

EP408 Computational Physics Project - Percolation

Sean Cummins
14432458

Department of Experimental Physics,
Maynooth University, Co Kildare

April 15, 2019

Contents

1	Abstract	3
2	Introduction and Theory	4
3	Algorithms	5
3.1	A first algorithm for cluster labelling	5
3.2	Hoshen - Kopelman Algorithm	6
4	Results	9
4.1	Determining spanning clusters using Hoshen - Kopelman Algorithm	9
4.2	Estimating p_c using Hoshen - Kopelman Algorithm	11
5	Conclusion	14
6	References	14
7	Code	15
7.1	A first algorithm for cluster labelling	15
7.2	Determining spanning clusters using the Hoshen - Kopelman Algorithm	18
7.3	Estimating p_c using the Hoshen - Kopelman Algorithm	22
7.4	Estimating β using the Hoshen - Kopelman Algorithm	26

1 Abstract

This short computational physics project is on Monte Carlo numerical simulations of introductory **site percolation** theory. In this project, percolation was simulated on $L \times L$ square lattices and cluster labelled using two algorithms.

The first was a straightforward although inefficient algorithm which served merely as an introduction. The second (which was used for the physics investigations) was the well known **Hoshen-Kopelman Algorithm** (a variant of **Union-Find**). In both cases the associated program was able to identify if there existed a spanning cluster.

Using the Hoshen-Kopelman Algorithm, the **critical probability** p_c was investigated and estimated for 25×25 and 100×100 square lattices. The estimates matched well against the infinite lattice analytical value of $p_c = 0.593$.

Finally, the **critical exponent** β corresponding to the fractional size of the spanning cluster relative to all occupied sites was estimated for 25×25 lattices and compared to infinite lattice analytical value of $\beta = 5/36 \approx 0.138$. The estimate for the critical exponent β was found to be bound in range (0.13,0.18).

2 Introduction and Theory

Percolation is a relatively new area of research having only been created in the 1960's. Since then, applications have been found in areas as diverse as:

- **Physics:** Micro-to-Macro links, Phase Transitions, Universality
- **Mathematics:** Fractals, Simple Models yet few analytical results found
- **Material Science:** Gels, polymers
- **Biology:** Epidemic, Forest fires
- **Computer Science:** Neutral networks

Percolation can be summed up as the answer to the following representative question:

"If liquid is poured though the top of a porous material, will it pass from hole to hole and reach the bottom?"

Common examples of this occurrence include the motion of groundwater through soil, the flow of oil through porous rock and water flowing coffee granules.

To model this we consider a simple situation - a lattice of squares.

Sites can be randomly occupied with a given probability p representing a hole and $(1 - p)$ representing an impermeable medium. Fluid can flow from one hole to another if that hole has a N, S, E or W neighbour. As we add more and more holes, **clusters** or collections of interconnected sites begin to form as seen in (b).

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

(a) Initialised 5×5 lattice

0	-1	-1	0	-1
-1	0	0	-1	0
-1	0	0	0	-1
0	-1	-1	0	-1
-1	0	-1	-1	0

(b) Label -1 if hole

0	-1	-1	0	-1
-1	0	0	-1	0
-1	0	0	0	-1
0	-1	-1	0	-1
-1	0	-1	-1	0

(c) Can fluid flow through this?

Adding more holes still and the clusters themselves begin to merge. If each cluster is given its own unique label, then we must relabel clusters that merge together.

We say the grid contains a **spanning cluster** if there exists an unbroken chain of hole sites from the top to the bottom of the medium. *If such a spanning cluster exists then we say the system percolates!!*

We can write a program to check for this condition for percolation. Give each cluster a unique label. Check if label is present in both the first and last row of matrix. If so than its a spanning cluster.

From a physics standpoint, this simple model is of particular interest as it exhibits a **phase transition** around a particular probability known as the *critical probability* or p_c .

Below the value of p_c (sub-critical phase) percolation is extremely unlikely. Above this value (super-critical phase) percolation is extremely likely. The transition is step-like at p_c (critical phase).

p_c is the fraction of occupied sites at the point when a spanning cluster appears. Estimating p_c requires us to know if there exists a spanning cluster and is therefore the motivation for the following.

3 Algorithms

3.1 A first algorithm for cluster labelling

The following algorithm describes a simple and straightforward implementation of cluster labelling and span cluster checking. It is by no means efficient and only works well in practice for lattices up to about 50×50 .

- Initialise all elements in an $L \times L$ grid to 0
- Initialise a random list of all sites in grid
- Randomly occupy site \rightarrow label site $N_1 = 1$
- Randomly occupy another site. Check if there is a neighbour (N,S,E,W). At this point there could only be one
 - If not \rightarrow label site $N_2 = N_1 + 1$
 - If so \rightarrow label site $N_2 = N_1$
- Randomly occupy another site. Check if there are neighbours.(Now there is the possibly of more than one)
 - If not \rightarrow label site $N_3 = N_2 + 1$
 - If so:
 - If there is only one cluster label among all neighbours then assign same number to new site
 - If there is more the one unique cluster label, then our new site is a **spanning site**
 - \rightarrow Assign new site the smallest of all the cluster labels involved.
 - \rightarrow Scan the whole grid merging all sites in the newly merged cluster with the same label.
- Is the spanning cluster condition satisfied?
 - Yes \rightarrow End
 - No \rightarrow Continue occupying and labelling sites as above.

p_c is the fraction of occupied sites among all sites at the point when a spanning cluster appears.

(By convention we always use the smallest of the clusters numbers for the new larger cluster)

As stated this algorithm is extremely CPU intensive due to the frequent relabelling of existing clusters which requires the entire grid to be rescanned. However the algorithm does demonstrate key elements used by more efficient algorithms like labelling and neighbour check.

If S is the number of lattice sites then the algorithm requires approximately S^2 CPU time. This chronic inefficiency is what led Hammersley and Handscomb in 1957 to state "*Direct simulation is out of the question*".

However a breakthrough occurred in 1976 when **Hoshen - Kopelmann** developed a algorithm whose CPU time requirement scaled $\approx S$ with S^2 .

3.2 Hoshen - Kopelman Algorithm

The Hoshen - Kopelman Algorithm is a simple, efficient algorithm for cluster labelling a pre-generated grid. It is a specific example of the more general Union-Find Algorithm.

The key idea is to keep a list of cluster labels (which we will call *csize*).

Indices on list denote labels of clusters. If a particular index on list contains a positive value, then it indicates that the label is the representative label for that cluster (Proper Label). The value itself indicates the cluster's size.

Negative values on list denote temporary labels which point to proper labels (i.e. $csize[4] = -2$ implies all grid elements labelled 4 are apart of cluster labelled 2.)

A temporary label could in turn point to another temporary label.

The algorithm uses recursion to quickly travel down tree of temporary labels to find proper label i.e. $7 \rightarrow 5 \rightarrow 1$ (all elements that are 7 and 5 are members of 1 and must be reassigned)

Now we only need to scan the grid at most twice. Once to label all proper and temporary labels and a second optional time to reassign temporary labels.

Furthermore we need only **raster scan** the pre-generated grid. Assuming we are scanning from left-to-right and row-to-row, we need only check the N and E neighbour for any site under consideration.

In program we define a boundary layer of zeros so program doesn't try to check a neighbour that is outside grid.

Example

We pre-generate a 5×5 grid under a certain probability of occupation for each site. Occupied sites are denoted as -1 . We now demonstrate the Hoshen - Kopelman for cluster labelling.

