A Study in Percolation Models and Algorithms

Victor Daniel Rice
Advisor Dr. Tim Callahan
Embry-Riddle Aeronautical University
Department of Arts and Sciences

April 30, 2014

Abstract

By adapting a previously written percolation model in C, the threshold probabilities for square, triangular, and cubic lattice types were confirmed. An algorithm to count the distribution of cluster sizes at a variety of percolation probabilities was developed, and the expected trends towards the so called infinite cluster was achieved. An equivalent bond percolation model was adapted to the original site algorithm, and by treating occupied bonds as springs, a total compression trend for the model was constructed, which implied that structures under the boundary conditions that were imposed does not have behavior that changes the total compression constant significantly at the percolation threshold.

1 Introduction

Percolation theory is the description of a collective system of clusters in a particular lattice structure. It is typical for percolation models to be discussed in terms of probabilities which represent the likelihood of each particular site or connecting bond in that lattice to be occupied. If sites or bonds are filled in a random order, it is analogous to saying that each bond or site is being filled with a particular probability. In example, if we have 100 sites to be filled, and we fill 30 of these sites in a random order, this is mathematically the same as filling each site with a 30% probability. This has been adapted into the numerical models in this project by using a random number generator to order sites or bonds to be filled through each iteration of the function *permutation*.

To examine the effects of percolation, the tendencies of cluster locations and sizes are typically examined. A cluster is defined as a group of occupied sites or bonds that affectively create a pathway through them without disruption from an unoccupied site or bond. Depending on what lattice structure is in question, there is a particular probability, typically called the percolation or threshold probability, in which there is a rapid increase in the size of the largest cluster in that lattice. This value can be most easily found numerically, and when the probability is below this threshold, that structure is said to be subcritical. Similarly, when the probability is greater than the threshold value, the structure is said to be supercritical. During the supercritical phase the lattice approaches percolation, or rather, in the case of non-continuous boundary conditions, when something can move through one edge of the lattice to the opposing edge through occupied bonds or sites. An example of this for a square lattice site percolation can be seen in Fig. I. (6)

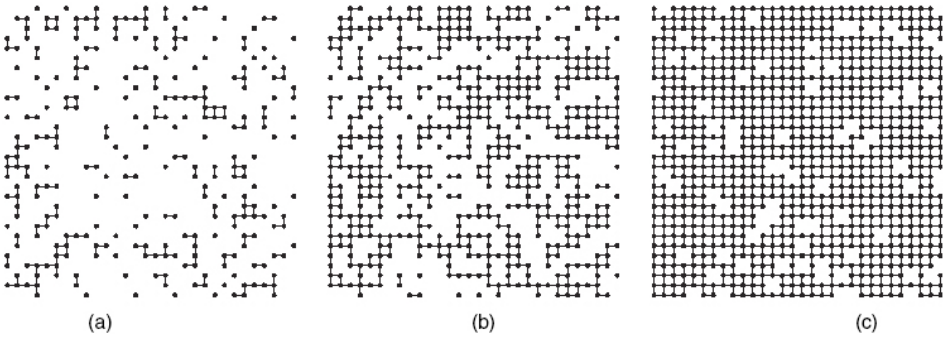

Figure 1: (a) Subcritical Probability (b) Threshold Probability P=.593 (c) Supercritical Probability

For this project I began by using a previously created model (7) to understand not only the fundamentals of programming in C, but the structure of the "Fast Monte Carlo Method". I recreated this model and annotated the code in my own words to describe my understanding of its structure and operation. The next step was to confirm the validity of the code, and for this I chose to run a model of a 150 X 150 square site lattice. The results of this can be seen as Fig. 3. After a full understanding of the methodology of this code was gained, I began making modifications.

The first modification that was made was the transition from the square lattice routine to a routine for a cubic lattice. This required changes to the *boundaries* routine in which the indexes for neighboring sites were defined. Once the percolation threshold values for this structure were confirmed (see Fig. 4) modifications for a two dimensional triangular lattice were made. The triangular modification proved to be an exercise in equivalent geometry analysis as well. Therefore, the structure was treated as a square lattice with diagonal bonds so as to insure the appropriate six nearest neighbors (Fig. 2). This allows us to use the same program for the previous two geometries with a simple modification the *boundaries* routine to have different neighbors.

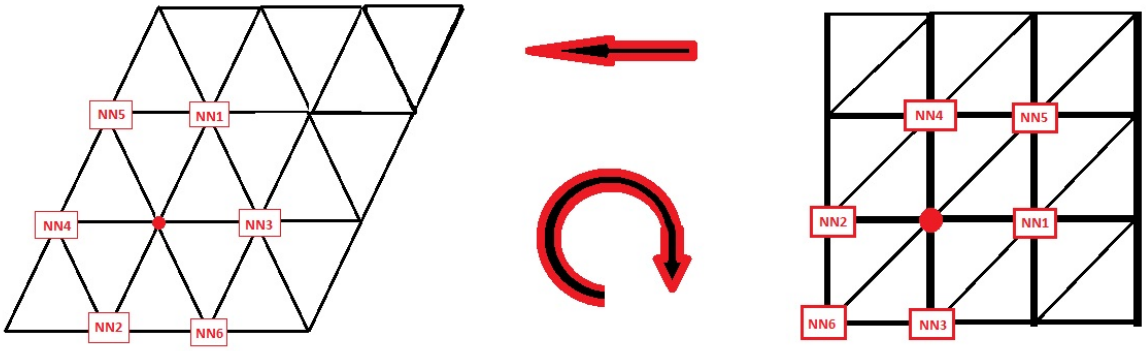

Figure 2: Bend and Rotate Equivalent Geometry for Triangular Lattice with Nearest Neighbors

After I had gained familiarity with a variety of geometries it was decided that it would be of interest to not only examine the size of the largest cluster in the structure as a function of probability, but to examine the distribution of cluster sizes at particular probabilities. This was done by writing a new subroutine that would be called at a particular iteration of *percolate*, which as a fraction of the total number of sites would represent that probability. The routine would then total how many clusters of each size were present at that point. This was done only for the square lattice, and by examining the cluster size distributions before, at, and after the threshold probability the expected trends were examined.

Up until this point, all interest was in the percolation of sites in different lattice structures. The next challenge was to examine an equivalent bond percolation problem and to treat each one of the bonds as a spring of constant value k. This required finding an equivalent geometry that would allow us to use the same *percolate* routine as before, but modifying *percolate* to store k-values of each vertical row of bonds.

A new routine, *ktot*, was nessisary to determine the total compression constant for each iteration.

