dimensional hypercube
- zero for $D > d$
  (e.g. there is no 4dimensional hypercube inside a 3dimensional)
- 2^d for $D = 0$ (there are $\underbrace{2 \cdot 2 \cdots 2}_{d}$ vertices; a point is 0-dimensional)

With this recursive "algorithm", table 2.1 can be constructed:

$$\mathrm{N}_{d,D} = 2\mathrm{N}_{d-1,D} + \mathrm{N}_{d-1,D-1} \tag{2.8}$$
$$\mathrm{N}_{d,0} = 2^d \tag{2.9}$$
$$\mathrm{N}_{d,D} = 0 \qquad \forall D > d \tag{2.10}$$

For example the number of vertices of a 4-space box is 2 times the number of vertices of the figure in the next lower space ($2 * 8 = 16 = 2^4$). The number of edges is 2 times the number of edges plus the number of vertices from the figure in the next lower space ($2 * 12 + 8 = 32$). The number of faces is 2 times the number of faces plus the number of edges in the figure in the next lower space ($2 * 6 + 12 = 24$). The number of 3-volumes is 2 times the number of 3-volumes plus the number of faces of the figure in the next lower space ($2 * 1 + 6 = 8$). The number of 4-volumes is 2 times the number of 4-volumes plus the number of 3-volumes from the figure in the next lower space ($2 * 0 + 1 = 1$).

Table 2.1: The number of D-dimensional subcubes of a d-dimensional hypercube up to d=4

D / d	0 vertices	1 edges	2 planes	3 cubes	4 4D-cubes
0	1				
1	2	1			
2	4	4	1		
3	8	12	6	1	
4	16	32	24	8	1

Some properties of these numbers can be seen straight-forward from the construction:

- $\mathrm{N}_{d,d} = 1$
  (trivial: there is always *one* d-dimensional object in a d-dimensional object)
- $\mathrm{N}_{d,d-1} = 2d$
  (the mentioned $2d$ *sides* that are (d-1)dimensional)

The Number of Subobjects

The above construction rule leads to the number of D-dimensional subcubes of d-dimensional hypercube. It is easy to show that this simplifies to one formula:

$$\mathrm{N}_{d,D} = \frac{2^{d-D}}{D!} \cdot \prod_{i=0}^{D-1} (d-i) \tag{2.11}$$
$$= \binom{d}{D} \cdot 2^{d-D} \tag{2.12}$$

The number of D-dim subobjects of a d-dim hypercube is thus given by (2.12):
"The number of different D-element-subsets of a d-element-set, multiplied by 2 to the power of the remaining dimensions" [1].

Easy to understand, because the D-dim subcube needs any D of the d provided dimensions (the binomial) as the orthogonal edge-directions from one corner-vertex - and it has 2 parallel versions in each of the $d - D$ codimensions.

[1] These d-dim sub-boxes can be used as the so-called simplicial complexes in algebraic topology invented by Euler in order to characterize surface (and more generally space) topology through simple invariants[21].

Properties of Those Numbers

Now we can try to understand high dimensional hypercubes by examining the behaviour of those numbers of subcubes as a function of the dimension.

Table 2.2: The number of D-dimensional subcubes of a d-dimensional hypercube is $\mathrm{N}_{d,D} = 2^{d-D} \cdot \binom{d}{D}$. Printed in boldface is the most frequent subcube in each dimension.

	D=0	1	2	3	4	5	6	7	8	9	10	11
d	vertices	edges	faces	cubes	4D-cubes	...						
0	1											
1	2	1										
2	**4**	**4**	1									
3	8	**12**	6	1								
4	16	**32**	24	8	1							
5	32	**80**	**80**	40	10	1						
6	64	192	**240**	160	60	12	1					
7	128	448	**672**	560	280	84	14	1				
8	256	1024	**1792**	**1792**	1120	448	112	16	1			
9	512	2304	4608	**5376**	4032	2016	672	144	18	1		
10	1024	5120	11520	**15360**	13440	8064	3360	960	180	20	1	
11	2048	11264	28160	**42240**	**42240**	29568	14784	5280	1320	220	22	1

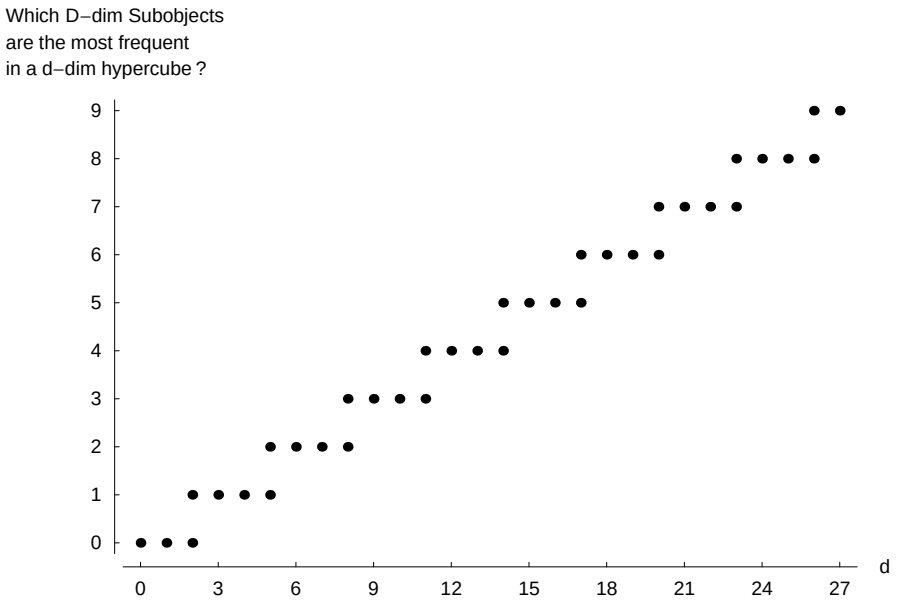


Figure 2.4: The dimensionality of the most frequent subobject $D_{max}(d)$) rises linearly with the dimension d of the hypercube. For example, while the most frequent subcube of the regular 3-dim dice are the 12 edges, in a 9-dim and 10-dim hypercube, the 3-dim subcubes are the most frequent.

Looking at table 2.2 one notices that the maximum number slowly moves to the right, e.g. for 9d-hypercubes the most frequent subobject is a 3D-cube with 10.5 cubes per vertex ($5376/512 = 10.5$). The position of the maximum is shown in figure 2.4. It rises linearly with the dimension d of the hypercube.

$$D_{max}(d) = \begin{cases} \text{Integer}(d/3) \wedge \text{ Integer}((d+1)/3) & \text{if } (d+1)\text{modulo } 3 = 0 \\ \text{Integer(d/3)} & \text{if } (d+1)\text{modulo } 3 \neq 0 \end{cases} \tag{2.13}$$

The height of the maximum, the actual number $\mathrm{N}_{d,D_{max}(d)} \cdot \frac{1}{2^d}$ of subobjects per hypercube vertex in d dimensions, is plotted in figure 2.5. For higher dimensions the asymptotic behaviour is exponential growth:

$$\mathrm{N}_{d,D_{max}(d)} \cdot \frac{1}{2^d} \sim \exp\left(0.4052 \cdot d - 4\right) \sim 1.5^d \tag{2.14}$$

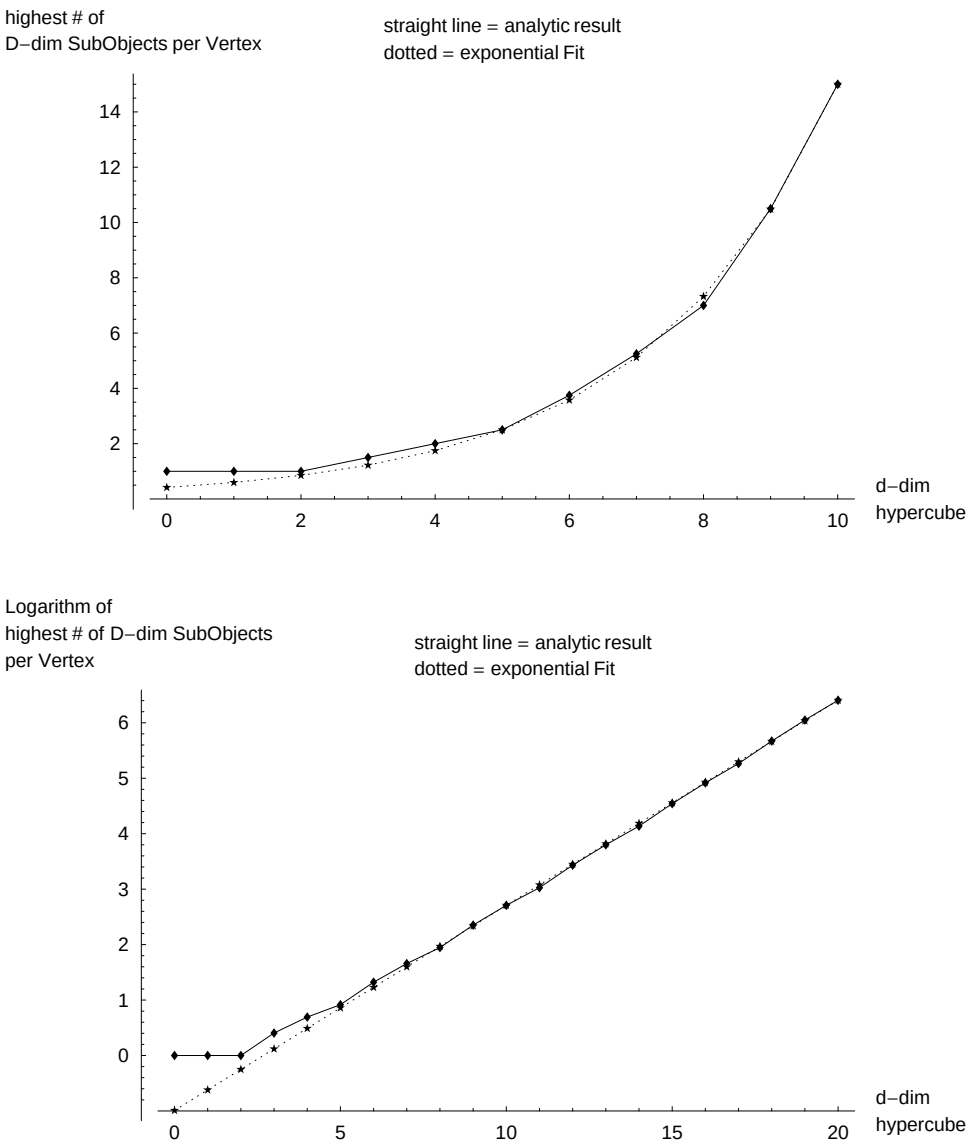


Figure 2.5: The height of the maximal number of subobjects per hypercube-vertex $\mathrm{N}_{d,D_{max}(d)} \cdot \frac{1}{2^d}$ up to dimension 10 plotted linearly and up to dimension 20, plotted logarithmically. The dotted line is the fitted asymptotic exponential (2.14)

2.2 The Critical Filling Factor η_c

As mentioned in chapter 1.1.3, the filling factor is the ratio of the volumes of all thrown objects to the provided space:

$$\eta = \frac{\pi_d R^d N}{l^d} = \pi_d R^d \rho \tag{2.15}$$

with radii R, number of spheres N, volume of d-dimensional hypercube l^d, point density ρ and the d-dim unitsphere volume π_d introduced in chapter 2.1.1. The main target of this thesis was to compute the critical thresholds η_c in d-dimensions.
The filling factor η can be varied in different ways, by varying N, R or l. This translates into throwing more spheres, expanding each sphere or compressing the hypercube.
In the following, the second way was chosen (varying the size of the spheres). The size of the hypercube was set to one $l = 1$, the number of spheres was kept constant in $N \in \{39, 78, 156, 312, 625, 1250, 2500, 5000, 10000, 20000, 40000, 80000, 160000, 320000\}$ and the radius was changed continuously to create the desired filling factor or to locate the exact filling factor, at which a cluster spans the space.

2.2.1 The Number Density n_c

As mentioned before, in chapter 1.1.3, the critical filling factor η_c can be transformed monotonically into other parameters, like the critical *number density* n_c [29]:

$$n_c = (2R)^d \frac{N}{L^d} \tag{2.16}$$

$$= \frac{2^d}{\pi_d} \eta_c \tag{2.17}$$

which is the total thrown volume of all *squares* in which the spheres are inscribed.
For example, with the numerical estimations $\eta_c^{2dim} = 1.1282$ and $\eta_c^{3dim} = 0.3416$, the number densities are $n_c^{2dim} = 1.4365$ and $\eta_c^{3dim} = 0.652$
The critical filling factor then becomes

$$\eta_c^{2dim} = \frac{\pi}{4} n_c \sim 0.785 \cdot n_c \tag{2.18}$$

$$\eta_c^{3dim} = \frac{\pi}{6} n_c \sim 0.524 \cdot n_c \tag{2.19}$$

$$\eta_c^{4dim} = \frac{\pi^2}{32} n_c \sim 0.308 \cdot n_c \tag{2.20}$$

$$\dots \tag{2.21}$$

For that prefactor, it is the ratio of box volume to sphere volume; please see the equation (2.7) and the plot 2.2 in chapter 2.1.1.

2.2.2 The Occupied Volume Fraction ϕ_c

Another measure for the density is the *occupied volume fraction* - it can be found with the knowledge that the Poisson distribution $\mu^{a\rho}$ (see chapter 2.3) gives the probability that an area a in a point process of density ρ is *empty* $(k = 0)$ with $\mu^{a\rho}(k = 0) = \exp(-a\rho) = \exp(-\eta)$ [42].

$$\phi_c = 1 - \exp(-\eta) \tag{2.22}$$

For example, with the numerical estimations $\eta_c^{2dim} = 1.1282$ and $\eta_c^{3dim} = 0.3416$, the occupied volume fractions are $\phi_c^{2dim} = 0.6764$ and $\phi_c^{3dim} = 0.2894$. So compared to site percolation on the lattice, the threshold in continuum percolation in terms of density is bigger in 2dim and smaller in 3dim.

2.3 Poisson Distribution

The Poisson distribution is a "rare events" approximation of the binomial distribution

$$P_B(k) = \binom{N}{k} p^k (1-p)^{N-k} \tag{2.23}$$

for $N \to \infty$, $p \to 0$ with $Np = \lambda = \text{const}$, so in practical situations for many events N and small probability p .

A point process with density $\rho = \frac{N}{l^d}$ generates independently N vectors that consist of d random coordinates uniformly distributed in $[0, l]$. Now introduce test areas with volume a and ask for the number of points in a. Naturally there are areas with less and areas with more points in it. The relative number of areas with k points is given by the Poisson distribution $\mu^{a\rho}(k)$:

$$\mu^{a\rho}(k) = e^{-a\rho} \frac{(a\rho)^k}{k!} \tag{2.24}$$

with ρ=point density, a=volume of test area, k=number of points in test area.

So the probability to have no points in the test area is $\mu^{a\rho}(0) = \exp(-a\rho)$, for one point $\mu^{a\rho}(1) = a\rho \exp(-a\rho)$, for two points $\mu^{a\rho}(2) = \frac{(a\rho)^2}{2} \exp(-a\rho)$, etc.

2.3.1 Properties of the Poisson distribution

As the $\mu^{a\rho}(k)$ give a *probability* to find k points in a, all relative numbers $\mu^{a\rho}(k)$ sum up to 1:

$$\sum_k \mu^{a\rho}(k) = 1 \tag{2.25}$$

The *mean number* of points $\overline{k}$ in test area a is naturally proportional to the volume of the area and the density of the point process: $a \cdot \rho$

$$\overline{k} = \sum_k k \cdot \mu^{a\rho}(k) = a\rho \tag{2.26}$$

2.3.2 Critical Number of Neighbours

A useful application of these Poisson properties is the answer to the question [20] about the mean number of neighbours of one sphere at criticality, an important number e.g. for spreading algorithms [58].

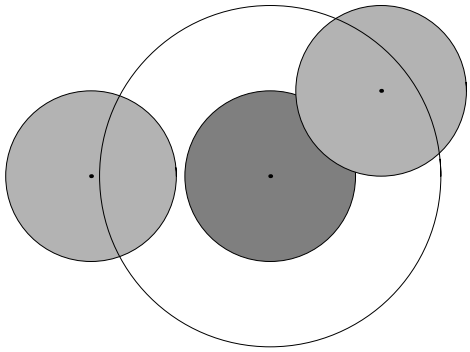


Figure 2.6: The centers of all neighbours (light gray) of a sphere (dark gray) lie in a "corona" of double radius.

Please see illustration 2.6 to understand the following condition for overlap (in this formulation only valid for equal-sized-spheres):

A sphere of volume $a = \pi_d R^d$ is embedded in a "corona" of volume $A = \pi_d (2R)^d = 2^d \cdot \pi_d R^d$, in which all centers of overlapping spheres must lie. It is the so-called "excluded volume" $V_{ex} = 2^d V$.

The Poisson distribution now tells us about the probability $\mu^{A\rho}(k)$ of k points in area A. For the mean

number of neighbours $\overline{NoN}$, it has to be summed over all probabilities $\mu^{A\rho}(k)$ to get the mean for one sphere and to be averaged over all spheres N:

$$\begin{aligned}
\overline{NoN} &= \frac{1}{N} \sum_{i=\text{all N spheres}} \overline{NoN_i} \\
&= \frac{1}{N} \sum_{allNspheres} \sum_{k} k \cdot \mu^{A\rho}(k) \\
&\overset{(2.26)}{=} \frac{1}{N} N \cdot A\rho \\
&= 2^d \cdot \pi_d R^d \cdot \rho \\
&\overset{(2.15)}{=} 2^d \eta
\end{aligned} \tag{2.27}$$

So with the numerical result for criticality in 2 dimensions $\eta_c = 1.1282$, we get:

$$\overline{NoN}_{2dim,critical} = 4\eta_c = 4.5128$$

In chapter 5.3.5, the simulation results for the other thresholds in d-dimensions will be expressed in terms of the critical mean number of neighbours.

2.4 Statistics

2.4.1 Mean Value and Fluctuation

Let M_i be n measurements. The mean value is

$$< M >= \overline{M} = \frac{1}{n} \sum_{i=1}^{n} M_i$$

the variance (or **fluctuation**) of the mean is

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (M_i - \overline{M})^2$$

This can be written as

$$\begin{aligned}
< (M - \overline{M})^2 > &= < M^2 - 2M\overline{M} + \overline{M}^2 > \\
&= < M^2 - 2\overline{M} < M > + < \overline{M}^2 > \\
&= < M^2 > -2\overline{M}^2 + \overline{M}^2 \\
&= < M^2 > -\overline{M}^2 \\
&= < M^2 > - < M >^2
\end{aligned} \tag{2.28}$$

Since at the transition point the observables fluctuate strongly, the maximum of the composite observable $< M^2 > - < M >^2$ might serve as an indicator for a transition point.

2.4.2 Skewness and Kurtosis

Usually the first and second moment of distributions are calculated (the *mean* and the *variance*). However, there are more (see table 2.3), namely the *skewness* (third central moment) and the *kurtosis* (fourth central moment), the latter frequently used in econometrics to describe "fat tails" of e.g. revenues probability distributions. For an example calculation of the four moments for a flat distribution of random numbers, please have a look at chapter 6.2.5.

In our case, small system sizes result in spanning-filling factor-distributions that are far from being symmetric, but have a steep increase for lower values (skewness $>> 0$) and have a strong "excess" (high kurtosis, a wider peak than a Gaussian).
Please have a look at the $N = 78$ and the $N = 20,000$ spanning-filling factor distributions in figures 5.5 and 5.6 in chapter 5 to see this property of small system sizes.

Table 2.3: The central moments of a distribution, their definitions and interpretations

mean	$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$	the mean value after n realizations $\{x_i\}$
variance	$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$	the variance around the mean value; the square of one standard-deviation if Gaussian
skewness	$\frac{1}{\sigma^3 \cdot n}\sum_{i=1}^{n}(x_i - \overline{x})^3$	if the distribution is crooked to the left or right; for symmetric distributions == 0
kurtosis	$\frac{1}{\sigma^4 \cdot n}\sum_{i=1}^{n}(x_i - \overline{x})^4$	"flatness"; length of tails compared with height of center - or width of the peak; for Gaussian == 3

2.4.3 The Standard Error of the Mean

The *standard deviation* σ of a distribution is the square root of the variance σ^2.

The *standard error of the mean* (SEM) [28, 30] is designated as: σ_M. It is the standard deviation of the sampling distribution of the mean. It is defined as:

$$\sigma_M = \frac{\sigma}{\sqrt{n}} \tag{2.29}$$

where σ is the standard deviation of the original distribution and n is the sample size.

So, if one needs a statistical error bar for the quality of a measurement, the standard deviations of all four moments have to be multiplied again by $\frac{1}{\sqrt{n}}$, so that a slow focussing of the 'measurement $\pm$ standard-deviation' takes place for increasing sample size.

2.5 Universality

The hope is that discriminating different universality classes helps to distinguish between fundamentally different systems and processes.

Systems are classified by the "steepness" of their singularities, which can be expressed by *critical exponents* of a polynomial fit around the critical threshold. Systems with identical or similar critical exponents are then regarded as being in the same *universality class*.

Important to notice is that similarity of those exponents is reported from models in totally different fields and length-scales. The classification of phase transitions might lead to a deeper understanding of universal structures and processes in nature. One system of a universality class can be studied and results be applied to other models of the same class.

Still, there are non-universal properties like the actual *location* of the singularity itself, which depend sensitively on the formulation of the model, but which are most important for practical purposes.
One example are site and bond percolation on different lattice types (square, triangular, honeycomb, ... lattices). While the thresholds differ a lot, the critical exponents are only dependent on the dimension of the embedding space.

A second example is the similarity of lattice and continuum percolation. While the *location* of the critical threshold in 2dim lattice site percolation $p_c = 0.593$ is very different from the location of the threshold in 2dim continuum percolation $\eta_c = 1.1282$ (or $n_c = 1.4365$ or $\Phi_c = 0.6764$), the *critical exponents* are widely accepted to be identical.

Most of the percolation models belong to one of the simplest classes, the *universality class of random, or uncorrelated, percolation* [25].

2.5.1 The Critical Exponents β, γ, ν, D

The Percolation probability $P_\infty(p)$, the correlation length $\xi(p)$ (= linear length of the biggest cluster), and the mean size of the finite clusters $\chi(p)$ behave like power laws in the vicinity of the critical threshold

[53]:

$$P_\infty(p) \sim (p-p_c)^\beta \quad \text{for } p > p_c \tag{2.30}$$

$$\chi(p) \sim \begin{cases} (p_c-p)^{-\gamma} & \text{for } p < p_c \\ (p-p_c)^{-\gamma'} & \text{for } p \geq p_c \end{cases} \tag{2.31}$$

$$\xi(p) \sim \begin{cases} (p_c-p)^{-\nu} & \text{for } p < p_c \\ (p-p_c)^{-\nu'} & \text{for } p \geq p_c \end{cases} \tag{2.32}$$

usually with $\nu = \nu'$ and $\gamma = \gamma'$.

For the first 6 dimensions in (hyper)cubic lattice percolation, Stauffer and Aharony give a table of the exponents (and the position of the thresholds $p_c(dim)$) in [53];

dim	p_c^{site}	p_c^{bond}	β	γ	ν
1	1	1			
2	0.592746	0.5	$5/36 \sim 0.139$	$43/18 \sim 2.39$	$4/3 \sim 1.33$
3	0.3116	0.2488	0.41	1.80	0.88
4	0.197	0.1601	0.64	1.44	0.68
5	0.141	0.1182	0.84	1.18	0.57
6 -ϵ	0.107	0.0942	-$\epsilon/7$ +1	$\epsilon/7$ +1	$5\epsilon/84$ +0.5
7	0.089	0.0787			
∞			1	1	1/2

The last line $dim \to \infty$ represents results for the "Bethe lattice" (or "Caley tree"). It can be regarded as an infinite dimensional structure, because in each "generation" of the branching process that results in this tree structure, the number of added vertices is proportional to the number of already existing vertices inside; so the "surface" of the structure is proportional to the "volume", a feature of infinite dimensional objects.

Please have a look at figure 2.7 to see the dimension-dependence of 3 exponents and their $dim \to \infty$ limit. There seems to exist an upper critical dimension $d^* = 6$ above which the critical exponents no longer change but have attained constant "mean field" values.

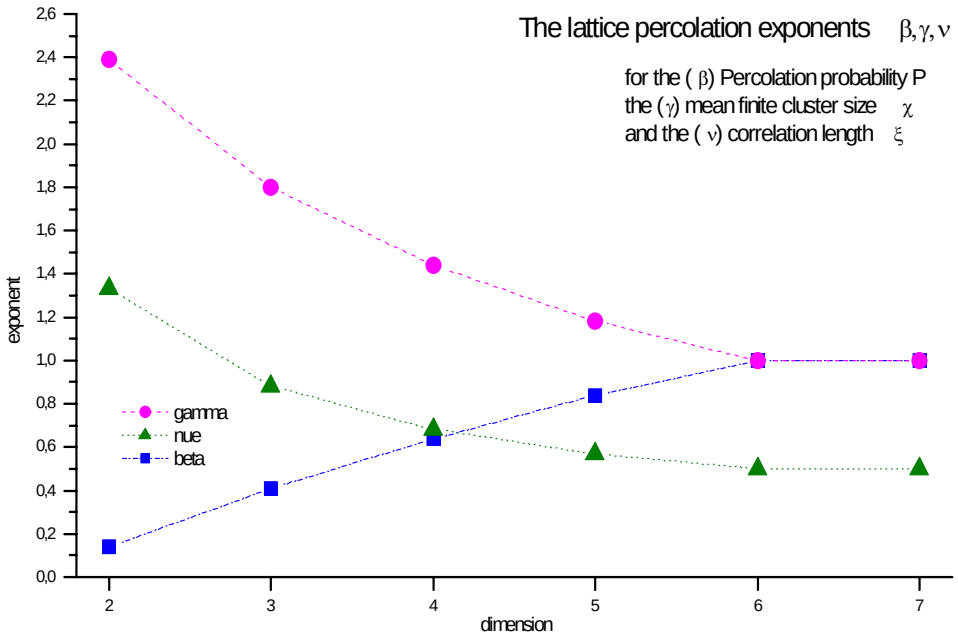


Figure 2.7: The critical exponents β, γ and ν for lattice percolation in the first 6 dimensions and their dim$\to \infty$ limit. A plot of the table in chapter 2.5.1. The source of the data is Stauffer & Aharony [53].

