Homework7

David Mathews

March 19, 2018

1 Coding

A. Percolation Probability

For the implementation of these calculations, I added several variables to the definition of the CLUSTER class. Also manipulated was the CLUSTER::meas function. This implementation only works for the Open Boundary Condition lattice. While it will output results for the Periodic Lattices, they will not be correct.

```
class CLUSTER
{
public:
  CLUSTER(const PARAMS&, const LATTICE&);//constructor
  void grow(const LATTICE&, MTRand&);
  void meas_clear(const LATTICE&);
  void meas(const LATTICE&);
  void binwrite(const PARAMS&, const LATTICE&);
  void print(const LATTICE& latt, int index);
  ~CLUSTER();// destructor
private:
  int size;
  vector <int> conf;
  vector <int> stack;
  double pr;
  int stck_pnt,stck_end;
  double avg_size;
  double prob_perc_x;
  double prob_perc_y;
  double prob_perc;
  ofstream dfout;
};
```

The additional variables in this function all have to do with the various probabilities that will be measured. Each are created as doubles to maintain precision and to be consistent with the average size variable.

For the implementation of the measure function, the following code was created.

```
void CLUSTER::meas(const LATTICE& latt)
{
  avg_size+=(double)size;
  //now determine if it percolated now
  //check bottom side
  bool bottest = false;
  bool toptest = false;
  bool lefttest = false;
  bool righttest = false;
  int Lx = latt.Lx;
  int i = 0;
  while(i<Lx && bottest==false){
    if(conf.at(i)==1)
      bottest=true;
    i++;
  }
  i=0;
  while(i<Lx && toptest==false){
    if(conf.at(Lx*Lx-1-i)==1)
      toptest=true;
    i++;
  }
  i=0;
  while(i<Lx && lefttest==false){
    if(conf.at(Lx*i)==1)
      lefttest = true;
    i++;
  }
  i=0;
  while(i<Lx && righttest==false){
    if(conf.at(Lx*(i+1)-1)==1)
      righttest = true;
    i++;
  }
  if(bottest==true && toptest==true)
    prob_perc_y+=1.0;
  if(lefttest==true && righttest==true)
    prob_perc_x+=1.0;
  if((lefttest==true && righttest==true) || (bottest==true && toptest==true))
    prob_perc+=1.0;
}
```

In this code, I utilized while loops to avoid unnecessary iterations. For this code, I simply iterate across each boundary and check to see if any of the locations along the edge are members of the cluster. If there is at least one member of the cluster on a particular edge, then that edge is counted as being reached. If both opposing edges are reached by the cluster, than the cluster percolates in that direction. If both directions have percolated, then the probability of the cluster percolating over both axis is updated as well.

To debug this method, I generated different clusters that either touched or didn't touch particular boundaries of the lattice. I added a print function to the CLUSTER::meas function and had it print if a particular edge was found or not and verified with the Python visualization scripts we created last lab that the cluster did indeed reach that boundary.

To make sure the cluster measurement information was cleared properly, the CLUSTER::meas_clear function was updated as shown below.

```
void CLUSTER::meas_clear(const LATTICE& latt)
{
  avg_size=0.;
  prob_perc=0.;
  prob_perc_x=0.;
  prob_perc_y=0.;
}
```

To print this new information to a file, the CLUSTER::binwrite function was updated.

```
void CLUSTER::binwrite(const PARAMS& p, const LATTICE& latt)
{
  dfout<<avg_size/((double)p.Nclust)<<" "<<prob_perc_x/((double)p.Nclust);
  dfout<<" "<<prob_perc_y/((double)p.Nclust)<<" "<<prob_perc/((double)p.Nclust)<<endl;
}
```

Error Analysis

```
import numpy as np

#open analysis file and sort into arrays
data=np.loadtxt("data.out")
asizes=data[:,0]
xprobs=data[:,1]
yprobs=data[:,2]
tprobs=data[:,3]

asize=np.mean(asizes)
errasize=np.std(asizes)
xprob=np.mean(xprobs)
errxprob=np.std(xprobs)
yprob=np.mean(yprobs)
erryprob=np.std(yprobs)
tprob=np.mean(tprobs)
errtprob=np.std(tprobs)

print(str(asize)+' : '+str(errasize))
print(str(xprob)+' : '+str(errxprob))
print(str(yprob)+' : '+str(erryprob))
print(str(tprob)+' : '+str(errtprob))
```

As Numpy already has an error formula implementation in the STD function I simply call that function along with the Numpy mean function. These are printed out to terminal.