0	-1	-1	0	-1
-1	0	-1	-1	-1
-1	0	0	0	-1
0	-1	-1	0	-1
-1	0	-1	-1	-1

(a) $csize = [0]$

0	1	-1	0	-1
-1	0	-1	-1	-1
-1	0	0	0	-1
0	-1	-1	0	-1
-1	0	-1	-1	-1

(b) $csize = [0, 1]$

0	1	1	0	-1
-1	0	-1	-1	-1
-1	0	0	0	-1
0	-1	-1	0	-1
-1	0	-1	-1	-1

(c) $csize = [0, 2]$

0	1	1	0	2
-1	0	-1	-1	-1
-1	0	0	0	-1
0	-1	-1	0	-1
-1	0	-1	-1	-1

(d) $csize = [0, 2, 1]$

0	1	1	0	2
3	0	-1	-1	-1
-1	0	0	0	-1
0	-1	-1	0	-1
-1	0	-1	-1	-1

(e) $csize = [0, 2, 1, 1]$

0	1	1	0	2
3	0	1	-1	-1
-1	0	0	0	-1
0	-1	-1	0	-1
-1	0	-1	-1	-1

(f) $csize = [0, 3, 1, 1]$

0	1	1	0	2
3	0	1	1	-1
-1	0	0	0	-1
0	-1	-1	0	-1
-1	0	-1	-1	-1

(g) $csize = [0, 4, 1, 1]$

0	1	1	0	2
3	0	1	1	1
-1	0	0	0	-1
0	-1	-1	0	-1
-1	0	-1	-1	-1

(h) $csize = [0, 6, -1, 1]$

0	1	1	0	2
3	0	1	1	1
3	0	0	0	-1
0	-1	-1	0	-1
-1	0	-1	-1	-1

(i) $csize = [0, 6, -1, 2]$

0	1	1	0	2
3	0	1	1	1
3	0	0	0	1
0	-1	-1	0	-1
-1	0	-1	-1	-1

(j) $csize = [0, 7, -1, 2]$

0	1	1	0	2
3	0	1	1	1
3	0	0	0	1
0	4	-1	0	-1
-1	0	-1	-1	-1

(k) $csize = [0, 7, -1, 2, 1]$

0	1	1	0	2
3	0	1	1	1
3	0	0	0	1
0	4	4	0	-1
-1	0	-1	-1	-1

(l) $csize = [0, 7, -1, 2, 2]$

0	1	1	0	2
3	0	1	1	1
3	0	0	0	1
0	4	4	0	1
-1	0	-1	-1	-1

(a) $csize = [0, 8, -1, 2, 2]$

0	1	1	0	2
3	0	1	1	1
3	0	0	0	1
0	4	4	0	1
5	0	-1	-1	-1

(b) $csize = [0, 8, -1, 2, 2, 1]$

0	1	1	0	2
3	0	1	1	1
3	0	0	0	1
0	4	4	0	1
5	0	4	-1	-1

(c) $csize = [0, 8, -1, 2, 3, 1]$

0	1	1	0	2
3	0	1	1	1
3	0	0	0	1
0	4	4	0	1
5	0	4	4	-1

(d) $csize = [0, 8, -1, 2, 4, 1]$

0	1	1	0	2
3	0	1	1	1
3	0	0	0	1
0	4	4	0	1
5	0	4	4	1

(e) $csize = [0, 13, -1, 2, -1, 1]$

0	1	1	0	1
3	0	1	1	1
3	0	0	0	1
0	1	1	0	1
5	0	1	1	1

(f) $csize = [0, 13, -1, 2, -1, 1]$

This completes the Hoshen - Kopleman Algorithm.

We see $csize[1] = 13$, $csize[3] = 2$ and $csize[5] = 1$ corresponding to clusters 1,3,5 are proper labels whose $csize$ values denote the cluster sizes.

We also see $csize[2] = -1$, $csize[4] = -1$ are re-direction or temporary labels whose $csize$ values points to the cluster they are apart of.

In the final step, for all temporary labels, we travelled up the tree of pointers until we found a positive proper label.

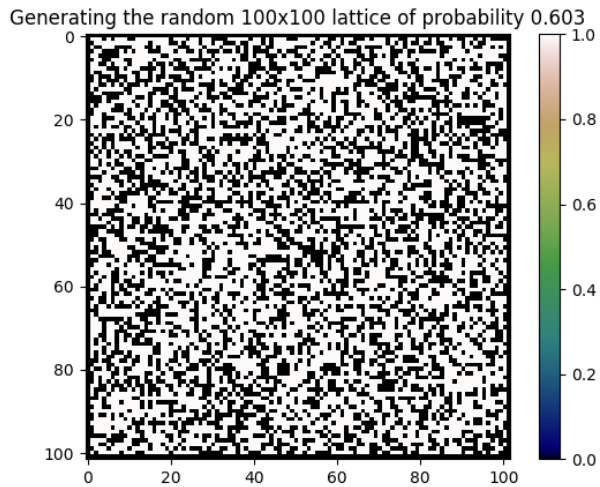
Like before we always use the smallest of the clusters numbers for the new larger cluster.

With the Hoshen - Kopelman Algorithm, 1000x1000 grids can be cluster labelled in less than 10s!!

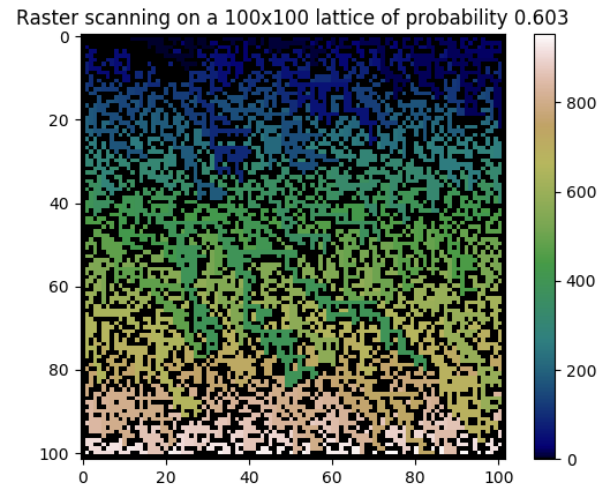
We now present the results from using the algorithm.

4 Results

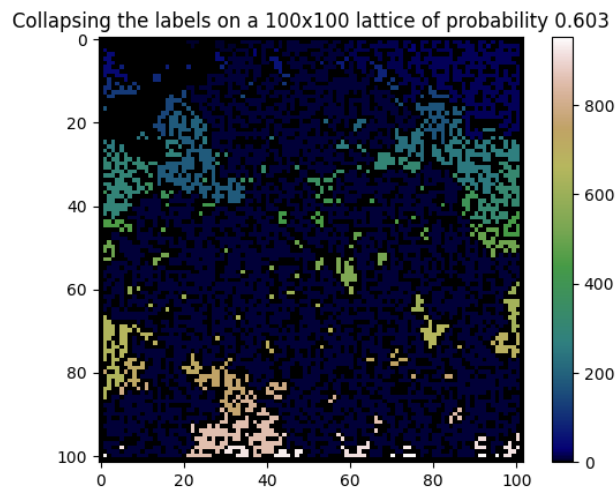
4.1 Determining spanning clusters using Hoshen - Kopelman Algorithm