The exponents are not fully independent, however. Only two of the critical exponents can be set independently. The theory of phase transitions gives relations between the exponents β, γ and ν, the embedding

dimension d and the fractal dimension D (of the infinite cluster at criticality, e.g. $D = 91/48$ for $d = 2$ and $D = 2.50$ for $d = 3$), the so-called "scaling relations", two of them being e.g.:

$$d\nu = 2\beta + \gamma \tag{2.33}$$

$$d = D + \frac{\beta}{\nu} \tag{2.34}$$

2.5.2 Finite Size Scaling

An important remark is the total lack of phase transitions in finite systems, because the finite attributions of all the constituents can't sum up to a singularity in an observable - a finite sum of finite summands is finite!

With growing system size N, however, one might get a finite-size signature of an upcoming singularity for the so-called "thermodynamical limit" $N \to \infty$, a drift of the measurements towards infinity or towards a non-smooth curve at the - then critical - point. In this way, it becomes possible to extrapolate from finite simulations without - to an infinite system with critical transitions.

On the lattice, it is known how to do a finite size scaling of the system order parameter Θ, e.g. the largest cluster mass, by using the lattice length L as the system size, with which the order parameter has to be rescaled [36]:

$$\Theta \sim L^{-\beta/\nu} F[(p_c - p)L^{1/\nu}] \tag{2.35}$$

For continuum percolation, one of the questions is if in d-dimensions L can be set as $L = N^{1/d}$ to do a similiar finite size scaling.

2.6 Time Complexity of Algorithms

The *complexity* of a problem can be understood in many ways. In computer science, mostly three aspects of complexity of algorithms are distinguished:

- time complexity
  The limiting behavior of the execution time of an algorithm when the size of the problem goes to infinity [66].
- space complexity
  The limiting behavior of the use of memory space of an algorithm when the size of the problem goes to infinity [66].
- algorithmic complexity,
  the "easiness of writing" [66] an algorithm that produces the wanted result. There might be an easy-to-code, but slow and memory-consuming version - and a rather difficult solution to the same problem that saves time and/or memory.

While the third criterion is a qualitative one, the first two can be measured and be denoted in big-O notation (or big-Omicron notation), giving the dependency on the size of the problem, usually expressed as the number n of objects treated by the algorithm.

2.6.1 A Simple Example for Computational Complexity

Look at the example: "What is the sum of n numbers?"
The algorithmic complexity of the following version is something like "three lines of Mathematica© code", if the numbers are stored in `array`:

```
n=Length[array];
sum=0;
Do[sum += array[i], {i,1,n} ] ;
```

The time and space complexity can be analyzed without actually measuring; because of the n accesses to the `array` and the $n - 1$ summations (each taking some *constant* CPU time for one access and one

addition), the execution time is **linearly proportional** to the number n of numbers, so we can write in the big-O notation:

$$\text{time}(n) \sim O(n)$$

If you only ask about the necessary memory for the actual summation (if the n numbers are not taken from the `array`, but generated during the summation, e.g. randomly), we see that only 2 variables have to be stored: *sum* and i, so the memory usage **will not grow** if we increase the number n of numbers. We can write

$$\text{memory}'(n) \sim O(1)$$

If in our analysis, we include the memory necessary to *store* all the n numbers, of course, we need memory for an `array` with a length **proportional to** n:

$$\text{memory}''(n) \sim O(n) + \text{memory}'(n) = O(n) + O(1) = O(n)$$

2.6.2 A Short Overview about Frequent Complexity Functions

The following table gives some possible asymptotics, their names and and some examples for *time complexity*:

$O(1)$	constant	random access to an element of an array
$O(\log(n))$	logarithmic	random access in a sorted list by binary search
$O(n)$	linear	random access in a linked list; sequential search
$O(n\log(n))$	"n log n"	sorting a list by a good algorithm
$O(n^2)$	quadratic	sorting a list by bubble sort
$O(n^k)$	polynomial	(probably [70]:) is the number n a prime number?
$O(\exp(n))$	exponential	
or $O(c^n)$	exponential	obviously with $c > 1$
$O(n!)$	factorial	
$O(n^n)$		

some remarks:

- Two different solutions to the same problem might be in the same complexity class $O(f(n))$, but still one is faster than the other, because the proportionality constant is smaller.
- There are hard problems for which one can be happy about approximations with polynomial time complexity, but usually algorithm should perform better than with quadratic asymptotics.
- Sorting problems are often (lower) bounded by O(n log n)
- Storing data in trees gives the possibility to find one leaf in O(log n) time
- It is still an open problem to answer the question whether the classes of polynomial time algorithms and non-polynomial time algorithms are identical or not [21] ("The P versus NP Problem" was even declared one of the "Millenium Prize Problems"[11]).

As you will see in chapters 3.1.1 and 3.2.4, for this work it was urgently necessary to create an algorithm that solves the "d-dimensional sorting problem" in less than quadratic time. Otherwise it would have been impossible to study larger systems and repeat each simulation many times to get a good statistics.

Chapter 3

Algorithm

The main goal of this chapter is to describe different approaches to identify clusters among N spheres in d-dimensional spaces and how to measure the main observables.

This "core-algorithm" can then be used for different questions:

- Where is the exact location of the threshold?
- What is the analytical content (the functional dependence) of the observables in the whole range of filling factors $\eta \in \{0, \infty\}$?

Those parts of the program are described later in chapter 6.5; for more technical details about the data structure and other technicalities, please consult the chapter 6.

However, if you are not interested in programming or if you lack the knowledge of C++, nevertheless this chapter 3 is worth reading, because it will explain the approach without many technical terms.

3.1 Complexity $O(N^2)$

3.1.1 The Naive Algorithm

The easiest program is often the best, the advantage of an obvious proof for correctness makes up for disadvantages in e.g. time complexity (see chapter 2.6).

For our question, in the beginning I could find a very easy solution, using *recursion*, i.e. a subroutine that calls itself until a break-out condition is reached - in this case, until all possible neighbours have been visited.

This algorithm visits all possible bonds between N points:

$$N-1+N-2+\cdots+2+1=\sum_{i=1}^{N-1} i=\frac{1}{2}N(N-1)\sim N^2 \tag{3.1}$$

and checks if a pair of points is close enough to have overlapping spheres (see equation (1.9)). It starts with sphere number 1 as the comparer, and when it finds an overlapping neighbour (that is not already part of the cluster), it calls itself with that neighbour as the comparer. So the tree of connectivity is traversed "depth-first" in this essentially two-lines-of-code algorithm[1]:

```
SUB neighbour (this,first,last,clusternumber)
   this.clusternumber = clusternumber;          // set visited flag to actual clusternumber
   FOR other = first to last                    // visit all candidates (LOOP-start)
      IF other.clusternumber == 0 THEN          // check only unvisited candidates for overlap
         IF overlap(this, other) THEN           // if overlap, then it is a neighbour
            CALL neighbour (other, first, last,
                            clusternumber)      // recursion
   NEXT other                                   // LOOP ends here
END neighbour
```

[1] Throughout this text, a pseudocode is used for computer programs (if it's not Mathematica©). Pseudocode is easy to understand even without programming experiences. Please consult appendix A.1 for explanations

That subroutine marks the one sphere `this`[2] as part of the actual cluster, and then checks for overlap with all those candidates between sphere number `first` and sphere number `last` (that are not found yet to be part of a cluster). If it finds an overlap, it calls itself with the found neighbour (*recursion*!).

The main routine calls that subroutine for all spheres:

```
SUB find_all_clusters (N)
   clno=1                                  // the first cluster is named ‘‘1’’
   FOR i=1 TO N                            // LOOP: each sphere could be the first of a cluster
      IF i.clusternumber==0 THEN           // if it’s not visited already
         CALL neighbour (i, i+1, N, clno)  // traverse the cluster depth-first
         clno ++                           // increase the clusternumber for the next cluster
   NEXT i                                  // LOOP ends here
   return clno                             // return the next clusternumber
END find_all_clusters
```

The main program calls this subroutine for all those spheres `i` from 1 to N that are not already in a cluster (so still `i.clusternumber==0`). This condition is always checked first to prevent unneccessary overlap-checking (see chapter 6.4.2). After one recursion has crawled back up, there is no sphere left between `first=i+1` and `last=N` that could be part of the cluster that started with `i`, so a new cluster with an increased clusternumber begins. At the end, this iteration returns the number of found clusters + 1.

This program works well, is easy to understand, and up to now, it functions as a validation of other, more-complicated algorithms. Because of the 2 nested loops that visit all possible bonds, however the time complexity is $O(\frac{N(N-1)}{2}) \sim O(N^2)$. For small N this is no problem and my very first results were actually calculated using just this simple approach. The need for bigger systems however, brought up the idea of a divide-and-conquer-approach described in chapter 3.2.

I have not found an alternative to search in this 'naive' way in the vicinity of a sphere, so until now, that naive clusterfinder is still used in the 'inner loop' of the main algorithm, inside the small cells that the whole problem is broken into (these 'areas' or 'cells' of the divide-and-conquer typically contain 50-250 spheres).

3.1.2 The First Transition - Find a Hold in Parameter Space

When one starts with a new problem, even with a tested algorithm, it is not clear right from the start if and where a signature of a transition can be found. The first thing to do is to scan a huge part of the possible parameter space between far-undercritical and far-overcritical domains.

So at first, $N = 10000$ coordinates were thrown in a box $[0, 10000] \times [0, 10000]$ and then after each measurement, the radii of all spheres were increased a little bit, scanning a wide intervall for R; in figure 3.1 the intervall $R \in [54; 70]$ is shown.

After the cluster-finder program (described above) was executed, it was easy to identify the *size of the biggest cluster*. As you see in figure 3.1, for this realization, there was a clear transition for the biggest clustersize at a radius about $R \sim [59; 64]$, which translates into a filling factor

$$\begin{aligned} \eta &= \pi_2 R^2 \frac{N}{l^2} \\ &= \frac{\pi}{10000} R^2 \\ &\sim [1.09; 1.29] \end{aligned}$$

which actually includes the best know estimate $\eta_c = 1.128$.

What has happened at the jump at radius $R = 62.6$? (At least) two clusters that were of a size < 4600 for radius $R = 62.5$, with the increase of radii, merged into one cluster of size 7900. This merging of existing structures into a much bigger structure because of a slight change in radius is the reason for this sharp transition.

[2] the simplified writing `this.clusternumber` means: access the property `clusternumber` of the `this`'sphere in the array of spheres

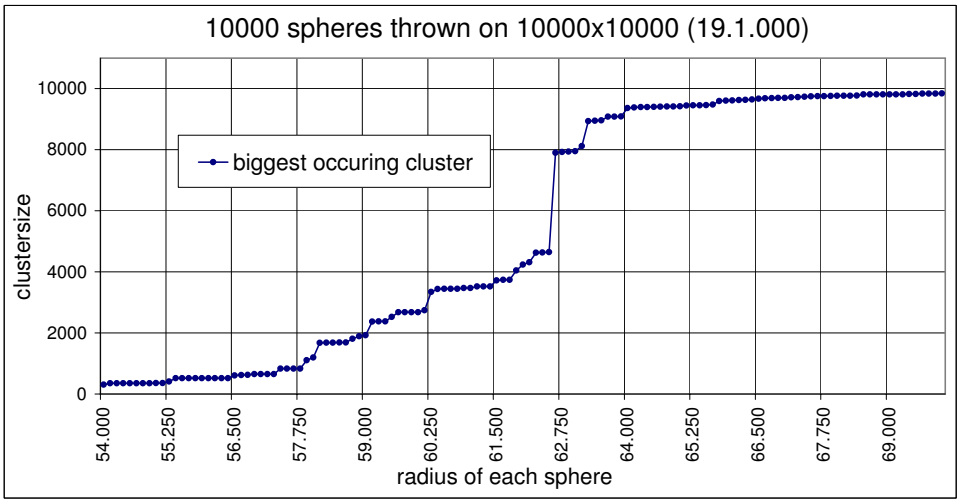


Figure 3.1: The first found transition for one realization of 10000 random coordinates in $\mathbb{R}^2 \cap \mathcal{HC}_{d=2}(l = 10000)$ and decorating spheres with 128 different radii. It takes places somewhere between $\eta = 1.09 \quad (R = 59)$ and $\eta = 1.29 \quad (R = 64)$.

3.1.3 Smooth the Curve

A single realization of thrown coordinates is only *one sample* of what can happen at that specific parameter-combination. Hence, the outcome cannot be generalized, the curve in figure 3.1 shows the properties of that one sample, but not the general properties of the parameter-combination.

To smooth that curve and find the general behaviour, we have to perform statistics by repeating measurements and calculating the **average** value of an observable. Please have a look at figure 3.2 to see the smoothing effect of 40 repetitions. Each point in that diagram represents the average over 40 runs with the same parameter, but each time newly thrown coordinates.

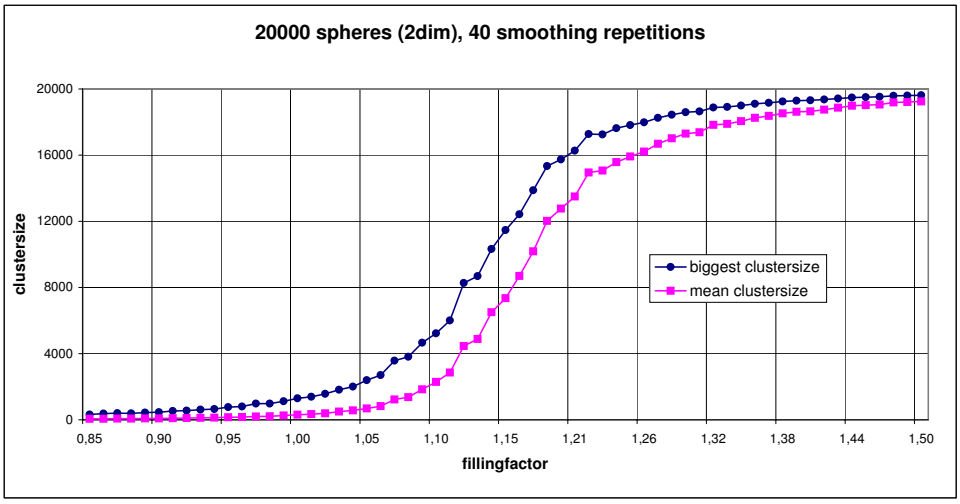


Figure 3.2: This diagram contains *mean values* of the two observables "biggest-clustersize" and "mean-clustersize". For each of the 60 different radii ($\rightarrow$ filling factors), 40 samples were created and analyzed, so this diagram contains 2400 realizations.

3.1.4 Dimensionality Dependence

The algorithm exhibits the expected 'time$(N) = c \cdot O(N^2)$'-behaviour, the proportionality factor c could be lowered by a factor 2 using some programming tricks (see chapter 6.4).

In higher dimensions, much bigger vectors have to be handled, the overlap-checking with the 'length-of-the-distance-vector'-calculation becomes slower. As a rule of thumb, in the C++ program, the time goes linear with the dimension, e.g. it is doubled for the step $dim = 2$ to $dim = 8$ when $N = 1000$:

$$\text{time}(2dim) \sim 1/2 \cdot \text{time}(8dim)$$

3.1.5 Recursion vs. Iteration

In most computer science literature like [50], recursions are described to be a) stack-consuming and b) slow. The underlying reason is the necessity (for the compiled program) to store all local variables of a recursion-level on the stack until that recursion-level is reached again on the way up. Problem a) can be solved by increasing the stack size (a rule of thumb that seems to be enough for the branching rate and depth within this special divide-and-conquer program up to 500,000 spheres is 2MB stack-size), but problem b) seemed to be worth considering.

Recursive programming gives an advantage in algorithmic-complexity (the code is easier to write and understand) - but if the time-complexity suffered severely from that, it would be a bad trade-off. An interesting *iterative* formulation of the same method ("check all spheres with all others once") including a Fortran code can be found in [69, Stoddard]:
At first, a link-table `L(i)` is initialized with `L(i)=i`. Then, in nested loops, the spheres are checked with each other - and whenever there is an overlap, the two link-table-entries are *swapped*. The resulting link-table consists of closed loops for each cluster that can be followed `L(i)` → `L(L(i))` → `L(L(L(i)))` → ... until `= i` again (then the loop is closed).

I have implemented the Stoddard-iteration and measured the needed time against the recursion - there is no difference! Obviously the C++ - compilers treat that recursive code in a way that gives no disadvantage over iterative formulation!

3.2 Divide-and-Conquer

The naive-count shows a $O(N^2)$ behaviour in time-complexity because each sphere is checked with *all* others. In this easy-but-too-simple approach, the obvious locality of the problem is not exploited at all; some spheres are very far away from each other and could just not be neighbours - but they are still checked for overlap!

But how can one care only about close enough neighbours? The spheres have to be sorted somehow!

The first idea to sort them globally in a linear way (by their distance from the origin) before finding the clusters, gave a little advantage for high densities, but almost none around the critical threshold or below (see figure 3.3). Moreover, a linear sorting can only improve one direction - so while it might do some good for 1-dimensional clustering, it is ineffective for higher dimensions.

What we need is sorting in several directions simultaneously. The resulting structure is a set of many small areas containing only a few spheres that are already very close to each other - so each overlap-check can be done with a higher probability for `true` - and most bonds (of far-away spheres) are not checked anymore.
Those many, but small problems are then solved using the naive-cellcount described above. Such algorithms are called "*divide-and-conquer*".
Then the counted areas have to be combined: Whenever two spheres from neighbouring boxes overlap, their clusters merge into one bigger cluster. At some stage the clusternumbers of all spheres have to be relabeled. A natural choice is to let the lowest clusternumber survive for the combined cluster [3]. So whenever two small clusters merge, all spheres get the lower clusternumber of the two.

Compared to the one-area naive-count, one will find the same solution much faster - but only above a certain total number of spheres, because the dividing and combining takes time itself. Moreover, the optimal balance between too many spheres per area (the disadvantages of the naive-count) and too many areas with only few spheres (too much time for dividing and combining) had to be found experimentally because it depends on the divide-and-combine algorithm and the effectiveness of the implementation.

[3] like in the Hoshen-Kopelman algorithm described below

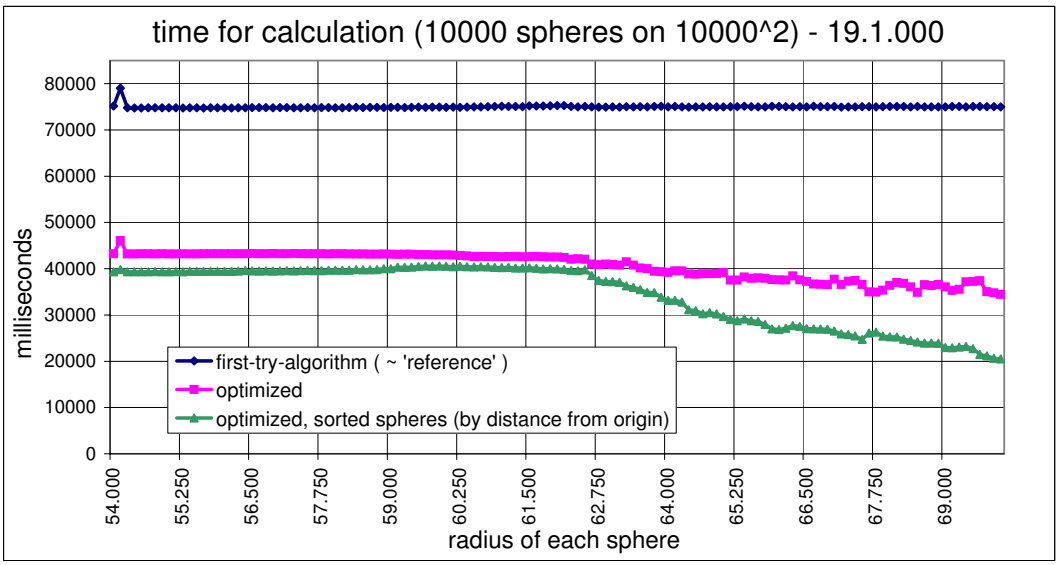


Figure 3.3: Execution time of the naive count of figure 3.1 for (a)[top] and (b)[middle] randomly chosen and (c)[bottom] linearly sorted spheres in dim=2. (b) and (c) are optimized code versions (see chapter 6.4). One can clearly see that linear sorting doesn't help much in the interesting range around the critical point, only above.

3.2.1 Combine

Let's consider two neighbouring 'areas' or 'cells' in which all clusters are found independently (their clusternumbers are disjunct). When these two cells are to be combined, not all spheres are relevant, only those inside two adjacent 'edges' of width R (see figure 3.4). The width of the edge-stripes must be two times the radius of the biggest sphere. Two clusters that merge into one will be called *neighbour-clusters*.

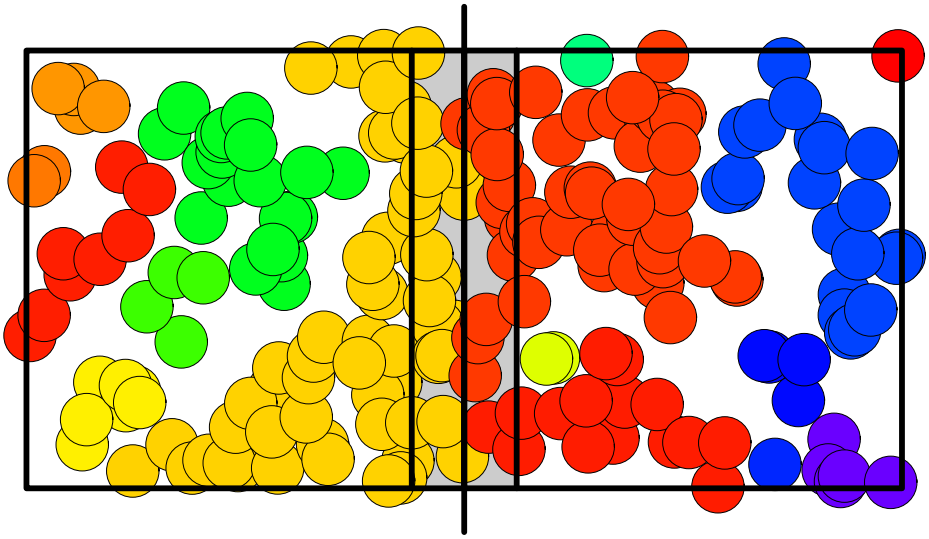


Figure 3.4: When combining two areas, only 'edgespheres' must be checked for overlap. They lie in a stripe of width $2R$ along the to-be-connected edges of the cells.

An array of lists of neighbour-clusters is created. The two lists of edgespheres are nestedly iterated and all possible bonds are checked for overlap. Whenever two spheres overlap, their clusternumbers are added to the list of neighbours of *both clusternumbers* in the array.

Now this cluster-connectivity structure has to be followed and merged to find all connected clusters that will be identified. This information is stored in a second array of lists in which the smallest occurring clusternumber is the index.

Now we can relabel all spheres of the neighbour clusters to their new clusternumber, then relabel the clusters and thus reduce the clustertable by using the now-empty clusternumbers.

Criticism

The described method works well, regardless of the dimension. Of course, the performance decreases for higher dimensions because most spheres then lie in the $(d-1)$dimensional 'surface' of the adjacent cells. Apart from implementing totally different, more efficient algorithms (see chapter 3.4), my program could be further optimized especially in the "combine"-routine, because of this high-dimensions property. (see chapter 7 for ideas).

An obvious improvement (that I see now looking into the structure) of the last step described above, would be to relabel the clusters and spheres in one step, similiar to the Hoshen-Kopelman clustertable-approach that does only one linear relabeling at the end (chapter 3.4.3).

3.2.2 Splitting

While developing my algorithm, the dimension-independence was a crucial request that demanded a thorough abstraction level.
How are the cells stored? Which cells are neighbours? Which surfaces of the cells are adjacent with the neighbouring "edge-stripes"? Which surfaces are at the outside of the big cube and don't need to be combined? Remember chapter 2.1.2: A d-dimensional hypercubic cell has $2d$ surfaces, it is surrounded by 3^d-1 other cells. So for a 2dim central cell ('central' means: not at some of the surfaces), there are 8 cells around that are relevant for combining, but for 10dim, central cells have 59048 neighbouring cells to be checked. And for open boundaries, this is only true for central cells, because e.g. a cell in one of the 2^d corners of the overall hypercube only has 2^d-1 filled neighbour cells, the remaining 3^d-2^d cells are outside the used area of coordinates and thus empty. Of course, it gets rather complicated to think about all other combinations in between corner- and central-cells.

Again for algorithmic simplicity, I decided *not to store the cell-structure explicitly*, but to partially solve the problem on every level inside a recursion that splits each cell into halves until a breaking condition is reached. The total number of splits can be chosen, e.g. if 8 splits are wanted ($\rightarrow 2^8 = 256$ cells) in 3dim, the recursion will do 3, 3 and 2 splits in the 3rd, 2nd and 1st dimensional direction. So in the recursion, two variables are varied: the number of splits and the dimensional direction.

The routine is called "counters::dividecount" (please see the sourcecode in chapter A.3). At first it is called with the whole set of unclustered spheres, the total number of wanted splits s is given and the starting direction for splitting is $D = dim$.
Until s/D splits are done, the routine divides the spheres by their D-coordinate in the middle of the intervall and calls itself for the "left" and the "right" part of it, with an increased splits-counter. Then, on a recursion level on which sufficient splits for this direction are done, the routine calls itself for the next lower direction $D \rightarrow D-1$. After the last direction ($D = 1$) is reached and all splitting recursions are opened, the remaining spheres in this cell are analyzed for clusters, using the naive clusterfinder of chapter 3.1.1.

On the way up in the recursion, the combining takes place. Because the splitting coordinate of the halves is known on each recursion level, it is easy to call the combine-routine of the previous chapter 3.2.1.

To *imagine* how this algorithm works in 2 dimensions: A sheet of paper is folded $s/2$ times along one direction, then $s/2$ times along the other direction. The small "cells" are counted naively - and during the unfolding of the paper the adjacent cells are checked for to-be-connected clusters within edges along the fold-rim.

