Teknillinen Korkeakoulu
Tietoliikenne- ja tietoverkkotekniikan laitos
S-38.3138 Tietoverkkotekniikan erikoistyö

Percolation threshold in the Poisson Boolean continuum model

Tekijä: Riku Lääkkölä
69896S
riku.laakkola@tkk.fi
Ohjaaja: TkT Pasi Lassila
Valvoja: Prof. Jorma Virtamo
Jätetty: 29.8.2008

Abstract

This is a report of the work done as a summer trainee at the Department of Communications and Networking. The objective was to implement methods to study and visualize the Poisson Boolean continuum percolation of nodes with fixed radii of 1/2. Examination was restricted to homogeneous distributions of nodes.

Contents

List of Figures

List of functions

Chapter 1

Introduction

1.1 Percolation

In general, percolation theory studies idealized random media [2], for example porous materials. Percolation theory is interested in the existence of an "open path" between the edges of the porous object. In other words, if fluid is poured on top a porous rock, what is the critical level of porosity (the percolation threshold) that allows an open path to form and lets the fluid pass through the rock.

The porosity of the material can be represented as a lattice, where either each vertex or each edge connecting the vertices is "open" at a certain probability and "closed" otherwise. These representations are called site percolation and bond percolation, respectively. Another representation is continuum percolation, where the pores are represented as disks or spheres that are placed in random points (according to Poisson process) at a certain density. Here the fluid can pass through overlapping disks, i.e., the disks' centers are less than the sum of their radii apart.

The percolation threshold in a model like this has earlier been measured by simulation accurately to six significant digits in [3] where, however, the model is somewhat simpler than in an actual wireless network. In the context of a wireless ad-hoc network of randomly placed nodes with fixed transmission radii, the intuitive approach would be a Poisson Boolean continuum model, where connectivity between two nodes exists if the disks overlap each other's centers. The percolation threshold refers to the critical value of node density beyond which an infinite connected cluster is formed at a probability of 1. The effects of interference (noise, etc.) on connectivity have been studied in [1].

In [3] the authors introduced two methods for calculating the threshold using a continuum model consisting of an inhomogeneous distribution of nodes. Both of the methods try to locate the average x-coordinate of the edge of the percolating cluster. The value calculated in the reference is represented, for disks with radius R at a density λ as

$$\lambda R^2 = 0.359072 \pm 0.0000004. \tag{1.1}$$

In this model the only thing affecting the connection between two nodes is the distance.

In [3] the algorithms applied dynamic generation of nodes, which increases computational efficiency, as the least possible amount of useless nodes is generated by dividing the area into subsquares. In the less efficient of the algorithms, the *Gap-traversal method*, the average location of the frontier is sought from a square area whose top and bottom edges are connected (forming a cylinder). The more efficient *Frontier-walk method* starts its walk from the top and can be continued arbitrarily long, because all the new nodes are dynamically generated.

1.2 Objective

In this report the model is similar to the one used in [3], with the exception that the disk radius R is fixed to $1/2$. Thus, the reference value of the threshold corresponds to

$$\lambda = 1.436288 \pm 0.0000016. \tag{1.2}$$

The purpose of the work presented is not to obtain a more accurate value for the threshold, but to create tools for studying and visualizing the phenomenon, using Wolfram Mathematica. All the necessary algorithms are portrayed in a general form with figures for clarification. The corresponding Mathematica input is listed in detail in Appendix A.

1.3 Outline

The structure of this document is the following. The concept of percolation is defined in brief, earlier work on the subject is cited and the objective of the work is described in Chapter 1. The simulated model is described more in detail and all the algorithms used in calculations are depicted in a general form in Chapter 2. The process of visualizing the simulations and results is described along with the gathering of data for the results in Chapter 3. The gathered results are analyzed in Chapter 4 and the actual Mathematica input used in implementing the algorithms is listed in Appendix A.

Chapter 2

Model and algorithms

2.1 Model description and assumptions

The methods demonstrated here examine Poisson Boolean continuum percolation, which can be characterized as follows. Consider a graph of nodes, which are given a certain radius or range and thus form disks, on a plane distributed according to a Poisson point process with a certain intensity λ. Connectivity between two nodes is determined by the radii R of the disks and the distance between them. Percolation refers to the study of a phase transition phenomenon in the connectivity properties of the graph: as the parameter average number of neighbors per node is increased, at a certain critical point the graph turns from a largely disconneted one to a mostly connected one due to the emergence of a so called connected giant component.

In our model, connectivity exists if the two disks overlap (i.e., the distance between the nodes is less than both of their radii). This leads to the following condition for connectivity between any two nodes at locations $\mathbf{x}$ and $\mathbf{y}$:

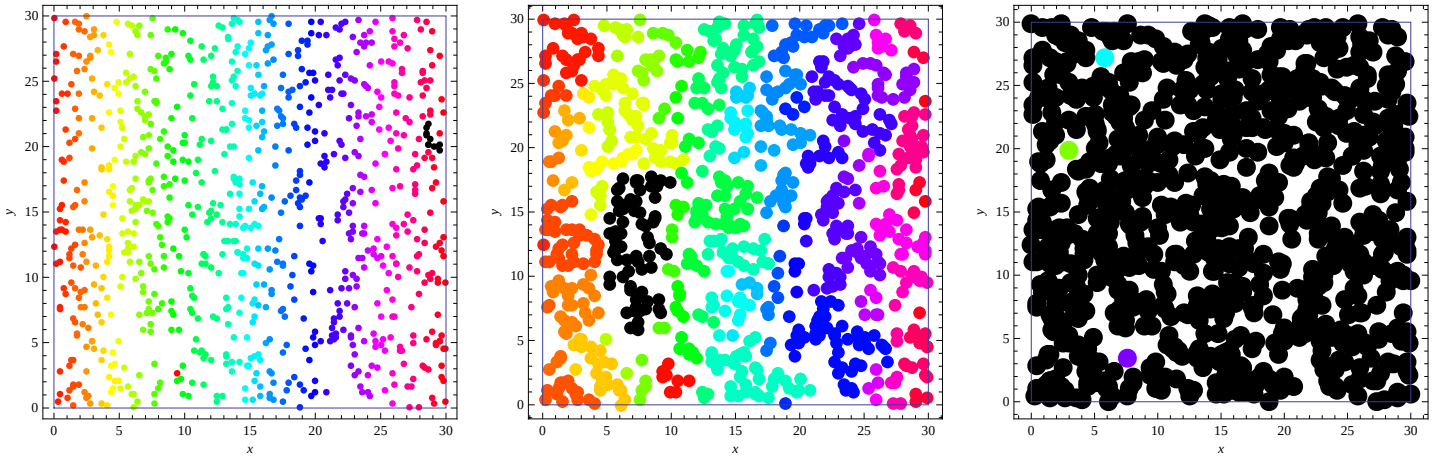
$$|\mathbf{x} - \mathbf{y}| < 2R.$$

The model used here differs from the model of wireless networks, where the distance may only be R or less.We assume that the radius for each disk is fixed so that $R = 1/2$, and the above connectivity requirement is simply

$$|\mathbf{x} - \mathbf{y}| < 1.$$

Additionally, in order to minimize the effects caused by the edges, the square plane is "bent" into a torus by considering the opposing edges of the square to be connected to each other.

The largest connected cluster of nodes (in terms of amount of nodes, not coverage), or the "giant component", and its features are the major interests, and the results gathered here are based on the behavior of the relation between the number of nodes in the giant component and the number of all nodes, or the "mass-ratio", in simulated graphs (the effect



(a) (b) (c)

Figure 2.1: A set of nodes in the same coordinates with three different radii and the giant component in black. $R =$ (a): 1/4, (b): 1/2, (c): 3/4.

that increasing the R of the disks has on the percolation can be seen in Figure 2.1). The mass-ratio is examined both as a function of area size with a constant λ and vice versa.

2.2 Algorithms

2.2.1 Generating and sorting nodes

Generating a list of homogeneously distributed nodes in a square with a side length of a and λ nodes per unit square is a simple algorithm (Function A.1): A pair of random real numbers is generated in the range $[0, a]$ and appended to a list. This is done λa^2 times. However, this will create a list in which the nodes are in no particular order.

To enable creating efficient algorithms for, e.g., discovering the "neighbours" of a node, the area has to be divided into subsquares (of side length 1, as $R = 1/2$) into which the coordinates of the node are placed (Function A.2). This is done by creating a two-dimensional list so that the index (x, y) in the list represents the subsquare bounded by the coordinates $(x - 1, y - 1)$ and (x, y), and then going through the unsorted list of coordinates and appending the nodes into their correct subsquares (the correct index is found by rounding up the coordinates of the node). This creates a structure where, for example, the index (i, j, n) would mean the nth node in subsquare (i, j).

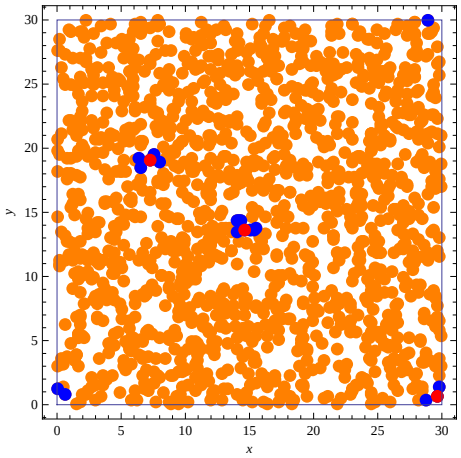


Figure 2.2: The neighbours (blue) of a node (red).

2.2.2 Connectivity

Neighbours

For discovering all the nodes that are connected to a particular node, its direct "neighbours" (Figure 2.2, the nodes that lay at a distance ≤ 1 of the node) have to be discovered first. For the neighbours of a node in subsquare (x, y) — in the parameters of Function A.3 the index $\{i, j, n\}$ represents the nth node in the subsquare (i, j) and the function returns a list of similarly formatted indices of the neighbours — only the nodes located in the square bounded by $(x-2, y-2)$ and $(x+1, y+1)$ have to be considered possible. This naturally means the nine subsquares $(x, y), (x \pm 1, y), (x, y \pm 1), (x \pm 1, y \pm 1), (x \pm 1, y \mp 1)$.

All the nodes in these subsquares are examined and the nodes that satisfy the neighbour-condition are kept. The neighbours-algorithm considers the opposing edges of the square area to be connected (effectively creating a torus-like topology). This has been taken into account in lines 8 and 12 – 17 of Function A.3.

Connected cluster

All the nodes connected to a particular node (Figure 2.3) can now be discovered by finding the neighbours of the node, then finding their neighbours and so on, maintaining a list of all the unique nodes that are discovered. A list of new neighbours (nodes whose neighbours haven't been discovered yet) has to be maintained (by comparing each result of the neighbours-algorithm to the list of nodes already discovered). The algorithm stops, when the list of new neighbours is empty. Function A.4 is used in a manner similar to Function A.3.

