EP408 Computational Physics Project - Percolation

Sean Cummins
14432458

Department of Experimental Physics,
Maynooth University, Co Kildare

April 15, 2019

Contents

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-Kopleman 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 though 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 a is 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 called *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.

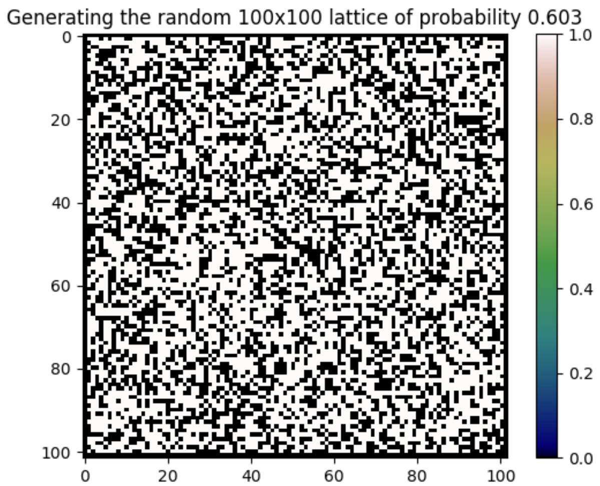
Like before we always use the smallest of the clusters numbers for the new larger cluster.

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

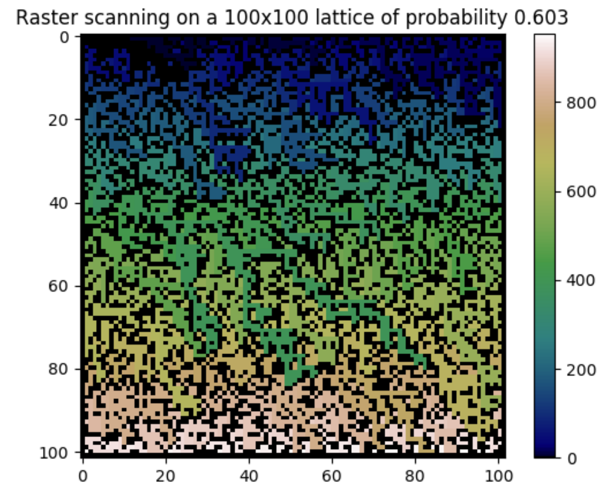
We now present the results from using the algorithm.

4 Results

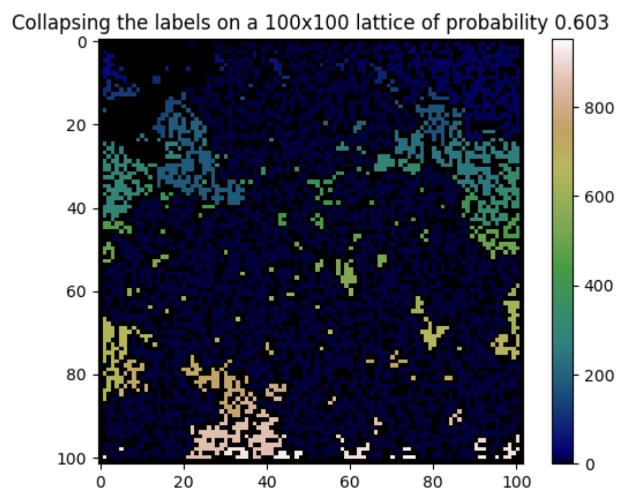
4.1 Determining spanning clusters using Hoshen - Kopelman Algorithm



(a) Random Grid Generator — 0.473999977112 seconds —



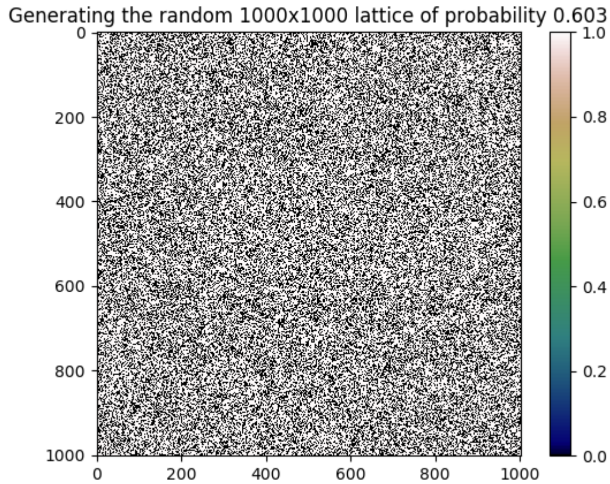
(b) Raster Scan — 0.461000204086 seconds —



