Homework 3

Maedeh Karkhane Yousefi

October 14, 2021

1.Ballistic Deposition With Relaxation(3.2)

The goal of this exercise was to make the layer in each time step, as flat as possible.

In order to do so, one way is that the particle check it's two closest neighbors and find the minimum height that it can have after being deposited. So, we actually give the particle a chance to change it's final position for the best.

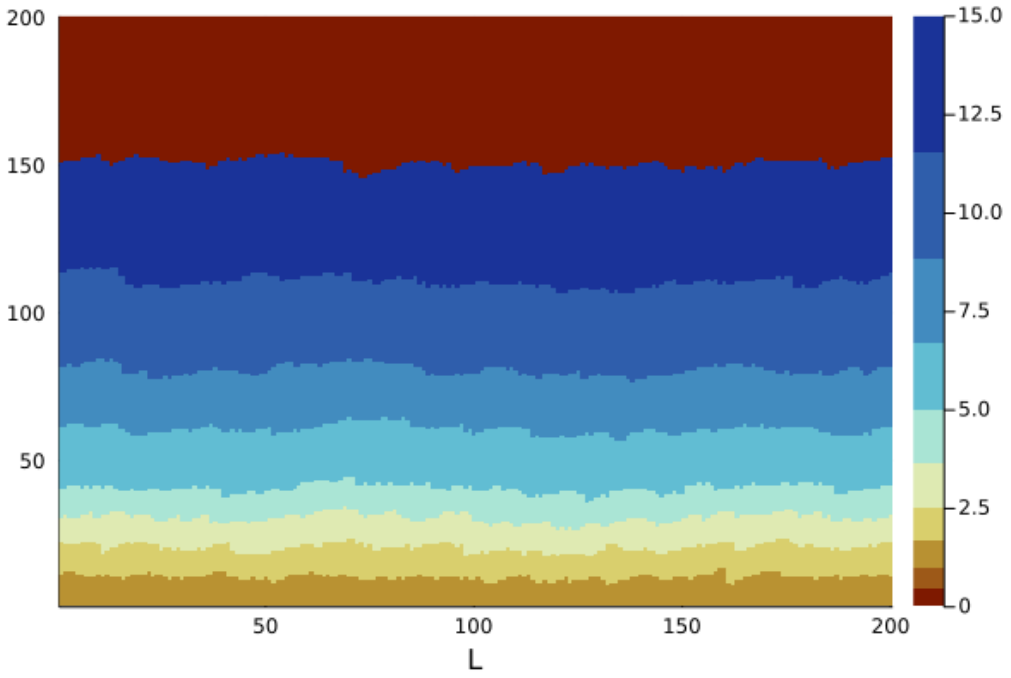
I considered a *200×200 Matrix* as the whole system. Each particle have an initial random column and can check it's side columns' heights to decide on it's final to fall into. Afterwards, starting from the bottom, checks each entry for an empty place, respectively. empty and full are differentiated by 0 and other numbers. If the position is empty, the deposition happens and we can move on to the next particle; if not, the particle loops through the array, further.

I've got 5 functions for the whole operation. The boundary conditions are controlled by returning the first column instead of *column=L+1* and the last column instead of *column=0*, whenever there is a invasion of limits. the *height calculator function* loops through rows for the given column, presuming each filled entry as a height increment. Finally, in the *column choosing function* we have an Ordered Dictionary, which stores these three columns and their heights, respectively, and returns the minimum height.

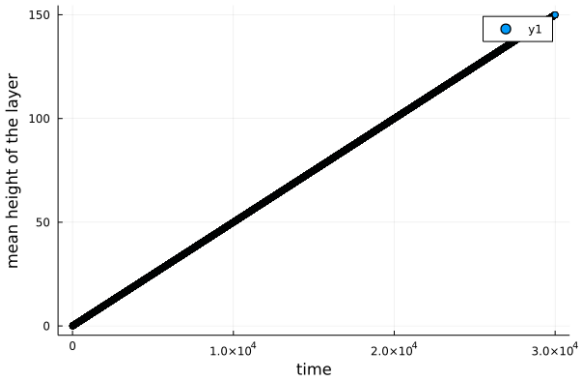
In order to plot the mean height of a layer in each time step, I introduced a *mean calculator function*, to get the height of each column, in each step, and by using the *mean() function*, return the mean number of each layer.