Criticism

The algorithm is very flexible and robust, especially because the work to store all cells and their cutting-surfaces is done implicitly by the recursion - but therefore it needs some space on the stack to store all intermediate data. It is not hard-coded for a certain dimensionality, so the algorithm works in any dimension.

The big disadvantage over "boxing" is the frequent check of coordinates. A single sphere is checked $2\frac{s}{dim}$ times in each coordinate direction, once for cutting in halves and once if being an edgesphere.

One advantage over "boxing" is the surprising effect that the box-size can be smaller than $2R$ without loosing possible overlaps of spheres in more than next-neighbouring cells. As the combine-routine selects all edgespheres in an edge of width $2R$ around the cutting-surface, those next-nearest-neighbour combinations will simply be found on a subsequent combine-level on the way up through through the recursion. We will see in the next chapter 3.2.3 that this gives the possibility to examine the optimal number of cuts by trespassing that value and choosing "too-many" cuts.

3.2.3 Optimal Number of Cuts

The described algorithm can be called with a wanted total number of splits s, that give the number of cells ($= 2^s$). If we choose $s = 0$, the original naive-clusterfinder of chapter 3.1.1 is recovered. For more splits s, the execution time sinks up to an *optimal number of splits*, then it rises again as you can see in figure 3.5. For too few cells, the $O(N^2)$-behaviour of the naive-clusterfinder decides about the duration; for too many cells, the execution time is dominated by unneccessary combine-calls with their expensive overlap-checking.

The optimal number of cells is a little lower than the limiting number of boxes of the boxing approach (see chapter 3.4.1), probably because of the inelegant combine, especially in higher dimensions.

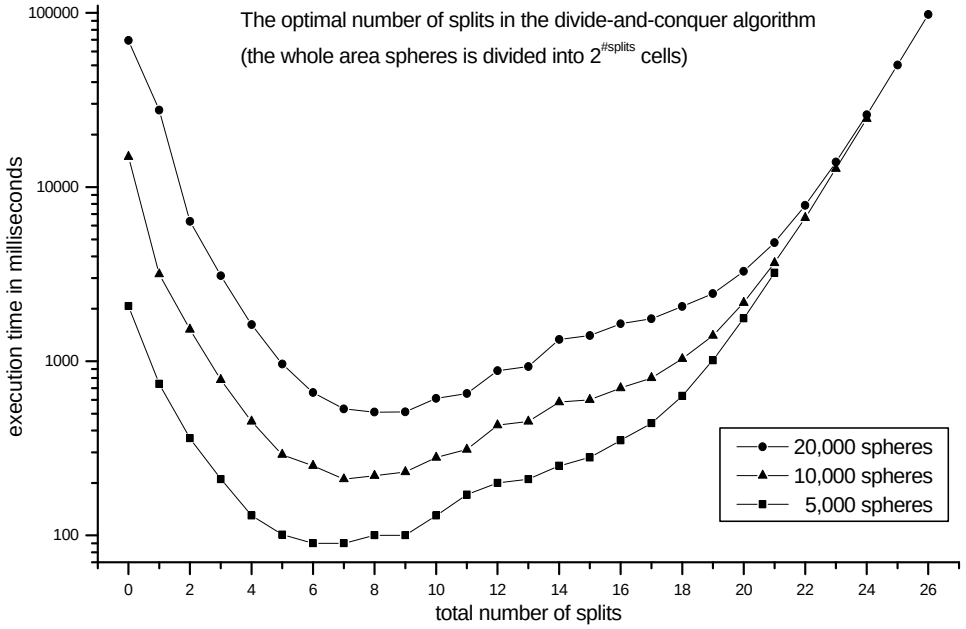


Figure 3.5: The optimal number of splits $s_{optimal}(\rightarrow 2^s =$ number of cells) depends on the number of spheres N and their dimension d. In this example, the execution time is shown for $N \in \{5,000; 10,000; 20,000\}$ and $dim = 2$ at criticality. $s = 0$ gives the naive clusterfinder of chapter 3.1.1, the optimal number of splits for these configurations are $s_{optimal} = 6$, 7 and 8, which accelerates the execution by factors of 23, 71 and 136 compared to the naive counter.

3.2.4 Complexity $O(N^x)$ with $x < 2$

All this tedious work gave a rewarding result: In the very first test of the new algorithm, it returned the same cluster-structure, but performed about 180times faster than the naive clusterfinder.

I analyzed the time complexity, and found a behaviour much better than $O(N^2)$. Of course, the gain strongly depends on the dimension and the number s of splits - in figure 3.6, you see the N-dependence of the execution time for 2 and 6 dimensions (with optimal number of splits). The exponent of the $O(N^\alpha)$ behaviour can be as good as $\alpha \sim 1.2$.
This improved time complexity made possible the examination of bigger systems with better statistics.

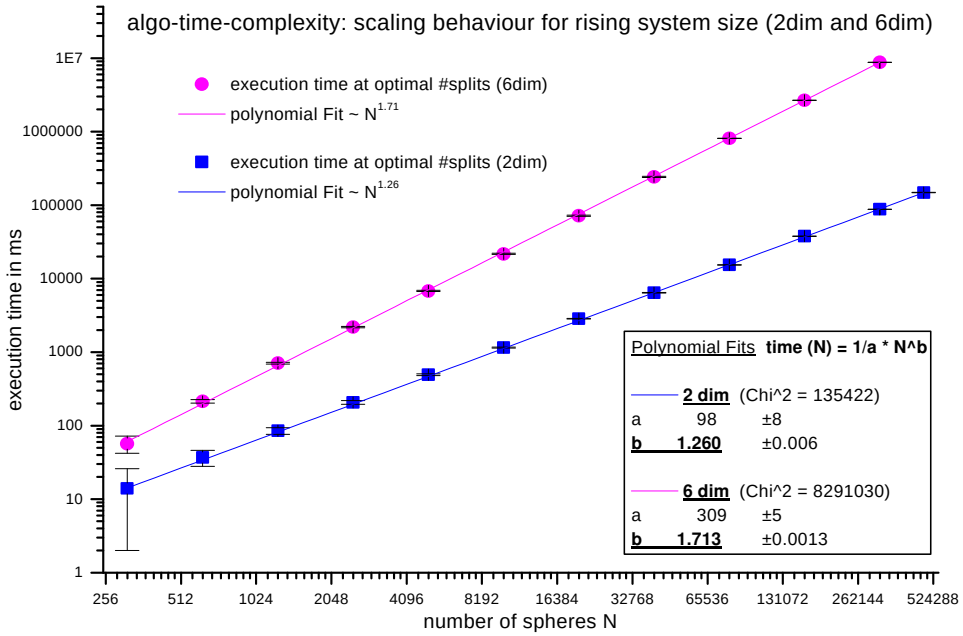


Figure 3.6: Execution time for rising number of spheres N, in 2 and 6 dimensions. The splitting algorithm performs like $O(N^{1.2})$ for 2 dimensions and $O(N^{1.7})$ for 6 dimensions.

3.3 Collecting the Results

Now that we have found all clusters, we can finally analyze the cluster size distribution, spanning probability, etc.

3.3.1 The Spanning Cluster

As mentioned in chapter 1.1.2, the occurrence of spanning clusters marks the critical threshold. At that density, the previously unconnected small clusters form such a big structure that a path of overlap from one side of the box to the other is formed.

After all clusters have been found, the cluster numbers are written into each sphere, so that it is easy to extract all cluster numbers at the left and right edge. If the spheres are of size same size (radius R), the edge must be of that width R to find all spheres that overlap the border; in the case of differently sized spheres, the radius $s.r$ of the sphere s itself is compared with the coordinate $s.x[d]$ in the dth space-direction.

Then the intersection of both sets of cluster numbers gives all numbers of spanning clusters:

leftedge=$\{s \in \text{spheres}; s.x[d] < s.r\}$
rightedge=$\{s \in \text{spheres}; L - s.x[d] < s.r\}$
with d =direction (e.g. $d = 0$ for x-axis), L =length of hypercube

if [clusternumbers(leftedge) $\cap$ clusternumbers(rightedge)] $\neq \{\}$
$\Leftrightarrow \exists$ spanning cluster

Then a decision has to be made: What is ***the*** spanning point? It can be triggered by a cluster that connects *all* dimensional directions (e.g. in 3dim, a cluster that spans in x-, y- ***and*** z-direction), or by a cluster that spans at least one direction (e.g. in 3dim, a cluster that spans in x-, y- ***or*** z-direction), or by spanning in a certain direction.
All these criteria give the same critical point for the limit $N \to \infty$, but in general, for finite realizations, the second happens earlier than the third which happens earlier than the first.

I have chosen the third criterion, that the $d = 0$-direction ("x-axis-direction") must be spanned.

3.3.2 The Histogram of Clustersizes

The main outcome of a realization besides the spanning-question is the histogram of occuring clustersizes. This distribution is an important aspect of the cluster-structure, in addition to that it "compresses" the information of cluster-forms, locations, etc. to only one property: the sizes.

In figure 3.7 you see an averaged histogram of 250 subcritical ($\eta = 0.5 \cdot \eta_{crit}$) experiments with $N = 5000$ two-dimensional discs.

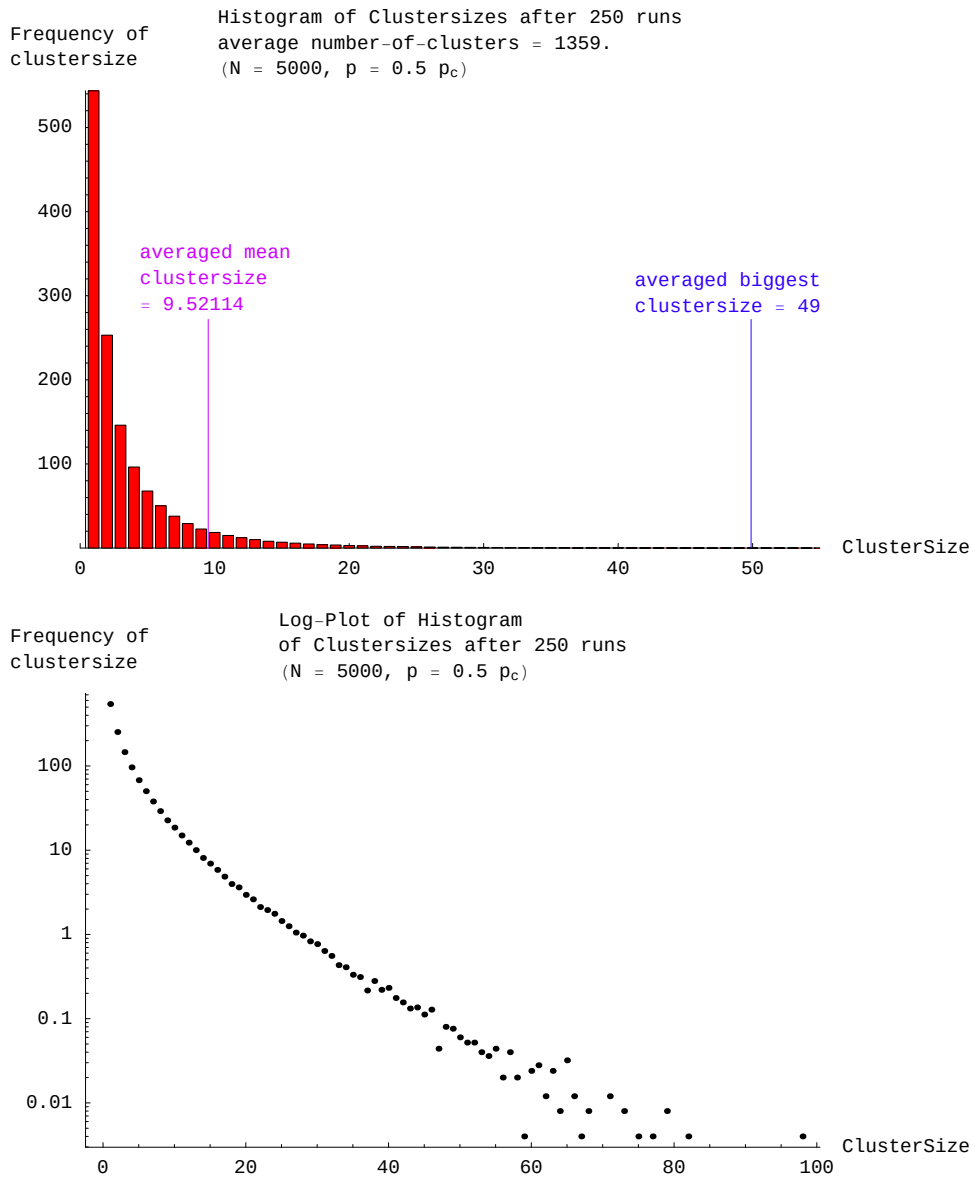


Figure 3.7: Histogram of clustersizes for $N = 5000$ at subcritical $\eta = 0.5 \cdot \eta_{crit}$ after 250 averaging runs. In the C++ program only the biggest and mean clustersize and the number-of-clusters is kept from these histograms (see chapter 3.3.3). In the LogPlot, you see that the frequency of clustersizes falls exponentially within a wide range.

3.3.3 Keeping and Dropping Data

Most of the observables have to be measured very often to get a good statistics because especially in the critical range, all fluctuations are big. The same experiment has to be done many times, so one crucial question is the amount of experimental results that are kept of each realization. It is necessary to discard lots of data.

After the ClusterFinder routine has finished its work, the resulting cluster structure is stored in the spheres and in an array of cluster lists. Only the visualization program (see chapter 4) uses these informations completely to show each sphere in a colour depending on the clustersize.
For the numerical part of the program, a histogram $H(s)$ of clustersizes s like figure 3.7 is generated out of the clusterarray of one realization and then all further information about the clusters is dropped.

Then this histogram is "compressed" even more, to only three parameters:

- the number of clusters $= \sum_s H(s)$
- the size of the biggest cluster $= \max\{s|H(s) \neq 0\}$
- the mean size of finite clusters $= \sum_{s,\text{if not spanning cluster}} H(s)s^2$

These three and the spanning-directions are written into a datafile and kept in memory for statistical analysis, and then the next realization is thrown.

3.4 Other Methods

I decided to use the recursive splitting algorithm described in chapter 3.2.2, mostly because I wanted to avoid the difficult task to define the neighbourhood of a box in d-dim space with open boundaries.

If one wants a fast algorithm in only 2 or 3 dimensions, other approaches might be useful:

3.4.1 Boxing

Especially for definite dimensions like 2- or 3dim problems, it is not too difficult to create a gridded structure with small boxes each holding only a tiny fraction of all spheres. The method divides each coordinate direction into n 'slots', the N spheres are sorted into the n^{dim} boxes in linear time by $N \cdot d$ divisions (for all coordinates). Then the partial problems are solved in each box using a naive ClusterFinder (mean time complexity $O((\frac{N}{n^{dim}})^2)$, afterwards the clusters in the boxes are combined with their neighbours.

Important to realize is that the boxes can be too small. If the boxlength is $< 2 \cdot$ radius, two spheres of next-next-nearest neighbour boxes might overlap. Because of the $O(N^2)$-behaviour inside the boxes, the optimal boxsize is the smallest possible using that condition.

3.4.2 Boxing Without a Naive Local ClusterFinder

Boxing can be done the same way as described in the previous chapter, but with boxes of which the d-dim diagonal is *smaller* than a diameter of the smallest sphere:

$$\text{box-diagonal} < 2R$$

$$\text{box-length} < \frac{1}{\sqrt{d}} 2R$$

Then all spheres inside a box *must* be overlapping, so no local in-box-ClusterFinder has to be started - all spheres in one box automatically belong to one cluster.

This advantage has a drawback, however. With such small boxes, not only next-nearest-neighbour boxes contain spheres that can overlap, also next-next-nearest-neighbour boxes have to be checked while combining the local clusters.

3.4.3 The Hoshen-Kopelman Algorithm

In 1978, Hoshen and Kopelman [24] found a good way of avoiding repeated relabeling within the 2dim lattice when two clusters merge. Their algorithm can thus scan the lattice in linear time, the cluster-relabeling happens only once at the end. Or not at all - because it's not necessary to go back to previously examined areas, one can even discard the already scanned lattice while it is still being examined, saving lots of memory by only keeping the cluster-information.

They keep a clusternumber-array which holds all clustersizes (positive numbers) and clusterlinks (negative numbers) at the same time. When the next lattice point is analyzed, and the neighbours to the top and to the left both are occupied and from *different clusters*, the cluster with the bigger number is merged into the cluster with the smaller number by adding its size the the other's size and giving it a negative entry that links to the other cluster. At the end of the run, the array's positive numbers are the clustersizes.

This approach can be adapted to continuum percolation by combination of boxing, local (naive) Clusterfinder and combining overlapping clusters in half of the surrounding $3^d - 1$ neighbour boxes, so in 2dim not only the left and top box (like on the lattice), but also left-top and left-bottom. The Mathematica© program in Appendix A.2.2 is my first attempt for 2dim continuum percolation - and it actually has a **linear** execution time behaviour of

$$\text{time}(N) \sim \frac{N}{900}\text{seconds}$$

Chapter 4

Visualization

I have written two completely different programs, one "numerical" with console- and datafile output (the results will be discussed in chapter 5), the other with a graphical frontend and output. Both use the same core algorithms.

(Not only) for this chapter, you should download the Win32 visualization-program from:

- `www.AndreasKrueger.de/thesis/exe`

It uses the strategies described in the previous chapter to find all clusters and shows them graphically. The program runs on any Windows Operating system, on a Dos-PC or in vmware virtual machines or Win-in-a-box; without any memory problems only in Windows NT, though.

In the newest version you can already start a non-stop show of clustering spheres - which gives a nice graphical effect and helps to understand percolation in a more intuitive way.

4.1 YGWYS - You Get What You See

I believe that visualization is a crucial step in understanding problems. Especially complex phenomena can be studied using our human visual system with its marvellous ability to grasp the essence of structures.

Not-so-obvious errors in algorithms might produce artefacts that can be *seen* and traced easily if the program output is visualized somehow.
Besides the didactic importance to be able to show to others what you are talking about and studying, these computer toys enable the *playing mode* in us with its intuitive intelligence.
Higher dimensional spaces might be impossible to imagine visually, but the attempt to show them helps to guess their properties.

During my work, programming has become more and more visualizing, especially since I have started to use Mathematica© (where lots of graphical routines are readily available), developing algorithms has become more and more a coding-looking-coding-looking approach, finding solutions by visually analyzing the question ... one might state: You Get What You See!

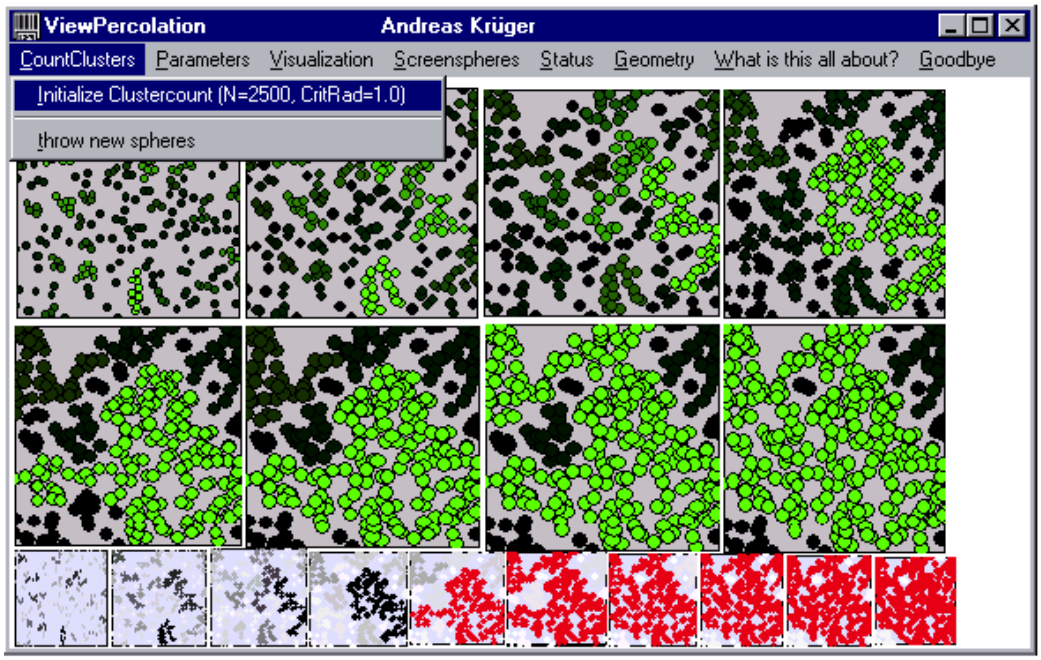
4.2 Principles of the Program

Please have a look at figure 4.1 to see the frontend of the program. It consists of menus to control the parameters and to switch between different visualization "shows", and of an empty area in which the spheres can be drawn as discs.

After the start and initialize, a box has to be pulled for to draw the spheres inside. Internally, it will always be a square, so if the box pulled on the screen is rectangular, the spheres become (oriented) ellipsoids.

First, all spheres are drawn once in white to get a feeling for the size of the problem, then the clusterfinder starts its work. On a 1Ghz CPU, it takes about 8 seconds for 160,000 spheres to be clustered in 2D.

The spheres of the biggest cluster are drawn in the strongest colour, all other spheres get less intense colours depending on the ratio of their clustersize to the biggest clustersize.



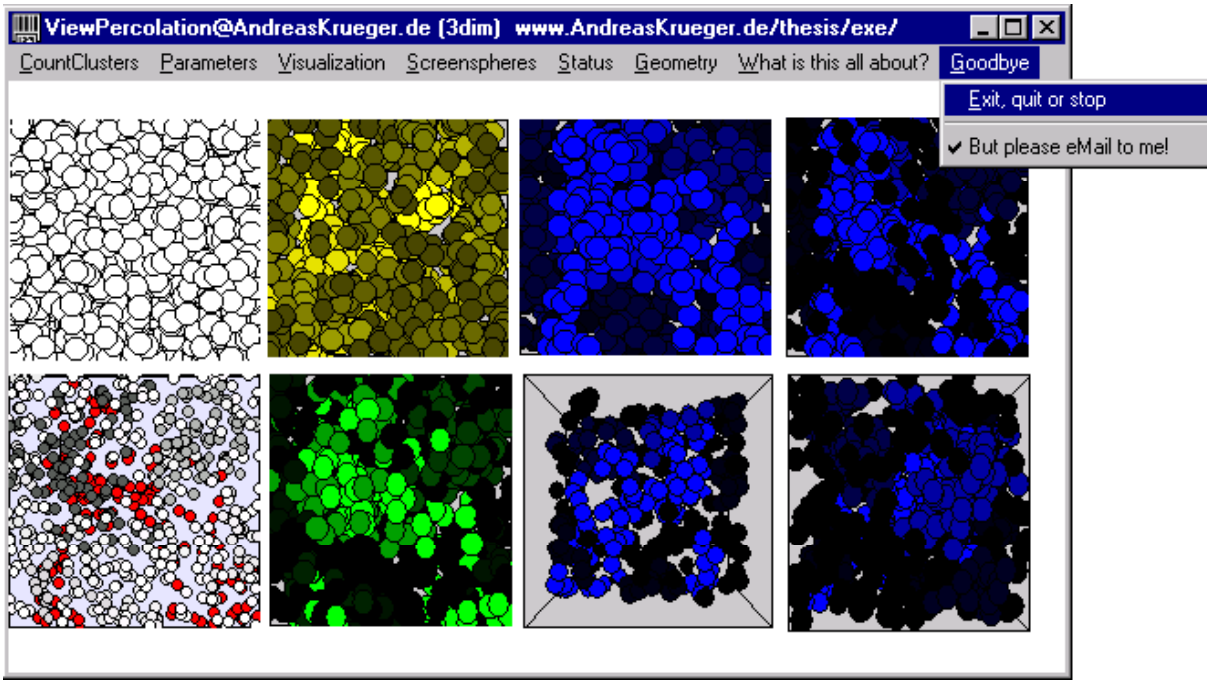


Figure 4.1: The frontend of the Win32 program for 2D with rising disc radius, and for 3D with examples of the different "shows". Especially the 3D version generates beautiful short-**films** (choose through the menus "initialize; N=160000; increasing z-sorted; 3D-effect"); and films can't be printed, so please download the program to your computer.

N.B.: At the moment there are no tick-marks when you enable a menu switch, but a status window can be opened to check the parameters.

The Parameters Menu

A wide range of numbers of spheres can be chosen, at the moment 39-160,000, the radius of all spheres can be set as a fraction of the theoretical critical filling factor ($\eta_{crit} \rightarrow$ "1.0" in all dimensions), especially accurately close to the critical point (0.95-1.05). Once clustered spheres are not clustered - only drawn -

again when you click-and-pull a new box; only after "throw new" or choosing a different N or R in the parameters menu, the algorithm is started again. This saves time when you try different visualizations.

The Visualize Menu

I experimented with different "shows" to draw the spheres:

- clusterwise: The spheres are drawn cluster-by-cluster
  The xy, xz or xZ plane is used for the screen coordinates of the projected spheres, the remaining coordinates are just dropped.
- random: The spheres are drawn in their original random order (using the xy plane for screen-disc coordinates)
- z order: the spheres are sorted by their z-coordinate, then they are drawn from back to front or from front to back
- 3D-effect: spheres in the back are smaller and their coordinates inside a smaller rectangle than spheres in the front (scaled by intercept theorems).
- stop after half of the spheres: one can hereby see a central cutting plane

At the moment the shows are executed one after the other, so a 5 seconds pause in between can be enabled to separate them.

The Screenspheres Menu

When a spanning cluster occurs (from left to right), the spanning spheres can be drawn in red during the first clusterwise-show, if you enable that option.
In more than 2dimensions, the screen with its two-dimensionality gets rather crowded, so I implemented an artificial reduction of the screenspheres. You can choose to have them *drawn* smaller to look beyond the last-drawn surface - the internal radius for overlap checking and clustering is not changed by that.
And last but not least, the surrounding black circles of the discs can be switched off, neccessary for to draw $N = 160000$ in dim$= 2$.

The Other Menus

In the first menu, you have to *initialize* the arrays once after each program start - and there, you can always **throw a new set** of points to look at other realizations. The clustering is then done with the next click-and-pull. The **Status** menu lets a N, R, r - window pop up to tell about the actual settings. The **Info** menu gives a short introduction and with **Goodbye**, you can leave the program or send me an eMail before.