Now it is simple to form a list of all the separate connected clusters (Figure 2.4). Function A.5 returns a list of the clusters, which are lists of coordinates, in the following manner: A

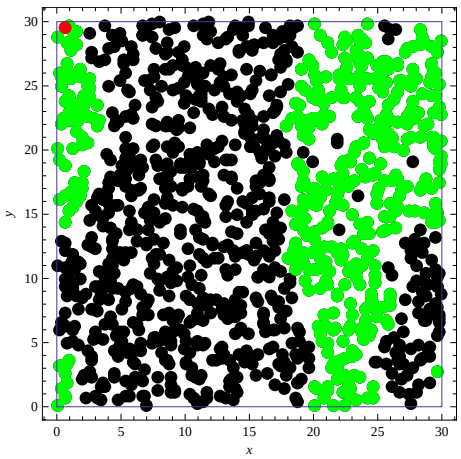


Figure 2.3: The nodes (green) connected to a node (red).

flat (not indexed by subsquares) list of all the nodes is created and Function A.4 is used to find the nodes connected to the first one (by first finding it from the sorted list). Now a list of indices is available so a list of corresponding coordinates can be established. This list is appended as an element in the list of clusters and its nodes are removed from the original flat list. Then this is repeated for the new first element of the flat list until the list is empty.

Function A.7 does the same thing, except that it returns the clusters as lists of indices. This is done by first creating a list of all the existing indices (with Function A.6 which simply loops through the sorted list of coordinates and appends all existing indices to a list), and then finding the indices of the nodes connected to the node first in that list, removing these nodes from the list of indices, doing the same thing to the new first element of the list of indices and repeating this until the list of indices is empty.

Function A.7 seems to be slightly faster than Function A.5. It is also in many cases somewhat more convenient to have a list of indices rather than a list of coordinates if other functions are to be applied to it.

2.2.3 Coastline

Angles

To discover the "coastline" of a cluster of nodes (the parts of the circular edge of each node that aren't covered by another node) we need to find out the points of the edge of the node where it intersects with its neighbours. For this we have Function A.8, which takes the sorted list of nodes and the index of a node as its parameters and returns a list of elements in the form $\{\{\alpha, \beta\}, \{i, j, n\}\}$ (Figure 2.5) where the angles represent points on the edge of the parameter node (the edge arc between the angles is covered by the overlapping node) and the index represents the overlapping node. The list is sorted by starting angle. The torus-topology is addressed in lines 16 – 23.

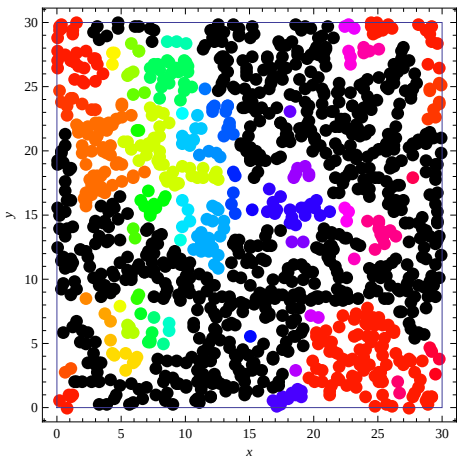


Figure 2.4: All the connected clusters in different colors, the largest in black.

The geometry for discovering α and β is fairly simple: Let ϕ be the angle between the line connecting the centres of the nodes and the line connecting the origin and the point of intersection, then

$$\begin{aligned} \cos\phi &= \frac{d/2}{1/2}, \\ \phi &= \arccos d, \end{aligned}$$

and γ is the directional angle of the vector connecting the centres of the nodes

$$\gamma = \arg(x_1 + iy_1).$$

We see that

$$\begin{aligned} \alpha &= \gamma - \phi, \\ \beta &= \gamma + \phi. \end{aligned}$$

Gaps

Now the parts (or "gaps") of the edge arc of the node which are not covered by another node and thus are parts of a coastline can be discovered. This is here done for a specially formatted list of the neighbours and their intersection angles of a particular node according to the following cyclic method (implemented in Function A.9, line numbers refer to this function), the indices of the nodes are also stored in various ways for additional calculations (mainly for discovering a single coastline instead of all the arcs in no particular order):

1. Get all the disks that have a starting angle $\alpha > 0$ and an ending angle $\beta < 0$, see line 3.

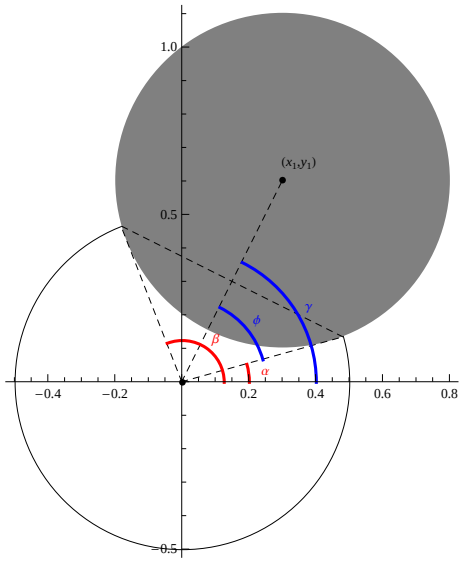


Figure 2.5: Finding the angles between which the disk in (x_1, y_1) overlaps the circle in the origin.

2. If such disks exist, set the starting angle of the cycle (ξ) to the greatest of their ending angles, and store the node whose ending angle this is as the last visited disk, otherwise, set ξ to $-\pi$ and the last visited disk to 0, see lines 6 – 10.