Like what I've written in my previous report, for this part, we can no longer assume time to be continuous. We shall assume that the time intervals' increment are exponential and therefore, the amount of particles falling between each time interval is increasing exponentially, too. repeating the same procedure, but including this consideration into account, leads us to call the function I've written for calculating the The Standard Deviation at the end of the loop. The *STD function* works like the *mean_ calculater function* except that it returns a list of *std*s.

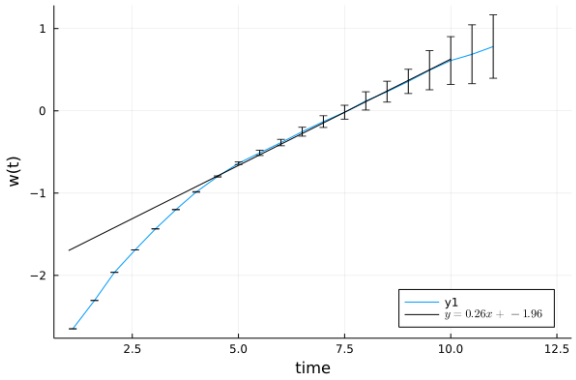
In order to show the the deviation I have set a *run_num*, by which the code loops through the whole procedure and it creates a list of multiple lists of *std*s. Getting the mean and the std of the same number of elements in every list, we are going to have related std list and mean list to plot the error bar, too.



(a) The dynamics is plotted for n=30000 particles in a *200×200 Matrix*.



(b) The Mean Height of a Layer over Time. Time range is considered to be equal to the whole number of particles(rate=1). n=30000, L=200, slope=0.005.



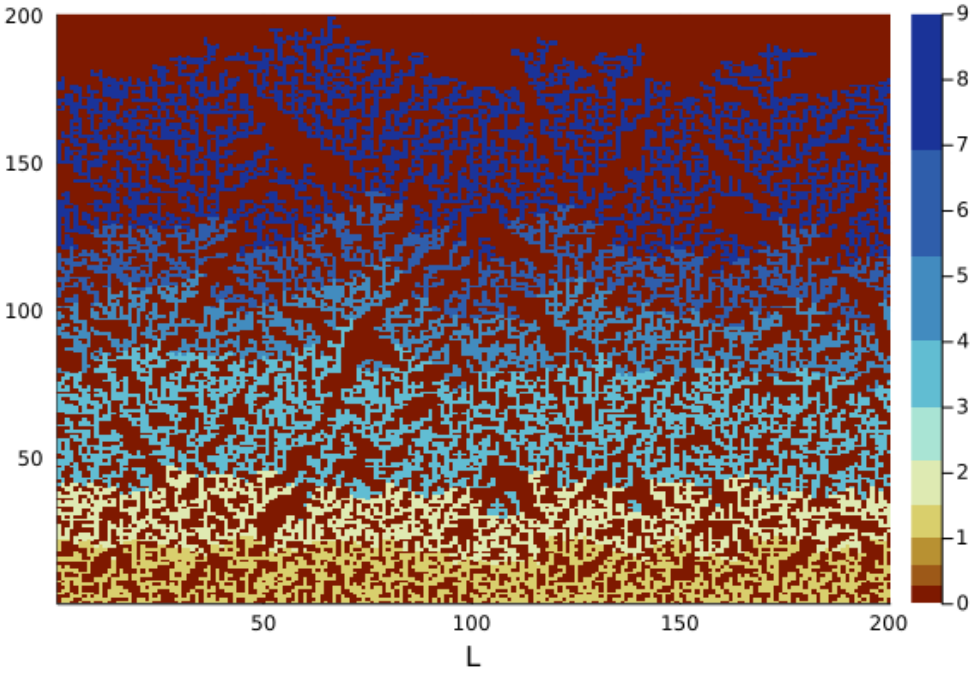
(c) w(t) over time. Time intervals are between 1 to 13, increasing by 0.5. The number of falling particles in each time interval are chosen by this time series(n). the code running is repeated for 100 times. β equals the slope of the line fit to the plot which is 0.26. Unfortunately the saturation time is not reached in this time range and if I increase the final time, I have runtime problem! So z and α are not clear.