The **Geometry** menu was the purpose of the original Win32-program that I found in the manual of my compiler. Windows-programming for menus, mouse and graphics is highly non-trivial, so I was thankful to have this instructive example - and I simply plugged-in all my routines.

4.3 Higher Dimensional Spaces

It is rather easy to show all discs in 2D, only for high N like 80000 or 160000 the surrounding black circle should be switched off (screenspheres menu).

For 3D, I suggest the following settings in the **visualisation menu**:

- Sort in third direction, ordered by **increasing z-coordinate**. Then the spheres are drawn from back to front.
- pseudo **3D-effect**; a central projection with the spheres smaller and closer at the back than at the front
- perhaps: **stop after half** of the spheres, then you can examine the central plane of the cube.

And perhaps in the **screenspheres menu**

- (for small N like 312 or 2500): **pause after drawing** one sphere. Then you can see the clusters slowly growing.
- **75% screenspheres-radius**; to be able to look through

In 4D, you can also use the above settings for 3D, moreover, you should reduce the screenspheres size further more (50% or 25%), but mainly you need your imagination to replace the unused fourth dimensional coordinate. The effect is that some spheres are in one cluster but don't seem to be, because they overlap in their fourth direction.

4.4 Plans

As mentioned before, the visualization program was very useful to find flaws in the ClusterFinder-routine and to be able to show percolation to non-mathematicians or non-physicists. But it is far from being intuitive and it uses only very few ideas to show the phenomena.

4.4.1 Ideas

The program GUI should really be redesigned and rewritten to be more useful and didactic. Some of the ideas are the following. If you find something that is not mentioned here, but you would like to see or have implemented, please tell me.

Visualization

- Radius lens: Show one disc enlarged or a magnified part of the whole area, to get a feeling for the growing radius
- Magnification: Zoom into a selected part of a cluster to see the overlap structure
- Colour lens: Colour the spheres by the position of their cluster in the histogram, not by the linear function only depending on the ratio to the biggest cluster. (At the moment most small clusters aren't coloured anymore as soon as a bigger cluster appears.) See appendix A.2.2 for a first fully functional - though slow - implementation of this idea.
- The biggest (or spanning) only: Up to half of the spheres, the smaller clusters are shown, above only the biggest (spanning) continues all the way up. This gives an impressive 3-dimensional pictures and is used e.g. in "grain growth" visualizations [27]. Or only the biggest (spanning) cluster is shown, no smaller ones at all.
- Clusternumbers: not by clustersize but by their clusternumbers, the spheres are shown in different colours
- Films: for a range of radii R (or number N), several pictures are generated and then shown as a short film.
- Continuous film: After one cube has been clustered and shown, a new set of spheres is thrown, clustered and shown. With sorting-by-z-coordinate, this will give the impression of continuously moving out.
- Legend: The palette of colours is related to numbers
- Rare events: Several spanning clusters, very early or late spanning, etc.

Statistics

- Display of observables: spanning or not, size of biggest cluster, mean clustersize, number of clusters
- Smoothing: Statistics over many runs are cumulated
- Histogram: The clustersizes can be observed while traversing the critical threshold

- Plots: The above measurements shown as plots or histograms during a film, e.g. a film in which the radius grows

Handling

- Input: All parameters should be input directly or by a slider, not only by constant menu values
- Find threshold: A spanning cluster-threshold-finder selects exactly that radius at which the transition takes place
- Mouse-click info: When a sphere is clicked-on, the clusternumber and size is given
- Progress and status bar: It shows how long the Clusterfinder will need
- Palette: create an own palette of colour transitions for each show

The Program Skeleton

- Save and load parameters, realizations (thrown coordinates) and palettes
- MDI: several windows in one application, a control-window to select the parameters with buttons and sliders
- Print pictures, histograms and parameters
- A symbol- and statusline for button-click commands, statistical results and progress bar
- A database connection for all previous runs - to be able to combine the visual output and a statistical analysis
- A context-help function and manual

4.4.2 OpenSource Computing and Visualization Project

The project would be a bigger one, rather time-consuming and probably too much for one person, but the result could be rewarding and useful, especially for lectures and talks about percolation and hyperspaces. Other people are working on the same or on similar programs, so why not create an environment for cooperative developing?

The program *must run on any platform and operating system*, so it should probably be in Java or C++ using Qt (Qt is the portable Frontend library for Windows and Linux that KDE is programmed with [41]). For the visualization, Open Source libraries on top of OpenGL (a fast graphics standard) could be used like SGL [47] or SDL [46]; and for drawing plots and histograms perhaps "root" from the Cern [67]?

Because computing power and simulation time is expensive, it could be interesting and challenging to create a simulation client that uses its host computer for simulations and communicates its results by TCP/IP to a database-server, on which several results are then combined. The reward for the owner of the computer (apart from the honour to work for the sciences) could be a beautiful screensaver on which parts of the simulations are drawn.
The SETI@home project [68] takes a similar approach, for which about 3.2 million members have sponsored 700,000 years (or $9.7 \cdot 10^{20}$ floating point operations) of computing power (until 29.8.2001) to calculate Fourier analysis' of incoming data from a telescope.

Chapter 5

The Numerical Results

This is the most important and thus most extensive chapter. It contains results that were measured for the first time in this accuracy.

It first motivates why exactly the thresholds were studied so excessively, then it names two possibilities how to locate a threshold.

In section 5.3, the long way of data analysis and compression from millions of single measurements to a dimension dependence formula is shown step by step; the data found by interval nesting is averaged and the distribution around the mean is examined. Then the mean is extrapolated with $N \to \infty$ to get the limit threshold for each dimension 1-11 (you find that most important table on page 52).
Once those 10 critical points were known, I constructed fit functions that sum up the knowledge about all of them (equations (5.3) and (5.4)). In the end they are related to the mean number of neighbours and the actually occupied volume fraction at criticality.

Section 5.4 then exploits that the same strategy can be used to identify other points below and above the critical threshold, with the goal of limiting later simulations to a definite and narrow interval of filling factors that are interesting to study. Figure 5.16 shows these results in one diagram.

At last, in section 5.5 the one dimensional case is treated; one can see clearly in the simulation data why that case is called 'degenerate' and can be solved analytically so easily.

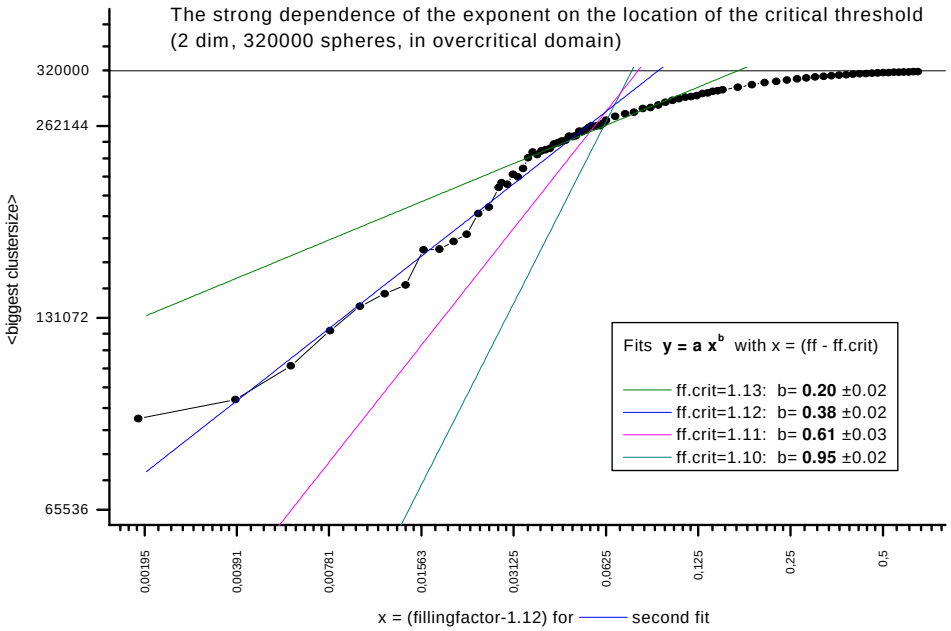


Figure 5.1: Fitting the experimental data to get the critical exponents of the singularity is impossible unless you have a very good knowledge of the position of the critical threshold. While varying η_c (here called `ff.crit`) by only 1%, the exponent differs by 100%.

5.1 Restriction to the Thresholds

The main characteristics of the phase transition are the exponents describing the singularity.

While trying to fit the simulation data around the critical point to a function $|\eta - \eta_c|^\omega$, a strong dependency of ω on the value of η_c can be noticed - and this dispersion makes the determination of the exponents very difficult (see figure 5.1).

So very accurate values of η_c are needed to find the critical exponents. When I decided to concentrate on that I didn't expect it to be so time consuming (because of coding and simulation time). Without an explicit decision before, it has now become the main result of this work, even when it is sometimes stated, that the search of critical thresholds is not the most interesting aspect of percolation theory [6] - it is still a necessity for any numerical treatment of the problem.

The exponents are still unknown to me, but they are widely accepted to be identical to the lattice percolation exponents - and those were excessively studied during the last decades and are known to a high accuracy.

5.2 Methods to Find η_c

5.2.1 Maxima of Fluctuation

Let M_i be measurements and $\sigma = \sum_i (M_i - \overline{M})^2 = < M^2 > - < M >^2$ the variance (or **fluctuation**) of the mean value $\overline{M}$ (see chapter 2.4.1). At the critical point, the outcome of measurements of e.g. the size of the biggest cluster B_i fluctuates most.

This maximum of fluctuation can be used as a criterion to locate the transition point, but it is a very expensive method. Because of the *second* moment involved, large deviations from the mean fluctuation have strong influence, so that a very good statistics is needed to smooth the fluctuation curve - many realizations have to be summed up.

At first I tried to use that method to locate the threshold; in figure 5.2 you can see that the fluctuation curve takes a shape similar to a Gaussian. The center of this Gaussian actually gives a good estimate already ($\eta_c = 1.132$), but too many repetitions had to be done, so I dropped that method again.

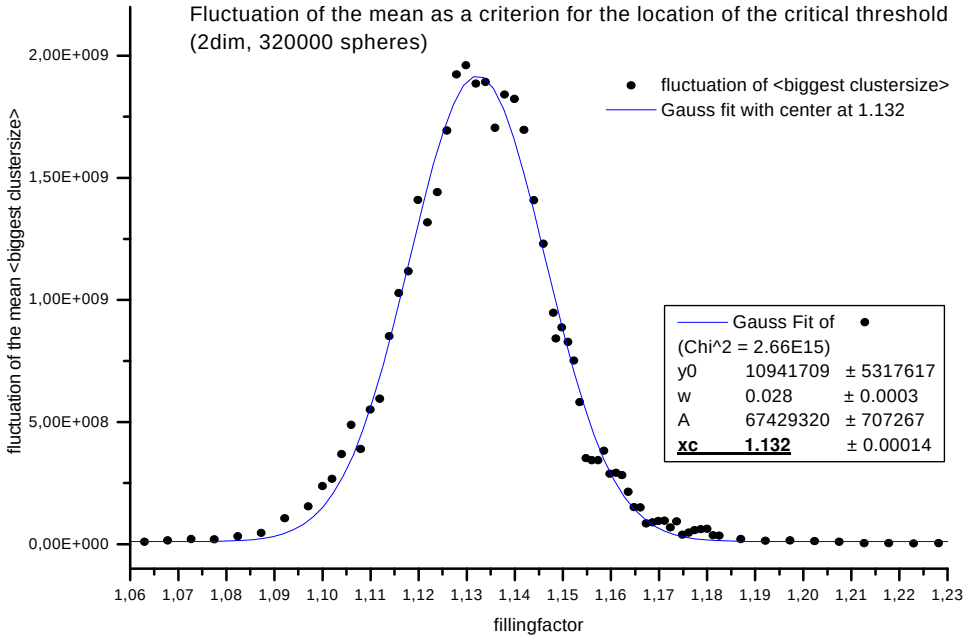


Figure 5.2: The fluctuation of the observable "size of the biggest cluster" exhibits a maximum at the critical point. This method gives the critical point - but is not very efficient due to the many repetitions necessary to get a smooth curve.

5.2.2 A Spanning-Cluster Occurs

The original definition of percolation: "occurrence of a cluster that spans any given space" (see chapter 1.1.2) is the direct prescription how to identify the critical percolation point.

In each realization of points, the question "spanning or not" can be used to adjust the radius of all decorating spheres to exactly the value (resp. intervall) below which there is no spanning cluster and above which there is a spanning cluster (please see chapter 5.3.1 for the intervall nesting routine); the center of such an intervall and the width of that intervall give the "critical" filling factor $\eta \pm \Delta\eta$ for that realization.

Then for many realizations, the numerical mean value $< \eta_c(N, d) >$ can be averaged, for which the spanning occurs in a d-dimensional system with N spheres.

Finally, the extrapolation for rising system size (= number-of-spheres N) gives the (thermodynamical limit of the) percolation point in that dimension, the critical point.

Measure the Mean of 0-1 Transitions

For a single realization of N sphere-centers thrown into a box l^d, there is exactly one radius (and thus filling factor), for which a spanning-cluster occurs. The filling factor-dependant condition "does (not) span the area" gives a false/true-transition at the wanted filling factor. This point can be found by intervall-nesting up to a given, but arbitrary accuracy.

Other such points might be similarly interesting. They can be found by the same algorithm as soon as that transition can be formulated as a false-true transition at that point. The following points are already implemented in the C++ program, but (partly) still lack numerical results:

1. a spanning cluster from "left" to "right" (first coordinate)
   - as described above, the (finite-size) **critical point**,

2. a spanning cluster connects *all* sides of the hypercube $\sim$ "**spanning-in-all-directions**",
   an alternative definition of the critical point

3. all spheres are in one cluster $\rightarrow$ number-of-clusters == 1
   (the highest "interesting" filling factor, let's call it "**saturation point**")

4. a definite number of clusters exists
   $\frac{\#clusters}{\#spheres} \leq a$ with $a \in]0; 1]$

   - $a == 1$ could be seen as the **onset of clustering**, but even for very small radii ~ 0, two spheres might be already overlapping, so a sharp ==1 condition is not practical for finite systems
   - $a == 0.9$ seems to be a better point (heuristically) to look for
     because the position is left of the critical point for all dimensions up to dim=11
   - $a == 0$ is not possible for finite N, the smallest possible is $a == \frac{1}{N}$
     (all N spheres are in 1 cluster - the saturation point)

5. the biggest cluster contains a definite mass (B:=mass of biggest-cluster)
   $\frac{B}{\#spheres} \geq a$ with $a \in]0; 1]$

   - $a == 1$ gives transition no.3, the saturation point
   - $a == 0.5$ could perhaps be an interesting point for practical applications because it can be measured easily:
     Half of the spheres are clustered in the biggest cluster

Of course, other conditions could be invented to study certain points of the percolation curve - as long as they are formulated in a false/true - transition, it will be easy to plug them into the existing C++ - program.

5.3 From the Appearance of a Spanning Cluster in One Realization to a Dimension-Dependence-Formula

N.B.:
We will soon extrapolate from finite to infinite systems (only there can be critical transitions!), but sometimes here the spanning cluster-transition point for finite systems is already called "critical".

5.3.1 Find the Transition in One Realization of Thrown Centerpoints by an Intervall-Nesting-Algorithm

Now let's come back from the generalization of the previous chapter to the one special point on the percolation curve, the appearance-of-spanning-cluster transition-point.

For a given number N of points in a given area $l^d = 1$, the step-function is a function of the radius only:

$$\begin{aligned} stepfn = f(R) &= (\exists spanningcluster?) \\ &\in \{false, true\} \end{aligned} \tag{5.1}$$

This stepfunction (that switches to true for the percolating domain) can now be used for an intervall-nesting routine (steps 1-4 below, shown in figure 5.3), and an averaging over many of those nesting-results (steps 5-6):

1. Take the *center of the last intervall* as the new *left* border, if stepfn(center) == false and as the new *right* border, if stepfn(center) == true.
2. back to 1. until 3. is true
3. The intervall is halved until the wanted precision Δ_{target} is reached:
   The result is $R_i^{crit} \pm \Delta R_i$ with $\Delta R_i < \Delta_{target}$ and $\Delta R_i = \frac{1}{2} \cdot$ length of interval.
4. With the constant l and N, this radius is transformed into a filling factor:
   $R_i \pm \Delta R_i \quad \rightarrow \quad \eta_i \pm \Delta\eta_i$
5. Now the sum over many realizations n is taken and the mean $< \eta_{crit} >$ is found
   (back to 1. with a new centerpoint-realization, until 6. ist true)
6. until $\sqrt{\frac{<variance(\eta_{crit})>}{n}} <$ wanted accuracy

Then, the resulting $< \eta^{crit}(N, dim) > \pm\Delta$ is stored and a new configuration (N, dim) in parameter space can be simulated. It is a good idea to double N in the next step, because then one can assume that the variance of occuring filling factors sinks with the rising system size. If the minimum and maximum of filling factors of all realization for N have been recorded, they can be narrowed a bit and used as the new starting intervall for $N \rightarrow 2N$.

5.3.2 Mean, Variance, Skewness and Kurtosis of the Results

As you can see in figure 5.4, especially for low N the distribution of spanning-cluster-appearance-points doesn't seem to be symmetric around the mean value. There are some very late events but not as many early spanning-realizations.

Thus the *mean value* is not the *most probable* like in a symmetric distribution. This brought up the question of statistical observables for measured data distributions. Consult chapter 2.4.2 for the definitions of mean, variance, skewness and kurtosis.

Please have a look at figures 5.5 and 5.6 to see all four parameters for 78 and for 20000 spheres. The skewness in the first case (small number of spheres N) is obviously far from being zero - so the distribution is skewed to the left side, the mean value is thus only a rough estimator for the whole distribution - but the distribution tends to become symmetric for high N (figure 5.6).

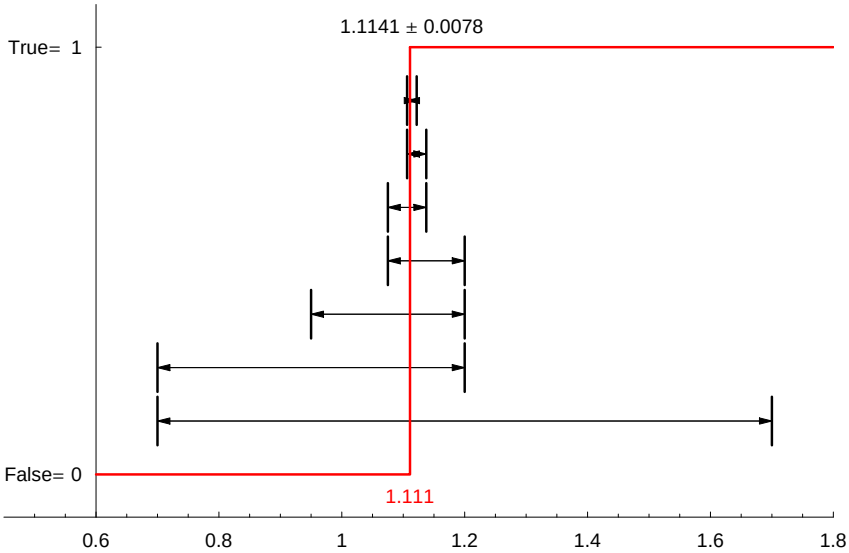


Figure 5.3: Find a certain filling factor that is located by a false-true stepfunction, using intervall-nesting. While a Fibonacci-search is the most effective to locate a minimum (divide the intervall into 1/2, 2/3, 3/5, 5/8, ..., 0.618) - in this case, locating a 0-1 transition, the simple 0.5-dividing is the best.

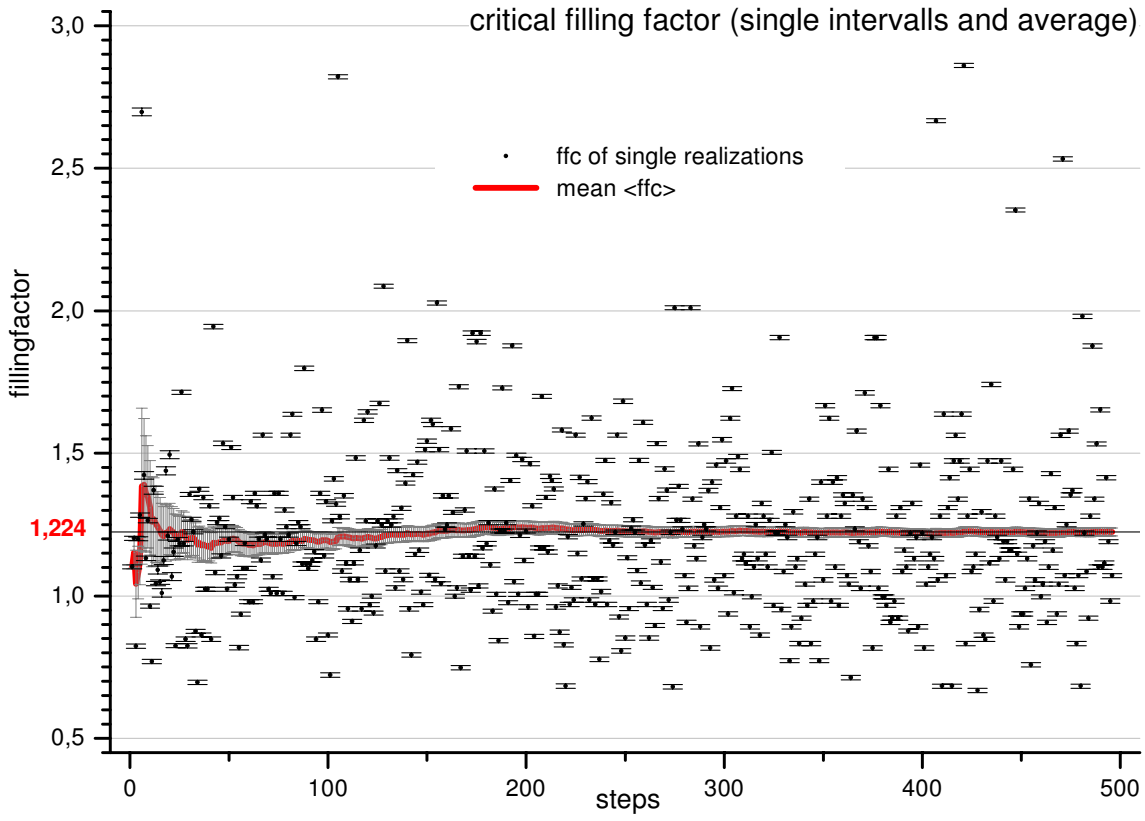


Figure 5.4: 500 intervalls for the spanning-transition and the mean value (red) for $N = 78$. Each datapoint with errorbar represents one realization of thrown coordinates. The position on the filling factor-axes was found by intervall nesting with varied radius for the decorating spheres until the intervall "doesn't span / spans" was shrunk to the shown error-bar size 1.2%.