(a) Random Grid Generator — 0.473999977112 seconds —



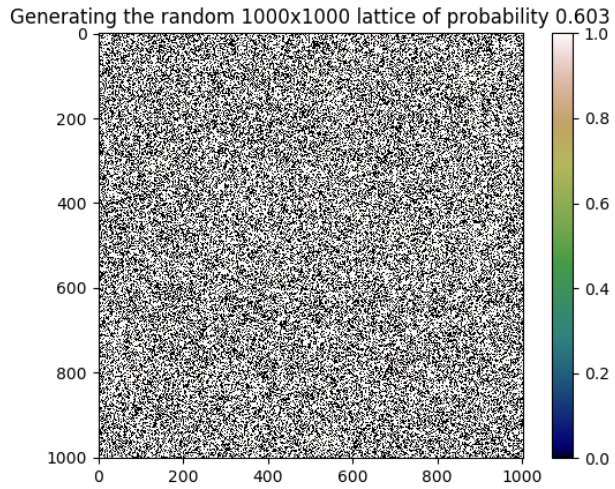
(b) Raster Scan — 0.461000204086 seconds —



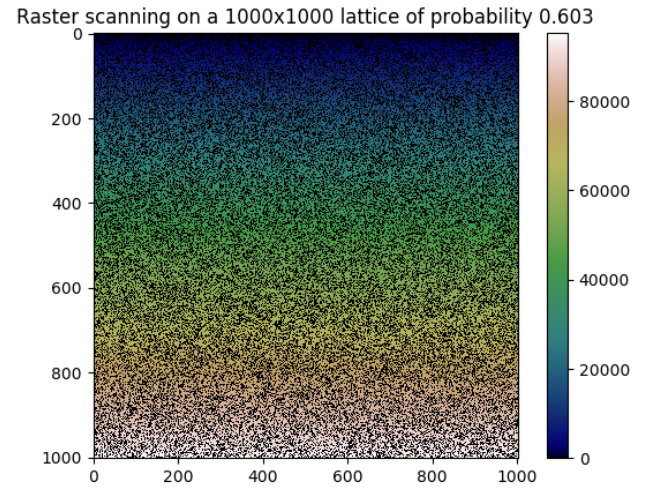
(c) Collapse — 0.451999902725 seconds —

Function in Hoshen - Kopelman program called Span cluster check correctly confirms a spanning cluster.

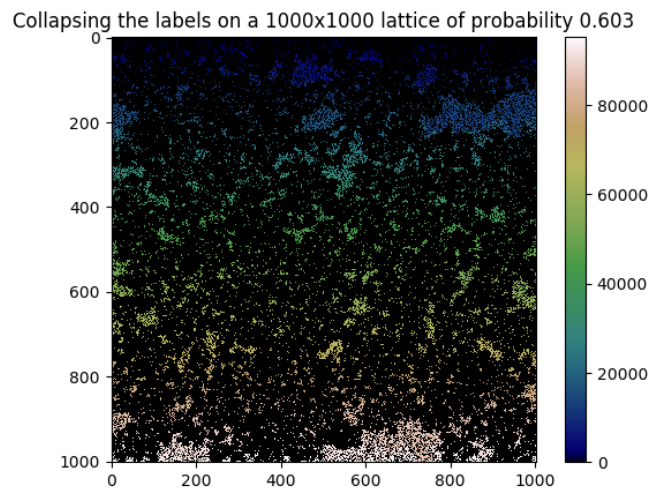
Determining spanning clusters using Hoshen - Kopelman Algorithm



(a) Random Grid Generator — 2.59899997711 seconds —



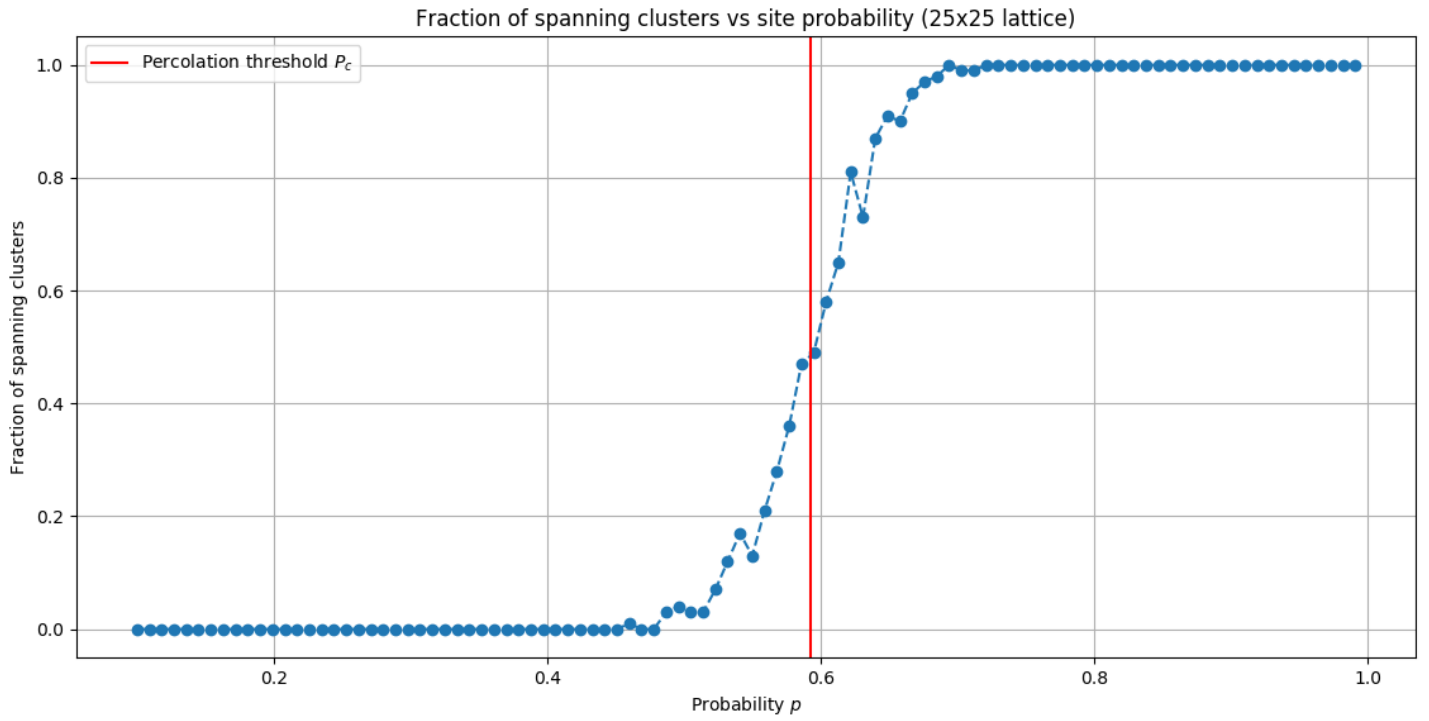
(b) Raster Scan — 4.44200015068 seconds —