Figure 1: Plots of Random Ballistic Deposition With Relaxation (exercise3.2)

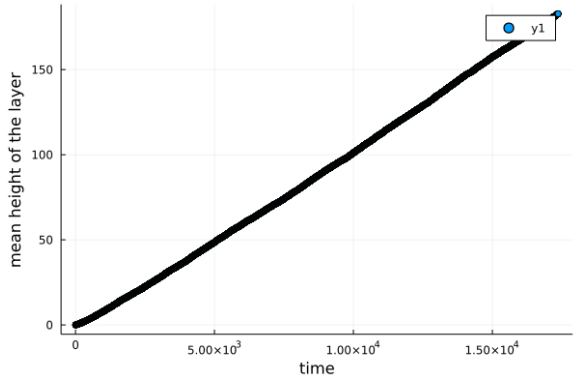
2. exercise 3.3

Same as the previous exercise we aim to increase the smoothness of the layer in each step. The same boundary conditions is considered in the same structure. After having a particle with a random column, it can choose which height to settle in without changing it's column. So, basically the maximum height is chosen, and if it's the height of neighbors, the particle deposits at the same height, and if it belongs to the same column, particle is falling in, the particle deposits at height plus one.

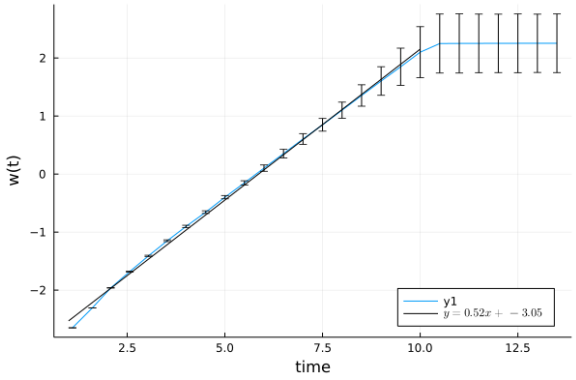
My *height calculator function* in this exercise processes quit different, compared with the previous code. Because we have holes in the dynamic, in this code the amount of filled entries no longer represent the height. This is one of the mistakes I made during coding this exercise, assuming everything is the same as the previous one, except for one tiny change. So, in this code I calculated the height by, starting from last row(L) and counting all the empty entries, until reaching the first filled entry. Subtracting the sum of counted empty entries from L, gives the height.



(a) The models dynamic of exercise 3.3. number of particles=30000, L=200, in a *L*×*L Matrix*.



(b) The Mean Height of a Layer over Time. n=30000, L=200, slope=0.01.