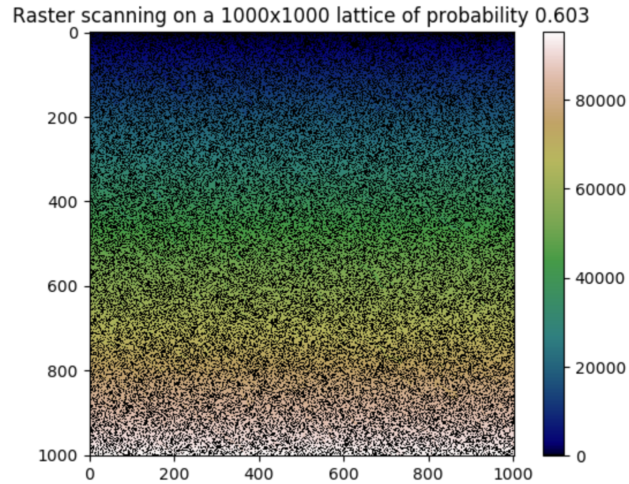
(c) Collapse — 0.451999902725 seconds —

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

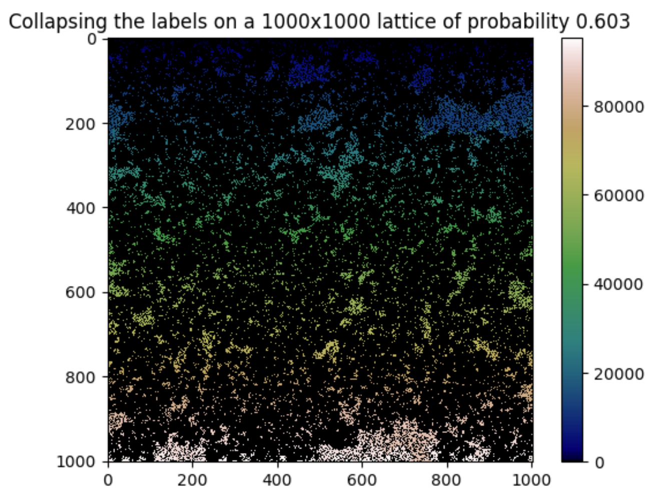
Determining spanning clusters using Hoshen - Kopelman Algorithm



(a) Random Grid Generator — 2.59899997711 seconds —



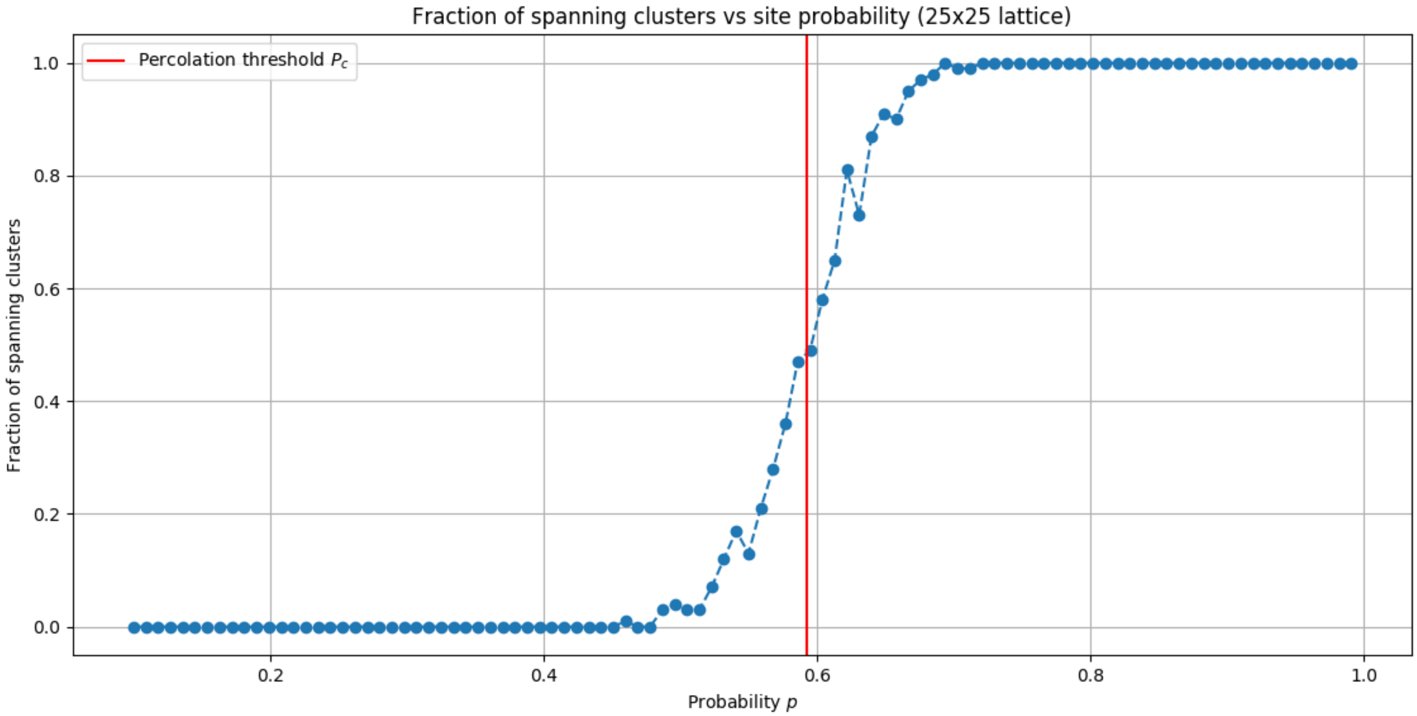
(b) Raster Scan — 4.44200015068 seconds —