The equivalent geometry for the bond percolation model requires us to separate the horizontal and vertical bonds from one another due to their differences in dimensionality. Take for example an original square site model of LXL sites. The number of horizontal bonds per row in this case is $L-1$, where are the vertical bonds still contain L entries per row. This means we can treat the system as two equivalent site percolation problems (Fig. 3). It was necessary to decide on what sort of boundary conditions this new model would have. In the previous models we had assumed periodic boundaries conditions for all sides of the structure, but in order to examine a total compression constant, it was necessary that terminal boundary conditions existed at the top and bottom of the vertical bonds for a compression to actually exist. A byproduct of this was that the number of rows of the vertical columns would be $L-1$, while the horizontal still had a total of L rows.

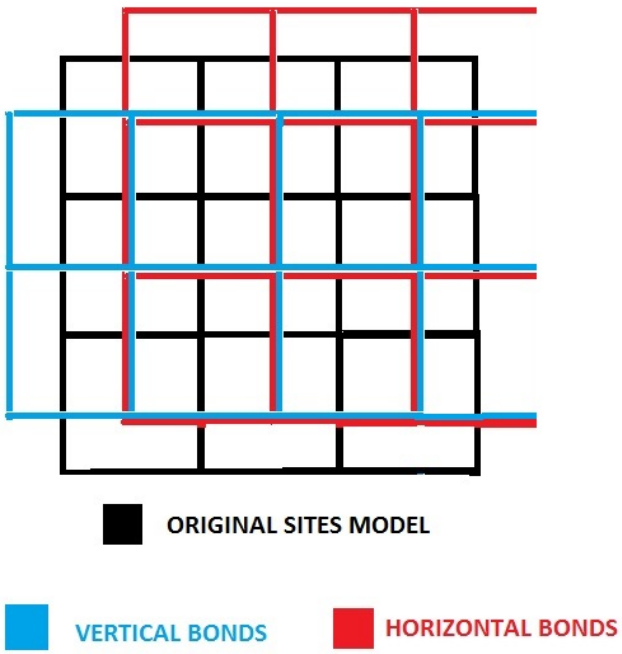

Figure 3: Equivalent Grid System for Site to Bond Transformation

The routine *percolate* was modified to hold a new variable, *krow* which would store the number of occupied bonds in a particular row with each iteration given by the index of *krow*. At the end of each iteration, *percolate* would then call the function *ktot* to determine the total compression spring constant value using the values of *krow* for vertical rows only (Eqn. 1). It was also determined that if a particular vertical row did not contain a single occupancy, then the compression constant would be zero given the instability of the material.

$$\frac{1}{ktot} = \frac{1}{kvalue} * (\frac{1}{krow[1]} + \frac{1}{krow[3]}. + ... \frac{1}{krow[L-1]}) \tag{1}$$

We expected that the total compression value should start at zero, jump to a small value at a particular probability, and then steadily increase until it reaches the original spring constant value k at probability equal to one.

2 Results and Analysis

The initial algorithm begins by defining its dimensionality (L) and the total number of sites ($N = L * L$). Then memory is allocated to the arrays $ptr[N]$, which will hold the cluster sizes and allow the routine to determine if a particular site is a root site, $nn[N]$[number of nearest neighbors] which will tell us what portions of *ptr* are which neighbors, and finally $order[N]$ which is used to randomize the order in which sites (or bonds) will be filled. In the function *boundaries* we define the location of the neighbors as well as invoke periodic boundary conditions to the sides of the lattice. For example, in the case of the square lattice we map the right neighbor of the far right column to the site on the far left of the same row, and for the top neighbor of the top most row, we define the top neighbor to be the entry at the very bottom of that column. These conditions were modified for each of the lattice types that were examined (See Appendix).

In the function *permutation*, through the use of two indexing variables and a placeholder variable, *temp*, *order* can be filled so that *percolate* will have a random order to assign occupied sites into the lattice with. The recursive function *findroot* is defined to be used when the algorithm examines whether an occupied site is a root site or not. *percolate* begins by defining six local integers and another integer *big*, which will hold the size of the largest cluster. Initially, all locations in *ptr* are defined to be empty, which is a global variable defined to be equal to $-(N + 1)$, this is done because negative values inside *ptr* mean that the indexed location is occupied, and the value of that number dictates how many sites are in the cluster whose root is that particular value of *ptr*. Therefore, it would be impossible to have a cluster larger than N. The function then fills the first chosen site with a value of -1 and defines the index of this value to be an integer $s1$. Then another integer $s2$ is defined under an iteration through all the numbers of nearest neighbors and calls *findroot* to tell if any of the neighbors are non-empty. If a particular neighbor is occupied, then the routine finds its root and adds the indexed site and all members of its cluster to the cluster of the neighbor. However, if the current site in question belongs to a cluster of a larger size than its neighbor, then the neighbor is added to the current sites cluster and assumes its root as well. If at any point in this routine a *ptr* value represents a cluster larger than the variable *big*, then it replaces that value. For each iteration in this function the largest cluster is output along with the iteration index.

If we divide each of the iteration indexes by the size of the structure, N, then we have normalized the maximum value to one, therefore each iteration value takes on the value of the associated percolation probability at that point. When we plot the largest cluster size as a function of this probability we can clearly see the point at which the system begins to percolate as a steep increase in the largest cluster size. In Figures 4, 5, and 6 we can see that for the square, cubic, and triangular geometries the standardized percolation threshold probabilities of 0.5927, 0.3116, and $\frac{1}{2}$ respectively, were accurately confirmed by these models.