(c) Collapse — 3.7539999485 seconds —

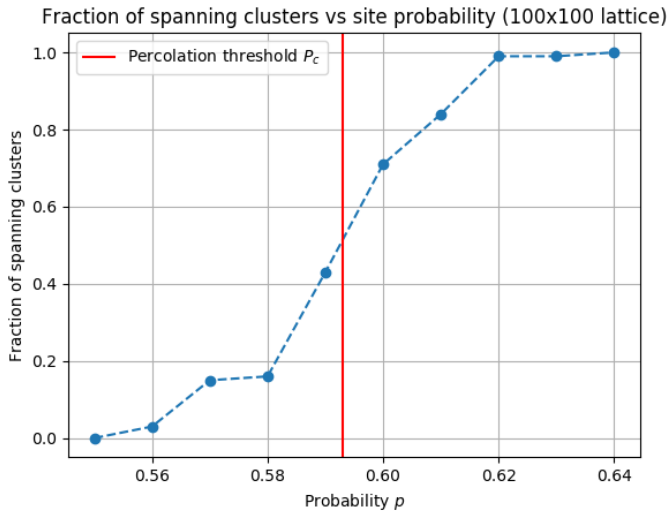
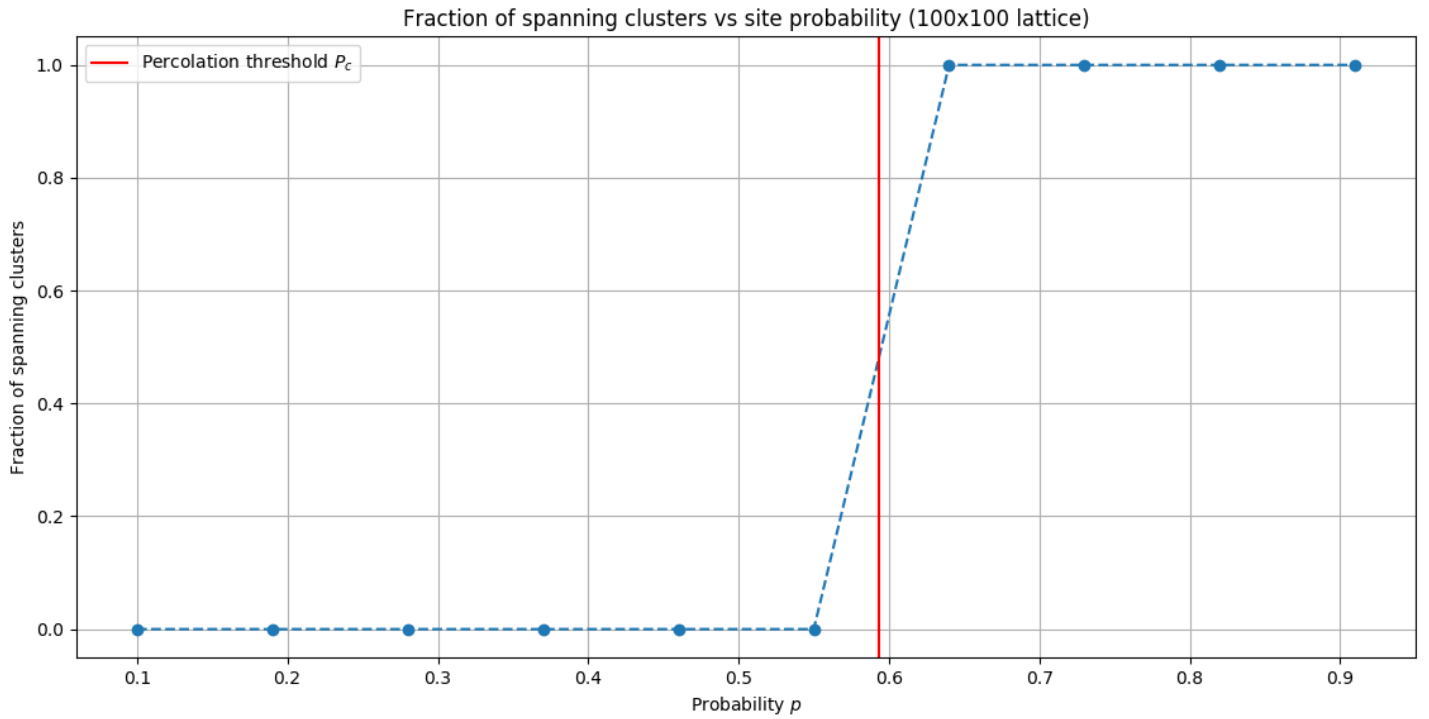
Span cluster check correctly confirms a spanning cluster.

4.2 Estimating p_c using Hoshen - Kopelman Algorithm

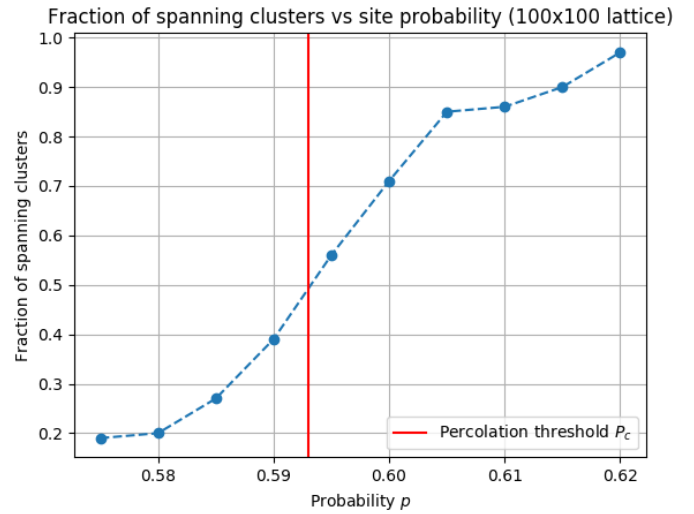


We see the step-like switch between $p < p_c$ and $p > p_c$ with the expected value for p_c in the middle.

Estimating p_c using Hoshen - Kopelman Algorithm



(a) Zoomed in image of above graph



(b) Even further zoomed in image of above graph

We see that for 100×100 grids, the critical probability for a square grid $p_c \approx 0.593$ corresponds almost exactly to 0.5 - the fraction of spanning clusters from 100 runs. This implies past p_c there is a greater than 50% chance that the system will percolate.

Estimating β using Hoshen - Kopelman Algorithm

The local structure determines the value of p_c . In the case of square cells, $p_c = 0.593$. For triangular cells, its $p_c = 0.5$. However what is interesting is when one considers $p > p_c$.

Past p_c (in the super-critical phase), the behaviour transcends local to become universally global depending on dimension only. That is to say, regardless of local structure, once past p_c all systems behave the same.

We call this concept **Universality** and can be seen as a link between microscopic and macroscopic phenomena.

In the super-critical phase, systems such as these can often be described with power-laws with **Critical Exponents**. These exponents encode important information about the system. We will attempt to estimate one; β - the order parameter.