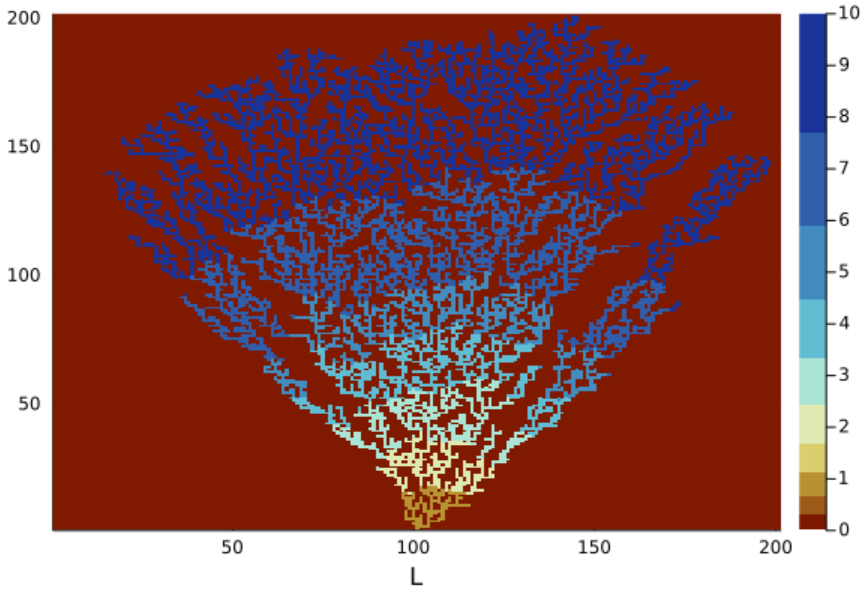


(c) w(t) over time. Time intervals are between 1 to 14, increasing by 0.5. The number of falling particles in each time interval are chosen by this time series(n). the code running is repeated for 100 times. The saturation time is reached after approximately 19 time steps. $t_s \sim 10.5$,β=0.52, z=0.443797, α ∼0.231.

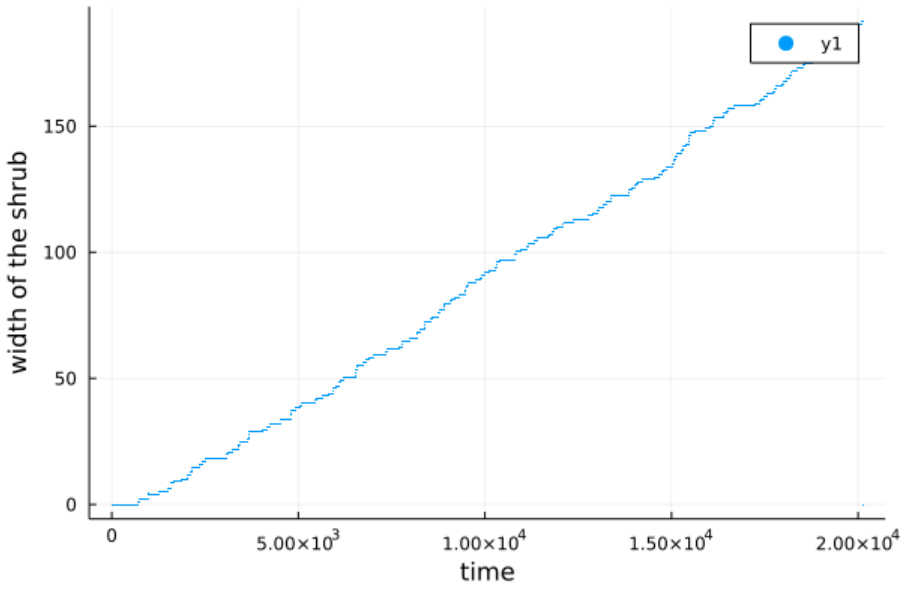
Figure 2: plots of exercise 3.3

3. exercise 3.4

In this exercise I considered the starting point to be at row=1 and column=L/2. Everything in this code is the same as the previous code, except for the fact that no particle with column other than L/2 and height=1 (because in the previous code, it was presumed that if the maximum height doesn't belong to the height of the neighbors, the particle lands on the last particle(which is at the maximum height) can land on the first row.



(a) The models dynamic of exercise 3.4.