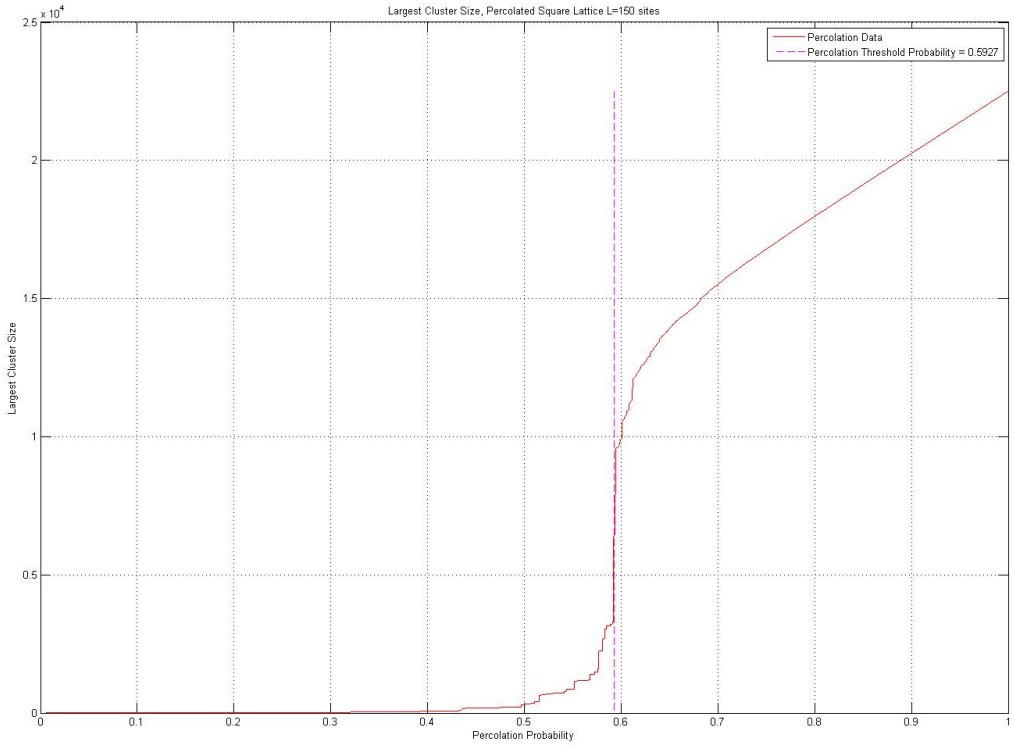

Figure 4: Square Lattice 150 X 150 Largest Cluster Size

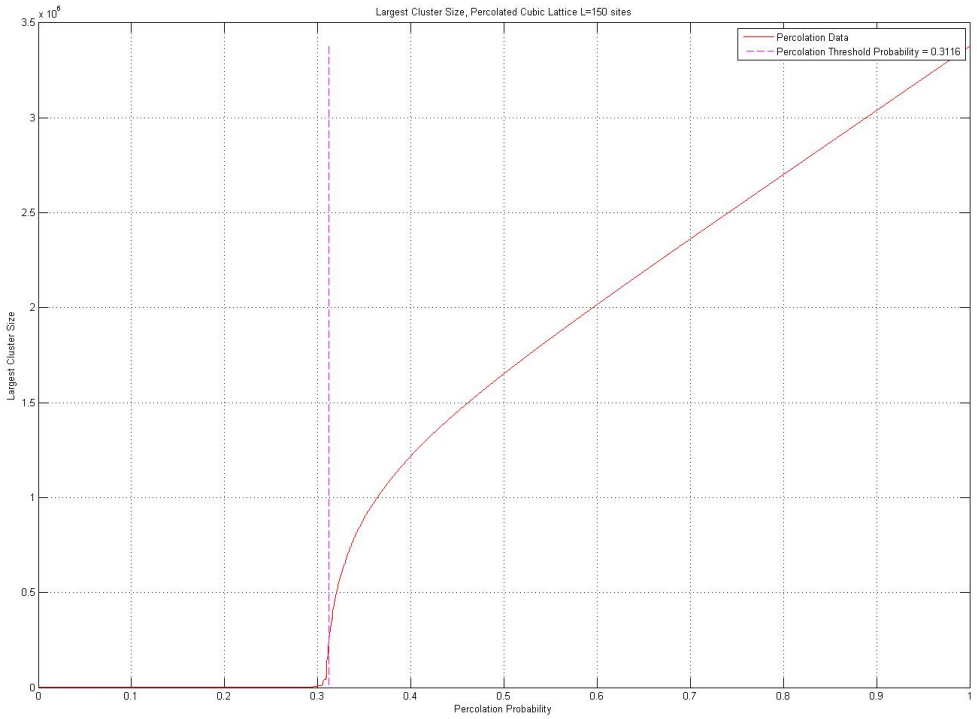

Figure 5: Cubic Lattice 150 X 150 X 150 Largest Cluster Size

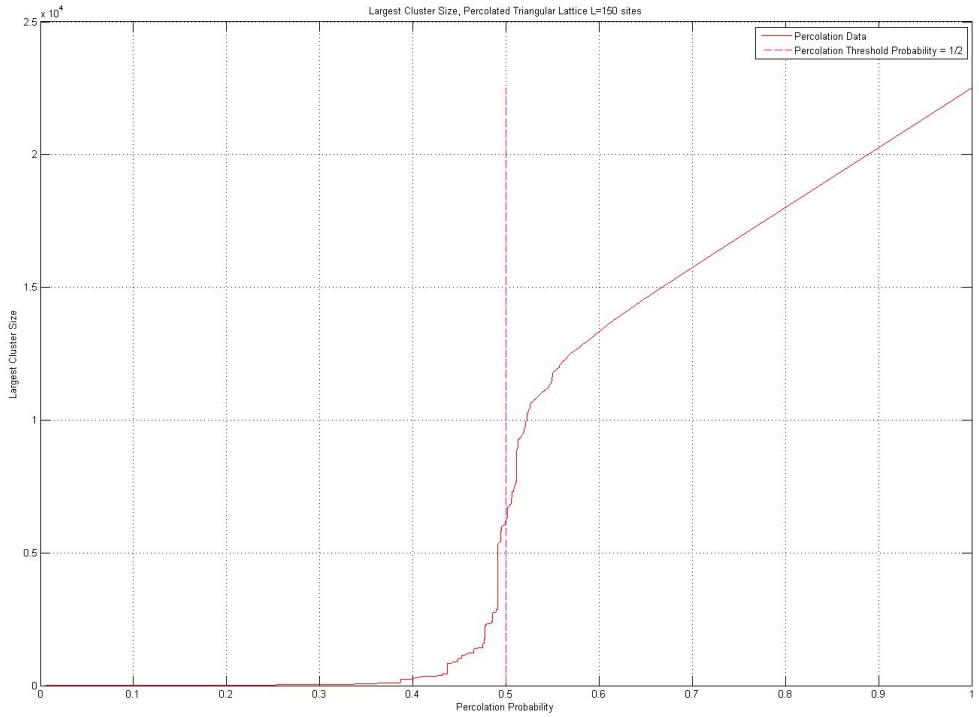

Figure 6: Triangular Lattice L=150 Largest Cluster Size

The routine *clustersizes* placed between *findroot* and *percolate* required the use of two indexing variables. At the beginning of the program another global integer, $cs[N+1]$, was defined to count the cluster sizes. The reason for defining cs to be size $N+1$ was because the index of cs represented the size of the cluster in question, and clusters of size zero are not of interest. The values contained within cs represent the number of clusters of that particular size inside of the lattice which are determined by examining the contents of ptr at a particular probability (iteration) of *percolate*.

By running from index values of 0 to N and seeing if the ptr value of that associated index was negative, we could determine if it was occupied or not, as well as the size of its cluster. If that site was indeed occupied, the routine indexes cs with $-ptr$ and adds 1 to that portion every time the condition is satisfied, effectively counting the number of clusters that size. The function then prints all nonzero values of cs as well as the associated index. Fig. 7 shows histograms of the distributions at probabilities of .50, .55, .60 and .65, and perhaps more obvious trends are shown upon visual examination of the data tables found in section 4.6. As we pass the percolation probability, the distribution of clusters decreases as sites continue to join the largest cluster. Beneath the threshold probability a variety of independent clusters of many different sizes exist.