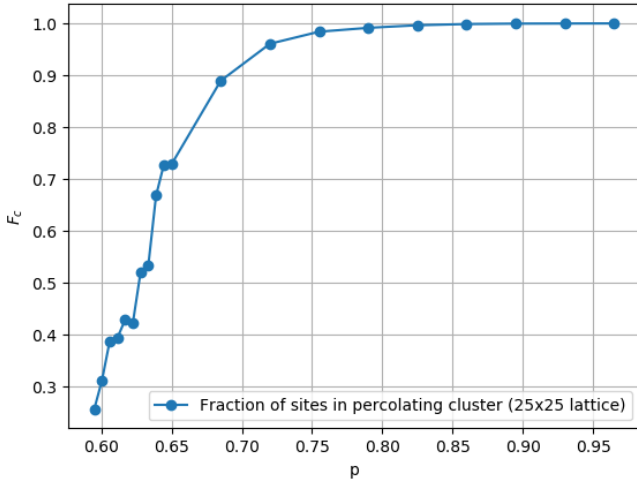
We consider the fraction of sites F that are in the spanning cluster as a function of p . This relationship is described the following power law.

$$F = F_0(p - p_c)^\beta$$

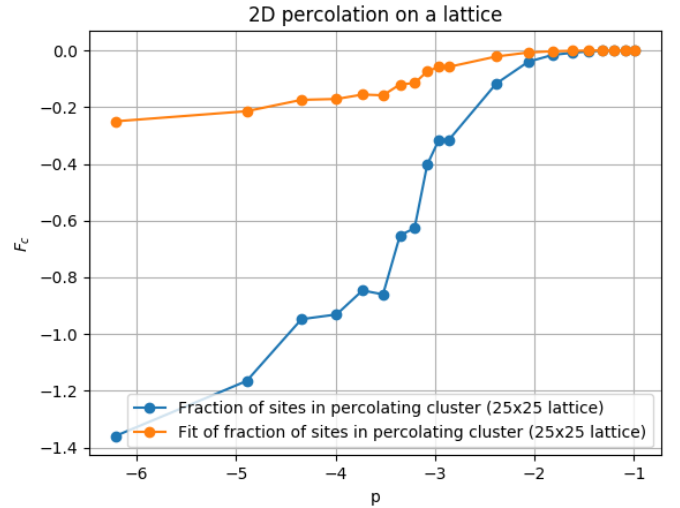
The equation naturally contains p_c and as a result the quality of our estimate for β is inextricably bound to the quality of our estimate for p_c .

For an infinite square grid $\beta = 5/36 \approx 0.138$.

When we plot F vs p and $\log(F)$ vs $\log(p)$, we obtain the following :



(a) Run for 20 probabilities. Singular at $p = p_c$



(b) β estimated to be (0.13,0.18) after running many iterations.

For each probability, 100 spanning grids were averaged. Log-log plot shows roughly straight line suggesting power relationship. β estimated to be (0.13,0.18)

5 Conclusion

In this computational physics project, the Hoshen - Algorithm was used to efficiently cluster label and span check square grids ranging in size from 25×25 to 100×100 . The programs written were correctly able to determine if a pre-generated grid of a given occupation probability possessed a spanning cluster.

With this estimates were made for the critical probability p_c for 25×25 and 100×100 grids by sweeping over a the full range of probabilities and estimating the point when the phase transition occurred. Despite being far smaller size than the analytical infinite grid, estimates from the graphical plots show the value of p_c for these sizes is still quite close.

Finally the estimate for the critical exponent β was found to be bound in range (0.13,0.18). The expected value is 0.138.

6 References

References

- [1] J. Hoshen and R. Kopelman *Percolation and cluster distribution. I. Cluster multiple labelling technique and critical concentration algorithm*

7 Code

7.1 A first algorithm for cluster labelling

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Fri Dec 01 16:47:50 2017
4
5 Percolation - site model
6
7 Very inefficient. Very slow for grid sizes > 10x10
8
9 @author: Sean Cummins
10 """
11
12 import numpy as np
13 np.set_printoptions(threshold=np.nan)
14 #for size < 10x10 , whole grid can be printed to screen
15 import random
16
17 #-----#
18
19 def Percolate(grid_length):
20
21     grid=np.zeros((grid_length+2,grid_length+2))
22     #+2 each size is boundary so all checks for each element are valid
23
24     """
25     RANDOM SITES
26     Generate as many unique random sites as can fit
27     """
28     possible_coordinates = [(x, y) for x in range(1,grid_length+1)\
29                             for y in range(1, grid_length+1)]
30     random_sites = random.sample(possible_coordinates, pow(grid_length,2))
31
32     site=0
33
34     span_cluster=False
35
36     while span_cluster==False:
37
38         site=site+1#0 is empty space. >0 are sites and clusters
39
40         grid[random_sites[site]] = site#generate random site with unique value
41
42         """
43         CLUSTER CHECK
44         check each element in grid for neighbours
45         """
46         for i in range(1, grid_length+1):
47
48             for j in range(1, grid_length+1):
49
50                 #if element is site
51                 if grid[i,j]!=0:
52
53                     #the # of neighbours defines the action
54                     cluster_num=[]
55                     #check element left,right, up and down and collect
56                     #neighbours unique identifying value
57                     if (grid[i,j+1]!=0):
58
59                         cluster_num.append(grid[i,j+1])
60
61                     if (grid[i,j-1]!=0):
```

```

62
63         cluster_num.append(grid[i,j-1])
64
65     if (grid[i+1,j]!=0):
66
67         cluster_num.append(grid[i+1,j])
68
69     if (grid[i-1,j]!=0):
70
71         cluster_num.append(grid[i-1,j])
72
73     #if 0 neighbours do nothing, move on
74     #if 1 neighbour, label both to same number
75     if (len(cluster_num)==1):
76
77         grid[i,j]=cluster_num[0]
78
79     """
80     SPANNING SITE
81     if more than 1 neighbour, go back and check whole grid
82     relabelling all sites with values equal to site
83     neighbour's values
84     """
85     if (len(cluster_num)==2):
86
87         for i_ in range(1, grid_length+1):
88
89             for j_ in range(1, grid_length+1):
90
91                 if (grid[i_,j_]==cluster_num[0])\
92                     or (grid[i_,j_]==cluster_num[1]):
93
94                     grid[i_,j_] = grid[i,j]
95
96     if (len(cluster_num)==3):
97
98         for i_ in range(1, grid_length+1):
99
100             for j_ in range(1, grid_length+1):
101
102                 if (grid[i_,j_]==cluster_num[0]) \
103                     or (grid[i_,j_]==cluster_num[1]) \
104                     or (grid[i_,j_]==cluster_num[2]):
105
106                     grid[i_,j_] = grid[i,j]
107
108     if (len(cluster_num)==4):
109
110         for i_ in range(1, grid_length+1):
111
112             for j_ in range(1, grid_length+1):
113
114                 if (grid[i_,j_]==cluster_num[0]) \
115                     or (grid[i_,j_]==cluster_num[1]) \
116                     or (grid[i_,j_]==cluster_num[2]) \
117                     or (grid[i_,j_]==cluster_num[3]):
118
119                     grid[i_,j_] = grid[i,j]
120
121
122
123
124
125
126

```