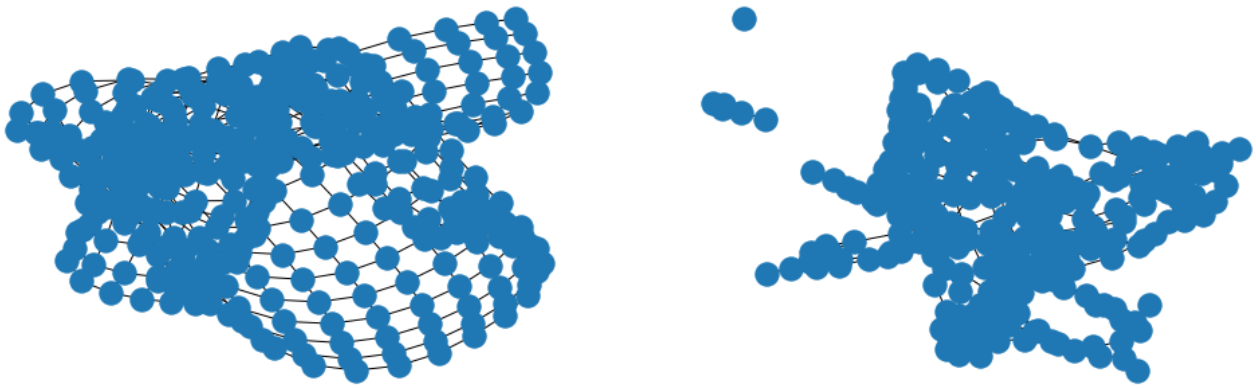
(b) Width of the Shrub over Time. It appears that when time goes on and more particles fall, the width of the tree increases linearly.

Figure 3: exercise 3.4. n=30000 particles, L=201, in a $L \times L$ *Matrix*

Percolation

4. exercise 4.1

For coding of this exercise I imported *networkx* in *python* to represent percolation in a grid. So instead of what's written in the example, to assume that all the nodes are at first *off* and then each node can become *on* if it's random probability is less than a particular probability, I presumed that all the nodes are initially *on* for simplicity and then some nodes can become *on* with a random probability if the probability is less than a certain *1-P*.



(a) The Initial Grid with 20 nodes.

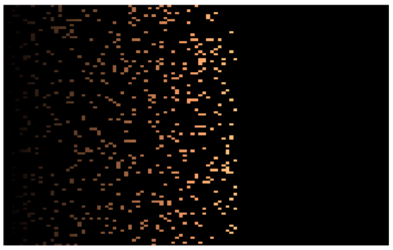
(b) The Final Grid after some nodes are removed, so that only *on* nodes are left.

Figure 4: Figures of the Grid before and after elimination of *off* nodes.The probability that a node can change to *on* state *(P)*=0.8. For this set of results, there *is* an infinite cluster.

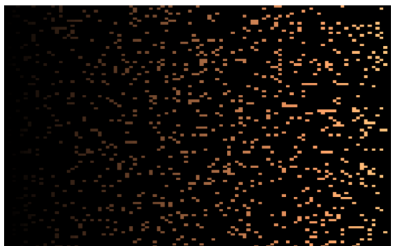
5. exercise 4.2

The algorithm of this exercise is just like what is written in the book. I considered a $L \times L$ *Matrix* with all it's entries set to be initially zero. After setting all the first column's elements to one and the last column's elements to *int_ max=100000*, the loops go through each elements of the second column to 1-L column, doing everything just as the book has said. *color_ change_ list* is the colors of the neighbors or the better to say the attributes of the neighbors that we need to have for the algorithm to work properly. The results are also shown as a gif. Here are some snapshots of them:

(a) First snapshot for p=0.1.

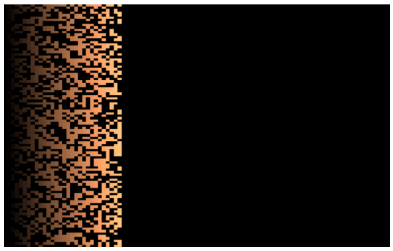


(b) Second snapshot for p=0.1.