3. Cycle through the list of node-angle-elements (in ascending order by starting angle) in the following manner until the end of the list is reached or $\xi \geq \pi$ (legend: ind_a = index of the last visited node, ind_b = index of the node of the current element, α = starting intersection angle of the current element, β = ending intersection angle of the current element):

   (a) If $\alpha < \xi$, append an element to the list of gaps in the following format: $\{ind_i, \{\xi, \alpha\}, \{ind_a, ind_b\}\}$, where ind_i is the index of the initial node whose gaps are under examination, see line 15.

   (b) If $\alpha > 0 \bigwedge \beta < 0$, set ξ to π, see line 16.

   (c) Set ξ to $\max(\xi, \beta)$, see line 17.

   (d) Set last visited node to ind_b, see line 18.

4. If $\xi < \pi$ (which means that also the first gap in the gathered list must begin from $-\pi$), alter the first gap in the list in the following manner: set starting angle to ξ and index of node intersecting in the start of the gap to the index of the last visited node, see lines 21 – 23.

Now it is simple to gather all the arcs (in no particular order) of all the coastlines in a set of points via some mapping (Function A.10). Or alternatively one might be interested in the coastlines of a single group of nodes (a separate connected cluster, for example). Function A.11 has been used to generate the list of arcs used in plotting Figure 2.6.

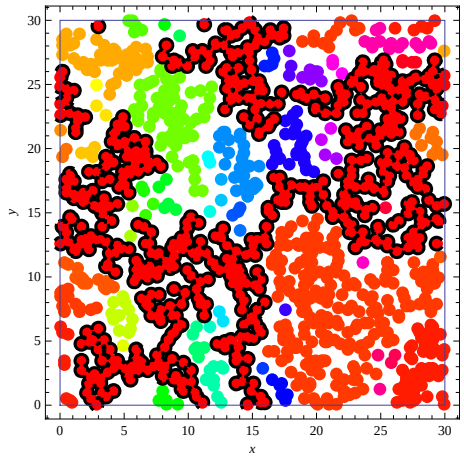
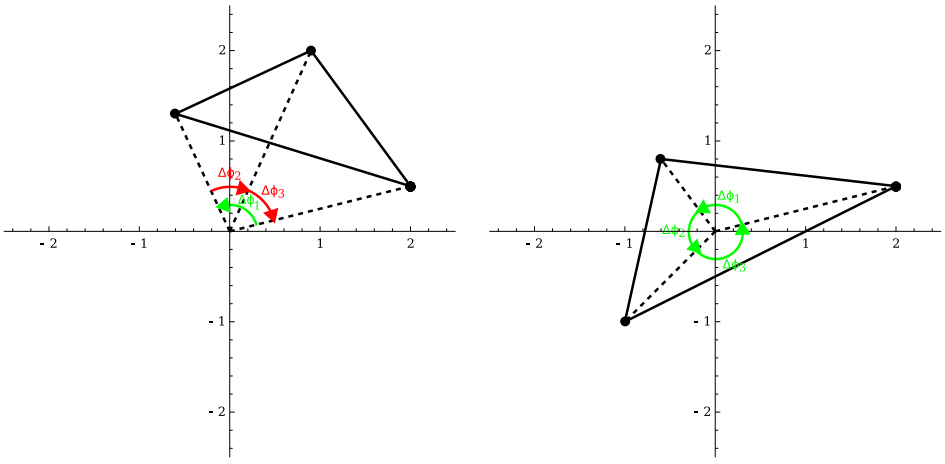


Figure 2.6: The coastline of the largest connected cluster.



(a) (b)

Figure 2.7: (a): The origin *is not* encircled by the line, so $\sum \Delta\phi = 0$, (b): The origin *is* encircled by the line, so $\sum \Delta\phi = n2\pi$ (here $n = 1$).

2.2.4 Miscellaneous

Encircled

A potentially interesting feature of a point on a plane is whether or not it is encircled by a line, which connects a given set of points (the first point in the set is connected to the last). This feature can easily be discovered by setting the point in question as the origin and then going through the given set of points forming the line in order, simultaneously calculating the sum of change in the directional angle between successive points (Function A.12). The sum should be equal to zero if the point is not encircled, and to a multiple of 2π if it is.

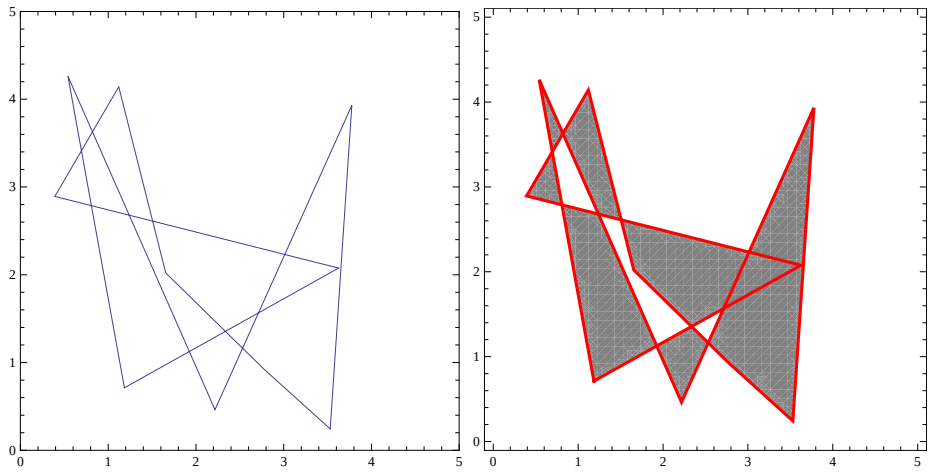


Figure 2.8: Points located in the darker areas of the plot are encircled by the line (i.e., Function A.13 returns $True$). Note that there are some white areas which seem to be encircled, but actually, if you imagine that the line is a solid string on the ground and put a stake in one of the white areas in question, you could remove the string by pulling it from one point without lifting it. Putting a stake in one of the darker areas would not allow this.