```

127     """
128     SPANNING CLUSTER CHECK
129     See if there are elements in bottom row that are the same as in top row
130     that are not 0. If so than there is a spanning cluster. Stop placing
131     sites and exit loop above
132     """
133     compare_edges=np.in1d(grid[1],grid[-2])
134
135     for element in range(len(compare_edges)):
136
137         if compare_edges[element]==True:
138
139             if grid[1,element]!=0:
140
141                 span_cluster=True
142
143     percolate_thres = float(site)/pow(grid_length,2)
144
145     return percolate_thres
146
147 #-----#
148
149 iterations=100
150
151 data=[]
152
153 grid_length=10
154
155 for iteration in range(iterations):
156
157     percolate=Percolate(grid_length)
158     data.append(percolate)
159     #print data[iteration]
160
161 av_perc_thres=np.average(data)
162 print 'Percolation threshold for ',iterations,' iterations is ', av_perc_thres

```

7.2 Determining spanning clusters using the Hoshen - Kopelman Algorithm

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Thu Dec 07 09:56:19 2017
4
5 An implementation of the Hoshen-Kopelman Algorithm for cluster labelling a
6 grid. Cluster sizes are stored as positive values in the cluster list (csize).
7 Redirection labels are stores as negative values in the cluster list (csize).
8 The grid is processed in the following manner:
9
10 -Random_Grid_Generator : Pre-generates a square grid of length "grid_length"
11 -Raster_Scan : Scans and labels grid as per the Hoshen-Kopelman Algorithm
12 -Collapse : Collapses chain of redirection labels in csize
13 -Span_cluster_check : Checks if there exists a spanning cluster (Boolean)
14                       and returns label (cluster size in csize)
15 -run : runs all functions together in chain
16
17 Included are optional 2-D histogram plots and runtime for the grid at all
18 stages of processing.
19
20 @author: Sean Cummins
21 """
22
23 import numpy as np
24 np.set_printoptions(threshold=np.nan)
25 import matplotlib.pyplot as plt
26 import time
27
28 #-----#
29
30 #Pre-generates a a grid with a given occupation probability
31 def Random_Grid_Generator(grid_length,thres_prob,plot):
32
33     start_time = time.time()
34     grid=np.zeros((grid_length+2,grid_length+2))#add a boundary of zeros
35
36     for row in range(1,grid_length+1):
37         for column in range(1,grid_length+1):
38
39             random_prob=np.random.random()
40             grid[row,column]=random_prob
41
42             if grid[row,column]>thres_prob:
43                 grid[row,column]=0
44             else:
45                 grid[row,column]=1
46
47     if plot==True:
48         fig, ax = plt.subplots()
49         im=ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
50         fig.colorbar(im, orientation='vertical')
51         plt.title('Generating the random ' + str(grid_length) + 'x' + \
52                 str(grid_length) + ' lattice of probability ' + str(prob) )
53
54     print("Random_Grid_Generator--- %s seconds ---" % (time.time() - start_time))
55
56     return grid
57
58 #-----#
59
60 #Scan grid row-by-row left-to-right
61 def Raster_Scan(grid,grid_length,plot):
62
63     start_time = time.time()
64     largest_label=0.0
```

```

65     csize=[]#cluster size list
66     csize.append(0)#start indexing at 1
67
68     for row in range(1,grid_length+1):
69         for column in range(1,grid_length+1):
70             if (grid[row,column]!=0.0):
71
72                 above=int(grid[row-1,column])
73                 left=int(grid[row,column-1])
74
75                 #check neighbours
76                 if (left==0) and (above==0):
77
78                     largest_label=largest_label+1.0
79                     grid[row,column]=largest_label
80                     csize.append(1)
81
82                 if (left!=0) and (above==0):
83
84                     grid[row,column]=left
85                     root_left=find(left,csize)
86                     csize[root_left]=csize[root_left]+1
87
88                 if (left==0) and (above!=0):
89
90                     grid[row,column]=above
91                     root_above=find(above,csize)
92                     csize[root_above]=csize[root_above]+1
93
94                 if (left!=0) and (above!=0):
95
96                     root_left=find(left,csize)
97                     root_above=find(above,csize)
98
99                     if left<above:#always choose smallest label
100
101                         grid[row,column]=left
102                         csize[root_left]=csize[root_left]+1
103
104                     if above<left:
105
106                         grid[row,column]=above
107                         csize[root_above]=csize[root_above]+1
108
109                     if root_left<root_above:
110                         #transfer size of cluster to proper label before making
111                         #temporary
112                         csize[root_left]=csize[root_left]+csize[root_above]
113                         csize[root_above]=-root_left
114
115                     if root_above<root_left:
116
117                         csize[root_above]=csize[root_above]+csize[root_left]
118                         csize[root_left]=-root_above
119
120                     if left==above:
121
122                         grid[row,column]=above
123                         root_above=find(above,csize)
124                         csize[root_above]=csize[root_above]+1
125
126
127 if plot==True:
128     fig, ax = plt.subplots()
129     im=ax.imshow(grid, cmap='gist_earth', interpolation='nearest')

```

```

130     fig.colorbar(im, orientation='vertical')
131     plt.title('Raster scanning on a ' + str(grid_length) + 'x' + \
132             str(grid_length) + ' lattice of probability ' + str(prob))
133
134     print("Raster_Scan--- %s seconds ---" % (time.time() - start_time))
135
136     return grid , csize
137
138     #-----#
139
140     #finds proper label index for grid element
141     def find(element,csize):
142
143         if csize[element]>0:
144
145             return element
146
147         if csize[element]<0:
148
149             root_reached=False
150             csize_index=-csize[element]
151
152             while root_reached==False:
153
154                 root=csize[csize_index]
155
156                 if root>0:
157
158                     root_reached=True
159
160                 if root<0:
161
162                     csize_index=-root
163
164             return csize_index
165     #-----#
166
167     #rescan grid assigning proper labels to grid elements which have temporary
168     #labels
169     def Collapse(grid,grid_length,csize,plot):
170
171         start_time = time.time()
172
173         for row in range(1,grid_length+1):
174             for column in range(1,grid_length+1):
175                 if grid[row,column]!=0:
176                     if csize[int(grid[row,column])]<0:
177
178                         root_reached=False
179                         csize_index=-csize[int(grid[row,column])]
180
181                         while root_reached==False:
182
183                             root=csize[csize_index]
184
185                             if root>0:
186                                 grid[row,column]=csize_index
187                                 root_reached=True
188                             if root<0:
189                                 csize_index=-root
190
191         if plot==True:
192             fig, ax = plt.subplots()
193             im=ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
194             fig.colorbar(im, orientation='vertical')

```

```

195     plt.title('Collapsing the labels on a ' + str(grid_length) + 'x' + \
196               str(grid_length) + ' lattice of probability ' + str(prob))
197
198     print("Collapse--- %s seconds ---" % (time.time() - start_time))
199
200     return grid
201
202 #-----#
203 #check if there exists a spanning cluster by comparing elements from 1st row
204 #of grid with last
205 def Span_cluster_check(grid):
206
207     start_time = time.time()
208     span_cluster=False
209     span_value=False
210     compare_edges=np.in1d(grid[1],grid[-2])
211
212     for element in range(len(compare_edges)):
213         if compare_edges[element]==True:
214             if grid[1,element]!=0:
215
216                 span_cluster=True
217                 span_value=grid[1,element]
218
219     print span_cluster, ", Span_cluster_check--- %s seconds ---" % \
220           (time.time() - start_time)
221
222     return span_cluster,span_value
223
224 #-----#
225 #Run all in sequence
226 def run(grid_length,prob):
227
228     grid=Random_Grid_Generator(grid_length,prob,True)
229     grid,csize=Raster_Scan(grid,grid_length,True)
230     grid=Collapse(grid,grid_length,csize,True)
231     span_check=Span_cluster_check(grid)
232
233     return grid,csize,span_check[0]
234
235 #-----#
236 grid_length=1000
237 prob=0.603
238
239 simulate=run(grid_length,prob)