2 Analysis

Critical Point

In order to approximate the critical point, I wrote a Python script that will call my percolation code multiple times with different probabilities to percolate and different sizes of lattices. This code also had a version of the error analysis code to generate proper error bars on each location. The raw code is shown below. Many of the inclusions and values printed to the user were for my own personal use such as the timing functions and are not necessary.

```
import matplotlib.pyplot as plt
import numpy as np
import os
import sys
import time
#set range of sizes of lattice
startsize = 10
stopsize = 14
numsteps=stopsize+1-startsize
sizes=np.logspace(startsize, stopsize,num=numsteps,base=2)
startprob=0.58
stopprob=0.60
stepprob=0.001
probs=np.arange(startprob,stopprob,stepprob)
nclust=10
nbin=10
seed=10
bound='sqlatt_OBC'

#what column to plot
ycol=3

averages=np.zeros(len(probs))
errors=np.zeros(len(probs))
for j in range(len(sizes)):
  for i in range(len(probs)):

    pfile=open('param.dat','w')
    pfile.write('%d %f %d %d %d %s\n' %(sizes[j],probs[i],nclust,nbin,seed,bound))
    pfile.close()
    start = time.time()
    os.system('./perc_hw > log.log')
    print(str(sizes[j])+' : '+str(probs[i])+' : '+str(time.time()-start))

    data = np.loadtxt('data.out')
    averages[i]=np.mean(data[:,ycol])
    errors[i]=np.std(data[:,ycol])
  plt.errorbar(probs, averages,xerr=0.0,yerr=errors,label='size='+str(sizes[j]))
plt.legend(loc='upper right')
plt.show()
```

In this code, the size of the lattice is always kept as a power of two in a particular range. All of the ranges can be adjusted at the beginning of the code for simplicity. Depending on the test, different configurations were used.

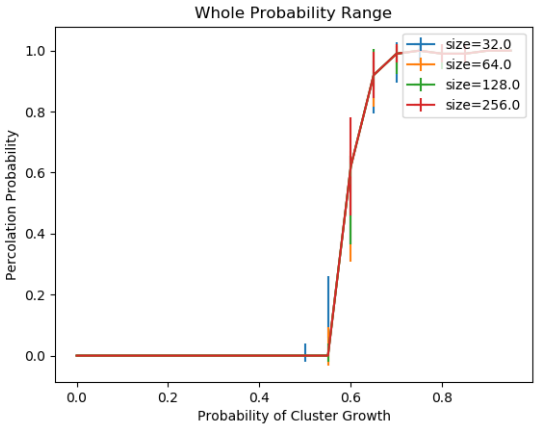

Figure 1: Entire Probability range with different sizes of lattice

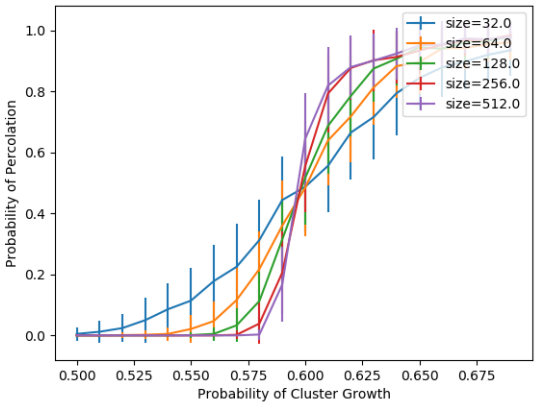

Figure 2: Zoomed in around the critical point

From these two plots I figure the critical point is going to be somewhere around 0.575 to 0.6 range. This is in agreement with the last assignment. It also appears that as the size of the grid is increased the jump from 0 to 1 is becoming more sharp and more step function like. This behavior is as we expected. The error bars seem to grow in size around the critical point. This makes sense as lattices with sizes less than infinity will not always percolate and the test will not return the same result each time whereas further from the critical point it is more likely for the cluster to either percolate or not percolate.

Based on the graph below, I believe the critical point is 0.595 ± 0.005. I choose this range as this is central to the curvature of the transition from 0 to 1 and it appears that the central region is being compressed from either side inwards. I chose the errors of 0.005 as I am not particularly confident in my decision and believe from this last graph that the critical point is between 0.590 and 0.6 but again am not positive.