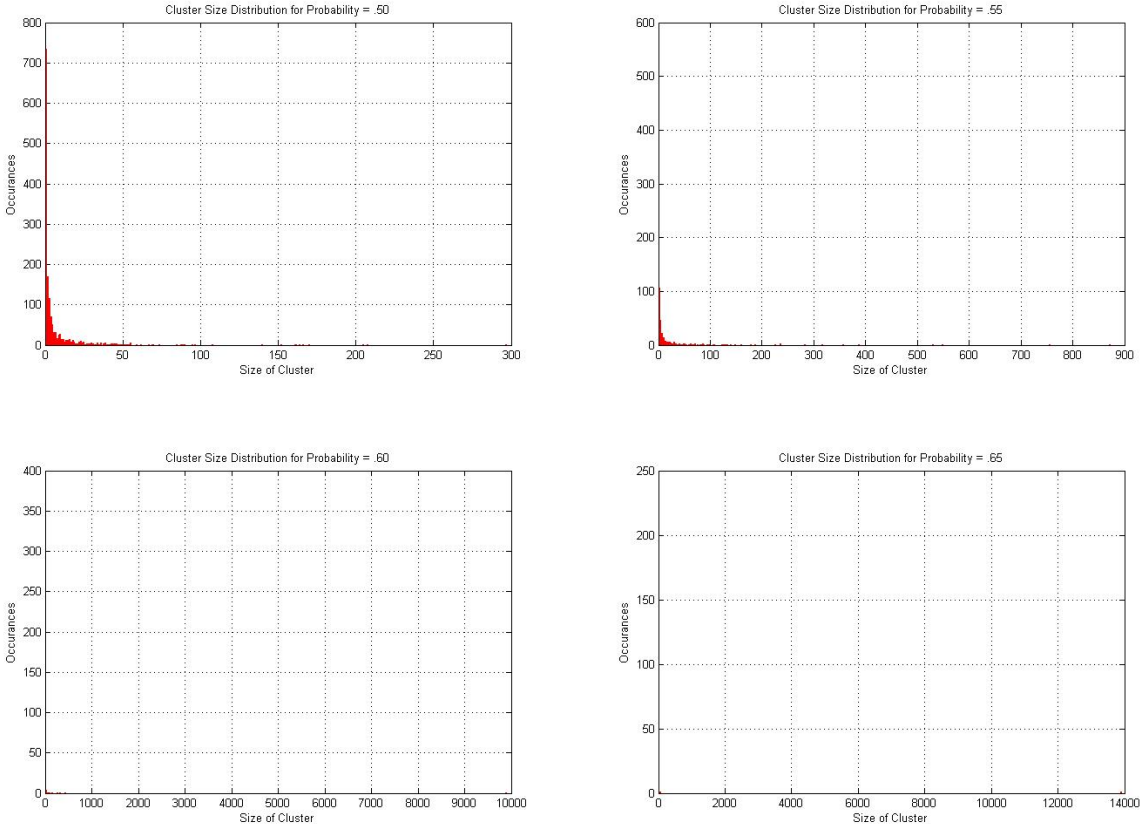

Figure 7: Cluster Size Distribution

At the percolation threshold probability for the square lattice (Fig. 8), the largest cluster occupies 28.4% of the structure. If we do not include clusters of size 1 in the total occupancy of the structure, the new corresponding percolation probability of .5759, gives a difference of 0.0168 in the probability, or rather, a 2.8% change in threshold probability.

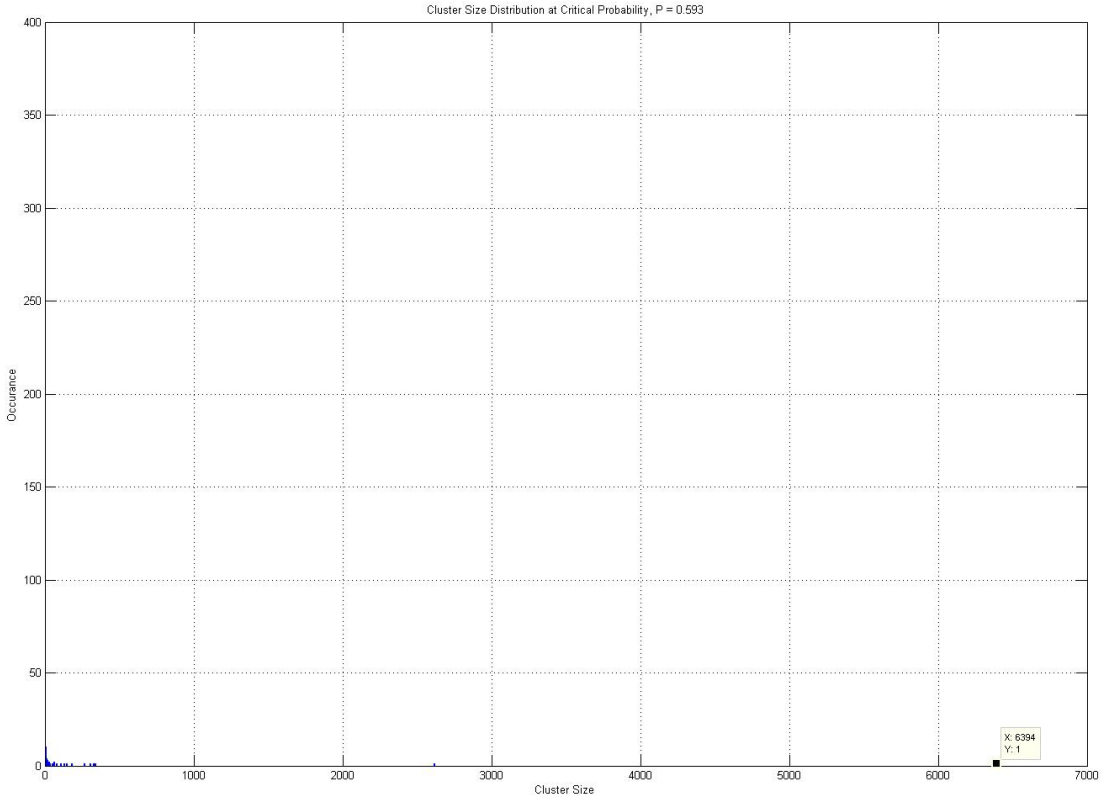

Figure 8: Cluster Size Distribution at Critical Probability

For simplicities sake, when modifying the algorithm to examine compression constant values, *percolate* was modified to once again fill *ptr* with values as determined by *order*, but then a local integer *row* gave us the row index for that particular location determined by *order*. If the row was a vertical row, 1 was added to that particular row location in *krow*. At this point *ktot* was called. *ktot* ran through all values of *krow* and for those belonging to vertical bonds, determined the value of *krow* through float division and added this value to a running total of the other nonzero rows. The result of this routine was one over the total compression constant and therefore by printing one over that value, the total compression constant of the system for a spring value, $k = 1$, was output for each probability. We can see that the behavior that we predicted is indeed realized in this model. It was also determined that compression, under the assumptions we had made about discontinuities in the structure, begins only at probabilities greater than 0.01313 (See Figure 10). We can also examine that there is no significant correspondence between the trend of *ktot* and the percolation threshold probability.