Interior points

Function A.14 calculates whether or not a node is an "interior point" — i.e. the circular edge of the node is completely overlapped by its neighbours, and thus is not a part of any coastline. The algorithm is to some extent similar to Function A.9, but somewhat simpler, as the result is simply a Boolean value. First, the algorithm checks simple initial conditions (and returns $False$ immediately, if any of these conditions IS satisfied):

- The node has no neighbours.
- The starting angles of all the neighbours are either negative or positive.

Then the largest negative ending angle whose pair is a positive starting angle is set as the starting angle (ξ) of the cycle ($-\pi$, if none exist), and the cycle is started. As soon as a gap is found (the starting intersection angle of the next neighbour is smaller than the current angle of the cycle), the cycle stops, and $False$ is returned. If no gaps are found and the cycle gets to π, $True$ is returned.

Chapter 3

Plotting and results

Much effort was also put into creating illustrative plots of the sets of disks. Some of the plotting functions created are described here for the sake of clarity.

3.1 Plotting disks and coastlines

In roded to write shorter plotting functions, Functions A.15, A.16 and A.17 were created for wrapping all the coordinates or coordinates in specified indices into a graphics primitive which plots a disk of radius 1/2 when used with the Graphics-command. Now it is simple to create plotting functions that display certain nodes in different colors by calling the appropriate function for the required indices, and then using the created helper functions combined with appropriate color directives (e.g., Function A.19).

Plotting the coastline out of the index-angle lists formed by Function A.11 does not differ greatly from plotting disks, as it is only needed to wrap the coordinates and angles of the arc into a *Circle*-primitive, which can then be plotted in the same manner as *Disk*-primitives. This is implemented in Functions A.21 and A.22.

Creating an illustrative plot of, e.g., all the separate connected clusters and the coastline of the giant component (Figure 2.6) is now possible with a fairly short function (Function A.23) which takes a list of the indices of the nodes sorted by clusters (output of Function A.7) and the list that contains the coordinates in the corresponding indices as an argument, finds the largest of the clusters, plots all the clusters using Function A.20 (edited so that it plots the giant component in red) and then plots the coastline of the giant component in black.

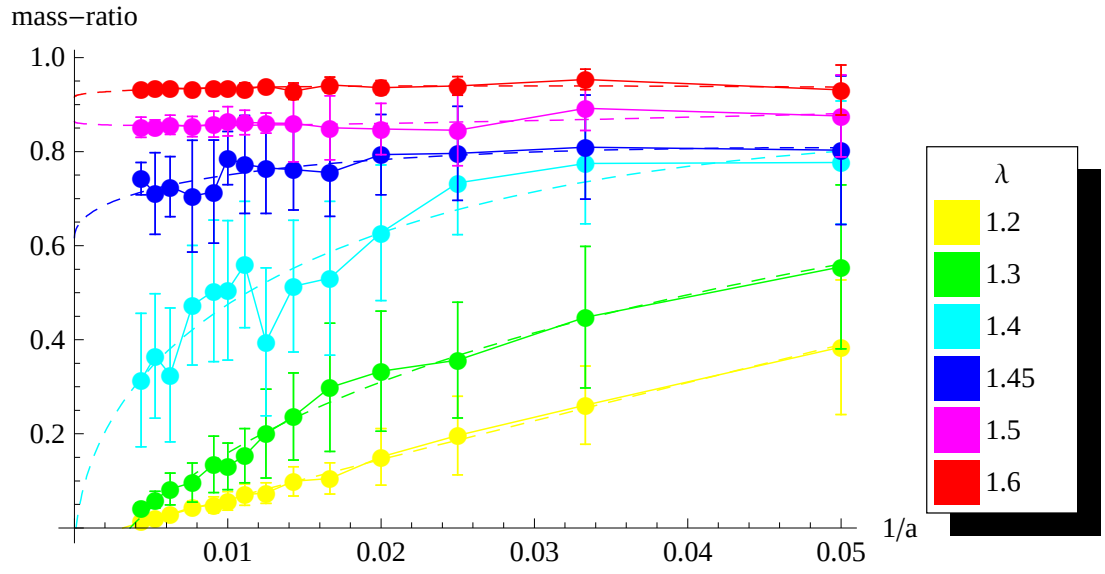


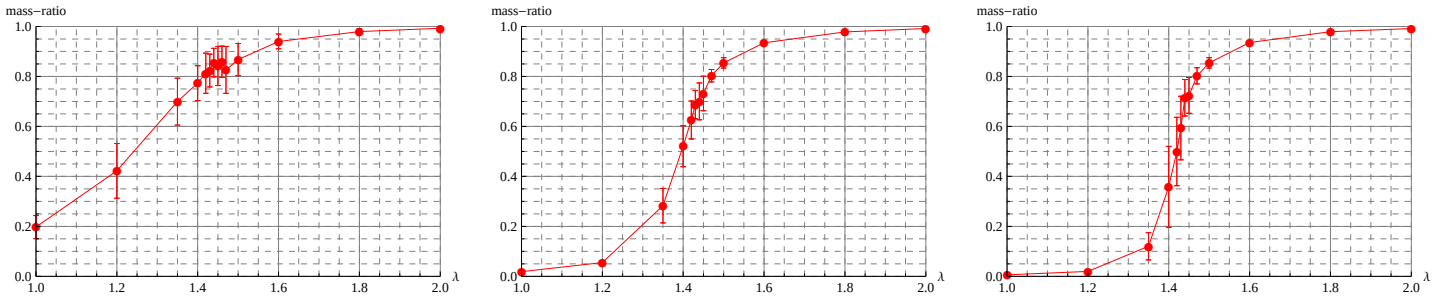
Figure 3.1: Mass-ratio as a function of $1/a$, where a is the length of the square's side. Each curve represents a different λ-value and the dashed curves are least square fitted functions $f_i(\lambda) = m + n\sqrt{\lambda}$.