(c) Collapse — 3.7539999485 seconds —

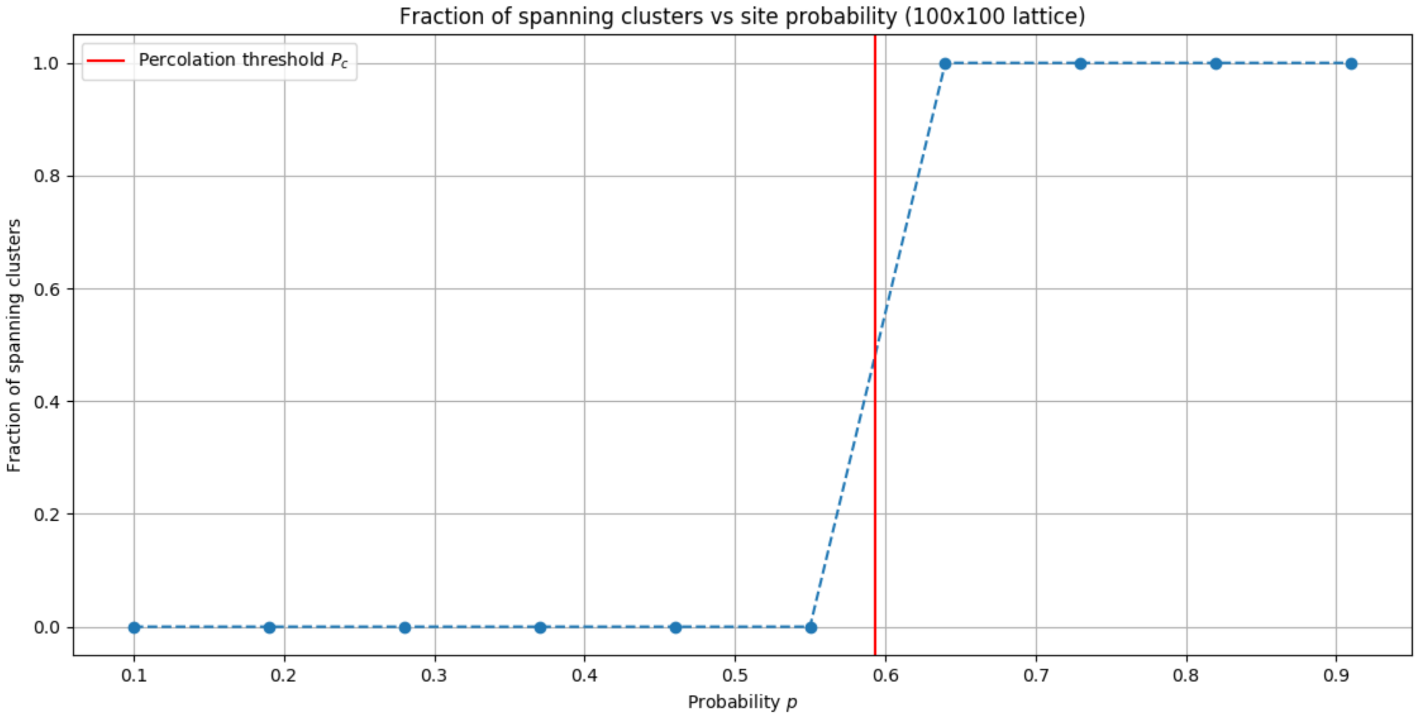
Span cluster check correctly confirms a spanning cluster.

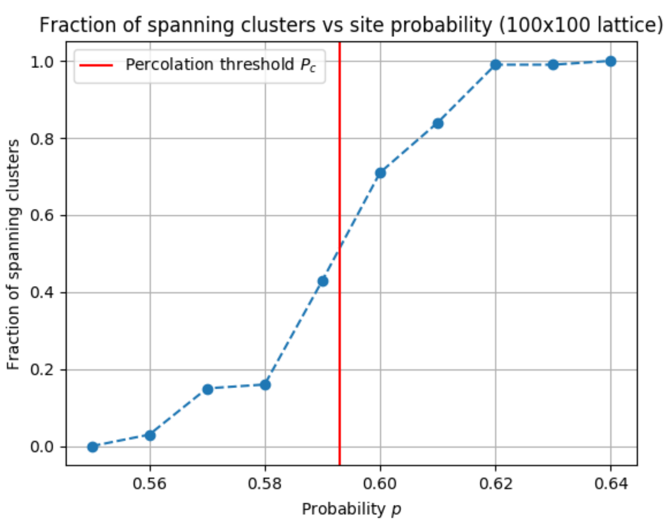
4.2 Estimating p_c using Hoshen - Kopelman Algorithm



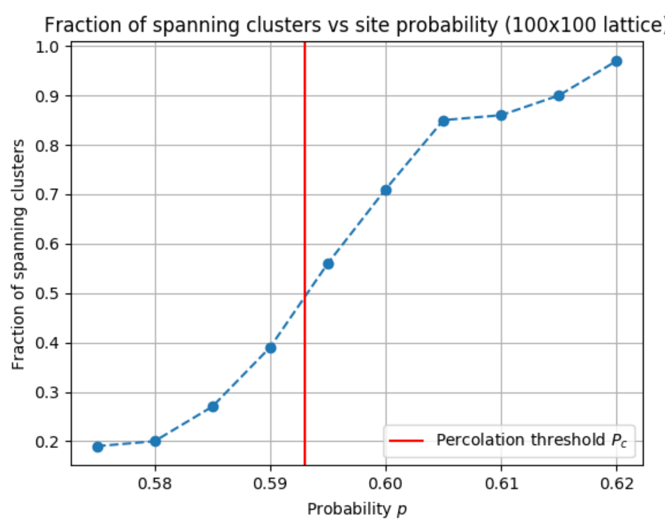
We see the step-like switch between $p < p_c$ and $p > p_c$ with the expected value for p_c in the middle.