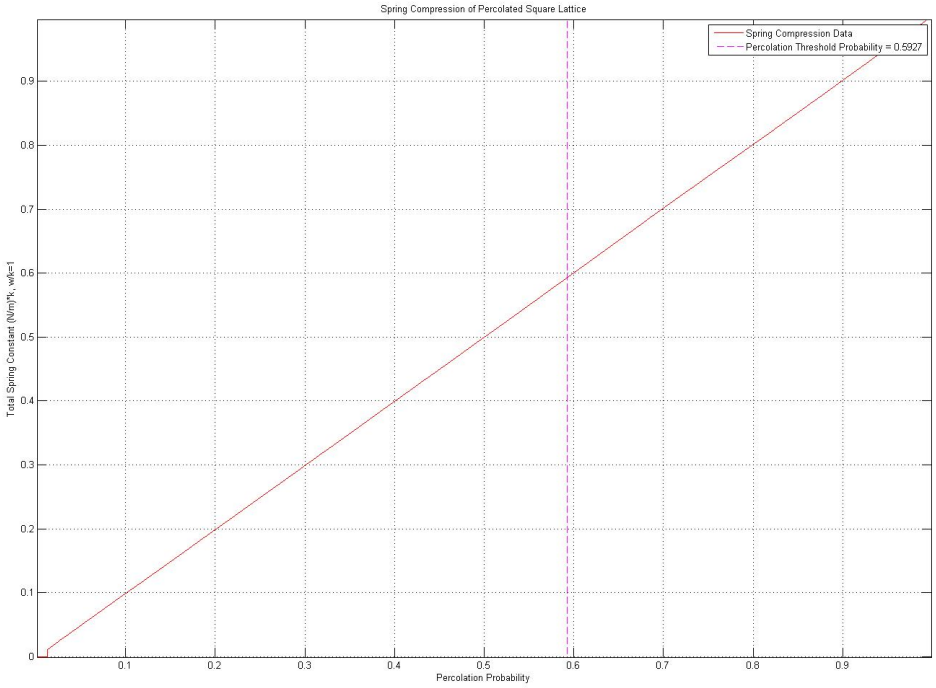

Figure 9: Compression Data

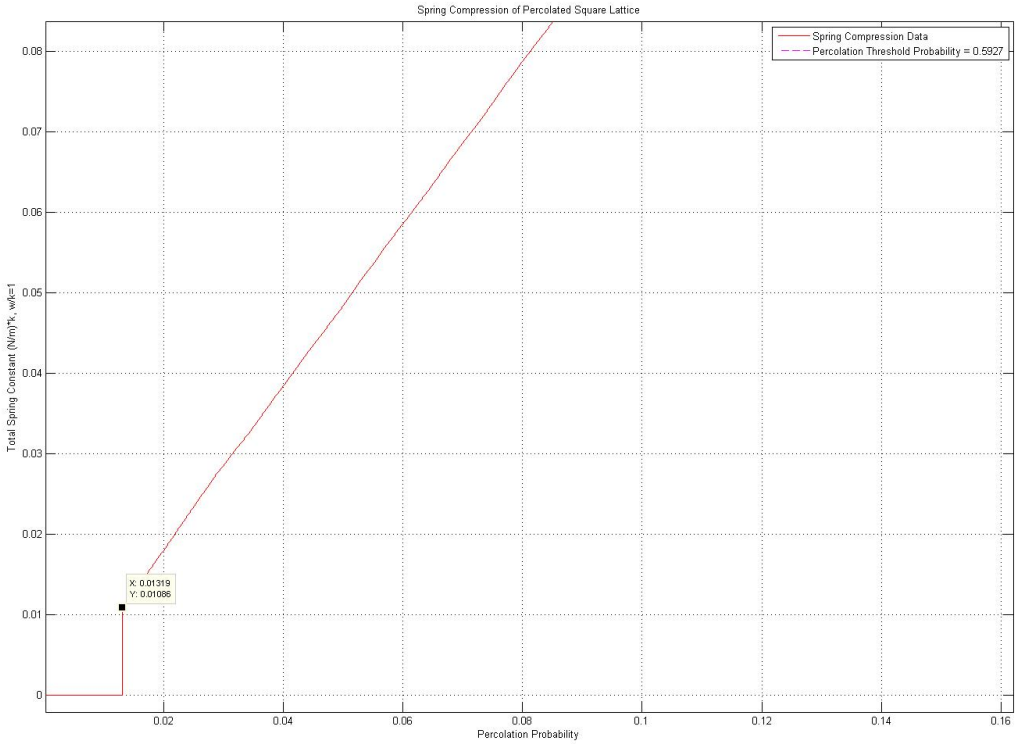

Figure 10: Initial Compression Threshold

3 Conclusion

With this project I was able to modify the Santa Fe algorithm for different geometries in 2 and 3 dimensions, count distributions of cluster sizes at a variety of probabilities, and examine a total compression constant trend for an equivalent percolating bond system of springs. However, there are still a variety of values which would have been informational to examine, but in the interest of time, I was unable to.

The *boundaries* routine was modified in the compression model to allow the original version of percolate to be implemented. *boundaries* was divided into two separate parts, one for the vertical bonds, and one for the horizontal bonds. Each had their own set of boundary conditions, including terminal constraints on the top and bottom of the lattice. Terminal constraints are achieved by a conditional statement which sets the absent neighbors location to *empty*, so that in *percolate* these values will never be added to a cluster. A graphical representation of this new *boundaries* function can be seen as Fig. 11. The purpose of this would have been to once again count the largest cluster size, and with some more in depth changes, allow us to isolate vertical springs belonging to that cluster. With this, a comparison could be made regarding how much of the total compression belongs to the largest cluster at particular probabilities.