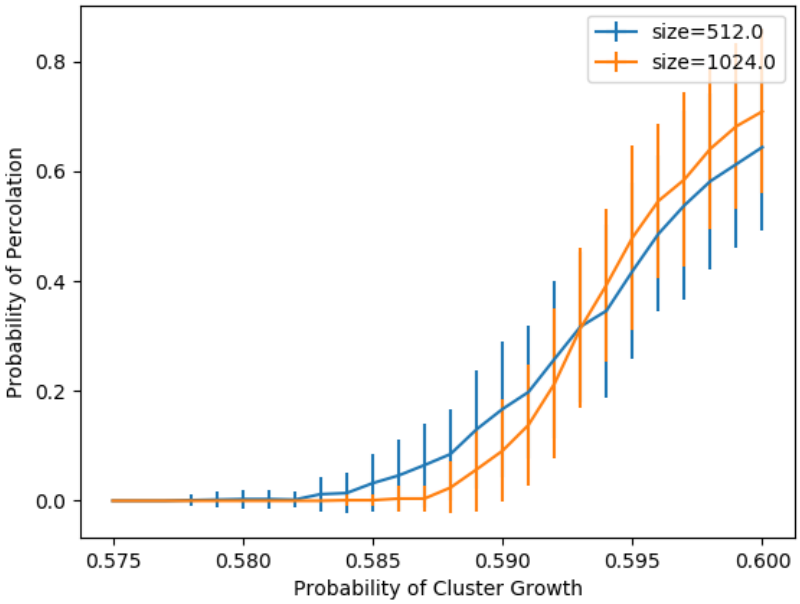

Figure 3: More zoomed in around critical point

Anomalous Dimension

For this part of the assignment, I wrote another code. In this code, 3 different probabilities are used and different size clusters are created for each one. A curve was generated with this information for each probability and this curve was fit on the graph. The code I wrote is shown below.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import os
import time
prs=[0.4,0.595,0.7]
startsize=10
stopsize=200
stepsize=5
sizes=np.arange(startsize,stopsize,stepsize)
nclust=10
nbin=100
seed=10
bound='sqlatt_OBC'
ycol=0

#curve fitting function
def func(x,a,b,c):
  return a*x**(b)+c
averages=np.zeros(len(sizes))
errors=np.zeros(len(sizes))
for i in range(len(prs)):
  for j in range(len(sizes)):
    pfile=open('param.dat','w')
    pfile.write('%d %f %d %d %d %s\n' %(sizes[j],prs[i],nclust,nbin,seed,bound))
    pfile.close()
    start = time.time()
    os.system('./perc_hw > log.log')
    print(str(sizes[j])+' : '+str(prs[i])+' : '+str(time.time()-start))
    data=np.loadtxt('data.out')
    averages[j]=np.mean(data[:,ycol])
    errors[j]=np.std(data[:,ycol])
  print(averages)
  plt.errorbar(sizes, averages, xerr=0.0, yerr=errors,label='pr= '+str(prs[i]))
  #do curve fitting
  param,blah=curve_fit(func,sizes,averages,bounds=(0,[1000,3,1000]))
  print(param)
  plt.plot(sizes,func(sizes,param[0],param[1],param[2]))

plt.legend(loc='upper right')
plt.xscale("log",basex=2)
plt.yscale("log",basey=2)
plt.xlabel('Size of Lattice')
plt.ylabel('Average Size of Cluster')
plt.show()
```

Utilizing this code, I created the following plot.

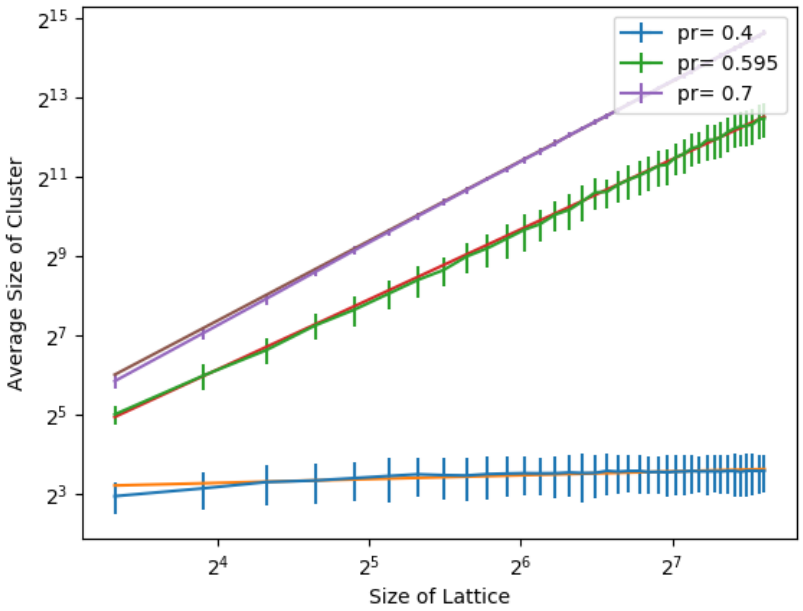

Figure 4: Size of the average cluster for different probabilities and sizes

I fit the output data to a function of the form $a * x^b + c$ where x is the size of the lattice and a, b, and c are optimization parameters. In this situation, we really don't care what a and c are and basically only want b.

Probability	b parameter
0.4	0.096
0.595	1.766
0.7	2.012

As expected, the dimension of the clusters for probabilities less than the percolation threshold is basically 0. For the dimension of clusters above the percolation threshold this is again as expected around 2. Looking at the dimensionality of my guess of 0.595 I find that it is around 1.766. This is a strange fractional number.