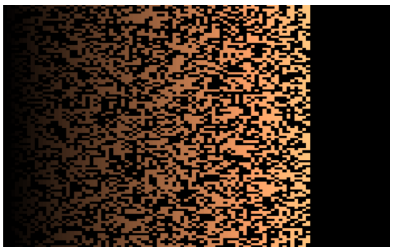


(c) Third snapshot for p=0.1

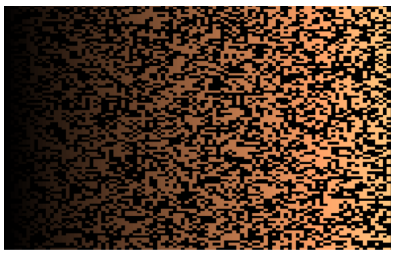
Figure 5: Percolation using coloring algorithm. *100×100 Matrix*,int_ max=100000, p=0.1.



(a) First snapshot for p=0.5.



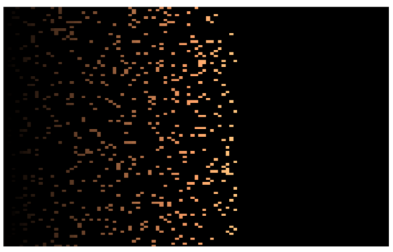
(b) Second snapshot for p=0.5.



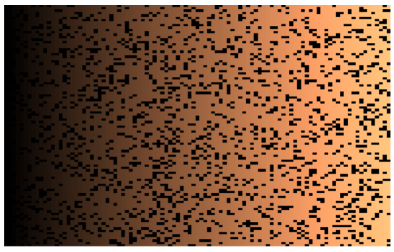
(c) Third snapshot for p=0.5

Figure 6: Percolation using coloring algorithm. *100×100 Matrix*,int_ max=100000, p=0.5.

(a) First snapshot for p=0.8.



(b) Second snapshot for p=0.8.

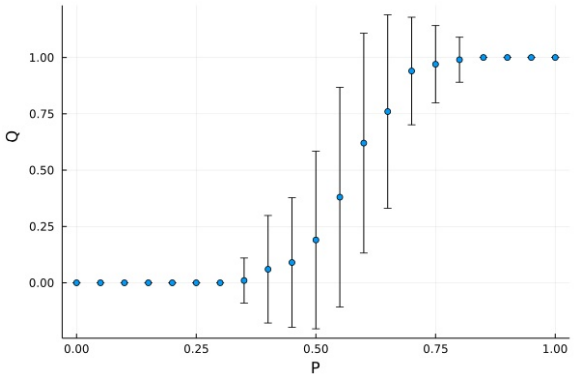


(c) Third snapshot for p=0.8

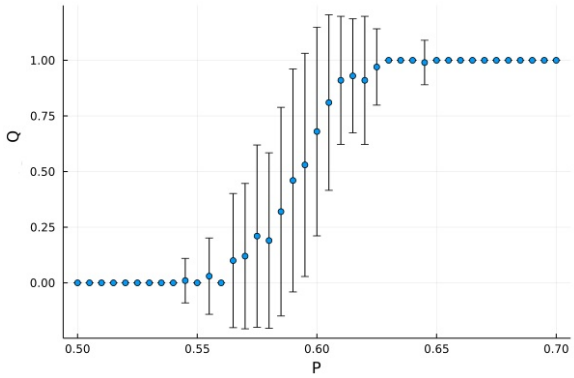
Figure 7: Percolation using coloring algorithm. *100×100 Matrix*,int_ max=100000, p=0.8.

6. exercise 4.3

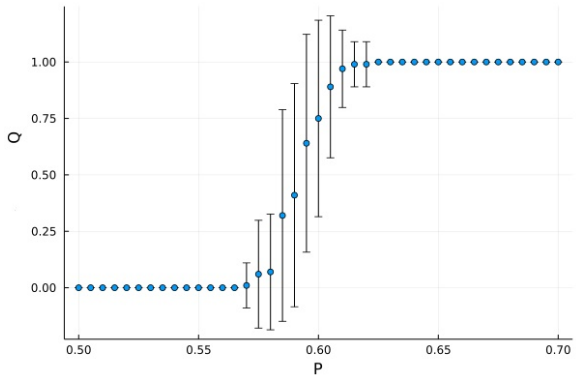
I used the Hoshen-Keoplman algorithm to code this exercise, following the steps as written in the book. I wrote 4 functions. one for building the array of neighbors, consisting of two neighbors(the left and the upper neighbor, if they are on). returning this to the percolation function, everything else goes as is said in the book. I had some difficulty writing the Recursion Function for returning the alternative labels associated with remaining entries in the feature, so at last I used a whole different strategy for this function, looping through the L array until the index and the amount of the label are equal to each other. At last, I wrote a function to check whether there is a intersect between the first and the last column or not. if there is, it returns *1* to to the *for loop* that evaluates this process with different values of P, and repeating it for each probability several times. Finally, we have all the values of each run as a mean and STD number, pushed into the main list. Giving this mean final mean list to the *scatter()*, we can have the plots with error bars gained by the final STD list.