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.

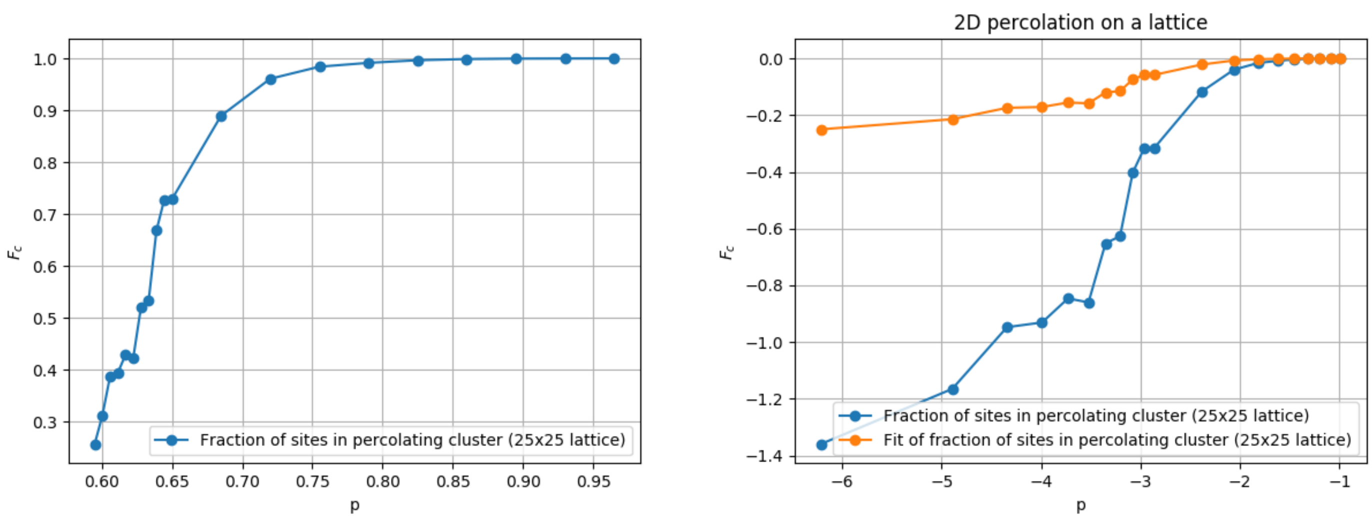
We consider the fraction of sites F that are in the spanning cluster as a function of p. This relationship is described the following power law.

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

The equation naturally contains p_c and as a result the quality of our estimate for *beta* 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

```python
# -*- coding: utf-8 -*-
"""
Created on Fri Dec 01 16:47:50 2017

Percolation - site model

Very inefficient. Very slow for grid sizes > 10x10

@author: Sean Cummins
"""

import numpy as np
np.set_printoptions(threshold=np.nan)
#for size < 10x10 , whole grid can be printed to screen
import random

#---------------------------------------------------------------------------#

def Percolate(grid_length):

    grid=np.zeros((grid_length+2,grid_length+2))
    #+2 each size is boundary so all checks for each element are valid

    """
    RANDOM SITES
    Generate as many unique random sites as can fit
    """
    possible_coordinates = [(x, y) for x in range(1,grid_length+1)\
                            for y in range(1, grid_length+1)]
    random_sites = random.sample(possible_coordinates, pow(grid_length,2))

    site=0

    span_cluster=False

    while span_cluster==False:

        site=site+1#0 is empty space. >0 are sites and clusters

        grid[random_sites[site]] = site#generate random site with unique value

        """
        CLUSTER CHECK
        check each element in grid for neighbours
        """
        for i in range(1, grid_length+1):

            for j in range(1, grid_length+1):

                    #if element is site
                    if grid[i,j]!=0:

                        #the # of neignbours defines the action
                        cluster_num=[]
                        #check element left,right, up and down and collect
                        #neighbours unique identifying value
                        if (grid[i,j+1]!=0):

                                cluster_num.append(grid[i,j+1])

                        if (grid[i,j-1]!=0):
```

```python

            cluster_num.append(grid[i,j-1])

    if (grid[i+1,j]!=0):

            cluster_num.append(grid[i+1,j])

    if (grid[i-1,j]!=0):

            cluster_num.append(grid[i-1,j])

    #if 0 neighbours do nothing, move on
    #if 1 neighbour, label both to same number
    if (len(cluster_num)==1):

        grid[i,j]=cluster_num[0]

    """
    SPANNING SITE
    if more than 1 neighbour, go back and check whole grid
    relabelling all sites with values equal to site
    neighbour's values
    """
    if (len(cluster_num)==2):

            for i_ in range(1, grid_length+1):

                for j_ in range(1, grid_length+1):

                    if (grid[i_,j_]==cluster_num[0])\
                    or (grid[i_,j_]==cluster_num[1]):

                        grid[i_,j_] = grid[i,j]

    if (len(cluster_num)==3):

        for i_ in range(1, grid_length+1):

            for j_ in range(1, grid_length+1):

                if (grid[i_,j_]==cluster_num[0]) \
                or (grid[i_,j_]==cluster_num[1]) \
                or (grid[i_,j_]==cluster_num[2]):

                    grid[i_,j_] = grid[i,j]

    if (len(cluster_num)==4):

        for i_ in range(1, grid_length+1):

            for j_ in range(1, grid_length+1):

                if (grid[i_,j_]==cluster_num[0]) \
                or (grid[i_,j_]==cluster_num[1]) \
                or (grid[i_,j_]==cluster_num[2]) \
                or (grid[i_,j_]==cluster_num[3]):

                    grid[i_,j_] = grid[i,j]
```

```
        """
        SPANNING CLUSTER CHECK
        See if there are elements in bottom row that are the same as in top row
        that are not 0. If so than there is a spanning cluster. Stop placing
        sites and exit loop above
        """
        compare_edges=np.in1d(grid[1],grid[-2])

        for element in range(len(compare_edges)):

            if compare_edges[element]==True:

                if grid[1,element]!=0:

                    span_cluster=True

    percolate_thres = float(site)/pow(grid_length,2)

    return percolate_thres

#----------------------------------------------------------------------------#

iterations=100

data=[]

grid_length=10

for iteration in range(iterations):

    percolate=Percolate(grid_length)
    data.append(percolate)
    #print data[iteration]

av_perc_thres=np.average(data)
print 'Percolation threshold for ',iterations,' iterations is ', av_perc_thres
```