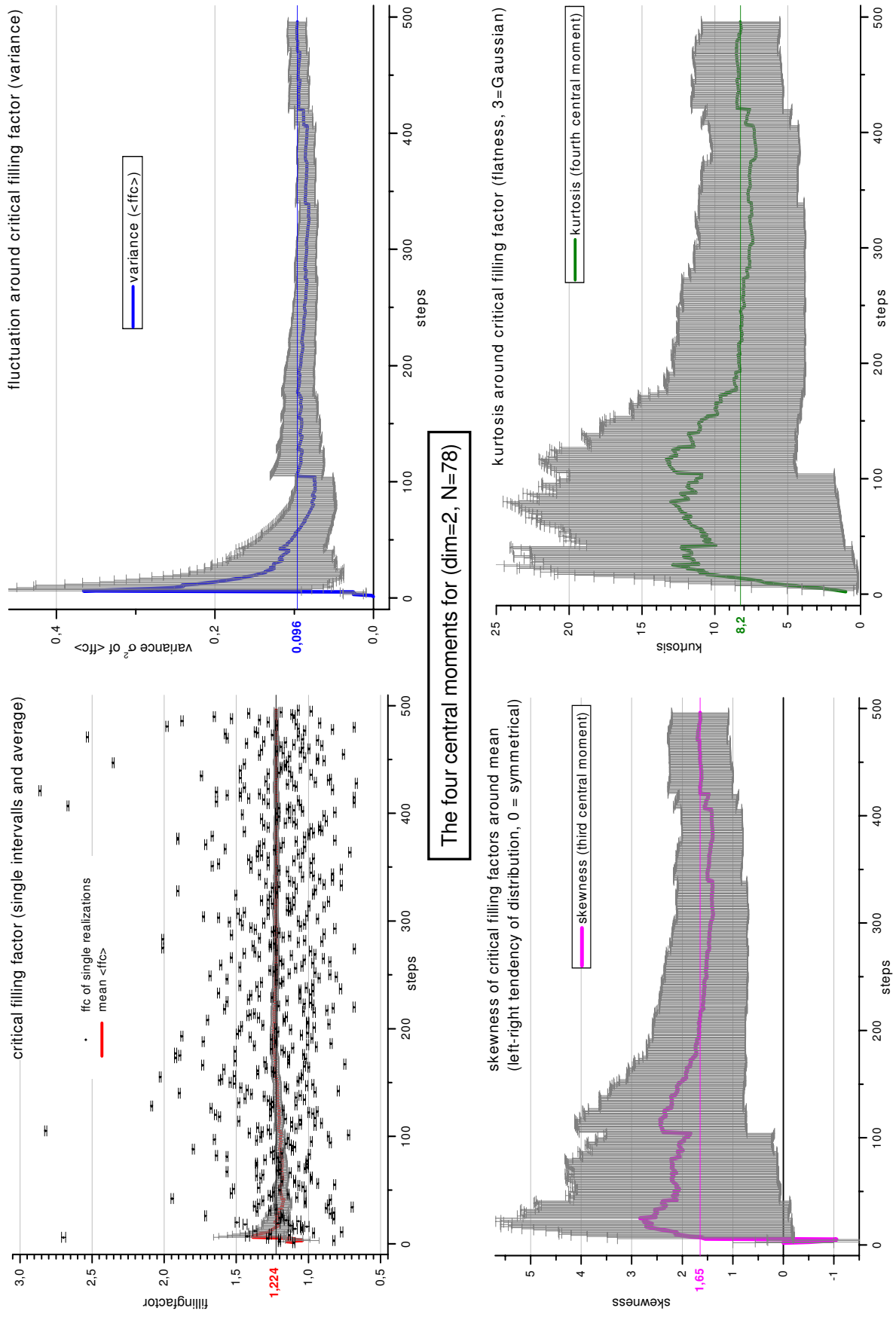


Figure 5.5: Mean, variance, skewness and kurtosis for $N = 78$ spheres and 500 realizations (dim=2)

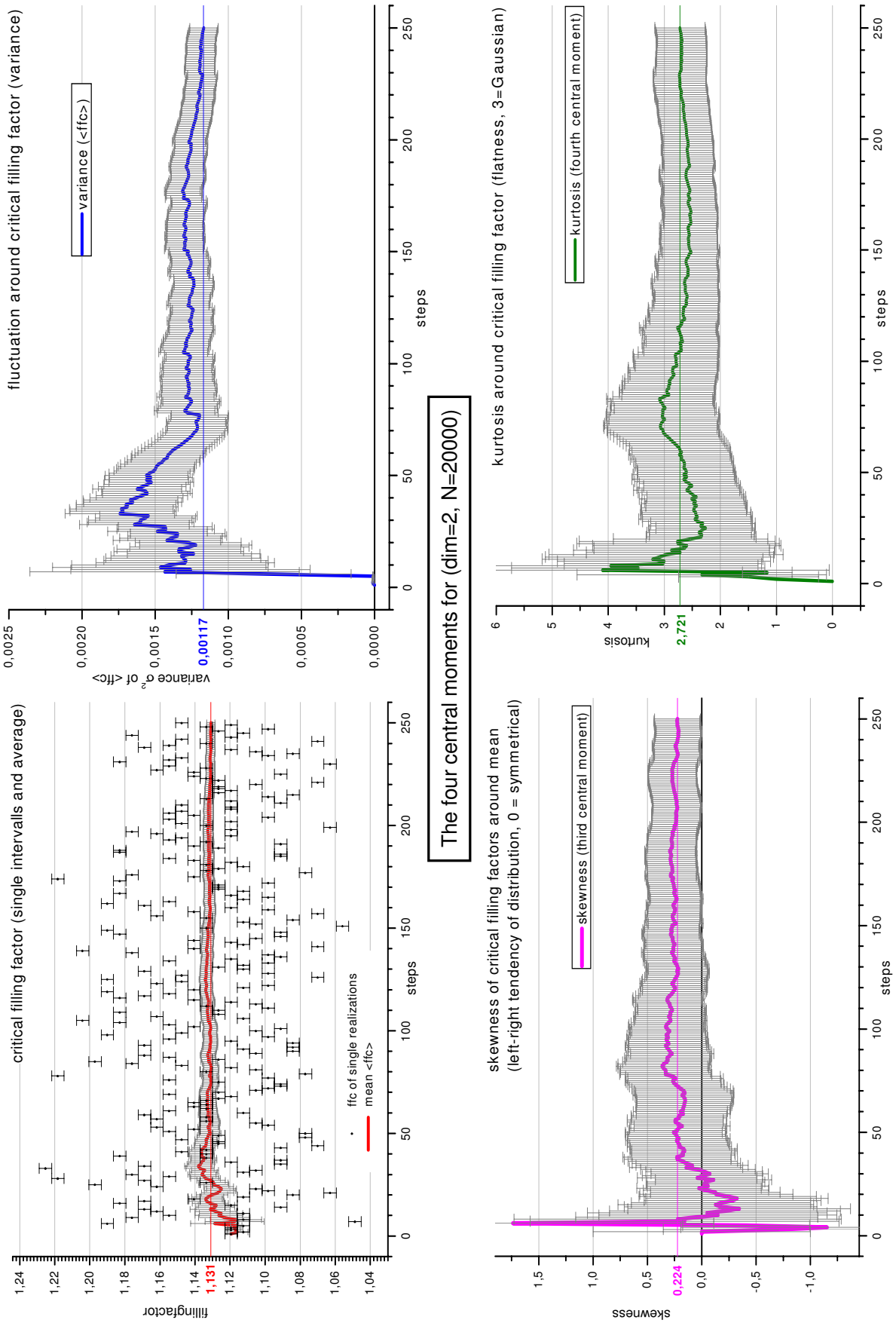


Figure 5.6: Mean, variance, skewness and kurtosis for $N = 20000$ spheres and 250 realizations (dim=2)

5.3.3 The Measurements: Min, Max, Mean and Variance

As seen before, the intervall of occuring measurements for the critical filling factor shrinks for rising system size (number-of-spheres N). In figure 5.7, the minimal and the maximal occuring filling factors are shown together with the mean values, the bars giving the standard deviation.

The contracting behaviour of that intervall is an important property of the critical point in percolation: The variance tends to zero for $N \to \infty$.

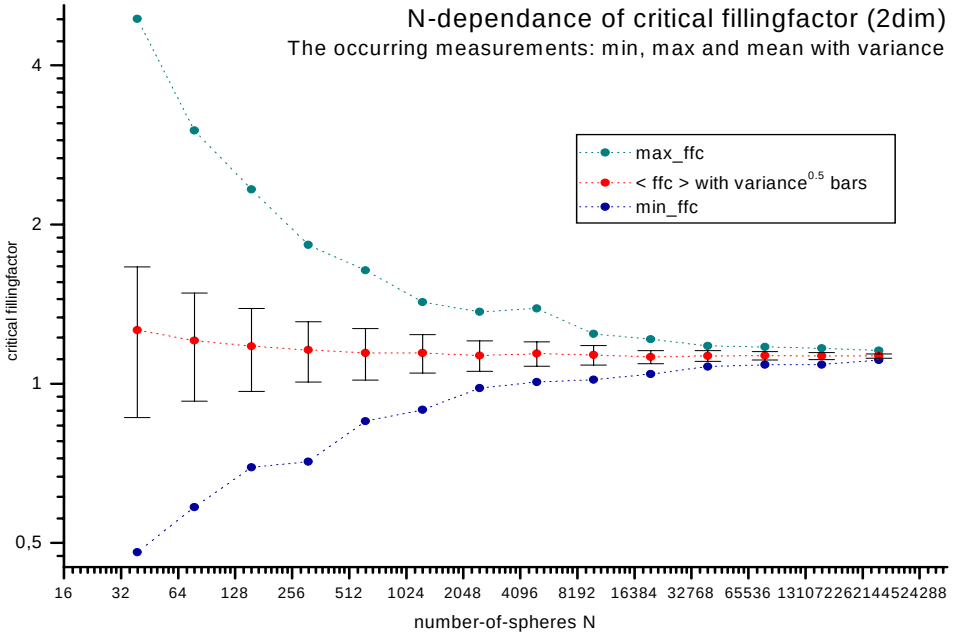


Figure 5.7: The maximal, minimal and average value of the percolation threshold, shown as functions of the rising system size (number-of-spheres N).

5.3.4 The Extrapolation $N \to \infty$

All the realizations (of the previous chapters) for one given number N of spheres result in one mean value $< \eta_{crit}(N, d) >$ for each of the spanning-transition thresholds of a d dimensional system with N spheres.

Now all these mean values can be studied, one can plot the N-dependence of the transition point. The spanning cluster occurs a little bit earlier for many spheres N than for few (see figure 5.8).

For the limiting case of an infinite number of spheres ("thermodynamical limit") this dependence must now be extrapolated to $N \to \infty$. An easy and natural fitting function (that fits the measured data remarkably well!) is an allometric power law with negative exponent c (to have a vanishing ∞-limit) and an offset-value for the $N \to \infty$ case:

$$\eta_{crit}(N) = a + bN^c, \tag{5.2}$$

For the numerical data in 2 dimensions this gives $a = \eta_{crit}(N \to \infty) = 1.1282$, the results for the critical threshold in higher dimensions are given in table 5.1.

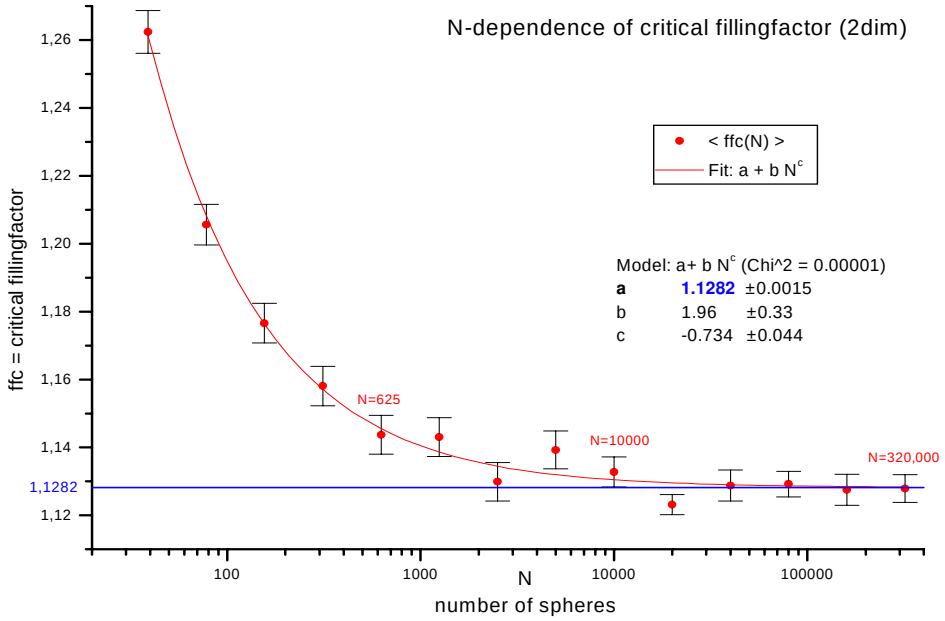


Figure 5.8: N-dependence of the critical filling factor (same data as figure 5.7, but the mean value only). Each datapoint represents one averaged mean value (i.e. the red line in in figure 5.4 for $N = 78$). The more the spheres, the earlier the spanning cluster appears (in terms of filling factor). The fit-function (5.2) extrapolates to $N \to \infty$; and the resulting ∞-filling factor for two-dimensional percolation is $\eta_{crit} = 1.1282$

Table 5.1: **The most important results in this paper:** The critical thresholds and the other two fit-function parameters b and c in the allometric fit (5.2) for dimensions 2-11. For the interpretation of the last column, please see chapter 5.3.5

dimension	$a = \eta_{crit}(\infty)$	b	c = the exponent	$\eta_{crit}(\infty) \cdot 2^d$
2	1.1282 ± 0.0015	1.96 ± 0.33	-0.734 ± 0.044	4.51 ± 0.01
3	0.3416 ± 0.0024	0.738 ± 0.14	-0.462 ± 0.042	2.73 ± 0.02
4	0.1300 ± 0.00095	0.614 ± 0.085	-0.520 ± 0.031	2.08 ± 0.02
5	0.0543 ± 0.00030	0.433 ± 0.039	-0.560 ± 0.025	1.74 ± 0.01
6	0.02346 ± 0.00028	0.235 ± 0.025	-0.534 ± 0.025	1.50 ± 0.02
7	0.0105 ± 0.00017	0.122 ± 0.011	-0.519 ± 0.022	1.34 ± 0.02
8	0.00481 ± 0.00011	0.0714 ± 0.0074	-0.524 ± 0.025	1.23 ± 0.03
9	0.00227 ± 0.00005	0.0489 ± 0.0048	-0.564 ± 0.023	1.16 ± 0.03
10	0.00106 ± 0.000032	0.0273 ± 0.0028	-0.567 ± 0.025	1.09 ± 0.03
11	0.000505 ± 0.000015	0.0161 ± 0.0013	-0.588 ± 0.020	1.03 ± 0.03

5.3.5 The Threshold Formula $\eta_c = \eta_c(\mathbf{dim})$

The Critical Filling factor

Once the extrapolations of chapter 5.3.4 were done for the numerical data in every dimension, the dimension dependence of $\eta_{crit}(d) = \eta_{crit}(d, N \to \infty)$ could be examined (first column in table 5.1). Figure 5.9 shows the results on a linear-linear and on a log-lin scale. The latter is important for higher dimensions, because the filling factor quickly drops to very low values in high dimensions.

Also shown are the two best fit function I could find:

$$\eta_{crit,1}(d) \cong [a_1 + b_1(d-1)^{c_1}]^d \tag{5.3}$$

$$\eta_{crit,2}(d) \cong [a_2 + b_2(d-1)^{c_2}]\frac{1}{2^d} \tag{5.4}$$

with $a_1 = 0.480, b_1 = 0.588, c_1 = -1.462$
and $a_2 = 0.4139, b_2 = 4.1038, c_2 = -0.8231$

At first, these (heuristical) fit functions were just born from the necessity to re-implement the numerical results into the C++ program for further runs. Nevertheless, one can see that they reflect details of the examined dimension-dependence:

- The $\eta_{crit} \to \infty$ for the 1-dimensional case (see chapter 5.5) is modelled into the $(d-1)$.
- For $d \to \infty$, η_{crit} becomes 0.48^d if the first fit function holds

I'd very much appreciate better fit-functions of table 5.1 if you find one!

N.B.: A similiar approach was undertaken by Galam and Mauger [19]; for many (very different!) lattice percolation models (square, honeycomb, ..., d-dimensional), they succeeded in fitting the site- and bond-percolation thresholds into one dimension-dependent invariant with three parameters.

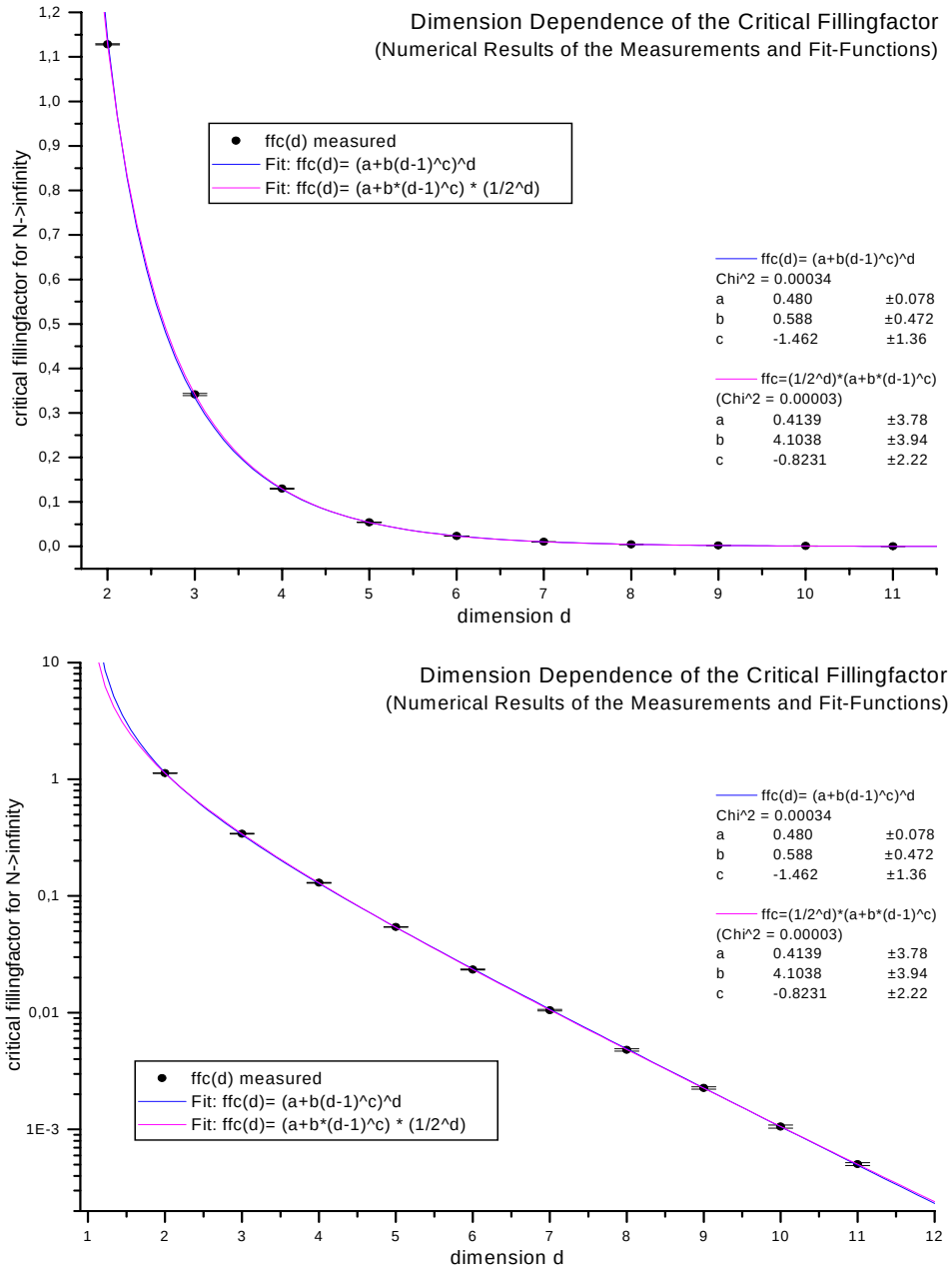


Figure 5.9: The measured critical filling factor $\eta_{crit}(d)$ and the fits (5.3) and (5.4) in a lin-lin plot and in a log-lin plot to see the higher dimensional behaviour.

The Critical Number of Neighbours

With the insights of chapter 2.3.2 one can get the *mean number of neighbours per sphere at criticality*, by calculating the Poisson mean number of other sphere-centers within a sphere of *doubled* radius. The last column of table 5.1 gives these numbers, they are plotted in figure 5.10.

The results are more accurate, but mostly in agreement with [1], where Alon, Drory and Balberg give preliminary results of their Monte Carlo simulations for $dim \in [2, 6]$ and $N = 20000$.

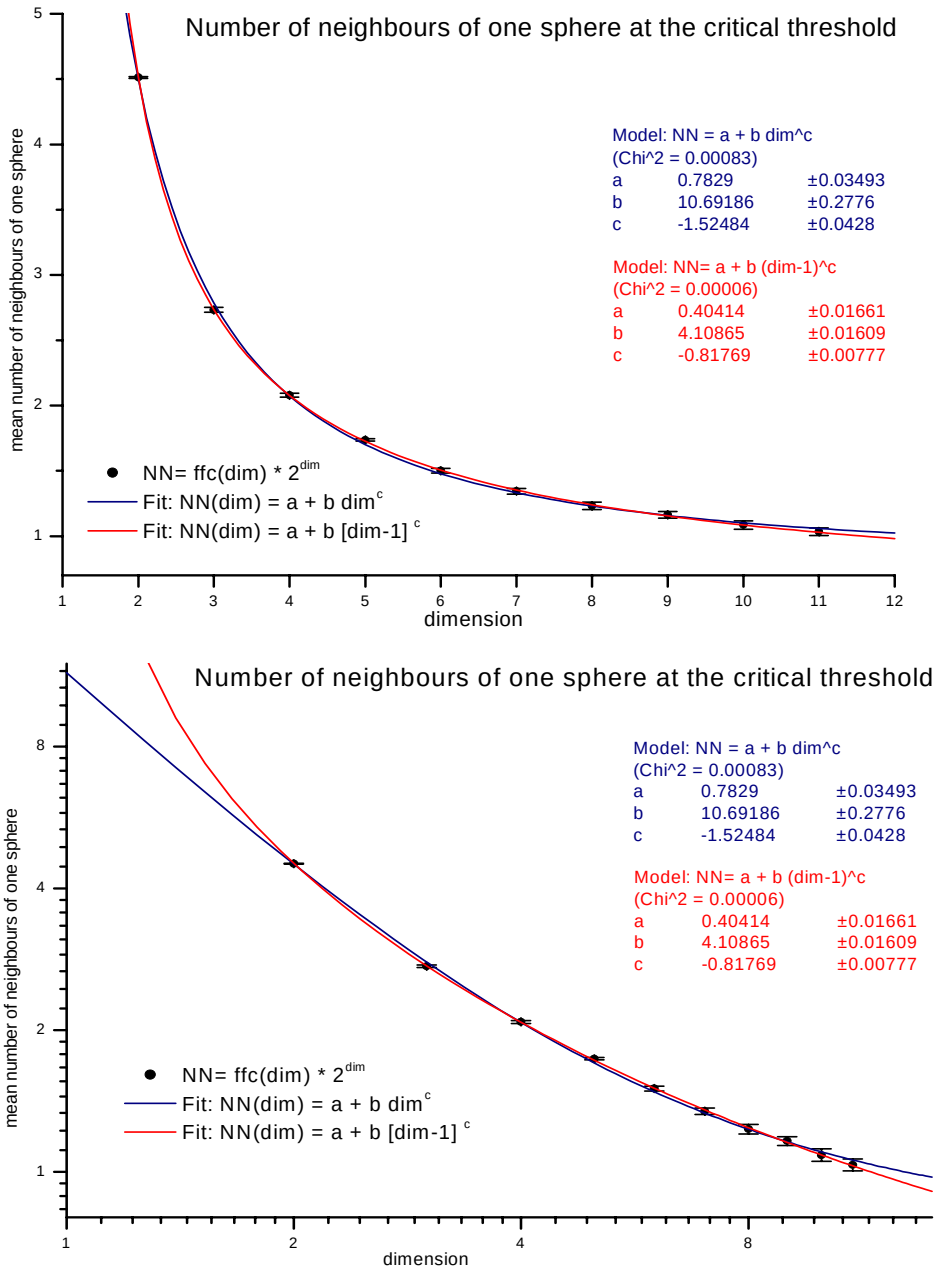


Figure 5.10: The necessary number of neighbours per sphere at the critical point in d dimensions as a linear and as a log-log plot. The second Fit-function seems to fit the data better. (N.B.: The y-axis labels are wrong: They should be "number of neighbours")

Blanchard, Gandolfo et. al. have recently published a paper [20] on "k-clusters" as the building blocks of the infinite cluster at criticality - and they find for 2 dimensions that k must be larger than 4 to get into the critical domain. Their "k-clusters" (similiar to the "k-mers" of Quintanilla and Torquato [42]) with k neighbours in a mean distance $r_0 < R$ from the center-sphere of size R are regarded as the 'building blocks' of a clustering process, which needs at least $k \geq 4$ neighbours to have long-range connectivity, i.e.

where the k-clusters merge into one spanning cluster.

that in a wider corona there is no further sphere, so true k-clusters are studied.

At first, it seems illogical that a mean number of neighbours < 2 is enough to get a spanning cluster (in dimensions $d > 4$, see table 5.1 and figure 5.10), but one mustn't forget all those spheres that are not part of the spanning cluster but still contribute to this average; moreover, there are rare but existing cases in the Poisson distribution of neighbour-degrees higher than the mean value that contribute to connectivity. And finally, the critical point only gives the density for the **onset** of percolation, at which a spanning cluster becomes *possible* for the first time with probability $P > 0$, but not with $P = 1$.

To clarify these questions, it could be interesting to actually measure the neighbour-degree in a simulation, a) to see if the Poisson argument holds and b) to only count those spheres inside the spanning cluster to measure the neighbour degree inside the critical spanning cluster.

The Critical Particle Phase Fraction

While the critical filling factor gives the ratio of thrown spheres over provided space (overlapping regions are counted multiply), the critical particle phase fraction (see chapter 2.2.1) tells us how much of the provided space has to be actually filled until that percolation appears.

The filling factor is used in most publications, but some of them give their results in terms of the Occupied Volume Fraction, the most accurate result for 2 dimensions currently being in [43]:

$$\phi_c^{\text{lit}}(d = 2) = 0.676339(4)$$

This is in excellent agreement with the results gotten in this work, because for 2 dimensions, I measured (see table 5.2):

$$\phi_c^{\text{here}}(d = 2) = 0.67638 \pm 0.0005$$

In figure 5.11 the dimension-dependence of this parameter is shown, a plot of table 5.2. You can see, that for 2 dimensional continuum percolation, 67.6% of the provided space have to be filled, for 3dimensional percolation 28.9% ...

The data seems to follow a straight line in log-lin plot, so it was fitted to the falling exponential

$$\begin{aligned} \text{FIT}[\phi_{crit}(d)] &= \exp\left(-0.7998 \cdot (dim - 1.38263)\right) && (5.5) \\ &= 0.4494^{dim} \cdot 3.0218 && (5.6) \end{aligned}$$

but actually it curves perceivably around that straight line and diverges from it for low dimensions (you can clearly see that in the "Fit"-row of table 5.2).

Table 5.2: The measured particle phase fraction (chapter 2.2.1) $\phi_{crit}(d)$ for criticality shows an exponential decay for rising dimensions.

dim	1	2	3	4	5		7	8	9	10	11
$\phi_{crit}(d)$	1	0,67638	0,28937	0,12190	0,052852	0,023187	0,0104451	0,00479845	0,00226743	0,00105944	0,000504873
$\pm$	0	0,000485	0,00171	0,000834	0,000284	0,000270	0,000168	0,000109	0,0000499	0,0000320	0,0000145
Fit	1,35	0,61	0,274	0,123	0,0554	0,0249	0,0112	0,00503	0,00226	0,00102	0,000456

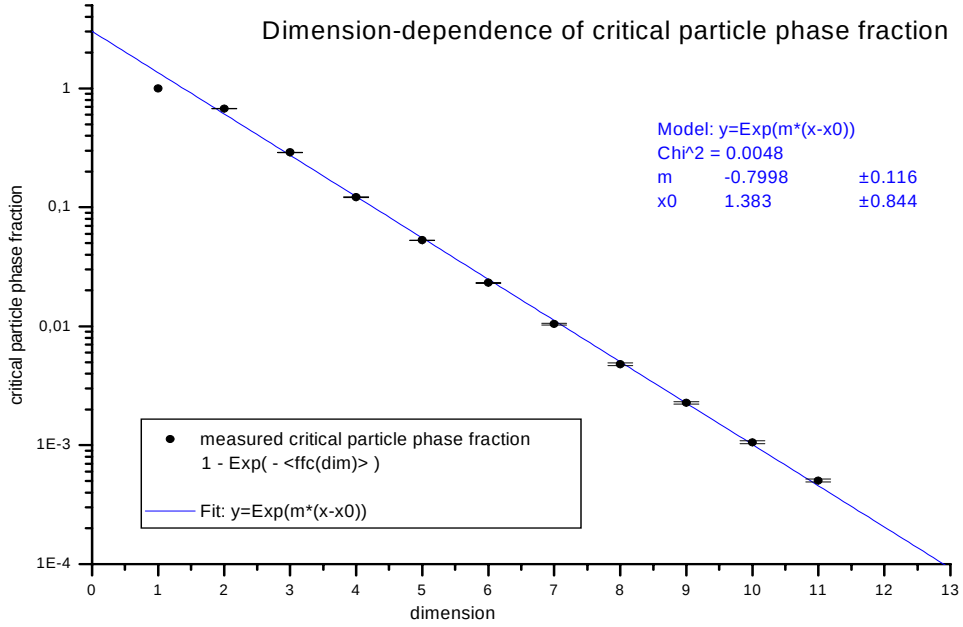


Figure 5.11: The measured critical particle phase fraction $\phi_{crit}(d)$ (see chapter 2.2.1) in a log-lin plot (to show the higher dimensional behaviour). The fit function only represents the general trend, especially for low dimensions, it does not fit well (see table 5.2).