```

7.3 Estimating p_c using the Hoshen - Kopelman Algorithm

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Thu Dec 07 09:56:19 2017
4
5 Estimate  $p_c$ 
6
7 Mod to HKA to run and average 100 times for a given probability and sweep  $n$ 
8 probabilities.
9
10 @author: Sean Cummins
11 """
12
13 import numpy as np
14 np.set_printoptions(threshold=np.nan)
15 import matplotlib.pyplot as plt
16 import time
17
18 #-----#
19
20 def Random_Grid_Generator(grid_length,thres_prob,plot):
21
22     #start_time = time.time()
23     grid=np.zeros((grid_length+2,grid_length+2))
24
25     for row in range(1,grid_length+1):
26         for column in range(1,grid_length+1):
27
28             random_prob=np.random.random()
29             grid[row,column]=random_prob
30
31             if grid[row,column]>thres_prob:
32
33                 grid[row,column]=0
34
35             else:
36
37                 grid[row,column]=-1
38
39     if plot==True:
40         fig, ax = plt.subplots()
41         ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
42
43     #print("Random_Grid_Generator--- %s seconds ---" % (time.time() - start_time))
44
45     return grid
46
47 #-----#
48
49 def Raster_Scan(grid,grid_length,plot):
50
51     #start_time = time.time()
52     largest_label=0.0
53     csize=[]
54     csize.append(0)
55
56     for row in range(1,grid_length+1):
57
58         for column in range(1,grid_length+1):
59
60             if (grid[row,column]!=0.0):
61
62                 above=int(grid[row-1,column])
63                 left=int(grid[row,column-1])
64
```

```

65         if (left==0) and (above==0):
66             largest_label=largest_label+1.0
67             grid[row,column]=largest_label
68             csize.append(int(largest_label))
69
70         if (left!=0) and (above==0):
71             grid[row,column]=left
72
73         if (left==0) and (above!=0):
74             grid[row,column]=above
75
76         if (left!=0) and (above!=0):
77             if left<above:
78                 grid[row,column]=left
79                 root_left=find(left,csize)
80                 root_above=find(above,csize)
81                 if root_left<root_above:
82                     csize[root_above]=-csize[root_left]
83                 if root_above<root_left:
84                     csize[root_left]=-csize[root_above]
85
86             if above<left:
87                 grid[row,column]=left
88                 root_left=find(left,csize)
89                 root_above=find(above,csize)
90                 if root_left<root_above:
91                     csize[root_above]=-csize[root_left]
92                 if root_above<root_left:
93                     csize[root_left]=-csize[root_above]
94
95         if left==above:
96             grid[row,column]=above
97
98
99     if plot==True:
100         fig, ax = plt.subplots()
101         ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
102
103     #print("Raster_Scan--- %s seconds ---" % (time.time() - start_time))
104
105     return grid , csize
106
107 #-----#
108
109 def find(element,csize):
110
111     if csize[element]>0:
112
113         return csize[element]
114
115     if csize[element]<0:
116
117         root_reached=False
118         csize_index=-csize[element]
119
120         while root_reached==False:
121
122             root=csize[csize_index]
123
124             if root>0:
125
126                 root_reached=True
127
128             if root<0:

```

```

130         csize_index=-root
131
132     return csize[csize_index]
133 #-----#
134
135 def Collapse(grid,grid_length,csize,plot):
136
137     #start_time = time.time()
138
139     for row in range(1,grid_length+1):
140
141         for column in range(1,grid_length+1):
142
143             if grid[row,column]!=0:
144
145                 if csize[int(grid[row,column])]<0:
146
147                     root_reached=False
148                     csize_index=-csize[int(grid[row,column])]
149
150                     while root_reached==False:
151
152                         root=csize[csize_index]
153
154                         if root>0:
155
156                             grid[row,column]=csize_index
157                             root_reached=True
158                             #plt.imshow(grid, cmap='gist_earth', interpolation='nearest')
159                             #plt.draw()
160                             #plt.pause(0.0001)
161
162                         if root<0:
163
164                             csize_index=-root
165
166
167                 """
168                 if csize[int(grid[row,column])]<0:
169
170                     grid[row,column]=-csize[int(grid[row,column])]
171                 """
172
173     if plot==True:
174
175         fig, ax = plt.subplots()
176         im=ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
177         fig.colorbar(im, orientation='vertical')
178
179     #print("Collapse--- %s seconds ---" % (time.time() - start_time))
180
181     return grid
182 #-----#
183
184 def Span_cluster_check(grid):
185
186     #start_time = time.time()
187     span_cluster=False
188     span_value=False
189
190     compare_edges=np.in1d(grid[1],grid[-2])
191
192     for element in range(len(compare_edges)):
193
194

```



```

195         if compare_edges[element]==True:
196
197             if grid[1,element]!=0:
198
199                 span_cluster=True
200                 span_value=grid[1,element]
201
202             #print span_cluster, ", Span_cluster_check--- %s seconds ---" % (time.time() - start_time)
203
204         return span_cluster,span_value
205
206 #-----#
207
208 def n_runs(grid_length,prob_i,prob_f,n):
209
210     averages=[]
211     dprob=(prob_f-prob_i)/n
212
213     for i in range(n):
214
215         n_runs=[]
216         prob=prob_i+i*dprob
217         print prob
218
219         for j in range(100):
220
221             grid=Random_Grid_Generator(grid_length,prob,False)
222             grid,csize=Raster_Scan(grid,grid_length,False)
223             grid=Collapse(grid,grid_length,csize,False)
224             span_check=Span_cluster_check(grid)
225             n_runs.append(span_check[0])
226
227         average=np.average(n_runs)
228         averages.append((prob,average))
229
230     return averages
231
232 #-----#
233 grid_length=100
234
235 #Range of probabilities to sweep
236 prob_i=0.575
237 prob_f=0.625
238 n=10
239
240 pc=0.593
241
242 n_runs=n_runs(grid_length,prob_i,prob_f,n)
243
244 p,num_spans = zip(*n_runs)
245 plt.plot(p,num_spans,'o--')
246 plt.axvline(x=pc,c='r',label='Percolation threshold $P_c$')
247 plt.grid()
248 plt.xlabel('Probability $p$')
249 plt.ylabel('Fraction of spanning clusters')
250 plt.title('Fraction of spanning clusters vs site probability (' + str(grid_length) + 'x' + str(grid_length) + ' lattice)')
251 plt.legend()
252 plt.show()

```

7.4 Estimating β using the Hoshen - Kopelman Algorithm

```
1  #- coding: utf-8 #-
2  """
3  Created on Thu Dec 07 09:56:19 2017
4
5  Estimate beta
6
7  @author: Sean Cummins
8  """
9
10 import numpy as np
11 np.set_printoptions(threshold=np.nan)
12 import matplotlib.pyplot as plt
13 import time
14 from scipy.optimize import curve_fit
15
16 #-----#
17
18 def Random_Grid_Generator(grid_length,thres_prob,plot):
19
20     start_time = time.time()
21     grid=np.zeros((grid_length+2,grid_length+2))
22
23     for row in range(1,grid_length+1):
24         for column in range(1,grid_length+1):
25             random_prob=np.random.random()
26             grid[row,column]=random_prob
27             if grid[row,column]>thres_prob:
28                 grid[row,column]=0
29             else:
30                 grid[row,column]=1
31
32     if plot==True:
33         fig, ax = plt.subplots()
34         im=ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
35         fig.colorbar(im, orientation='vertical')
36         plt.title('Generating the random ' + str(grid_length) + 'x' + \
37                 str(grid_length) + ' lattice of probability ' + str(prob) )
38
39     #print("Random_Grid_Generator--- %s seconds ---" % (time.time() - start_time))
40
41     return grid
42
43 #-----#
44
45 def Raster_Scan(grid,grid_length,plot):
46
47     start_time = time.time()
48     largest_label=0.0
49     csize=[]
50     csize.append(0)#start indexing at 1
51
52     for row in range(1,grid_length+1):
53         for column in range(1,grid_length+1):
54             if (grid[row,column]!=0.0):
55                 above=int(grid[row-1,column])
56                 left=int(grid[row,column-1])
57                 if (left==0) and (above==0):
58                     largest_label=largest_label+1.0
59                     grid[row,column]=largest_label
60                     csize.append(1)
61                 if (left!=0) and (above==0):
62                     grid[row,column]=left
63                     root_left=find(left,csize)
64                     csize[root_left]=csize[root_left]+1
```