7.2 Determining spanning clusters using the Hoshen - Kopelman Algorithm

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Dec 07 09:56:19 2017

An implementation of the Hoshen-Koplemann Algorithm for cluster labelling a
grid. Cluster sizes are stored as positive values in the cluster list (csize).
Redirection labels are stores as negative values in the cluster list (csize).
The grid is processed in the following manner:

    -Random_Grid_Generator : Pre-generates a square grid of length "grid_length"
    -Raster_Scan : Scans and labels grid as per the Hoshen-Koplemann Algorithm
    -Collpse : Collapses chain of redirection labels in csize
    -Span_cluster_check : Checks if there exists a spanning cluster (Boolean)
                          and returns label (cluster size in csize)
    -run : runs all functions together in chain

Included are optional 2-D histogram plots and runtime for the grid at all
stages of processing.

@author: Sean Cummins
"""

import numpy as np
np.set_printoptions(threshold=np.nan)
import matplotlib.pyplot as plt
import time

#-----------------------------------------------------------------------------#

#Pre-generates a a grid with a given occupation probability
def Random_Grid_Generator(grid_length,thres_prob,plot):

    start_time = time.time()
    grid=np.zeros((grid_length+2,grid_length+2))#add a boundary of zeros

    for row in range(1,grid_length+1):
        for column in range(1,grid_length+1):

            random_prob=np.random.random()
            grid[row,column]=random_prob

            if grid[row,column]>thres_prob:
                grid[row,column]=0
            else:
                grid[row,column]=1

    if plot==True:
        fig, ax = plt.subplots()
        im=ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
        fig.colorbar(im, orientation='vertical')
        plt.title('Generating the random '  + str(grid_length) + 'x'+ \
                  str(grid_length) + ' lattice of probability ' + str(prob) )

    print("Random_Grid_Generator--- %s seconds ---" % (time.time() - start_time))

    return grid

#-----------------------------------------------------------------------------#

#Scan grid row-by-row left-to-right
def Raster_Scan(grid,grid_length,plot):

    start_time = time.time()
    largest_label=0.0
```

```python
    csize=[]#cluster size list
    csize.append(0)#start indexing at 1

    for row in range(1,grid_length+1):
        for column in range(1,grid_length+1):
            if (grid[row,column]!=0.0):

                above=int(grid[row-1,column])
                left=int(grid[row,column-1])

                #check neighbours
                if (left==0) and (above==0):

                    largest_label=largest_label+1.0
                    grid[row,column]=largest_label
                    csize.append(1)

                if (left!=0) and (above==0):

                    grid[row,column]=left
                    root_left=find(left,csize)
                    csize[root_left]=csize[root_left]+1

                if (left==0) and (above!=0):

                    grid[row,column]=above
                    root_above=find(above,csize)
                    csize[root_above]=csize[root_above]+1

                if (left!=0) and (above!=0):

                    root_left=find(left,csize)
                    root_above=find(above,csize)

                    if left<above:#always choose smallest label

                        grid[row,column]=left
                        csize[root_left]=csize[root_left]+1

                    if above<left:

                        grid[row,column]=above
                        csize[root_above]=csize[root_above]+1

                    if root_left<root_above:
                        #transfer size of cluster to proper label before making
                        #temporary
                        csize[root_left]=csize[root_left]+csize[root_above]
                        csize[root_above]=-root_left

                    if root_above<root_left:

                        csize[root_above]=csize[root_above]+csize[root_left]
                        csize[root_left]=-root_above

                    if left==above:

                        grid[row,column]=above
                        root_above=find(above,csize)
                        csize[root_above]=csize[root_above]+1


    if plot==True:
        fig, ax = plt.subplots()
        im=ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
```

```
        fig.colorbar(im, orientation='vertical')
        plt.title('Raster scanning on a '  + str(grid_length) + 'x'+ \
                  str(grid_length) + ' lattice of probability ' + str(prob))

    print("Raster_Scan--- %s seconds ---" % (time.time() - start_time))

    return grid , csize

#----------------------------------------------------------------------------#

#finds proper label index for grid element
def find(element,csize):

        if csize[element]>0:

            return element

        if csize[element]<0:

            root_reached=False
            csize_index=-csize[element]

            while root_reached==False:

                root=csize[csize_index]

                if root>0:

                    root_reached=True

                if root<0:

                    csize_index=-root

            return csize_index
#----------------------------------------------------------------------------#