5.4 Other Points Than the Critical Threshold

As mentioned before, not only the critical point might be interesting to study. Especially the saturation point (where all spheres belong to one cluster) is important to know (above no further simulations are needed). It lies far above the critical point.

For all later algorithms that will scan a whole range of filling factors below, at and above the critical threshold, a lower and a higher limit for the scan is needed.

I didn't succeed in finding an equally natural point as the saturation point, but *below* the critical threshold. Thus, I experimentally chose a point that stays below the critical point at least for the first 11 dimensions, the "10%-clustering-point", where the number of clusters is 90% of the number of spheres N, so *almost* all spheres are still separated from each other.

Moreover, it turned out that also the data for the rather natural saturation point could not be expressed only by the space dimension, independently of the number of spheres. So in the end, I chose another point shortly before that saturation point, the "99%-clustering-point", where the number-of-clusters is only 1% of the number-of-spheres N, so *almost* all spheres are clustered.

5.4.1 The 10%-Clustering-Point

The "10%-clustering-point" simulation data for a rising number of spheres N was analyzed in the same way as the finite-size-scaling for the critical threshold $\eta_c(dim)$ described in chapters 5.3.1-5.3.4; so for each configuration (dim, N) the mean filling factor was found as an average over many repetitions, then all the mean values were extrapolated to (dim, N $\to \infty$) and after that the shown dimension dependence was analyzed.

A rather good heuristical fit-function for the dimension-dependence seems to be

$$\eta_{10\%-clustering}(dim) = a \cdot c^{dim} \tag{5.7}$$

with $a = 0.16762, c = 0.53439$.

Please have a look at figure 5.12 to see a plot of the experimental data and that fit-function.

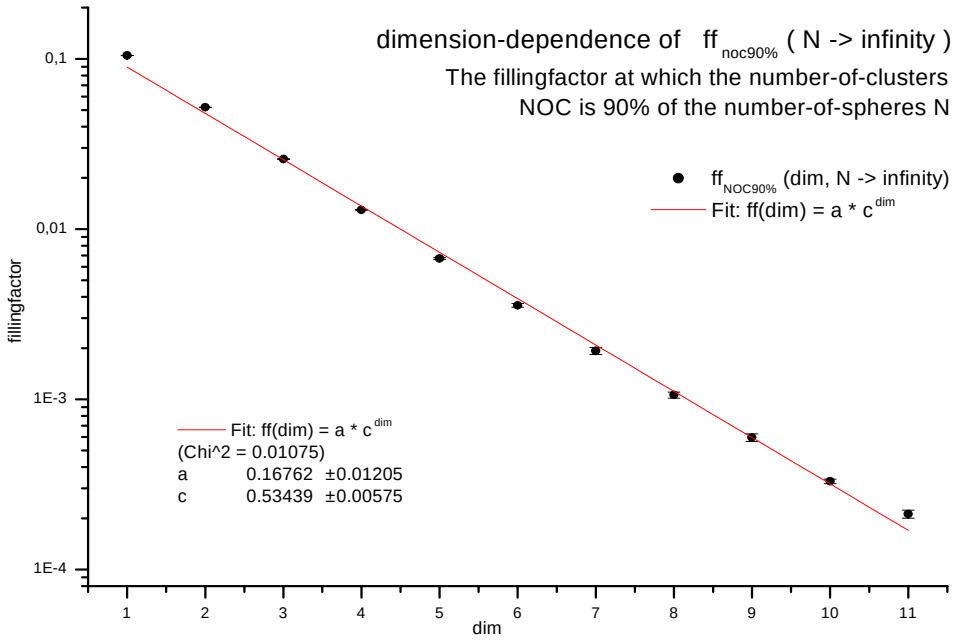


Figure 5.12: The 10%-clustering point at which the number-of-clusters is 90% of the number-of-spheres N. Experimental data after finite-size scaling, and the fit-function (5.7)

5.4.2 The Saturation Point

The saturation point is the mean filling factor for which further increase in sphere-size won't change anything anymore because all spheres are already in one cluster. So it marks the upper end of the "percolation intervall" that is interesting to look at.

The simulation data in each dimension for a rising number of spheres N *could not* be analyzed in the same way as the finite-size-scaling for the critical threshold $\eta_c(dim)$ described in chapters 5.3.1-5.3.4 or the 10%-clustering-point of chapter 5.4.1.
Why? The mean value for the saturation filling factor *keeps on growing* for increased system size (number-of-spheres N) - within the range of simulated N (up to 320,000) without bounds; it does not curve to a plateau, so it cannot be decided if it saturates to some $N \to \infty$-limit value or if it keeps on growing.

As an example, in figure 5.13 you can see the N-dependence of the saturation point in 2 dimensions.
Rather good heuristical fit-functions for the N-dependence seem to be logarithmic

$$\eta_s(N) = a + b \cdot \log_{10} N \tag{5.8}$$

with a, b depending on the dimension. These fit-parameters are given in table 5.3.

For $dim \geq 3$, those fitparameters a and b can be themselves fitted by exponential fit-functions with the same basis, so the rows (dimensions) 3-11 of table 5.3 can be expressed in one function; the saturation filling factor is approximately

$$\eta_s(dim, N) \quad \cong \quad c^{dim} \cdot (f + g \cdot \log_{10} N) \qquad \text{for } dim \geq 3 \tag{5.9}$$

with $c = 0.7405, f = 1.913, g = 0.692$.

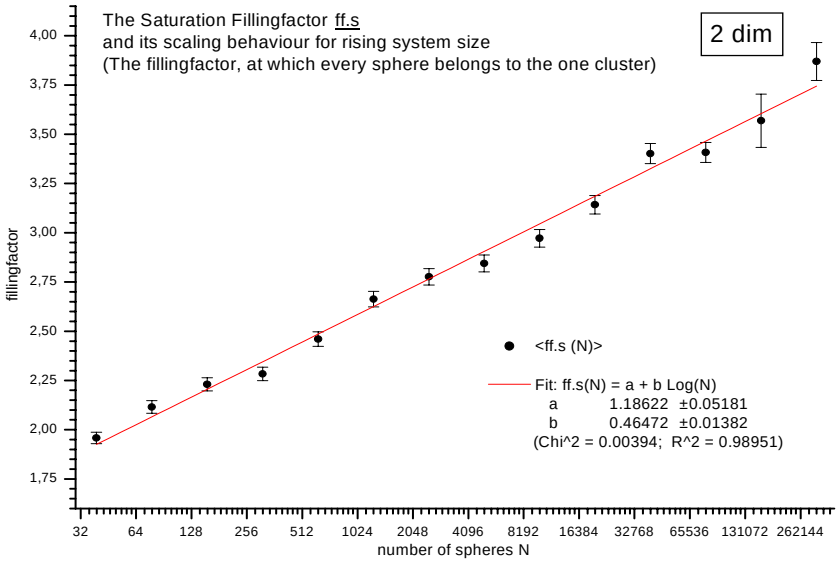


Figure 5.13: The saturation filling factor $\eta_s(N)$ in 2 dimensions at which all spheres are in one cluster. Experimental data and a logarithmic finite-size-scaling fit-function (see equation (5.8)).

5.4.3 The 99%-Clustering-Point

As the saturation point could not be extrapolated to the $N \to \infty$ limit (because it kept on growing), another filling factor was needed that could serve as the upper border of the "interesting filling factor-range":
Where 99% of the clustering has taken place can be situated by the filling factor, at which the number-of-clusters is only 1% of the number-of-spheres.

That definition of a threshold results in data that *does saturate* for high N - but it obviously works only for $N > 100$. A good fitting-function for the N-dependence reflects this:

$$\eta_{99\% clustering}(N) \quad = \quad a + b(N - N_0)^c \tag{5.10}$$

with N_0 a little above 100.

Table 5.3: Fit-Results for the N-dependence of the Saturation filling factor in $\eta_s(N) = a + bLog_{10}(N)$ for dimensions 1-11. Above $dim \geq 3$ the fitparameters - and thus the saturation filling factor - can be summarized in equation 5.9

dimension	Fitparameter a	Fitparameter b
1	0.4886 ± 0.11645	2.31157 ±0.03105
2	1.18622± 0.05181	0.46472 ±0.01382
3	0.81494± 0.02935	0.29044 ±0.00861
4	0.59509± 0.02018	0.20218 ±0.00586
5	0.42317± 0.02597	0.15092 ±0.00762
6	0.32889± 0.02813	0.10536 ±0.00825
7	0.22469± 0.01135	0.08473 ±0.00333
8	0.17796± 0.00931	0.05942 ±0.00273
9	0.12318± 0.00643	0.04843 ±0.00189
10	0.09297± 0.00602	0.03654 ±0.00177
11	0.06777± 0.00533	0.02696 ±0.00163

These fits were done for all the measured dimensions 1-11. Fitparameter a is again the $N \to \infty$ limit (because c is negative and thus the second term vanishes).

You can see the dimension-dependence of that $N \to \infty$-limit for all the measured dimensions in figure 5.14. That upper border of the "interesting filling factors" quickly falls for rising dimensions and can be fitted by:

$$\eta_{99\%clustering}(dim) \quad = \quad f \cdot g^{dim} \tag{5.11}$$

with $f = 3.976$ and $g = 0.5337$.
The data is not as good as for the other filling factors, but the general trend seems to be fulfilled. Moreover, the basis of this exponential fit function for the 99%-clustering point is almost identical with the 10%-clustering point (see equation (5.7)).

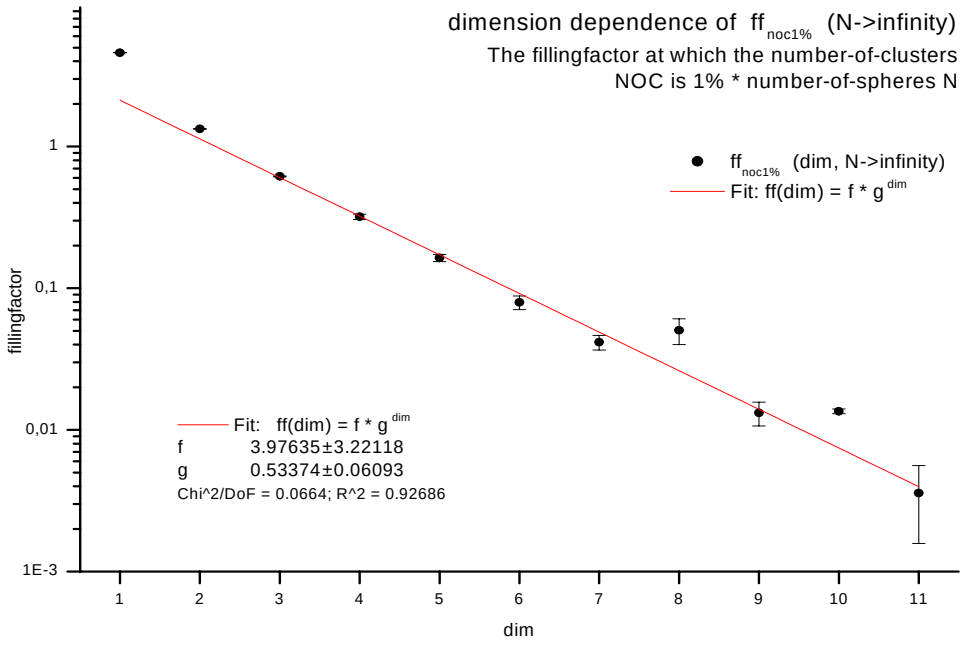


Figure 5.14: The 99%-clustering point at which the number-of-clusters is 99% of the number-of-spheres N. Experimental data after finite-size scaling, and the fit-function (5.11)

5.4.4 The Percolation Intervall

N-dependence

The critical filling factor lies between the 10%-clustering-point and the 99% clustering point (which is below the saturation point) - of course only in the non-degenerate situation of $dim > 1$ (see chapter 5.5 for that case $dim = 1$).

In figure 5.15 the dependence of those four filling factors on N is shown for $dim = 2$ and $dim = 7$. You can clearly see the ever-growing saturation-point at the top of the diagrams (chapter 5.4.2). In 2 dimensions, the critical filling factor "ffc" is rather close to the 99%-clustering point (called "ffnoc1", because the number-of-clusters is 1% of the number-of-spheres), while in high dimensions like dimension 7, it is closer to the beginning of the percolation intervall marked by the 10%-clustering point ("ffnoc90") - and the saturation (where *all spheres* are connected) happens much later.

Dimension Dependence

With those 3 filling factors that can be extrapolated to $N \to \infty$, the *percolation intervall* and its *dimension dependence* is shown in figure 5.16. It contains the "interesting" filling factors where the transition from (almost) **no clustering** over **critical clustering** to (almost) **full clustering** takes place.

Remarkable features:

- With rising dimensions, the critical filling factor lies closer and closer to the beginning of that intervall and the saturation above it in the overcritical domain takes (relatively) longer and longer.
- The two borders of the intervall, the 10%- and the 99%-clustering point, exhibit the same dimension-dependence (same slope).
- The degeneration of the one-dimensional case causes the crossing of the critical threshold and the 99%-clustering point below dim=2 (see chapter 5.5).

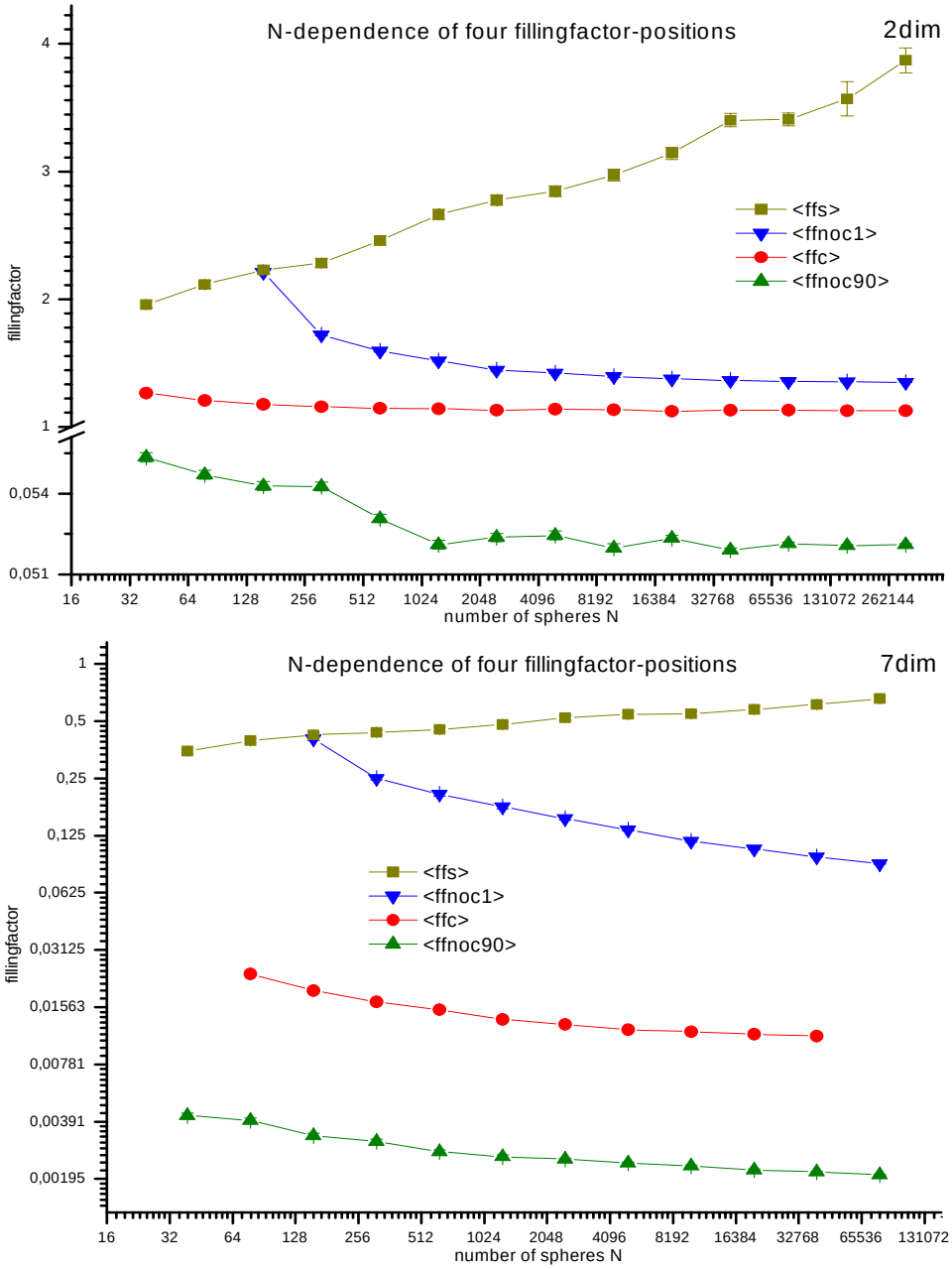


Figure 5.15: The N-dependence of the four measured filling factors in 2 and 7 dimensions. *ffnoc90*: Where the number-of-clusters NOC is still 90% of the number-of-spheres N ("10% clustering"); *ffc*: The critical filling factor where the spanning cluster occurs; *ffnoc1*: Where NOC is 1% of N (the "99%-clustering point"); *ffs*: The saturation filling factor where all spheres are in one cluster

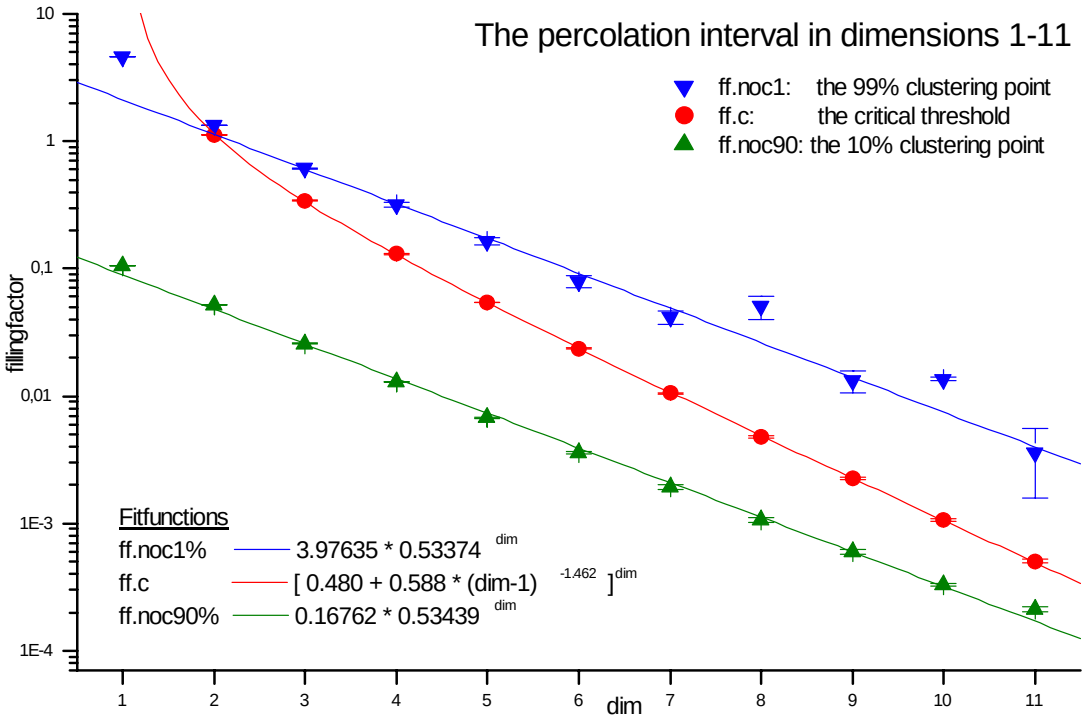


Figure 5.16: (**"THE" numerical result of this work in one diagram:**) The critical threshold is between the 10%- and the 99%-clustering-point in dimensions 1 to 11. This **percolation intervall** contains the "interesting" filling factors where the transition from (almost) no clustering over critical clustering to (almost) full clustering takes place.

5.5 The One-Dimensional Case

The one-dimensional case has to be treated differently because of its degeneration.
Easy to understand: The spanning cluster disappears as soon as only one single area of the 1-dimensional line is *not* filled by a sphere. So the *critical point* and the *saturation point* become identical. There is no overcritical domain.
The critical *occupied volume fraction* has to be $\phi_c = 1$, so from $\phi_c = 1 - e^{-\eta_c}$ (equation (2.22)) we know that the critical filling factor has to become $\eta_c = \infty$.

So the N-dependence of the critical point at which a spanning-cluster appears is totally different from the N-dependence in higher dimensions, please have a look at figure 5.17 and compare it to the $dim = 2$ and $dim = 7$ figures 5.15.
Here, in 1 dimension, the (finite-size) critical filling factor (or spanning-transition point) is *growing* without bounds for rising system size N. Rather good fit-functions for the measured critical and saturation point are logarithmic

$$\eta_{c/s}(N) \cong a + b \cdot \log_{10} N \qquad \text{for } dim = 1 \tag{5.12}$$

with $a_c = 1.16, b_c = 2.16$
or $a_s = 0.49, b_s = 2.31$

The 10% clustering point (which marks the lower edge of the interesting region) levels at ~ 0.105, the 99%-clustering point (where almost all spheres belong to the one cluster) levels at ~ 4.60 for high N, which for the mentioned reasons is *below* the critical point $\eta_c =$<ffc>.

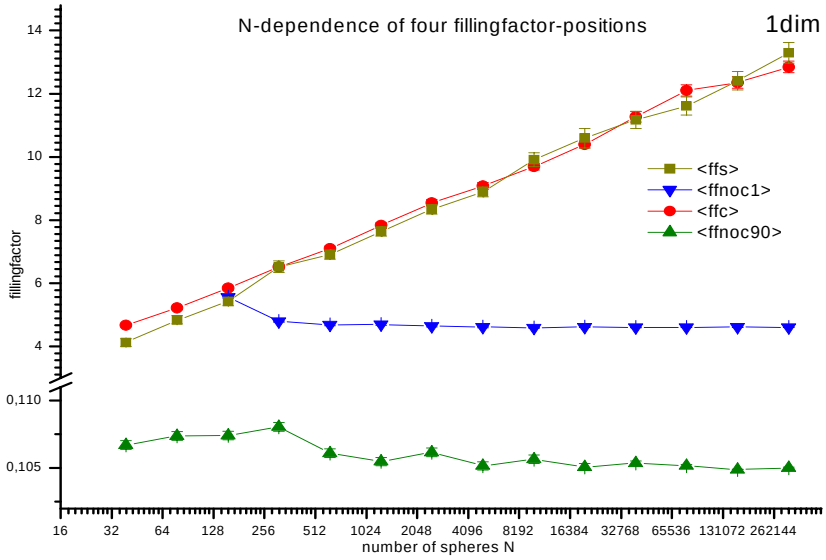


Figure 5.17: The 1 dimensional case is degenerate. Please compare to figure 5.15. In 1 dimension, the critical filling factor and the saturation filling factor are identical, they scale logarithmically with rising system size (equation (5.12)).

Chapter 6

Technicalities and Programming

As you can guess from the title, this chapter should be read with some insight into programming, as it is kind of a user manual for the source code. The less experienced reader might want to just skip it and move on to the last chapter.

However, if you are interested in recycling parts of my sources, I recommend especially sections 6.2.4 and 6.2.5 and the Appendix.

Introduction

Learning to program in C++ [55] was one of the main challenges of this work, now the sourcecode consists of 10000 lines, of which 8000 are code, the rest are lots of comments and some blank lines.

At the start I tried to model everything myself, even the data containers like linked lists, because I wanted to understand the basics of programming and to optimize them to the limit. But soon I realized, that I should use at least the STL (Standard Template Library) with its containers to save time for more important work (see chapter 6.3).

Now, the majority of the code is *not* for the algorithm described in chapter 3 anymore, but for analysis, statistics, data-handling, batch-starters, etc. - all the "onion-shells" around the core program.

In this chapter I will try to give some introduction to the C++ code; hopefully it will show programming principles and will give hints for this and other projects. There are lots of comments inside the sourcecode itself, so if you are interested to study it or to use parts of it for your programs, you will soon find it at

- `www.AndreasKrueger.de/thesis/code`

It is tested with GNU g++ 2.95 and MS Visual C++ 6 and doesn't need any compile options. Just compile `clcount.cpp`. Only the reserved stack size in MS Visual C++ has to be raised to 2,000,000,000 before.

6.1 The Code

6.1.1 The Files

While trying to keep order, I partitioned the code into now 24 files. There is not only a chronological, but also a structural order, in which the program can be understood. Just have a look at the include's in `configuration.h` to already understand a gross of it:

After building the `vector` and the `sphere`, I experimented with different containers like arrays and linked lists. Then the naive counters followed, now they have become the `cellcounter`s. The `cluster`s are to be handled and `analyze`d (spanning cluster, histogram, etc.). For some experiments `quicksort` and `boxing` were necessary, but the main work (and number of codelines) is done in `divide and conquer` - the new algorithm.

All the experimental data is kept in a new class, I called `measure`, it holds "data ± error" and cares for (simple) error propagation (most of the important mathematical routines +, -, ·, /, sin, exp, etc. are implemented with the respective derivatives for Gaussian error propagation) - one of the definitely

recycable parts of the program, as well as the `statistics`-part, in which the mean, variance, skewness and kurtosis of any data is calculated.

Just a transport class for a large amount of data is kept in `results`, while `datafiles` does the obvious.