```

65         if (left==0) and (above!=0):
66             grid[row,column]=above
67             root_above=find(above,csize)
68             csize[root_above]=csize[root_above]+1
69         if (left!=0) and (above!=0):
70             root_left=find(left,csize)
71             root_above=find(above,csize)
72             if left<above:
73                 grid[row,column]=left
74                 csize[root_left]=csize[root_left]+1
75             if above<left:
76                 grid[row,column]=above
77                 csize[root_above]=csize[root_above]+1
78             if root_left<root_above:
79                 csize[root_left]=csize[root_left]+csize[root_above]
80                 csize[root_above]=-root_left
81             if root_above<root_left:
82                 csize[root_above]=csize[root_above]+csize[root_left]
83                 csize[root_left]=-root_above
84             if left==above:
85                 grid[row,column]=above
86                 root_above=find(above,csize)
87                 csize[root_above]=csize[root_above]+1
88
89
90     if plot==True:
91         fig, ax = plt.subplots()
92         im=ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
93         fig.colorbar(im, orientation='vertical')
94         plt.title('Raster scanning on a ' + str(grid_length) + 'x' + \
95                 str(grid_length) + ' lattice of probability ' + str(prob))
96
97     #print("Raster_Scan--- %s seconds ---" % (time.time() - start_time))
98
99     return grid , csize
100
101 #-----#
102
103 def find(element,csize):
104
105     if csize[element]>0:
106         return element
107     if csize[element]<0:
108         root_reached=False
109         csize_index=-csize[element]
110         while root_reached==False:
111             root=csize[csize_index]
112             if root>0:
113                 root_reached=True
114             if root<0:
115                 csize_index=-root
116
117         return csize_index
118 #-----#
119
120 def Collapse(grid,grid_length,csize,plot):
121
122     start_time = time.time()
123
124     for row in range(1,grid_length+1):
125         for column in range(1,grid_length+1):
126             if grid[row,column]!=0:
127                 if csize[int(grid[row,column])]<0:
128                     root_reached=False
129                     csize_index=-csize[int(grid[row,column])]

```

```

130         while root_reached==False:
131             root=csize[csize_index]
132             if root>0:
133                 grid[row,column]=csize_index
134                 root_reached=True
135             if root<0:
136                 csize_index=-root
137
138     if plot==True:
139         fig, ax = plt.subplots()
140         im=ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
141         fig.colorbar(im, orientation='vertical')
142         plt.title('Collapsing the labels on a ' + str(grid_length) + \
143                 'x'+ str(grid_length) + ' lattice of probability ' + str(prob))
144
145     #print("Collapse--- %s seconds ---" % (time.time() - start_time))
146
147     return grid
148
149 #-----#
150
151 def Span_cluster_check(grid,csize):
152
153     start_time = time.time()
154     span_cluster=False
155     span_size=False
156     compare_edges=np.in1d(grid[1],grid[-2])
157
158     for element in range(len(compare_edges)):
159         if compare_edges[element]==True:
160             if grid[1,element]!=0:
161                 span_cluster=True
162                 span_size=csize[int(grid[1,element])]
163
164     #print span_cluster, ", Span_cluster_check--- %s seconds ---" % (time.time() - start_time)
165
166     return span_cluster,span_size
167
168 #-----#
169
170 def run(grid_length,prob):
171
172     grid=Random_Grid_Generator(grid_length,prob,False)
173     grid,csize=Raster_Scan(grid,grid_length,False)
174     grid=Collapse(grid,grid_length,csize,False)
175     span_check=Span_cluster_check(grid)
176
177     return grid,csize,span_check
178
179 #-----#
180
181 def n_runs(grid_length,prob_i,prob_f,n):
182
183     averages=[]
184     dprob=(prob_f-prob_i)/n
185
186     for i in range(n):
187
188         prob=prob_i+i*dprob
189         print prob
190         n_runs=[]
191
192         for j in range(100):
193
194             occupied_sites=0.0

```

```

195         check=False
196         while check==False:
197
198             grid=Random_Grid_Generator(grid_length,prob,False)
199             grid,csize=Raster_Scan(grid,grid_length,False)
200             grid=Collapse(grid,grid_length,csize,False)
201             span_check=Span_cluster_check(grid,csize)
202
203             perc_cluster_size=span_check[1]
204             check=span_check[0]
205
206             for i in range(len(csize)):
207                 if csize[i]!=0 and csize[i]>0.0:
208                     occupied_sites=occupied_sites+csize[i]
209
210             frac_perc_cluster_size=perc_cluster_size/occupied_sites
211
212             n_runs.append(frac_perc_cluster_size)
213             """
214             fig, ax = plt.subplots()
215             im=ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
216             fig.colorbar(im, orientation='vertical')
217             """
218
219             average=np.average(n_runs)
220             averages.append((prob,average))
221
222         return averages
223
224 #-----#
225
226 grid_length=25
227
228 #Take many values over rapidly changing range
229
230 prob_i=0.595
231 prob_f=0.65
232 n=10
233
234 n_runs_1=n_runs(grid_length,prob_i,prob_f,n)
235
236 #This range changes more slowly
237
238 prob_i=0.65
239 prob_f=1.0
240 n=10
241
242 n_runs_2=n_runs(grid_length,prob_i,prob_f,n)
243
244 p_1,F_1 = zip(*n_runs_1)
245 p_2,F_2 = zip(*n_runs_2)
246
247
248 """
249  $y = N * x ** a$ 
250  $\ln(y) = \ln(N * x ** a)$ 
251  $\ln(y) = a * \ln(x) + \ln(N)$ 
252 """
253
254 pc=0.593
255
256 F=np.concatenate((np.asarray(F_1),np.asarray(F_2)),axis=0)
257 p=np.concatenate((np.asarray(p_1),np.asarray(p_2)),axis=0)
258
259 plt.figure()

```

```

260 plt.plot(p,F,'-o',label='Fraction of sites in percolating cluster ('\
261         + str(grid_length) + 'x'+ str(grid_length) + ' lattice)')
262 plt.xlabel('p')
263 plt.ylabel('$F_c$')
264 plt.grid()
265 plt.legend()
266
267
268 F=np.log(F)
269 p=np.log(p-pc)
270
271 plt.figure()
272 plt.plot(p,F,'-o',label='Fraction of sites in percolating cluster (' \
273         + str(grid_length) + 'x'+ str(grid_length) + ' lattice)')
274
275
276 #power fit data - obtain estimate for b (beta)
277 def func_powerlaw(x,b):
278     return b*x
279
280 target_func = func_powerlaw
281 popt, pcov = curve_fit(target_func, p, F,maxfev=1000)
282
283 plt.plot(p, target_func(F, *popt), '-o',label='Fit of fraction of sites in percolating cluster ('\
284         + str(grid_length) + 'x'+ str(grid_length) + ' lattice)')
285
286 plt.title('2D percolation on a lattice')
287 plt.xlabel('p')
288 plt.ylabel('$F_c$')
289 plt.grid()
290 plt.legend()
291 plt.show()
292
293 print popt

```