#rescan grid assigning proper labels to grid elements which have temporary
#labels
def Collapse(grid,grid_length,csize,plot):

    start_time = time.time()

    for row in range(1,grid_length+1):
        for column in range(1,grid_length+1):
            if grid[row,column]!=0:
                if csize[int(grid[row,column])]<0:

                    root_reached=False
                    csize_index=-csize[int(grid[row,column])]

                    while root_reached==False:

                        root=csize[csize_index]

                        if root>0:
                            grid[row,column]=csize_index
                            root_reached=True
                        if root<0:
                            csize_index=-root

    if plot==True:
        fig, ax = plt.subplots()
        im=ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
        fig.colorbar(im, orientation='vertical')
```

```
        plt.title('Collapsing the labels on a '  + str(grid_length) + 'x'+ \
                  str(grid_length) + ' lattice of probability ' + str(prob))

    print("Collapse--- %s seconds ---" % (time.time() - start_time))

    return grid

#---------------------------------------------------------------------------#
#check if there exists a spanning cluster by comparing elements from 1st row
#of grid with last
def Span_cluster_check(grid):

    start_time = time.time()
    span_cluster=False
    span_value=False
    compare_edges=np.in1d(grid[1],grid[-2])

    for element in range(len(compare_edges)):
        if compare_edges[element]==True:
            if grid[1,element]!=0:

                span_cluster=True
                span_value=grid[1,element]

    print span_cluster, ", Span_cluster_check--- %s seconds ---" % \
    (time.time() - start_time)

    return span_cluster,span_value

#---------------------------------------------------------------------------#
#Run all in sequence
def run(grid_length,prob):

    grid=Random_Grid_Generator(grid_length,prob,True)
    grid,csize=Raster_Scan(grid,grid_length,True)
    grid=Collapse(grid,grid_length,csize,True)
    span_check=Span_cluster_check(grid)

    return grid,csize,span_check[0]

#---------------------------------------------------------------------------#
grid_length=1000
prob=0.603

simulate=run(grid_length,prob)
```

7.3 Estimating p_c using the Hoshen - Kopelman Algorithm

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Dec 07 09:56:19 2017

Estimate p_c

Mod to HKA to run and average 100 times for a given probability and sweep n
probabilities.

@author: Sean Cummins
"""

import numpy as np
np.set_printoptions(threshold=np.nan)
import matplotlib.pyplot as plt
import time

#----------------------------------------------------------------------------#

def Random_Grid_Generator(grid_length,thres_prob,plot):

    #start_time = time.time()
    grid=np.zeros((grid_length+2,grid_length+2))

    for row in range(1,grid_length+1):
        for column in range(1,grid_length+1):

            random_prob=np.random.random()
            grid[row,column]=random_prob

            if grid[row,column]>thres_prob:

                grid[row,column]=0

            else:

                grid[row,column]=-1

    if plot==True:
        fig, ax = plt.subplots()
        ax.imshow(grid, cmap='gist_earth', interpolation='nearest')

    #print("Random_Grid_Generator--- %s seconds ---" % (time.time() - start_time))

    return grid

#----------------------------------------------------------------------------#

def Raster_Scan(grid,grid_length,plot):

    #start_time = time.time()
    largest_label=0.0
    csize=[]
    csize.append(0)

    for row in range(1,grid_length+1):

        for column in range(1,grid_length+1):

            if (grid[row,column]!=0.0):

                above=int(grid[row-1,column])
                left=int(grid[row,column-1])

```

```python
                if (left==0) and (above==0):
                    largest_label=largest_label+1.0
                    grid[row,column]=largest_label
                    csize.append(int(largest_label))

                if (left!=0) and (above==0):
                    grid[row,column]=left

                if (left==0) and (above!=0):
                    grid[row,column]=above

                if (left!=0) and (above!=0):
                    if left<above:
                        grid[row,column]=left
                        root_left=find(left,csize)
                        root_above=find(above,csize)
                        if root_left<root_above:
                            csize[root_above]=-csize[root_left]
                        if root_above<root_left:
                            csize[root_left]=-csize[root_above]

                    if above<left:
                        grid[row,column]=left
                        root_left=find(left,csize)
                        root_above=find(above,csize)
                        if root_left<root_above:
                            csize[root_above]=-csize[root_left]
                        if root_above<root_left:
                            csize[root_left]=-csize[root_above]

                    if left==above:
                        grid[row,column]=above


    if plot==True:
        fig, ax = plt.subplots()
        ax.imshow(grid, cmap='gist_earth', interpolation='nearest')

    #print("Raster_Scan--- %s seconds ---" % (time.time() - start_time))

    return grid , csize

#-----------------------------------------------------------------------------#

def find(element,csize):

        if csize[element]>0:

            return csize[element]

        if csize[element]<0:

            root_reached=False
            csize_index=-csize[element]

            while root_reached==False:

                root=csize[csize_index]

                if root>0:

                    root_reached=True

                if root<0:

```

```
                    csize_index=-root

            return csize[csize_index]
#----------------------------------------------------------------------------#

def Collapse(grid,grid_length,csize,plot):

    #start_time = time.time()

    for row in range(1,grid_length+1):

        for column in range(1,grid_length+1):

            if grid[row,column]!=0:

                if csize[int(grid[row,column])]<0:

                    root_reached=False
                    csize_index=-csize[int(grid[row,column])]

                    while root_reached==False:

                        root=csize[csize_index]

                        if root>0:

                            grid[row,column]=csize_index
                            root_reached=True
                            #plt.imshow(grid, cmap='gist_earth', interpolation='nearest')
                            #plt.draw()
                            #plt.pause(0.0001)

                        if root<0:

                            csize_index=-root


                    """
                    if csize[int(grid[row,column])]<0:

                        grid[row,column]=-csize[int(grid[row,column])]
                    """

    if plot==True:

        fig, ax = plt.subplots()
        im=ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
        fig.colorbar(im, orientation='vertical')

    #print("Collapse--- %s seconds ---" % (time.time() - start_time))

    return grid
#----------------------------------------------------------------------------#

def Span_cluster_check(grid):

    #start_time = time.time()
    span_cluster=False
    span_value=False

    compare_edges=np.in1d(grid[1],grid[-2])

    for element in range(len(compare_edges)):

```

```
        if compare_edges[element]==True:

            if grid[1,element]!=0:

                span_cluster=True
                span_value=grid[1,element]

    #print span_cluster, ", Span_cluster_check--- %s seconds ---" % (time.time() - start_time)

    return span_cluster,span_value

#----------------------------------------------------------------------------#

def n_runs(grid_length,prob_i,prob_f,n):

    averages=[]
    dprob=(prob_f-prob_i)/n

    for i in range(n):

        n_runs=[]
        prob=prob_i+i*dprob
        print prob

        for j in range(100):

            grid=Random_Grid_Generator(grid_length,prob,False)
            grid,csize=Raster_Scan(grid,grid_length,False)
            grid=Collapse(grid,grid_length,csize,False)
            span_check=Span_cluster_check(grid)
            n_runs.append(span_check[0])

        average=np.average(n_runs)
        averages.append((prob,average))

    return averages

#----------------------------------------------------------------------------#
grid_length=100

#Range of probabilities to sweep
prob_i=0.575
prob_f=0.625
n=10

pc=0.593

n_runs=n_runs(grid_length,prob_i,prob_f,n)

p,num_spans = zip(*n_runs)
plt.plot(p,num_spans,'o--')
plt.axvline(x=pc,c='r',label='Percolation threshold $P_c$')
plt.grid()
plt.xlabel('Probability $p$')
plt.ylabel('Fraction of spanning clusters')
plt.title('Fraction of spanning clusters vs site probability (' + str(grid_length) + 'x'+ str(grid_length) + ' lattice)')
plt.legend()
plt.show()
```

7.4 Estimating β using the Hoshen - Kopelman Algorithm

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Dec 07 09:56:19 2017

Estimate beta

@author: Sean Cummins
"""

import numpy as np
np.set_printoptions(threshold=np.nan)
import matplotlib.pyplot as plt
import time
from scipy.optimize import curve_fit

#-----------------------------------------------------------------------------#

def Random_Grid_Generator(grid_length,thres_prob,plot):

    start_time = time.time()
    grid=np.zeros((grid_length+2,grid_length+2))

    for row in range(1,grid_length+1):
        for column in range(1,grid_length+1):
            random_prob=np.random.random()
            grid[row,column]=random_prob
            if grid[row,column]>thres_prob:
                grid[row,column]=0
            else:
                grid[row,column]=1

    if plot==True:
        fig, ax = plt.subplots()
        im=ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
        fig.colorbar(im, orientation='vertical')
        plt.title('Generating the random '  + str(grid_length) + 'x'+ \
                  str(grid_length) + ' lattice of probability ' + str(prob) )

    #print("Random_Grid_Generator--- %s seconds ---" % (time.time() - start_time))

    return grid

#-----------------------------------------------------------------------------#

def Raster_Scan(grid,grid_length,plot):

    start_time = time.time()
    largest_label=0.0
    csize=[]
    csize.append(0)#start indexing at 1

    for row in range(1,grid_length+1):
        for column in range(1,grid_length+1):
            if (grid[row,column]!=0.0):
                above=int(grid[row-1,column])
                left=int(grid[row,column-1])
                if (left==0) and (above==0):
                    largest_label=largest_label+1.0
                    grid[row,column]=largest_label
                    csize.append(1)
                if (left!=0) and (above==0):
                    grid[row,column]=left
                    root_left=find(left,csize)
                    csize[root_left]=csize[root_left]+1
```

```
                if (left==0) and (above!=0):
                    grid[row,column]=above
                    root_above=find(above,csize)
                    csize[root_above]=csize[root_above]+1
                if (left!=0) and (above!=0):
                    root_left=find(left,csize)
                    root_above=find(above,csize)
                    if left<above:
                        grid[row,column]=left
                        csize[root_left]=csize[root_left]+1
                    if above<left:
                        grid[row,column]=above
                        csize[root_above]=csize[root_above]+1
                    if root_left<root_above:
                        csize[root_left]=csize[root_left]+csize[root_above]
                        csize[root_above]=-root_left
                    if root_above<root_left:
                        csize[root_above]=csize[root_above]+csize[root_left]
                        csize[root_left]=-root_above
                    if left==above:
                        grid[row,column]=above
                        root_above=find(above,csize)
                        csize[root_above]=csize[root_above]+1


    if plot==True:
        fig, ax = plt.subplots()
        im=ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
        fig.colorbar(im, orientation='vertical')
        plt.title('Raster scanning on a '  + str(grid_length) + 'x'+ \
                  str(grid_length) + ' lattice of probability ' + str(prob))