3.2 Statistics and results

Originally, the objective was to gather statistics on the x-coordinate of the coastline of the percolating cluster in an inhomogeneous distribution of nodes (similarly to [3]), but as creating an algorithm that walks the outermost coastline proved to be a difficult task, the statistical analysis here is limited to comparing the ratio between the number of nodes in the giant component and the number of all the nodes in a homogeneous distribution (Function A.24).

Simulations were repeated several times for different values of λ and area sizes from 20×20 to 230×230 using Function A.25. The results of these calculations can be seen in Figure 3.1. The figure implies that even with the relatively small amounts of nodes ($\leq 10^5$) generated, the ratio converges towards zero with λ values below the reference percolation threshold (Equation 1.2), and towards a positive value with values above the threshold. To be able to see the convergence at more accurate values near the reference threshold, more repetitions with larger areas would be required, as results show considerable variance already with $\lambda = 1.4$ and 20 repetitions.

Similar simulations were also run with a constant area size and varying λ. The result was expected to be a curve that has near-zero values below the percolation threshold and a very steep upslope near the threshold. With a small area (20×20) however, the upslope is very gentle, but by increasing the size (to 100×100 and 200×200), the phenomenon becomes slightly more visible (Figure 3.2).



(a) (b) (c)

Figure 3.2: Mass-ratio as a function of λ in an area of size (a): 20×20, (b): 100×100, (c): 200×200.

In figures 3.1 and 3.2 the error bars represent the 95% confidence interval.

Chapter 4

Conclusions

The objective of the work was to create and implement algorithms for studying and visualizing the Poisson Boolean continuum percolation. The main interest was in the mass-ratio of the largest connected cluster.

As can be seen in figures 3.1 and 3.2, the behaviour of the mass-ratio of the giant component is as predicted: as the size of the plane grows arbitrarily large, below the percolation threshold the mass-ratio converges towards zero and above the threshold it maintains a constant value. However, the accuracy of the results obtained by the performed simulations is not great: when examining the mass-ratio as a function of area size, significant variance was present already at density $\lambda = 1.4$ (Figure 3.1), and in Figure 3.2 the upslope is still distributed to a relatively wide range of λ-values, even in the largest simulated area. The confidence intervals in both cases are also rather wide.

The accuracy of the results could be improved by enlarging the range of sizes in the simulated graphs, and additionally increasing the amount of performed repeated simulations to decrease variance and narrow the confidence intervals. But as the previous simulations already took a relatively long time, it would most likely be more reasonable to focus on creating more efficient algorithms and studying different approaches, such as locating the coastline in an inhomogeneous distribution of nodes.

Bibliography

[1] O. Dousse, F. Baccelli, and P. Thiran. Impact of interferences on connectivity in ad hoc networks. *Networking, IEEE/ACM Transactions on*, 13(2):425–436, April 2005.

[2] Geoffrey R. Grimmett. *Percolation*, volume 321 of *Grundlehren der mathematischen Wissenschaften*. Springer, 2nd edition, 1999.

[3] S. Torquato J. Quintanilla and R. M. Ziff. Efficient measurement of the percolation treshold for fully penetrable discs. *J. Phys. A: Math. Gen.*, 33, 2000.

Appendix A

Code Listings

A.1 Algorithm implementations

```
1 pUniform[a_, lambda_] :=
2  Table[a*{RandomReal[], RandomReal[]}, {lambda a^2}]
```

Function A.1: Creates a list of random points

```
1 sortPoints[p_, a_, b_] :=
2   Module[{sorted = Table[Table[{}, {a/b}], {a/b}], i = 1},
3    While[i <= Length[p],
4     AppendTo[sorted[[Ceiling[p[[i, 1]]], Ceiling[p[[i, 2]]]]],
5      p[[i]]];
6     i++;];
7    sorted];
```

Function A.2: Returns the list of points sorted into $\frac{a}{b} \times \frac{a}{b}$ subsquares

```
sortedNeighbours[p_, {i_, j_, n_}] :=
 Module[{le = Length[p], possibleSq = {},
   x = p[[i, j, n, 1]], y = p[[i, j, n, 2]], origin, t},
  origin = {x, y};
  possibleSq = {{i, j}, {i + 1, j}, {i + 1, j + 1}, {i,
     j + 1}, {i - 1, j}, {i, j - 1}, {i - 1, j - 1}, {i - 1,
     j + 1}, {i + 1, j - 1}};
  possibleSq = ((possibleSq /. (le + 1) -> 1) /. 0 -> le);
  t = Table[
    Position[p[[possibleSq[[m, 1]], possibleSq[[m, 2]]]],
      point_ /; (Norm[point - origin] < 1 ||
         Norm[point + {le, 0} - origin] < 1 ||
         Norm[point + {0, le} - origin] < 1 ||
         Norm[point + {le, le} - origin] < 1 ||
         Norm[point - {le, 0} - origin] < 1 ||
         Norm[point - {0, le} - origin] < 1 ||
         Norm[point - {le, le} - origin] < 1), 1]
               // Flatten, {m, 1, 9}];

  Flatten[
   Table[Table[{possibleSq[[r]], t[[r, s]]} // Flatten,
                {s, 1, Length[t[[r]]]}], {r, 1, 9}], 1]]
```