In `counters`, composite counters with all initialization, analysis and data handling are stored, and `ff` contains the routines like intervall nesting, etc. that are necessary to pinpoint the critical filling factor.

For both programs, the numerical and the visualization, the same `constants` are used, mainly fit-functions for η_c and the table of optimal cuts, but also the maximal dimension for the vector.

In `clcount.cpp` the rest is loaded for the numerical part of the program only:

In `optimizealgo`, the counters can be started with ranges of different cuts per direction to find the fastest number of cuts for the divide-and-conquer; the `frontend` contains some user interaction and there are `speedtest`s. The big `starter`-programs and `batch`-routines contain the longrunners that can keep a desktop busy for weeks.

Either the menu asks for interaction at the beginning or the most important parts of the program can now also be started by `commandline`-parameters.

6.1.2 The Namespaces

Object oriented programming (OOP) was a new concept to me, so that I learned it only after a while, when a big part of the code was already programmed. The `vector` itself is a pure class with all the respective advantages, the `sphere` and the cluster (called `cluson`) are classes with public members and `measure` is even a templated class. But most of the other routines were not initially designed to be part of a class with inheritance, etc.

When looking for C++ language elements to still structure the heap of code logically, I found `namespace`s - such a new concept even in C++ that compilers like the GCC 2.8.1 of 1997 (the one installed in our physics department) can't handle it. Everything within a namespace is taken out of the global namespace and has to be addressed with the scope operator `::`, which prevents name-collisions and structures the program a lot.

Across all of the files, the subprograms are divided into logical namespaces now, e.g. so that there is *no* subroutine `makehistogram(...)`, it has to be called like this: `analyze::makehistogram(...)`.

The namespaces are:

- counters
- analyze
- grid (for first experiments with boxing)
- statistics
- results
- datafiles
- ff (**f**illing **f**actor)
- optimize
- frontend
- starters

I hope this work of structuring will help when the program keeps on growing or will be used by other people than me.

6.2 Data Classes

Some more details about the containers, they were at the center of the initial planning. In my opinion, when developing a new idea and algorithm, a good starting point are the containers for the objects that are to be handled.

6.2.1 Aliases for the Fundamental Types

In order to keep it flexible which type of variables might be the fastest and leanest, throughout the program I avoided to use the built-in types like `float, double, int, long`, etc.. Instead, I defined the aliases `REAL, COORDFLOAT, NUMBER, COUNTER` and `LONGBITS`, which now stand for `double, float, long, int` and `unsigned long`.

So e.g. the d-dim vector class of the next chapter consists of d `COORDFLOAT`s and not directly of d `float`s, but for the simplicity of the explanations, in the following I will name the types canonically.

6.2.2 The `vector`

Vectors in d dimensions hold (at least) d coordinates, numbers of the field that is underlying the vector space, in this case the field is of type `float`.

To store a set of numbers in normal memory efficiently and with fast access to the elements, the [ ]-*array* of C++ is the best solution, with the one disadvantage: the size has to be known beforehand (at compile-time):

```
const MAXDIM=11;

class myVector {
   float x[MAXDIM];
   int dim;
}
```

Now, after initializing a variable of type `myVector`, a subset of the array can be used to store the `dim` coordinates, the remaining `dim-MAXDIM` variables are just not used.
If this is to be avoided, and the dimensionality is still only known at runtime, another possibility has to be chosen: The vector itself then only consists of a pointer to a float(array), and when initialized, the array is created on the *heap*:

```
class myVector2 {
   float *x;
}
myVector2::myVector2(int dim) {
   x=new float[dim];
}
```

This results in a truely variable-dimensional structure that can even change its size at runtime, so no memory is wasted like in the first version. Unfortunately, this vector performs much worse, because of pointer access to heap data; when `dim==MAXDIM`, the first formulation is even **two times faster**!

I decided to waste memory and save time.

All the vector manipulation routines like $-$, $<<$, null-vector and random-vector are members of or friends to this class, so that they can access the (private) array of coordinates.

The Perfect Coordinate: `float` or `double`?

Many simulations are done on the lattice, not only for topological advantages, but also because integer-calculations are said to be faster.

At first in experiments with the cellcounter, I tried all variations for the coordinates, and `float`s performed best in this test, even better than `long`s, probably because they are smaller (4 bytes).

The clusterfinder with `float`-coordinates only needed 88% of the time using `double` or `long` - coordinates.

I didn't choose `int` because that would result in a too coarse lattice. Now every calculation is done in `double` because of the higher precision - but the coordinates are stored in `float` - to save memory and time. Nevertheless, it would not be difficult to redo all the simulations in `double` coordinates by just changing `COORDFLOAT` to `double` (see chapter 6.2.1).

6.2.3 The `sphere`

A sphere-container must hold at least the center vector and the radius. For all the algorithms it is useful to store the cluster*number* with the sphere - and for visualization, the cluster*size* might also reside here, then the colouring is easier to do.

```
class sphere {
   myVector center;
   double radius;
   long clusternumber;
   long clustersize;
}
```

The sphere-class contains routines for distance-measurement and overlap-checking. The calculation of the unitsphere-volume is in the same file.

6.2.4 A Container for Error-Bar Numbers: `measure<T>`

When I had the first real-data measurements and tried to calculate results with them, I noticed the always needed error propagation. The class `measure` closes that often occuring gap by providing the user with all important mathematical functions - and the class cares for the necessary error propagation. It is `template`'d so that it can be used for all wanted types.

```
template <class T> class measure {
   T average;
   T error;
}
```

The operators $+, -, *, /$ and functions like exp, sin, fabs, ... are implemented.
An easy example is the multiplication of two measurements with independent error-bars:

$$\begin{aligned} c \pm \Delta c &= (a \pm \Delta a) \cdot (b \pm \Delta b) && (6.1)\\ &= (a \cdot b) \pm \sqrt{(b\Delta a)^2 + (a\Delta b)^2} && (6.2) \end{aligned}$$

So with example values

$$\begin{aligned} \hat{a} &= a \pm \Delta a = 2 \pm 0.1 && (6.3)\\ \hat{b} &= b \pm \Delta b = 7 \pm 0.5 && (6.4) \end{aligned}$$

the result is

$$\begin{aligned} \hat{c} &= \hat{a} \cdot \hat{b} && (6.5)\\ &= c \pm \Delta c && (6.6)\\ &= 14 \pm 1.22066 && (6.7) \end{aligned}$$

Now with my new class `measure`, a calculation like this can be very easy, because it provides a new *type* for variables:

```
typedef measure<double> measure;    // create a double-valued type (once!)
measure a = measure(2, 0.1);
measure b = measure(7, 0.5);
measure c = a*b;
cout << c <<endl;

// on the screen, it results in 14±1.22066
```

So once the variables are defined, the usual operator-symbols and mathematical routines can be used in the canonical way. The library is tested extensively in Windows and Unix - and my statistics package uses it just as a storage container for measurements.

N.B.:

The Gaussian error propagation has one drawback: The error increases for each operation. And as the implementation is not done in symbolic mathematics, but as numerical calculations, it can't find the "shortest" formula by itself. Let's look at an instructive example. The calculation

$$\hat{d} = \frac{\hat{a} \cdot \hat{b}}{\hat{b}} \tag{6.8}$$

should result in

$$\hat{d} = \hat{a} \tag{6.9}$$

$$= a \pm \Delta a = 2 \pm 0.1 \tag{6.10}$$

but the C++ program

```
measure d = a*b/b;
cout << d <<endl;
```

still results in $2 \pm \mathbf{0.225425}$, because the error propagates once for `*` and a second time for `/`.

So if you are doing long calculations with error-bar numbers, use `measure` with care and deliberation - the error might be too high.

6.2.5 Statistics

Experimental data of computer simulations is often interpreted only *after* the simulation runs, using other programs like gnuplot, Origin or SPSS. The advantage of runtime-statistics, however, is that a) the data can be directly plotted without further analysis, and b) the program itself can react to statistical results. So e.g. it can do more simulation runs until a wanted accuracy is reached.

I have coded an abstract library that calculates from any (number) data the four central moments of distributions: the mean, variance, skewness and kurtosis. It takes a (STL)-list of arbitrary length and returns the four moments and their standard-deviations (four `measure`-objects).

It is `template`'d, so that the ingoing data and the outcoming results can be of different types. The collection of the four moments is a class called `muvarskewkurt<result-type>`; the averaging-routine is called `calculate_moments<result-type, listelement-type>`.

Before the elements of the list `L` are summed to get the central moments, a preprocess-function can manipulate each element, so e.g. square it or shift it by a constant value, which prevents the necessity to create a second list of numbers, if one is interested in other aspects of the data. If this is not wanted (the standard case), the data is manipulated by the function `linear`, which does nothing (still, the shift value `manip_argument` and the return-type-`dummy` have to be given (see below) as dummy-values)

The following code should be self-explanatory, it creates 500,000 random numbers between 0 and 1 and calculates and prints the four moments; once for the distribution of the *uniformly distributed* numbers and a second time for the *squares* of these numbers:

```
#include <iostream>    // for cout
#include <list>        // a linked-list from the STL holds the data
using namespace std;

#include "statistics.h"
using namespace statistics;
void main(){
  cout <<"Test statistics with a list of 500,000 random numbers.";
  list <float> L;
  for (int i=0; i<500000; i++)  L.push_back( rand()/(float)RAND_MAX);
  int manip_argument=42; // dummy (unused due to parameterless data manipulation linear, square)
  double dummy=0.0;      // dummy to set the result-type to double
  muvarskewkurt<double> A, B;
  A = calculate_moments<double,float>(&manipdata<float>::linear, manip_argument, L, dummy);
  B = calculate_moments<double,float>(&manipdata<float>::square, manip_argument, L, dummy);
  cout <<"The distribution has these statistical properties:\n";
  cout <<"average mu, variance, skewness, kurtosis = \n";
  cout <<" linear:\n  "<< A <<"\n square:\n  "<< B <<endl;
}
```

After a few seconds, it prints:

```
Test statistics with a list of 500,000 random numbers.
The distribution has these statistical properties:
(average mu, variance, skewness, kurtosis) =
linear:
mu=0.500276±0.288418 var=0.0831846±0.0745583 skw=-0.00185585±1.96797 krt=1.80335±2.40927
square:
mu=0.333461±0.297904 var=0.0887464±0.0950127 skw=0.639098±2.49657 krt=2.1462±4.46682
```

While the uniform random numbers are distributed around 1/2 symmetrically (skewness ≈ 0) with a low kurtosis, the squares of these numbers have a mean of 1/3 and are skewed to the left with a higher, a little more Gaussian (would be 3) kurtosis.

6.3 Containers for Many Objects: Array Contra Linked-List - and the STL

What is the optimal structure for data? This question arouse after the single elements like `sphere` were ready to use.

Often the algorithms themselves decide about this question, because it might (not) be necessary to randomly access each element, or there might be collections of varying size.

6.3.1 Some Important Containers

Arrays are fast and not flexible

The standard-C array "[ ]" is a good solution for any given-size data collection, it allocates a successive chunk of memory and is very fast to access, because each element gets an definite address ($O(1)$-access).

On the other hand, it is impossible to increase the array size or insert elements at random positions.

Linked-List

A linked list is a linear structure of connected "nodes" that hold data. A single node contains data and at least one pointer to its successor (in a double-linked list, each node also knows its predecessor), so that it can be iterated from the start to the end, reading out the data of all nodes.

The big advantages of lists are the variable length and the ability to insert data-nodes after any given point (by changing only one pointer in the node before and pointing from the new element to the old successor), the disadvantage is the lack of address of the nodes - a single data-node can be found only by iterating the whole list until that node is reached.

I programmed a linked-list following the excellent book of Sedgewick [50], it worked well and it was fast. But it didn't have an iterator-friend-function, so only one iteration at a time could be done per list.

When I learned about the STL (see chapter 6.3.2), I gave it up to program my own standard containers, but it actually was a very instructive exercise to have done it.

Hash-Table

A hybrid structure of the above is a *hash* or *map*. A node of a map contains a key, the data and (at least) one pointer. The key can be of any type and directly adresses the data in the right node - and the pointer-structure makes it possible to iterate the whole structure linearly from any given key (node) on, because the elements are inserted using an order-function for the keys, so they are *sorted*.

As an example, if we want to store the population of countries, the key should be a `string` to hold the country name, the data would be of type `long`, and the sorting order would be lexical (using the STL-order function `less`):

```
#include <map>
using namespace std;
typedef map<string,long, less<string> > string2long;
string2long population;
```

```
population.insert(string2long::value_type("Germany", 80000000));
population.insert(string2long::value_type("China", 1300000000));
cout << "The population of Germany is ";
cout << population["Germany"] << endl;
\\ The result is
\\ The population of Germany is 80000000
```

6.3.2 STL - the Standard Template Library

The Standard Template Library STL provides C++-programmers with the standard containers to store data and algorithms to iterate and manipulate that data. In contrast to my first intention to create own structures to be able to fully control them, I now see the advantages of the STL and recommend it highly. The containers have standard-interfaces and -behaviour, the library is tested and optimized and it is ported to almost any platform.

If your program is designated to run on *any* platform, you can even compile your own STL by using the OpenSource product "STLport" [48]. Then the *same* STL will be used everywhere.

6.4 The Core-Algorithm

Some remarks about the *implementation* of the cellcount. It was important to optimize it thoroughly, because it is the "inner loop" of the whole program.

All the described improvements concerning the `overlap` -checking function speeded it up by a factor greater **3** (!) in the extreme case of 20 dimensions.

Most of the tricks presented here might be of general interest for your programming style of time-critical parts in your program.

6.4.1 Passing by Reference

If a subroutine needs data to process it, the data can be passed directly, then automatically, a copy of the data is made for each call of the subroutine.

Especially for large structures (e.g. a 10-dim sphere is 64 bytes), this takes a lot of time, so C++ provides a better option to pass data: "passing by reference", using the `&`-operator in the function-head parameter list. Through the reference, the data will be accessed on its original memory position without the need of copying it (similiar to an access by pointer, but not in pointer-syntax).

For example, the overlap-checking function needs sphere 1 and sphere 2 to check if their center-vectors are closer than the sum of radii. If the spheres are passed by reference:
`bool overlap(sphere& sph1, sphere& sph2){...}`, it only takes 85% of the time that a copy-of-sphere passing needs.

6.4.2 Avoid Implicit Temporary Variables

When analyzing execution times, I measured that the overlap-checking `overlap(sph1,sph2)` takes about 60% of the total running time! More detailed, the vector `operator-` that creates the distance vector between the two centers, takes 40% of that. How could that be optimized?

An operator- is binary: `a-b`, it needs a temporary (internal) variable to store the result until it is put into some target variable or used elsewhere. This temporary variable is created and destroyed for every single subtraction, which takes up a lot of time.

I improved the process by passing a third variable `temp` which is created once and used over and over again: Now `overlap(sph1, sph2, temp)` can use the *much* faster operator-= which is unary.

6.4.3 Square'ing Instead of Square-Root

To check two spheres for overlap, the distance is compared to the radii (equation (1.9)):

$$\sqrt{\sum (x_i - y_i)^2} = \| x - y \| < R_1 + R_2 \quad (6.11)$$

$$\Rightarrow \sum (x_i - y_i)^2 = \| x - y \|^2 < (R_1 + R_2)^2 \quad (6.12)$$

As (6.12) eliminates the square-root in favour of a (cheaper) multiplication, the second version takes only 90% as long as the first version (for 10000 spheres, at critical radius in 2dim).

6.4.4 Don't Check Visited Spheres

One of the most important improvements is algorithmically, not by changing some technical implementation details.

Please look at the lines 4 and 5 of the subroutine `SUB neighbour` in chapter 3.1.1.

There are two possible orders for the 2 condititions "overlap" AND "yet uncounted":

1. `if (overlap(sph1,sph2) && sph2.clno==0) THEN recursion`
2. `if (sph2.clno==0 && overlap(sph1,sph2)) THEN recursion`

Version 2 takes *much* less time because the already clustered spheres are *not checked for overlap* again.

For [8dim, 1000 spheres], in the undercritical and critical situation, this already saves about 50% of the time, but far in the overcritical domain, even only 6% of the running time of the other version is needed!

6.4.5 How to Store and Count Subsets of Spheres

During the divide-and-conquer recursion, the total set of spheres is divided into smaller and smaller sets. Those sub-sets have a differing number of spheres, so the spherenumbers of one cell have to be stored in a variable-sized object: a linked-list.

Thus, to do the naive cellcount in one cell, at first it seemed advantageous to iterate through all those spherenumbers using these lists, but unfortunately this takes about 200-300% of the time compared to an arraycount with spheres stored in a definite-sized array!

The solution I chose was to copy the spheres of one cell into a temporary (big enough) array. The copying takes some (very short) time - but then the array can be used to find the clusters of that cell with a normal arraycount. And of course, the temporary array is recycled, it is created once and destroyed only after all cells have been clustered.

6.4.6 Structure of One Throw

Obviously it is neither necessary nor possible to describe such a big program in detail, but for all starter-programs, it might be useful to understand what it does in one realization. A short schematic overview:

1. throw N spheres
2. find clusters recursively
   (a) reset variables (clusternumbers, clustertable, ...)
   (b) i. divide into cells (recursively) (DIVIDE-AND-CONQUER)
   ii. find clusters in one cell (CELLCOUNT)
   iii. combine cells (COMBINE)
   (c) results: clustertable, execution time
3. spanning?
4. clustertable → histogram of clustersizes
5. delete clustertable

6. histogram analysis
    - number of clusters NOC
    - size of biggest cluster B
    - mean size of finite clusters M
7. delete histogram

The (saved) results are the **spanning directions**, NOC, B, M and the **execution time**.

6.5 The Starter-Programs

Like every other big program, my program can be understood as a succession of concentric "onion shells", the overlap-checking of two spheres in the "onion-core", the cellcounter in a "onion-shell" around it; then further outside, the divide-and-conquer and so on.

The outer shell of that "onion" are the *starter-programs*, which interact with the user to take in the chosen parameters and execute all necessary steps for the simulation. Depending on the parameters, a single run of a starter-routine might take weeks to finish its task.

There are several types of starter-programs using all (or some of) the inner routines:

1. Throw one realization and solve the problem with different algorithms: analysis of time complexity.
2. Find the optimal number of cuts for the divide-and-conquer.
3. Find η_{crit} by intervall-nesting - and repetition over many of those realizations;
   repeating it until $\Delta\eta_{crit}$ is smaller than a wanted accuracy
   (this was used for the results of chapter 5)
4. Scan a filling factor range with a given stepsize (e.g. 90 steps in $\eta \in [0.8 - 1.7]$ in 2 dimensions) to see the behaviour of the spanning probability $P_{spanning}$, the number-of-clusters NOC, the size of the biggest cluster B, etc.
   For a succession of rising filling factors, the simulation is started again and again, statistical mean values are calculated, etc.
   This starter-program can be used to produce data like figures 3.2 and 7.1, the percolation functions.
5. Within a range of dimensions d and number-of-spheres N, scan filling factor-ranges (like in 4.)

Some of those starter-programs are now accessible through commandline parameters, so that without any user interaction, the program can be started for batch processing in a queue like condor [62].

6.5.1 How to Smooth the Curves

One of the advantages of runtime-statistics is that the algorithm can itself "observe" its results. Around the critical threshold, the fluctuations increase, so the 4. starter algorithm described above can do more runs in that region proportional to the variance of the result. The resulting curves are much smoother and this method saves time because it smoothes the curves only where that is necessary.

6.5.2 File Formats

Data can be analyzed in various ways. The targeted result at the moment, to find statistical estimators for the critical thresholds, is "hardcoded" in the program - and on the way up to that abstract numbers, a lot of simulation data is summarized and then discarded afterwards.

As I could not foresee which other aspects of the data might be also interesting later, I implemented data-save routines on several levels of the program. Up to now, about 300 MB of ASCII-coded data have been saved to files.

For example, during the "find-critical-filling factor" routine, three different filetypes are saved which represent three levels of statistical averages. A line in such a file consists either of:

1. The 5 observables measured in a single point set realization at one filling factor (sphere radius):
   (not)spanning, NOC, B, M, execution time
   (The results of chapter 6.4.6 during the intervall nesting)

2. The *spanning* filling factor of one point set realization with the accumulated mean, variance, skewness and kurtosis of the distribution of all previous critical filling factors in that configuration (see the plots in figures 5.5 and 5.6)

3. The best, averaged results for each (N, dim):
   mean, variance, skewness and kurtosis; total running time
   (=last line of 2.)

While there are hundreds of megabytes of the first datatype, only about 100 kilobytes of the last datatype exist. The plan is to create a database with all the tables and analyze it "diagonally" for other aspects than the one described above.

6.6 Compilers

The ANSI standard of C++ is platform-independent, nevertheless I had to learn that there are still different "dialects" among the compilers. It was a hard task to port the fully functioning program from *MS Visual C++* on Windows to *GNU g++* on Unix, because of slight differences in formulation. The about 150 "errors" after porting could be eliminated almost completely, so that now my written syntax of C++ has become platform-independent.

One inconsistency (introduction of templated friend functions in the head of a class) remained and had to be solved by a preprocessing directive for the compiler, so that here now two different code snippets exist for GNU and Microsoft C++ (see the class head of `measure.h`).

6.6.1 Microsoft Visual C++

The programming environment of MS Visual C++ is very comfortable, the detailed online help explains every C++ command and gives a lot of code examples for the STL. This remarkable comfort allows it to learn the language interactively; even by try-and-error, as the compiler results directly point back to the responsible code lines.

There are "browse-informations" generated that allow it to jump around in the C++ code by point-and-click (not only a class browser is provided, but any definition can be found automatically, etc.).

The "Win32 Debug" compiler settings generate executable code, that can be graphically debugged to find errors by visiting the variables during runtime - but in "Win32 Release"-settings, the execution time is then only ∼47%.

The execution time can be further sped up by the manual settings "no MFC" (95%) and "incremental binding": (96%).

6.6.2 GNU g++

One disadvantage of the MS Visual C++ is the "sloppy" C++ syntax; it accepts a lot of formulations, that other compilers don't accept, so one tends to write "Microsoft-C++" that is then more difficult to port to other platforms.

Therefore, for portable programming projects, I suggest to use 2 compilers at a time during coding to find the idiosyncrasies early enough: The OpenSource "`cygwin`" unix-shell for Windows [63] makes it possible to run GNU GCC in Windows, so that it is very easy to compile the same C++-code-files with MS Visual C++ *and* GNU g++.

GNU gcc [18] is closer to the ANSI C++ standard, it is for free and OpenSource, moreover, it is ported to many platforms, so try to program GNU GCC c++ style!

To compile my program with gcc, no commandline-switches are necessary, the optimizer `-O2` is suggested:

- `g++ clcount.cpp -O2 -ocontinuum_percolation.out`

You need a rather new compiler version, version 2.8.1 doesn't compile it because of various developments in recent C++ (like namespaces), v2.95.2 works perfectly!

Unfortunately, on the computer-cluster of our university, version 2.8.1 (of 1997!) was installed; I asked often and I waited more than *3 months* until gcc was upgraded to the new version 2.95.2.
Obviously, the computer admin of the physics department desperately needs a full-time assistant!

Chapter 7

Outlook

7.1 The Achievements

The percolation thresholds were measured numerically in the first 11 integer dimensions of euclidean spaces filled randomly with equally sized spheres.

Three other filling factors were located in the same way to establish knowledge about the percolation interval: the 10%-clustering point, the 99%-clustering point and the saturation point.

May these numerical results simplify future examination of continuum percolation.

7.2 The Consequent Pathway

7.2.1 The Next Research Goals

Critical Exponents

The next, most important step is to measure the critical exponents of these singularities at the critical point. Please revisit chapters 2.5 for the importance of the critical exponents; the experimental results for our *continuum* percolation model should be proving to be the same as those for *lattice* percolation shown in figure 2.7

Moreover, especially the "transition" of the critical exponents from the finite dimensional (and dimensional-dependent) values to the infinite dimensional values at dimension 6 could be interesting to examine.

A Percolation Formula

To learn more about critical systems and to model real-life-data phase transitions into percolation models, it might be also useful to study the observables in the whole range of filling-factors, also in off-critical situations. There is still no "percolation-formula" for the observables, like e.g.;

- P_∞ =function(R, N, dim), the percolation probability
- NOC=function(R, N, dim), the number of clusters
- $\chi(p)$ =function(R, N, dim), the mean size of the finite clusters
- B =function(R, N, dim), the size of the biggest cluster
- ...

Looking at figure 7.1, you can see that all the tried sigmoidal functions (Sigmoidal Richards, Gompertz, Weibull, Hill, Boltzman, Logistic) don't fit the experimental data for the size of the biggest cluster. Especially in the overcritical domain, the real percolation data saturates much more smoothly than any of the sigmoidal functions. How could a "percolation function" look like that reflects this behaviour better?

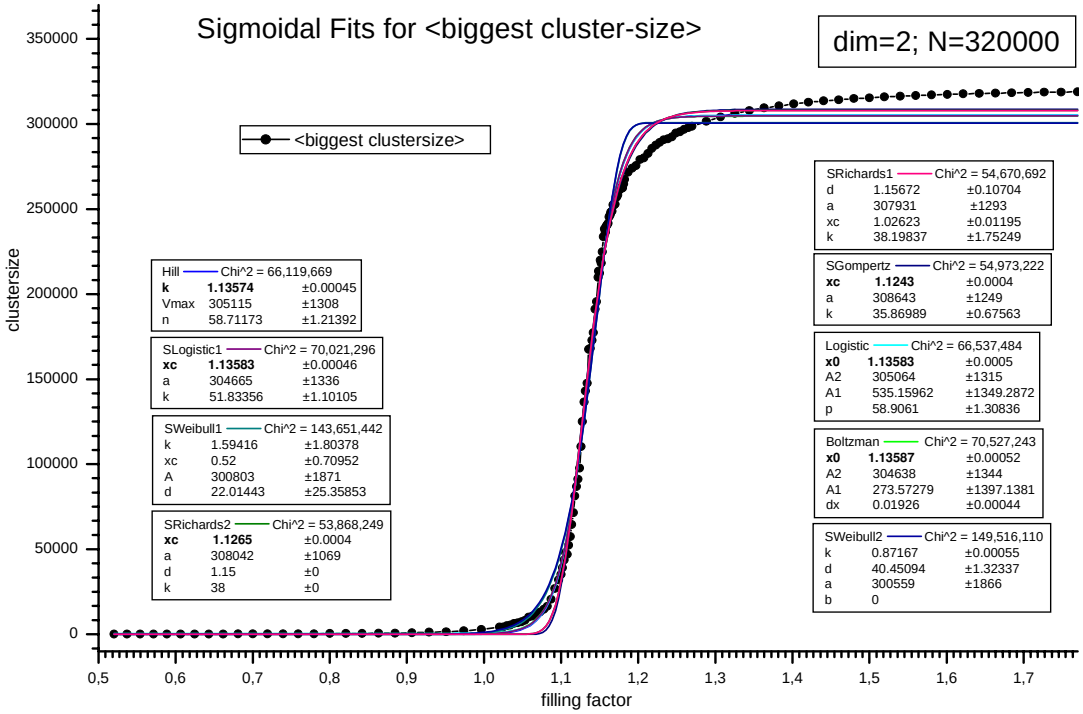


Figure 7.1: These sigmoidal functions don't fit the percolation data.