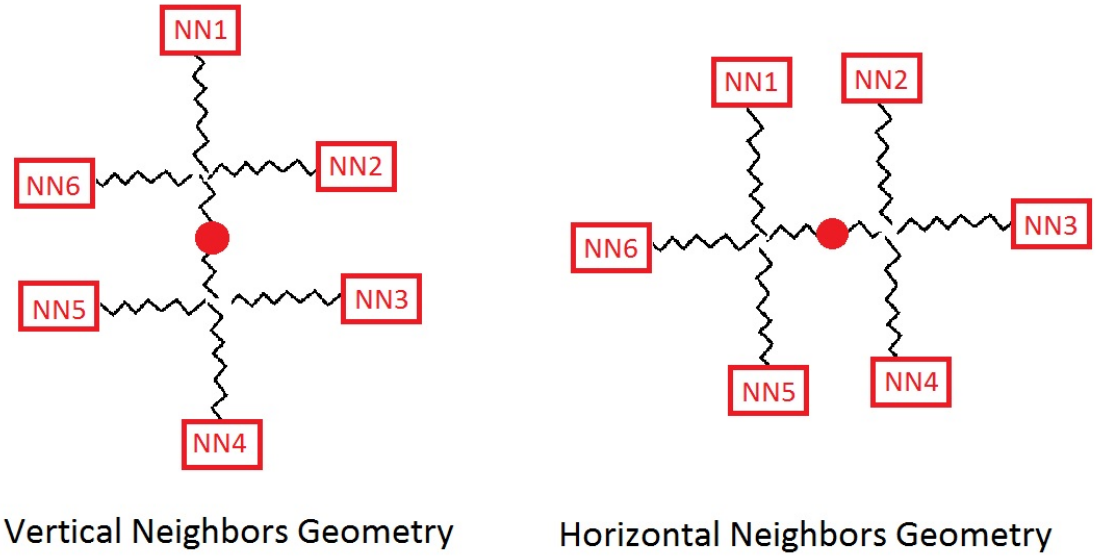

Figure 11: Vertical and Horizontal Boundaries Geometry

Another modification that I would have liked to make would have been to implement a similar counting routine as *ktot* to determine values of horizontal compression. This would allow us to test say, the compression in 2 dimensions at a variety of probabilities under accelerations of the lattice at particular angular orientations.

Finally, the last concepts I would have liked to examine would have been the resistor network problem. Given modified routines of the original *percolate* and our new *boundaries*, by treating occupied bonds as resistors, one could theoretically calculate a total network resistance at a variety of probabilities and examine once again the contribution of the largest cluster to this value.

4 Appendix

4.1 Square Lattice Model

```
/* The following script was written by M.E. Newman and R.M. Ziff of the Santa Fe Institute for "A Fast Monte Carlo Algorithm for Site or Bond Percolation" for a simple
square lattice */

#include<stdio.h>
#include<stdlib.h>

#define L 150  /*number of entries on side*/
#define N (L*L)  /*Number of Sites available for occupacy/*
#define EMPTY -22501  /*Empty will be a negative number greater than dimension of lattice by 1 */


int ptr[N];  /*Array of pointers*/
int nn[N][4];    /* To help us take a look at the neighbors*/
int order[N];  /*This list will allow us to randomize the order that sites are filled*/



/* The following defines boundaries condition which I intend to change pending this all runs correctly and i understand it's methodology fully */

void boundaries()
{
        int i;

        for (i=0; i<N; i++) {
                nn[i][0] = (i+1)%N;  /*first portion of entry is neighbor the the right, mod ( dimension of lattice)*/
                nn[i][1] = (i+N-1)%N;  /*second portion of entry in nn is the neighbor to the left */
                nn[i][2] = (i+L)%N;   /* third portion of entry in nn is the neighbor below */
                nn[i][3] = (i+N-L)%N;  /*fourth portion of entry in nn is the neighbor above */

        if (i%L == 0) nn[i][1] =i+L-1;    /*maps left side to right side*/
        if((i+1)%L == 0) nn[i][0] = i-L+1;   /*Maps right side to left side */
        }
}


/* We must generate a random order for which the sites will be occupied */
void permutation ()
{
int i, j;   /* initializes our position locator*/
int temp;  /*temporary integer for storage*/

for (i=0; i<N; i++) order[i]=i;
        for (i=0; i<N; i++)
        {
        j=i+(N-i)*1.0*rand()/RAND_MAX;  /*assigns value of j to a location a random number of steps through the list from i */
        temp=order[i];    /* holds value of original position of "i" value */
        order[i]=order[j];  /* swaps value of order between original position "i" and new position "j" */
        order[j]=temp;     /*  then assign the order new location to the order of old location ( nessisary for occupational of all sites) */
        }

}


/* We are going to need to have a method of finding the root of a cluster...*/
/* a Recussive method is chosen to do this */


int findroot(int i)
{
if (ptr[i]<0) return i;   /*negative values represent roots in this model */
return ptr[i]=findroot (ptr[i]);
}

/* */

void percolate ()
{
int i, j ;
int s1,s2;
int r1, r2;
int big=0;

for (i=0; i<N; i++) ptr[i]=-22501;
for (i=0; i<N; i++)
{
```

```
76          r1=s1=order[i];    /* r1 and s1 are defined to start at the position of 1st ordered location
                */
77          ptr[s1]= -1;  /*fills the 1st chosen site (recall that values of roots are negative)*/
78          for (j=0; j<4; j++) {
79                  s2= nn[s1][j];      /*Calls location of s2 to be the nearest neighbor or s1*/
80                  if (ptr[s2] != -22501) {
81                  r2=findroot(s2);  /* Assuming that the value of site located at s2 is occupied,
                        determine it's root*/
82                      if (r2 != r1)
83                      {
84                              if (ptr[r1]>ptr[r2]) {
85                              ptr[r2] += ptr[r1];   /*If ptr r1 contains less elements than ptr r2,
                                    then add the contents of ptr 1 to ptr 2 */
86                              ptr[r1]=r2;    /* ptr r1 then becomes value of its root, r2  */
87                              r1= r2;
88                              }
89                              else {
90                              ptr[r1] += ptr[r2];  /*But if ptr r1 contains more elements that ptr
                                    2, the complete the opposite action */
91                              ptr[r2]=r1;
92                              }
93                                      if (-ptr[r1]>big) big = -ptr[r1];  /* This keeps track of the
                                            largest clusters size by monitoring the contents of ptr
                                            */
94                                      }
95                      }
96                  }
97                  printf("%6i %6i\n",i+1, big);  /* which iteration is which and how large is the
                        biggest cluster*/
98          }
99  }
100
101
102 /* To run the program now we must just initilize the following functions and end main */
103
104 int main ()
105 {
106
107 boundaries ();
108 permutation ();
109 percolate ();
110
111 }
```