Function A.3: Returns the neighbours of the given point

```
sortedConnected[p_, {i_, j_, n_}] := Module[{all, newneighbours},
   all = newneighbours = sortedNeighbours[p, {i, j, n}];
   While[Length[newneighbours] > 0,
    all = Union[newneighbours, all];
    newneighbours =
     Complement[
      Union[Flatten[sortedNeighbours[p, #] & /@ newneighbours, 1]],
      all];];
   all];
```

Function A.4: Returns a list of nodes in p the given point is connected to

```
sortedIslands[p_] :=
  Module[{islandlist = {}, pp = Flatten[p, 2], island, pos},
   While[pp =!= {},
    island =
     p[[# /. List -> Sequence]] & /@
      sortedConnected[p, Position[p, pp[[1]]] // Flatten];
    AppendTo[islandlist, island];
    pp = Complement[pp, island];
    ];
   islandlist];
```

Function A.5: Returns a list of all the connected clusters (the coordinates of their nodes)

```
sortedIndices[p_] :=
  Module[{ind = {}, le = p // Length, i, j},
   For[i = 1, i <= le, i++,
    For[j = 1, j <= le, j++,
      AppendTo[ind, {i, j, #}] & /@ Range[p[[i, j]] // Length];];];
   ind];
```

Function A.6: Returns a list of all the existing indices in the sorted list

```
sortedIslandIndices[p_] :=
 Module[{pp = p, ind = sortedIndices[p], tmp, isls = {}},
  While[ind != {},
   AppendTo[isls, (tmp = sortedConnected[p, ind[[1]]])];
   ind = Complement[ind, tmp];];
  isls]
```

Function A.7: Returns a list of all the connected clusters (the indices of their nodes)

```
sortedAngles[p_, {i_, j_, n_}] := Module[
{le = p // Length, pp = p, phi, g, a, b,
 l = DeleteCases[sortedNeighbours[p, {i, j, n}], {i, j, n} ],
 i2, j2, n2, k, px, py},

   phi[x_] := ArcCos[Norm[x]];
   g[x_] := Arg[x[[1]] + i x[[2]]];
   a][x_] :=
    If[g[x] - phi[x] < -Pi,
                g[x] - phi[x] + 2 Pi,
                g[x] - phi[x]];
   b[x_] :=
    If[g[x] + phi[x] > Pi,
                g[x] + phi[x] - 2 Pi,
                g[x] + phi[x]] ;
   For[k = 1, k <= (l // Length), k++,
    i2 = l[[k, 1]]; j2 = l[[k, 2]]; n2 = l[[k, 3]];
    px = p[[i2, j2, n2, 1]]; py = p[[i2, j2, n2, 2]];
    If[Abs[i - i2] > 1,
                pp[[i2, j2, n2, 1]] = px + Sign[i - i2]*le];
    If[Abs[j - j2] > 1,
                pp[[i2, j2, n2, 2]] = py + Sign[j - j2]*le];
    ];
   Sort[{a[
        pp[[#[[1]], #[[2]], #[[3]]]] - pp[[i, j, n]]], b][
        pp[[#[[1]], #[[2]], #[[3]]]] - pp[[i, j, n]]], #} & /@ l]
   ];
```

Function A.8: Returns the intersection points of the parameter point and its neighbours

```
sortedFindGaps[in_] := Module[
{gaps = {}, i = 1, a, xi, le = Length[in[[2]]], upper, lastDisk},
upper = Position[in[[2]], {x_, y_, z_} /; x > 0 && y < 0]
                          // Flatten;

If[(upper // Length) > 0,
  last = (Sort[b[[upper]], #1[[2]] > #2[[2]] &] // First);
  xi = last[[2]]; lastDisk = last[[3]],
  xi = -Pi; lastDisk = 0
];

While[i <= le && x < Pi],
    a = in[[2, i++]];
    If[a[[1]] > xi,
     AppendTo[gaps, {in[[1]], {xi, a[[1]]}, {lastDisk, a[[3]]}}]];
    If[a[[1]] > 0 && a[[2]] < 0, xi = Pi];
    xi = Max[xi, a[[2]]];
    lastDisk = a[[3]];
];

If[xi < Pi,
    gaps[[1]] = {in[[1]], {xi, gaps[[1, 2, 2]] + 2 Pi},
                          {lastDisk, gaps[[1, 3, 2]]}}];

gaps];
```

Function A.9: Finds the parts of the circular edge of the node which aren't covered by other nodes

```
allCoastLineArcs[p_] :=
  Flatten[sortedFindGaps[#] & /@ ({#, sortedAngles[p, #]} & /@
      sortedIndices[p]), 1];
```

Function A.10: Finds all the arcs of all coastlines of a sorted set of disks

```
indListCoastLineArcs[indlist_, p_] :=
  Flatten[sortedFindGaps[#] & /@ ({#, sortedAngles[p, #]} & /@
      indlist), 1];
```

Function A.11: Finds all the arcs of all the coastlines of a group of disks in p indicated by the list of indices *indlist*.

```
angleAround[L_, p_] := Module[{nL = (# - p) & /@ L, le},
    le = Length[nL];
   Round[
    Plus @@ Append[
      Table[Arg[nL[[j + 1, 1]] + I nL[[j + 1, 2]]/
                (nL[[j , 1]] + I nL[[j , 2]])],
                {j, 1, le - 1}],
      Arg[nL[[1, 1]] + I nL[[1, 2]]/
                (nL[[le , 1]] + I nL[[le , 2]])]]]
   ];
```