    #print("Raster_Scan--- %s seconds ---" % (time.time() - start_time))

    return grid , csize

#-----------------------------------------------------------------------------#

def find(element,csize):

        if csize[element]>0:
            return element
        if csize[element]<0:
            root_reached=False
            csize_index=-csize[element]
            while root_reached==False:
                root=csize[csize_index]
                if root>0:
                    root_reached=True
                if root<0:
                    csize_index=-root

            return csize_index
#-----------------------------------------------------------------------------#

def Collapse(grid,grid_length,csize,plot):

    start_time = time.time()

    for row in range(1,grid_length+1):
        for column in range(1,grid_length+1):
            if grid[row,column]!=0:
                if csize[int(grid[row,column])]<0:
                    root_reached=False
                    csize_index=-csize[int(grid[row,column])]
```

```
                    while root_reached==False:
                        root=csize[csize_index]
                        if root>0:
                            grid[row,column]=csize_index
                            root_reached=True
                        if root<0:
                            csize_index=-root

    if plot==True:
        fig, ax = plt.subplots()
        im=ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
        fig.colorbar(im, orientation='vertical')
        plt.title('Collapsing the labels on a '  + str(grid_length) +\
                'x'+ str(grid_length) + ' lattice of probability ' + str(prob))

    #print("Collapse--- %s seconds ---" % (time.time() - start_time))

    return grid

#-----------------------------------------------------------------------------#

def Span_cluster_check(grid,csize):

    start_time = time.time()
    span_cluster=False
    span_size=False
    compare_edges=np.in1d(grid[1],grid[-2])

    for element in range(len(compare_edges)):
        if compare_edges[element]==True:
            if grid[1,element]!=0:
                span_cluster=True
                span_size=csize[int(grid[1,element])]

    #print span_cluster, ", Span_cluster_check--- %s seconds ---" % (time.time() - start_time)

    return span_cluster,span_size

#-----------------------------------------------------------------------------#

def run(grid_length,prob):

    grid=Random_Grid_Generator(grid_length,prob,False)
    grid,csize=Raster_Scan(grid,grid_length,False)
    grid=Collapse(grid,grid_length,csize,False)
    span_check=Span_cluster_check(grid)

    return grid,csize,span_check

#-----------------------------------------------------------------------------#

def n_runs(grid_length,prob_i,prob_f,n):

    averages=[]
    dprob=(prob_f-prob_i)/n

    for i in range(n):

        prob=prob_i+i*dprob
        print prob
        n_runs=[]

        for j in range(100):

            occupied_sites=0.0
```

```
            check=False
            while check==False:

                grid=Random_Grid_Generator(grid_length,prob,False)
                grid,csize=Raster_Scan(grid,grid_length,False)
                grid=Collapse(grid,grid_length,csize,False)
                span_check=Span_cluster_check(grid,csize)

                perc_cluster_size=span_check[1]
                check=span_check[0]

                for i in range(len(csize)):
                    if csize[i]!=0 and csize[i]>0.0:
                        occupied_sites=occupied_sites+csize[i]

                frac_perc_cluster_size=perc_cluster_size/occupied_sites

                n_runs.append(frac_perc_cluster_size)
                """
                fig, ax = plt.subplots()
                im=ax.imshow(grid, cmap='gist_earth', interpolation='nearest')
                fig.colorbar(im, orientation='vertical')
                """

        average=np.average(n_runs)
        averages.append((prob,average))

    return averages

#------------------------------------------------------------------------------#

grid_length=25

#Take many values over rapidly changing range

prob_i=0.595
prob_f=0.65
n=10

n_runs_1=n_runs(grid_length,prob_i,prob_f,n)

#This range changes more slowly

prob_i=0.65
prob_f=1.0
n=10

n_runs_2=n_runs(grid_length,prob_i,prob_f,n)

p_1,F_1 = zip(*n_runs_1)
p_2,F_2 = zip(*n_runs_2)


"""
y = N * x ** a
ln(y) = ln(N * x ** a)
ln(y) = a * ln(x) + ln(N)
"""

pc=0.593

F=np.concatenate((np.asarray(F_1),np.asarray(F_2)),axis=0)
p=np.concatenate((np.asarray(p_1),np.asarray(p_2)),axis=0)

plt.figure()
```

```
plt.plot(p,F,'-o',label='Fraction of sites in percolating cluster ('\
                    + str(grid_length) + 'x'+ str(grid_length) + ' lattice)')
plt.xlabel('p')
plt.ylabel('$F_c$')
plt.grid()
plt.legend()


F=np.log(F)
p=np.log(p-pc)

plt.figure()
plt.plot(p,F,'-o',label='Fraction of sites in percolating cluster (' \
                    + str(grid_length) + 'x'+ str(grid_length) + ' lattice)')


#power fit data - obtain estimate for b (beta)
def func_powerlaw(x,b):
    return b*x

target_func = func_powerlaw
popt, pcov = curve_fit(target_func, p, F,maxfev=1000)

plt.plot(p, target_func(F, *popt), '-o',label='Fit of fraction of sites in percolating cluster ('\
         + str(grid_length) + 'x'+ str(grid_length) + ' lattice)')

plt.title('2D percolation on a lattice')
plt.xlabel('p')
plt.ylabel('$F_c$')
plt.grid()
plt.legend()
plt.show()

print popt
```