Predicting a Percolation Transition

Connected with this question is the task to find indicators for for an upcoming onset of percolation, a criterion below p_c that tells us how far the system still is from its percolation transition (the momentary candidate being the *number of clusters* - see figure 1.3).

The Connection of Lattice and Continuum Percolation

It would be very useful to have a *transformation prescription* how to transform *lattice percolation* results (e.g. p_c) into *continuum percolation* results (e.g. η_c); the former can have only one object on each lattice vertex which has a definite maximal number of neighbours ($2d$ in d dimensions), while the latter is using soft spheres which can multiply overlap with a variable number of neighbours. In recent works, Galam and Mauger [19] have introduced an universal formula for percolation thresholds on many lattices, it could be interesting to subsume the continuum percolation thresholds into that formula.

The Critical Number of Neighbours

To see where exactly the Poisson distribution is a good approximation (chapter 2.3.2), the mean number of neighbours should be measured directly in the simulation - not only but especially at the critical point.

The parameter "mean number-of-neighbours" might be a more straightforward, instructive and useful observable than the rather abstract density "filling factor" if one wants to apply d-dimensional percolation to real-life problems.

The Dimension of the Critical Cluster and the Correlation Length

Two other observables have not been modelled into the program yet: The *fractal dimension* of the spanning cluster at criticality and the linear size of the biggest cluster, which gives the *correlation length* of the process. Those two complete the set of critical exponents for hyperscaling by D and ν.

Non-Integer Dimensions

It could be interesting to study percolation in non-integer dimensions, to do simulations of percolation on fractal structures. For example, because of the divergence at $dim = 1$ and the condition: dimension $d \in \mathbb{Z}^+$, it is difficult to decide between several possible fit-functions for the percolation threshold in d dimensions (see chapter 5.3.5).

A candidate for a structure with "tunable dimensionality" is a branching process (Caley) tree, but with differing branch lengths (If all branches have the same length, a binary tree has dimension ∞, but if the branch-length doubles in each generation, the dimension is 2. In between those extremes, one should be able to generate all possible dimensions).

7.2.2 The Computer Program

Apart from the ideas in chapter 4.4.1 how to redesign the visualization, the simulation core itself should be improved for clarity and better performance:

- The code "grew" to its actual form - it should be tidied up to a more logical structure.
- The combine routine could use $d-1$ dimensional divide-and-conquer of the adjacent surfaces to merge two d dimensional cells
- Boxing rather than multiple splitting for the divide-and-conquer to avoid multiple coordinate checking
- Adaptation of the Hoshen-Kopelman approach (chapter 3.4.3) to a) continuum percolation in b) variable dimensions to create a linear algorithm
- Experiments with different algorithmic approaches, e.g. iterating a distance-sorted connectivity list gotten by full triangulation
- Store all the data in one database instead of hundreds of datafiles

Others shall profit from the coding work, so one goal is to publish an OpenSource-library for Continuum Percolation.

7.2.3 Related Models

Distributions of Differently Sized Spheres, Spheres of Influence

Although most routines in the program are already working with individually sized spheres, all my simulations up to now have always used N spheres of the *same size*.

But which distributions of sphere-sizes are interesting? What is the "natural" extension of equally-sized spheres?

There is e.g. a model called "Spheres of Influence", in which the radii are a consequence of the actual point distribution. (Please have a look a figure 7.2; the nearest neighbours were found by using the Delaunay-Triangulation [51] of a point set.). Each sphere gets a radius that equals the distance to the nearest neighbour. Some initial simulations show, that this system might be always overcritical, the averaged filling factor in 2 dimensions seems to be exactly 1.0.

Density Distributions

Especially for applications, like modelling QCD quark-color-tubes (strings) in particle collisions [45, 9], it might be interesting to vary the density of points locally, e.g. there are more particles in the center than at the outside of the collision area.

Other Shapes

Spheres are *the* natural simplification of geometrical objects in physics.

But varying the shape of the objects - how does it affect percolation results? The first generalization are ellipsoids with a tunable aspect-ratio between the d different radii, once the overlap-function for ellipsoids

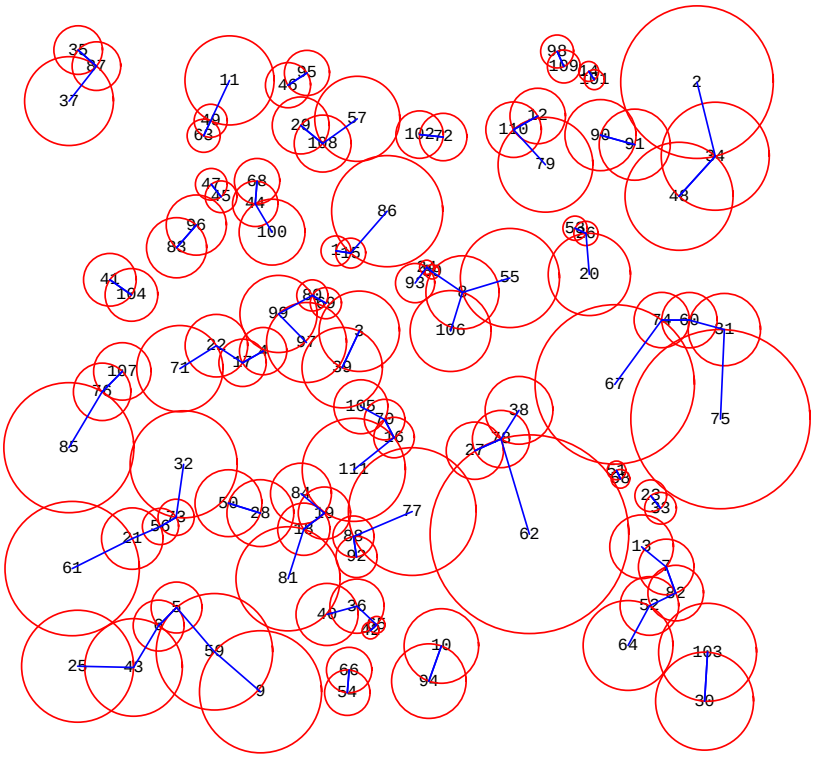


Figure 7.2: "Spheres of Influence" ($N = 111$). The radii are the distances to the nearest neighbour (blue lines). (When boundary-effects are masked) the mean filling factor after many runs seems to be just 1.0 and the system is very often just (percolating) over-critical.

[38] is defined, the C++ program can generate the results straight-forward; but quite as well, one might be interested in percolation of rods, squares, random-shape objects, etc.

The clusters themselves are shaped objects - what approach is the natural one to model this insight into a "percolation of clusters"-structure (instead of percolation-of-spheres) to use renormalization?

Objects with Properties

The spheres could vary among each other in *size* (see above) to model different interaction *ranges*, but also *qualitatively* by adding discriminating properties.

At the moment, the only property of a sphere that influences the connectivity is its radius. The model could be enhanced by sphere properties like *spin* or *color*, which would break up connections between overlapping but differently coloured spheres.

Characterization of Point Processes; Minkowski-Functionals and QuermaßVectors as Shape-Finders

A possible measure for quantifying point processes are the $d + 1$ skalar *Minkowski-Functionals* and the vector-valued *Quermaß-vectors*, used mainly in galaxy-distribution analysis [31, 4]; they decorate the point set in question with spheres, and then give a value for the cumulated volume, surface, mean curvature, etc.. They are suitable to compress the complex morphology of huge point sets to just $d + 1$ numbers (or vectors). Among these "shape-finders", the *Euler characteristics* is the last; interestingly, it has a zero shortly below the critical percolation threshold [7]!

I would like to implement these Minkowski-functionals to study them further in percolation and related processes, especially in non-uniform (non-Poisson) distributions.

Particle Interaction

Apart from the excluding condition of hard sphere models or lattice percolation (only one object per vertex is allowed), the objects in percolation do not interact at all, which is an over-simplification for most physical situations.

If one introduces local attractive interaction, a stronger clustering occurs, research is done e.g. in the field called "grain growth" [44, 27].

7.3 Network Analysis

For more than 100 years now, we are communicating over electronical networks, the Internet is the biggest (fully-connected!) artificial structure on this planet, and also our social relations seem to be modelled more adequately in a network-like manner than on a lattice or in euclidean spaces, because of the wide variation of the number of neighbours of one "node".
The language of percolation can be useful to study phenomena in simple networks. Many questions are interesting - like the *pathlength* for messages depending on the intensity of connectivity or the morphology (Poisson distribution contra scale-free networks [2, 35]). The pathlength-research corresponds to the so-called *small world* (SW) patterns in collaboration and citation networks [3, 35, 61] or in food webs [33].

When does a network break into 2 islands? This question might yield some insight into the stability of networks during normal use or under attacks [12].

What kinds of contact processes are used by computer-virusses that infect parts of our computer networks [37]- or by semantic "meme"s [13, 14] that travel through our society using the contagious process of forcing their victims to talk about the meme? What property makes them so successful that they can reach the overcritical domain so that the infinite cluster appears and the macroscopical phase transition "popularity" happens?
And why not create a "percolation meme" in form of a distributed-computing client that is dedicated to percolation simulations of meme networks; it "lives" on the Internet because its hosts have downloaded our software to share their CPU computing power.

7.4 Your Influence

Many thanks for reading this. Please tell me your opinion.

Which parts would you recommend to extract for publications?

For me, it now depends strongly on your resonance about this work, and on funding, in which direction I will intensify these studies.

Acknowledgement

Thank you, the reader.

Special thanks to all the people who helped me to understand the problems and to express them in increasingly less words from talk to talk:

Daniel Gandolfo, Philippe Blanchard, Helmut Satz, Stefan Reimann, Andreas Tupak, Friederike Schmidt, Olaf Lenz, Harald Lange, Andreas Ruschhaupt, Markus Prevot, Andreas Kaiser, Frank Meissner, Matthias Strobl, Dietrich Stauffer, Nèstor Armesto, Jonathan Bowen, Frank Brand, Ulf Ehlers, Michael Doering, Tyll Krueger, Oliver Preuss, Christian Pfeiffer, Christian Greiffenhagen - and many more.

In deep gratitude I would like to thank especially Iris an der Heiden, for all the patience with me and the endless talks about percolation, and most of all, to Egon & Ilse Krüger, who sponsored my long studies generously.

List of Figures

List of Tables

Appendix A

Appendix

A.1 Pseudocode

Pseudocode is used to explain parts of the algorithm in an abstract way, so that it should be possible to implement into any existing computer language.
The following part of the program can be called with different parameters, executes and returns a result:

```
SUB subname (parameter)
do something
create a result
subname=result
END subname
```

Call the subroutine N-times with i from 1 to N:

```
FOR i=1 TO N
CALL subroutine(i)
NEXT i
```

Conditioned execution of commands:

```
IF condition THEN
CALL subroutine1
CALL subroutine2
ELSE
CALL subroutine3
ENDIF
```

conditions (they return TRUE or FALSE):
`a == b` a equals b
`a <= b` a equals or lower than b
`overlap(a,b)` spheres a and b overlap or not
etc.

operators:
`a++` increase a by 1
`a--` decrease a by 1

access to properties of objects:
`sph1.R` radius R of sphere sph1

A.2 The Mathematica Programms

The main feature of Mathematica© notebooks is the executability. The reader can study a longer thought or experiment live by evaluating the whole paper cell by cell. For this reason, the following notebooks are

published on the Internet so that you can download and execute them directly (provided that you have Mathematica© v4.0 or higher). This is an instructive way to understand the facts stated throughout this paper and seems to me a more reasonable use than lengthening this document by another 50 pages on paper.

The main page for all the running computer programs is

```
www.AndreasKrueger.de/thesis/exe/
```

A.2.1 Lattice Percolation

At first mainly programmed for the images in the first part of this thesis, this notebook now contains a complete lattice clustercounter for 2 dimensions using the linear Hoshen-Kopelman strategy [24]. It clusters about 10000 vertices per second. A lattice can be filled with probability p for each vertex or for the whole lattice. The clusters are analyzed by the most important percolation observables - and can be output into graphics. A whole range of p can be scanned and in a long simulation (about one day) for several rising system sizes. Then a finite-size scaling is applied to find the ∞-limits.

Please download this notebook from

```
www.AndreasKrueger.de/thesis/exe/lattice-percolation.nb
```

A.2.2 Continuum Percolation

The filling factor and π_d is defined, a naive and a Hoshen-Kopelman clusterfinder is implemented, the time complexity of the algorithm is studied (This HK algorithm clusters about 1000 spheres per second). Then cluster analysis routines are applied, histograms of clustersizes shown, the optimal dividing for the boxing is found, and visualizations of the configurations can be plotted. At the moment, this is only implemented in 2 dimensions.
A second notebook contains the model for the titlepage with varying η and N in one configuration. It was produced mainly for visualization, but includes the necessary algorithms for summing up the clustermass as the total sum of areas (not as the total *number* of spheres) and for assigning colours by a partial sum of cluster masses in increasing mass order (it is the first fully functional - though slow - implementation of the idea "colour lens" mentioned in chapter 4.4.1).

Please download these notebooks from

```
www.AndreasKrueger.de/thesis/exe/continuum-percolation.nb
www.AndreasKrueger.de/thesis/exe/titlepage.nb
```

A.2.3 The Poisson Distribution - Number of Neighbours

The basic properties of the Poisson distribution are studied, then it is related to the number of neighbours per sphere in continuum percolation. This number of neighbours is compared to the number of neighbours in lattice percolation.

Please download this notebook from

```
www.AndreasKrueger.de/thesis/exe/poisson-distribution.nb
```

A.2.4 Hypercubes and Their Properties

In this notebook, d-dimensional Hypercubes can be drawn, a recursive formula for the number of subobjects is given and simplified first to an iterative formula, then to equation (2.12). Then the table of subobjects is analyzed.

Please download this notebook from

```
www.AndreasKrueger.de/thesis/exe/hypercubes.nb
```

A.3 The C++ Sourcecode of the Simulation Program

Now that you have finished reading this paper, another 135 pages are awaiting you - the sourcecode of the C++ simulation engine and the visualization program is approximately 9000 lines long. You can download it at

`www.AndreasKrueger.de/thesis/code`

Besides a version history, there are several variations; a plain-text file, a C++ syntax color-coded PDF file and an archive of all the separate include-files to compile the program on your computer.
I would really like to provide a fully linked sourcecode browsing like in Java, and I appreciate any hints leading to such a solution - is there anything available like "c++2html" ?

To study the sourcecode, at the moment I recommend the color-coded version, the C++ syntax is much easier to read (I am searching for a `cpp2html` - browser tool, please give me a shout if you know a good one). Please consult chapter 6 for explanations about the design structure of the program.

Bibliography

[1] Alon, U. and Drory, A. and Balberg, I. (1990) *Systematic derivation of percolation thresholds in continuum systems*, Physical Review A 42, 8, 4634

[2] Albert-László Barabási (2001) *The physics of the web*, http://www.physicsweb.org/article/world/14/7/9 as on 1.08.2001

[3] Barrat, A. and Weigt, M. (2000) *On the properties of small-world network models*, Eur.Phys.J.B 13, 547; cond-mat/9903411

[4] Beisbart, Buchert and Wagner (2000) *Morphometry of Spatial Patterns*, astro-ph/0007459

[5] Blanchard, Ph. (1993) *Zufallsgraphen, Perkolationstheorie und HIV-Ausbreitung*, Phys.Bl. 49, 12, 1116-1118

[6] Blanchard, Ph. and Gandolfo, D. (2001) *Percolation: Concepts, Tools and Applications to Real World Phenomena*, The science of complexity, (CD-Rom) ZiF University of Bielefeld

[7] Blanchard, Gandolfo and Ruiz,J. (2001) *Euler-Poincar Characteristic and Percolation in the Random Cluster Model*, The science of complexity, (CD-Rom) ZiF University of Bielefeld

[8] Bowen, J. (1996) *Hypercube images*, http://www.cs.reading.ac.uk/archive/hypercubes/ as on 2.10.2001

[9] Braun, M.A. and Pajares, C.(2000) *Implications of color-string percolation on multiplicites, correlations, and the transverse momentum*, Eur. Phys. Journ. C 16, 349-359 (2000)

[10] Broadbent, S.R. and Hammersley, J.M. (1957) *Percolation processes I. Crystals and mazes*, Proc. Cambridge Phil. Soc. 53, 629-641.

[11] Clay Mathematics Institute (2000) *Millenium Prize Problems: The P versus NP Problem*, http://claymath.org/prizeproblems/pvsnp.htm as on 17.04.2002

[12] Cohen, Erez, ben-Avraham and Havlin (2001) *Breakdown of the Internet under intentional attack*, cond-mat/0010251

[13] Dawkins, Richard (1977) *The selfish gene*, Oxford University Press, London

[14] Dawkins, Richard and Blackmore, Susan (2000) *The Meme Machine*, Oxford University Press;

[15] Ferreiro, E.G. and Pajares, C. (1998) *Fusion of strings vs. percolation and the transition to the quark gluon plasma*, Nuclear Physics A 642, 143c-148c

[16] Flory, P.J. (1941) *Molecular Size Distribution in Three Dimensional Polymers. I. Gelation*, J.Am.Chem.Soc. 63, 3083

[17] Forster, O. (1984) *Analysis 3*, Vieweg, Braunschweig

[18] gcc (2001) *GNU Compiler Collection*, http://gcc.gnu.org, as on 6.12.2001

[19] Galam, S. and Mauger, A. (1998) *Topology invariance in percolation thresholds*, Eur.Phys.J. B 1, 255-258

[20] Gandolfo, Blanchard, Dell'Antonio and Sirugue-Collin (2002) *Connectivity Properties of Continuum Percolation Processes on* $\mathbb{R}^2$, Journ of Stat. Phys. 106, 1-22

[21] Gandolfo, D. (2002) personal communications

[22] Grimmet, G. (1989) *Percolation*, Springer Verlag

[23] Hammersley, J.M. (1957) *Percolation processes. Lower bounds for the critical probability*, Ann.Math.Statist. 28, 790-795

[24] Hoshen, J. and Kopelman, R (1978) *Percolation and cluster distribution. I. Cluster multiple labeling technique and critical concentration algorithm*, Phys. Rev. B 14, 3438

[25] Isichenko, M.B. (1990) *Percolation, statistical topography, and transport in random media*, Reviews of Modern Physics 64, 4, 961-1042

[26] Kesten, H. (1982) *Percolation theory for mathematicians*, Boston MA, Birkhauser

[27] Kunaver, U. and Kolar, D. (1998) *Three-dimensional computer simulations of anisotropic grain growth in ceramics*, Acta mater., Vol 46, No. 13, pp. 4629-4640

[28] Lane, David M.(1993-2001) *HyperStat Online Textbook*, http://davidmlane.com/hyperstat as on 26.9.01

[29] Lorenz, Chr.D. and Ziff, R.M. (2001) *Precise determination of the critical percolation threshold for the three-dimensional "Swiss cheese" model using a growth algorithm*, Journal of chemical physics, vol 114, no. 8

[30] Lowry, Richard (1999-2000) *VassarStats*, http://faculty.vassar.edu/lowry/dist.html as on 26.9.01

[31] Mecke, Buchert and Wagner (1994) *Robust morphological measures for large-scale structure in the Universe*, Astron.Astrophys. 288, 697-704

[32] Meester, Ronald and Roy, Rahul (1996) *Continuum Percolation*, Cambridge University Press

[33] Montoya, J.M. and Solé, R.V. (2000) *Small World Patterns in Food Webs*, cond-mat/0011195

[34] Newbold, M. (1997) *Tesseract, a Java-applet of a 4-dimensional hypercube*, http://www.illusionworks.com/html/tesseract.html as on 2.10.2001

[35] Newman, Strogatz and Watts (2001) *Random graphs with arbitrary degree distributions and their applications*, cond-mat/0007235

[36] Nguyen, V.L. and Canessa, E.(1999) *Finite-Size Scaling in Two-dimensional Continuum Percolation Models*, cond-mat/9909200

[37] Pastor-Satorras, R. and Vespignani, A. (2001) *Epidemic Spreading in Scale-Free Networks*, Physical Review Letters, Vol 86, no. 14

[38] Perram, J.W.(1985) *Statistical Mechanics of Hard Ellipsoids I. Overlap Algorithm and the Contact Function*, Journ. of. Comp. Phys. 58, pp. 409-416

[39] Provatas, N. et.al. (1996) *Growth, Percolation, and Correlations in Disordered Fiber Networks*, cond-mat/9611046

[40] Provatas, Haataja, Seppälä, Majaniemi, Åström, Alava, AlaNissila (1996) *Growth, Percolation, and Correlations in Disordered Fiber Networks*, cond-mat/9611046

[41] *Trolltech Qt - The crossplattform c++ GUI Framework*, http://www.trolltech.com/products/qt/ as on 31.08.2001

[42] Quintanilla, J. and Torquato, S.(1997) *Clustering in a continuum percolation model*, Adv. Appl. Probab. 29, 327–336, http://citeseer.nj.nec.com/quintanilla96clustering.html as on 8.4.2003

[43] Quintanilla, J. and Torquato, S. and Ziff, R.M. (2000) *Efficient measurement of the percolation threshold for fully penetrable discs*, J.Phys. A 33, L399-L407

[44] Raabe, Dierk (1998) *Computational Materials Science*, Wiley VCH, Weinheim

[45] Rodrigues, A., Ugoccioni, R. and Dias de Deus, J.(1998) *Percolation approach to phase transitions in high energy nuclear collisions*, hep-ph/9812364

[46] SDL (2001) *Simple Direct MediaLayer, a cross-platform multimedia library*, http://www.libsdl.org/ as on 31.08.2001

[47] SGL (2001) *A 3D Scene Graph Library*, http://sgl.sourceforge.net/ as on 31.08.2001

[48] STLport Consulting (1997-2001) *STLport - a multiplatform ANSI C++ Standard Library implementation*, http://www.stlport.org/ as on 27.9.2001

[49] Satz, H. (1998) *Deconfinement and Percolation*, Nuclear Physics A 642, 130c-142c

[50] Sedgewick, R. (1984) *Algorithms*, Addison-Wesley

[51] Shamos, M.I. and Hoey, D. (1975) *Closest-point problems*, In Proc. 16th Annu. IEEE Sympos. Found. Comput. Sci., pages 151–162

[52] Solomon, Weisbuch, deArcangelis, Jan, Stauffer (2000) *Social percolation models*, Physica A 277, 239-247

[53] Stauffer, D. and Aharony, A. (1994) *Introduction to percolation theory, 2nd ed.*, Taylor and Francis Ltd., London

[54] Stauffer, D. and Jan, N. (2000) *Sharp peaks in the percolation model for stock markets*, Physica A 277, 215-219

[55] Stroustrup, B. (1997) *The C++ Programming language, Third Edition*, Addison-Wesley

[56] Suematsu, K. (1998) *Theory of gel point in real polymer solutions*, Eur.Phys.J. B 6, 93-100

[57] Sykes, M.F. and Essam, J.W. (1964) *Exact Critical Percolation Probabilities for Site and Bond Problems in Two Dimensions*, Journal of Mathematical Physics, Vol 5, no. 8

[58] Tupak, Andreas (2002) *Percolation in a family of constrained quadratic lattices*, Masterthesis, University of Bielefeld

[59] Vardi, Ilan (1998) *Prime Percolation*, Experimental Mathematics 7:3, p. 275

[60] caida.org (2001) *TOOLS : visualization : Walrus - Graph Visualization Tool*, http://www.caida.org/tools/visualization/walrus as on 17.07.2001

[61] Watts, D.J. and Strogatz, S.H. (1998) *Collective dynamics of 'small-world' networks*, Nature(London) 393, 440-42

[62] *The Official Condor Homepage* is http://www.cs.wisc.edu/condor

[63] Cygwin (2001) *Cygwin, a UNIX environment for Windows*, http://www.cygwin.com as on 6.12.2001

[64] Durrett, R. (1988) *Crabgrass, Measles, and Gypsy Moths: An Introduction to Interacting Particle Systems*, The Mathematical Intelligencer Vol.10 No. 2

[65] Meissner, Frank (2001) personal communications

[66] National Institute of Standards and Technology (2000) *Dictionary of Algorithms, Data Structures, and Problems* http://hissa.nist.gov/dads as on 15.08.2001

[67] root (2001) *An object-oriented Data Analysis Framework (Cern)*, http://root.cern.ch/ as on 31.08.2001

[68] SETI@home (2001) *The search for Extraterrestrial Intelligence*, http://setiathome.ssl.berkeley.edu/totals.html as on 31.08.2001

[69] Stoddard, S.D. (1978) *Identifying clusters in computer experiments on systems of particles*, J.Comp.Phys 27, 291-293

[70] Wagon, S. (1984) *Mathematica in Aktion*, Spektrum Akademischer Verlag, Heidelberg

Hiermit erkläre ich, die Arbeit eigenständig erstellt
und alle verwendeten Quellen angegeben zu haben.

Title page: A visualization of the two-dimensional case

- There are 28665 discs of a total fillingfactor (sum of areas) 0.894362
- with a fillingfactor growing in 9 steps between 25% and 145% of the critical fillingfactor 1.128 in y-direction
- with the number of discs growing in 9 steps by a factor of 77 in x-direction (while equally compensating the fillingfactor by decreasing radius)
- The used "colour lens" assigns the colours from green (small cluster mass) to red (big cluster mass) in proportion to the partial sum in the histogram of cluster masses
- The spanning cluster contains 10251 discs, its total sum of areas is 0.486329