Function A.12: Calculates the sum (rounded to the nearest integer) of change in directional angle between successive points in the list L from the perspective of the given point p.

```
encircledQ[L_, p_] := angleAround[L, p] =!= 0;
```

Function A.13: Returns $True$, if the point p is encircled by the line L, $False$ otherwise.

```
sortedInteriorQ[p_, {i_, j_, n_}] :=
Module[{interior = True, a = sortedAngles[p, {i, j, n}], xi,
   k = 1, le, upper, b},

le = Length[a];
If[le == 0 || a[[le, 1]] < 0 || a[[1, 1]] > 0,
  interior = False,
  upper = Position[a, {x_, y_} /; x > 0 && y < 0] // Flatten;
  xi = Max[-Pi, a[[upper, 2]]];
];

While[interior && k <= le && xi < Pi,
  b = a[[k++]];
  If[b[[1]] > xi,
    interior = False,
    If[b[[1]] > 0 && b[[2]] < 0, xi = Pi;, xi = Max[xi, b[[2]]];];
  ];
];
interior];
```

Function A.14: Returns $True$ if the specified node in p is an interior point, $False$ otherwise.

A.2 Plotting

```
dsk[p_] := (Disk[#, 1/2] & /@ p);
```

Function A.15: Produces a list of Disk-primitives of the unsorted list of coordinates. The list can be plotted using Graphics.

```
sdsk[p_] := (Map[Disk[#, 1/2] &, p, {3}]);
```

Function A.16: Produces a list of Disk-primitives of the sorted list of coordinates. The list can be plotted using Graphics.

```
inddsk[p_, indlist_] :=
 Disk[#, 1/2] & /@ (p[[# /. List -> Sequence]] & /@ indlist)
```

Function A.17: Produces a list of Disk-primitives of coordinates in p which are in the indices listed in *indlist*.

```
sdsk[p_, nodes_] := Module[{a, b, c, nei},
    b = {Red, inddsk[p, nodes]};
    nei = Flatten[sortedNeighbours[p, #] & /@ nodes, 1];
    c = {Blue, inddsk[p, nei]};
    a = {Orange, Complement[sdsk[p], c]};
    Graphics[{a, c, b}]];
```

Function A.18: Plots all the disks in p, so that the nodes in the indices given in *nodes* are colored red, their neighbours blue, and other nodes orange.

```
sdskc[p_, {i_, j_, n_}] := Module[{a, b, c, conn},
    b = {Red, Disk[p[[i, j, n]], 1/2]};
    conn = sortedConnected[p, {i, j, n}];
    c = {Green, inddsk[p, conn]};
    a = Complement[sdsk[p], c];
    Graphics[{a, c, b}]];
```

Function A.19: Plots all the disks in p, so that the node in the index (i, j, n) is colored red, the nodes connected to it green, and other nodes black.

```
islandplot2[islands_] :=
  Module[{le = Length[islands], i = 1, all, gco = gc[islands],
    gcgraph},
   all =
    Graphics[{Hue[i++/le], dsk[#]} & /@
      Delete[islands, Position[islands, gco]]];
   gcgraph = Graphics[{Black, #} & /@ dsk[gco]];
   Show[all, gcgraph]];
```

Function A.20: Plots a disk in each coordinate in the list of lists of coordinates *islands* so that the nodes in the same list are the same color.

```
sortedCoastLinePlot[p_] :=
  Graphics[{Thick, Black,
    Circle[p[[#[[1, 1]], #[[1, 2]], #[[1, 3]]]], 1/2, #[[2]]] & /@
     allCoastLineArcs[p]}];
```

Function A.21: Plots all the coastlines formed by the nodes in p.

```
sortedCoastLinePlot[indlist_, p_] :=
  Graphics[{Thick, Black,
    Circle[p[[#[[1, 1]], #[[1, 2]], #[[1, 3]]]], 1/2, #[[2]]] & /@
     indListCoastLineArcs[indlist, p]}];
```

Function A.22: Plots all the coastlines formed by the nodes in indices of p listed in *indlist*.

```
sortedGCCoastLinePlotInd[isls_, p_] :=
  Module[{le = Length /@ isls, GCindx},
   GCindx = Position[le, Max[le]] // Flatten // First;
   Show[islandplot2R[indToCoordList[#, p] & /@ isls],
    sortedCoastLinePlot[isls[[GCindx]], p]]];
```

Function A.23: Plots all the islands in different colours and also the coastline of the giant component in black

A.3 Statistics

```
1 gcRatio[isls_] := Max[Length /@ isls]/Length[Flatten[isls, 1]]
```

Function A.24: Calculates the value n_g/n_a where n_g is the number of nodes in the largest connected cluster and n_a is the total number of nodes.

```
1  repRatioGather[Lambda_, alist_, rep_, file_] :=
2  Module[{data = {}, a, tmp, i = 0},
3    For[i = 1, i <= (alist // Length), i++,
4     AppendTo[
5      data, {alist[[i]], Lambda,
6       Table[sortedIslandIndices[
7           sortPoints[pUniform[alist[[i]], Lambda], alist[[i]], 1]] //
8           gcRatio, {rep}]}];
9     Put[data, file];
10    ];
11   data]
```

Function A.25: Gathers mass-ratio data of networks with λ-value *Lambda* and area side lengths listed in *alist*, repeating each simulation *rep* times.