4.2 Cubic Modifications to Nearest Neighbors

```
1
2   /* The following script was written by M.E. Newman and R.M. Ziff of the Santa Fe Institute for "A
        Fast Monte Carlo Algorithm for Site or Bond Percolation */
3   /* Modified to 3-Dimensions by Victor Rice of Embry-Riddle Aeronautical University on 10/22/13*/
4
5
6   #include<stdio.h>
7   #include<stdlib.h>
8
9   #define L 150 /*number of entries on side*/
10  #define N (L*L*L)  /*Number of Sites available for occupacy/*
11  #define EMPTY -3375001  /*Empty will be a negative number greater than dimension of lattice by 1 */
12
13
14  int ptr[N];  /*Array of pointers*/
15  int nn[N][6];    /* To help us take a look at the neighbors*/
16  int order[N];  /*This list will allow us to randomize the order that sites are filled*/
17
18
19
20  /* The following defines boundaries condition which I intend to change pending this all runs
        correctly and i understand it's methodology fully */
21
22  void boundaries()
23  {
24          int i;
25
26          for (i=0; i<N; i++) {
27                  nn[i][0] = (i+1)%N;  /*first portion of entry is neighbor the the right, mod (
                        dimension of lattice)*/
28                  nn[i][1] = (i+N-1)%N;  /*second portion of entry in nn is the neighbor to the left */
29                  nn[i][2] = (i+L)%N;   /* third portion of entry in nn is the neighbor below */
30                  nn[i][3] = (i+N-L)%N;  /*fourth portion of entry in nn is the neighbor above */
31                  nn[i][4]= (i+L*L)%N ;  /*Checks portion "behind" site */
32                  nn[i][5]= (i-L*L)%N;   /* Checks portion "in front" of site  */
33
34  /*It is going to be nessisary to change this mapping conditions as well, must map front to back and
        back to front */
35
```

```
        if (i%L == 0) nn[i][1] =i+L-1;     /*maps left side to right side*/
        if((i+1)%L == 0) nn[i][0] = i-L+1;   /*Maps right side to left side */
        if((i+L*L) > N) nn[i][4]=i-(N-L*L);   /* Maps back side to front side */
        if((i-L*L) < 0) nn[i][5]=i+(N-L*L);   /* Maps front side to back side */
        }
}

.
.
.
```

4.3 Triangular Modificaitons to Nearest Neighbors

```
.
.
.


int ptr[N];  /*Array of pointers*/
int nn[N][6];    /* To help us take a look at the neighbors*/
int order[N];  /*This list will allow us to randomize the order that sites are filled*/



/* The following defines boundaries condition which I intend to change pending this all runs
     correctly and i understand it's methodology fully */

void boundaries()
{
        int i;

        for (i=0; i<N; i++) {
                nn[i][0] = (i+1)%N;  /*first portion of entry is neighbor the the right, mod (
                    dimension of lattice)*/
                nn[i][1] = (i+N-1)%N;  /*second portion of entry in nn is the neighbor to the left */
                nn[i][2] = (i+L)%N;   /* third portion of entry in nn is the neighbor below */
                nn[i][3] = (i+N-L)%N;  /*fourth portion of entry in nn is the neighbor above */
                nn[i][4] = (i+N-L+1)%N; /*fifth portion of entry in nn is the right top  diagonal*/
                nn[i][5] = (i+L-1)%N;    /*sixth portion of entry in nn is the left bottom diagonal
                    */
        if (i%L == 0) nn[i][1] =i+L-1;     /*maps left side to right side*/
        if ((i+1)%L == 0) nn[i][0] = i-L+1;   /*Maps right side to left side */
        if (i%L == 0) nn[i][5]=(i+2*L-1)%N;       /*maps bottom left corner to right side of row
            below insure no out of bounds index */
        if ((i+1)%L ==0) nn[i][4]=(i+N-2*L-1)%N;     /*maps top right corner to left side of row
            above, insures no negative index */
        }
}



.
.
.
```

4.4 Square Routine with Cluster Sizes Algorithm

```
.
.
.

void clustersizes ()
{ int m,i;

     for (m=0; m<=N; m++) {
           cs[m]=0;               /*Clears memory */
                       }

          for (i=0; i<N; i++){
               if (ptr[i] < 0) {
                     if (ptr[i] != -22501) {                    /*if site is occupied  */

                      cs[-ptr[i]]++;              /*index of cs is size of the cluster which i
                          belongs to and places it at index of value of the same size */
                                        }
                                 }
                              }

/* Let's output some values around this arena */
/* Only nonempty values concern us*/
for (i=1; i<=22500; i++){
     if (cs[i] != 0){
```

```
 printf("%6i %6i\n", i, cs[i]);
                      }
                           }
}

.
.
.
```

4.5 Simple Compression Routine

```
#include<stdio.h>
#include<stdlib.h>

#define L 500     /* Pick a Dimension of the lattice, # of sites per edge of the square */
#define N (L*L)  /* Total number of sites*/
#define empty (-(2*N-L)-1)    /*1 outside of total number of available bonds to store empty ptr values */

int order[2*N-L];      /*This vector used to percolate the system */
int ptr[2*N-L];    /*This will let us know if bonds are empty or occupied */
int nn[2*N-L][6];  /*The nearest neighbor vector for horizontal (index 1=1) and for the vertical ( index 1=2)*/
int krow[2*L-1];   /*How many occupied bonds in each row stored in this vecotr */

void boundaries()
{
int i;
int n = 1;
                  for (i=0; i<2*N-L; i++) {

                      if (n%2 != 0){     /* if this index is odd it belongs to horizontal routine */

                                    nn[i][0] = i-L;   /*Top left  neighbor */
                                    nn[i][1] = i-L+1;   /*Top right neighbor */
                                    nn[i][2] = i+1;    /* To the Right neighbor*/
                                    nn[i][3] = i+L+1;   /*Bottom Right Neighbor */
                                    nn[i][4] = i+L;    /*Bottom left neighbor */
                                    nn[i][5] = i-1;  /*To the Left neighbor*/

         /*Continious Boundary Conditions only exist for horizontal entries, so we must void top and bottom neighbors of the lattice*/

                                   if(i<=L-1)  nn[i][0] = empty , nn[i][1] = empty;
                                   if(i>=2*N-2*L) nn[i][3] = empty, nn[i][4] = empty;

          /*Still have to map the horizontal pieces though...*/

                                  if((i+1)%L==0) nn[i][2]= i-L+1;   /*Map right to left*/
                                  if(i%L==0) nn[i][5] = i+L-1;  /*Map left to right*/
                                   }
                     else {    /* here is where we will place the vertial routine */

                                    nn[i][0] = i-2*L;    /*Top center  neighbor*/
                                    nn[i][1] = i-L;     /*Top Right neighbor*/
                                    nn[i][2] = i+L;      /*Bottom Right Neighbor*/
                                    nn[i][3] = i+2*L;   /*Bottom Center Neighbor*/
                                    nn[i][4] =i+L-1;   /*Bottom Left Neighbor*/
                                    nn[i][5] = i-L-1;   /*Top Left neighbor*/

         /*Must void top and bottom for the vertical routine as well */

                                    if(i<=2*L-1) nn[i][0] = empty;
                                    if(i>=2*N-3*L) nn[i][3] = empty;

        /*Still have to map the horizontal pieces though */

                                   if(i%L==0)   nn[i][4]=i+2*L-1 , nn[i][5]= i-1;
                       }

 if(i==n*L) n++;   /*When we finish running routine through a row, then change rows */
                                        }

}

/* We must generate a random order for which the sites will be occupied (basically in order for us to percolate the system) */


void permutation ()
{
int i, j;   /* initializes our position locator*/
int temp;   /*temporary integer for storage*/

for (i=0; i<2*N-L; i++) order[i]=i;
```

```
71                  for (i=0; i<2*N-L; i++)
72                  {
73                  j=i+(2*N-L-i)*1.0*rand()/RAND_MAX;/*Assigns value of j to a location a random number
                         of steps through the list from i */
74                  temp=order[i];           /* Holds value of original position of "i" value */
75                  order[i]=order[j];   /* Swaps value of order between original position "i" and new
                        position "j" */
76                  order[j]=temp;   /* Then assign the order new location to the order of old location
                        */
77
78                  }
79  }
80
81
82  int findroot(int i)
83  {
84   if (ptr[i] < 0) return i;
85  return ptr[i]=findroot(ptr[i]);
86  }
87
88
89  float  ktot ()
90  {
91
92  float tot = 0;
93  int n;
94
95  for(n=0;n<2*L-1;n++){
96        if(n%2!=0){                      /*Only interested in the odd index rows*/
97          if (krow[n]==0) return 0;    /*If there is a break in the lattice structure, then no
                compression exists*/
98          else {
99              tot=tot+1./krow[n];      /*Keep a running total of 1/ktotal for the structure*/
100                 }
101               }
102                   }
103 return (1/tot);                   /*Return the total final spring constant value*/
104 }
105
106
107 void percolate ()
108 {
109 int j;
110 int k;
111 int i;
112 int row;
113
114 for (k=0; k<2*N-L; k++) ptr[k]=empty;   /*All bonds begin as empty*/
115
116       for (i=0; i<2*N-L; i++) {
117                   ptr[order[i]] = -1;          /*Fills the bonds in a random order (recall that values
                        of roots are negative)*/
118                   row = (order[i]+L)/L;      /*Domain of [1:2*L-1]*/
119
120                if (row%2!=0) krow[row]++;      /*It must belong to vertical if row is even*/
121                    printf("%6f\n",ktot());  /*For each iteration of "i" find the total value of the
                         compression constant*/
122                             }
123 }
124
125
126 int main()
127 {
128 boundaries ();
129 ktot();
130 permutation ();
131 percolate();
132 }
```

4.6 Data For Cluster Size Distributions

[H] Size at P=.50	Occurrence at P=.50	Size at P=.55	Occurrence at P=.55	Size at P=.60	Occurrence at P=.60	Size at P=.65	Occurrence at P=.65
1	734	1	518	1	356	1	219
2	169	2	107	2	73	2	32
3	115	3	64	3	29	3	13
4	70	4	46	4	26	4	11
5	50	5	28	5	19	5	8
6	32	6	20	6	10	6	2
7	32	7	22	7	3	7	1
8	17	8	12	8	6	8	1
9	25	9	6	9	5	9	2
10	27	10	9	10	5	10	1
11	14	11	13	11	8	11	2
12	14	12	6	12	1	13	1
13	7	13	5	13	4	16	1
14	11	14	7	14	3	17	2
15	12	15	7	15	2	20	1
16	14	16	3	16	1	23	1
17	8	17	4	17	1	24	1
18	12	18	5	18	2	32	1
19	7	19	6	19	4	69	1
20	3	20	5	21	2	13912	1
21	3	21	1	22	1		
22	7	22	1	23	1		
23	10	23	5	24	2		
24	5	24	4	25	1		
25	7	25	4	28	1		
26	2	26	2	29	1		
27	3	27	3	30	1		
28	3	28	3	31	2		
29	3	29	1	32	1		
30	6	30	3	33	1		
31	3	31	5	35	1		
32	1	33	2	38	1		
33	2	34	2	58	1		
34	5	35	2	62	1		
35	2	37	1	65	1		
36	5	40	1	66	1		
38	3	41	1	67	1		
39	5	42	3	69	1		
40	1	43	1	81	1		
41	2	46	1	111	1		
42	2	48	1	152	1		
43	3	49	3	267	1		
44	3	50	1	332	1		
45	3	52	2	437	1		
46	3	53	1	9906	1		
47	1	54	1				
48	1	60	1				
49	2	61	1				
50	1	62	1				
51	1	63	2				
52	1	64	1				
53	1	66	1				
54	1	69	1				
55	5	72	2				
59	1	76	1				
62	1	82	1				
67	1	84	1				
69	1	87	2				
70	1	88	1				
74	2	89	1				
85	2	95	1				
88	1	98	1				
89	1	106	1				
90	2	108	1				
95	1	124	1				
97	1	127	1				
108	1	131	1				
140	1	133	1				
152	1	134	1				
161	1	141	1				
162	1	149	1				
164	1	160	1				
166	1	179	1				
170	1	188	1				
205	1	226	1				
208	1	237	2				
297	1	284	1				
		317	1				
		358	1				
		387	1				
		530	1				
		550	1				
		756	1				
		873	1				

5 Citations

(1) **Meeser, Ronald ; Roy, Rahual** Continuum Percolation *Cambridge University Press,* 1996

(2) **Sahimi,Muhammad** Applications of Percolation Theory, *Taylor and Francis,* 1994

(3) **Staugger, Dietrich ; Aharony, Amnon** Introduction to Percolation Theory *Taylor and Francis,* 1994

(4) **Wang, J; Z. Zhou; W. Zhang; T. Garoni; Y. Deng** (2013). Bond and site percolation in three dimensions. *arXiv:1302.0421. Bibcode:2013PhRvE..87e2107W. doi:10.1103/PhysRevE.87.052107.*

(5) **Sykes, M. F.; J. W. Essam** (1964). "Exact critical percolation probabilities for site and bond problems in two dimensions". *Journal of Mathematical Physics 5 (8): 1117–1127. Bibcode:1964JMP.....5.1117S. doi:10.1063/1.1704215.*

(6) **Brown, Daniel; Elliot, Matthew ; Et. Al.** "phase transitions of computational power in the resource states for one-way quantum computation". *http://iopscience.iop.org/1367-2630/10/2/023010/fulltext/* 12 February 2008

(7) **Newman, M.E.J. ; Ziff, R.M.** "A Fast Monte Carlo Algorithm for Site or Bond Percolation" *http://www.santafe.edu/media/workingpapers/01-02-010.pdf* Santa Fe Institute, 2001