(a) L=10, $0 \leq p \leq 1$, $\Delta p = 0.05$, number of runs=100.



(b) L=100, $0.5 \leq p \leq 0.7$, $\Delta p = 0.005$, number of runs=100.

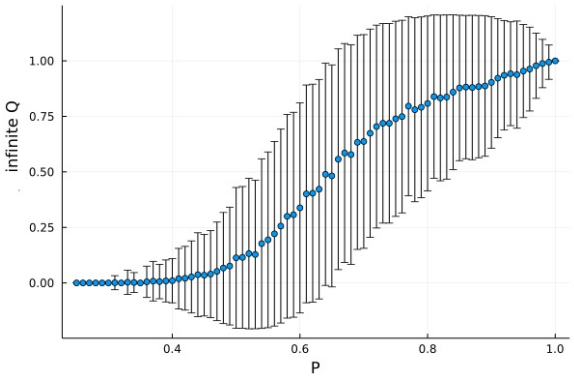


(c) L=100, $0.5 \leq p \leq 0.7$, $\Delta p = 0.005$, number of runs=100.

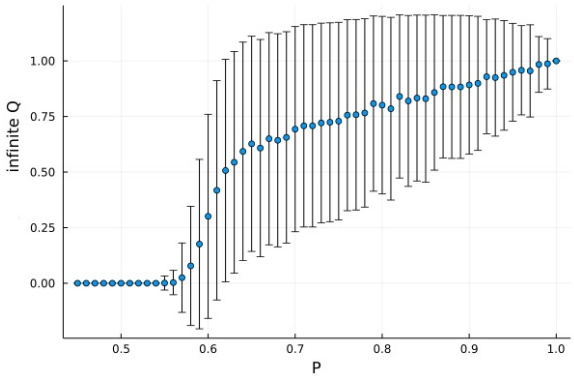
Figure 8: Q per P for different amounts of L.

7. exercise 4.4

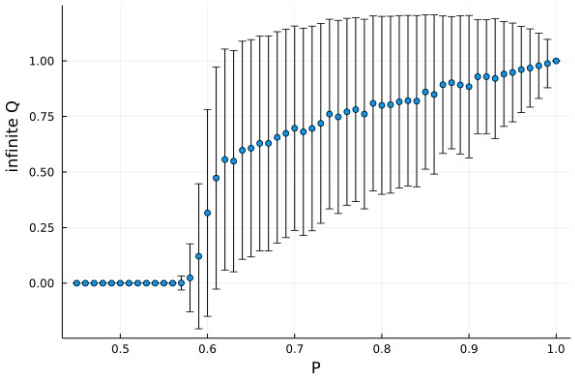
To gain these plots, just one tiny change is made in the function that returns the numbers *1* or *0* as an assurance that an intersect exists. We pick a random entry in the matrix and see if it's on and if it's label has a link to the intersect of the first and the last column. if true, then the function returns *1* indicating that the entry is connected to the an infinite cluster.



(a) L=10, $0.45 \leq p \leq 1$, $\Delta p = 0.01$, number of runs=1000.



(b) L=100, $0.45 \leq p \leq 0.1$, $\Delta p = 0.01$, number of runs=1000.



(c) L=100, $0.45 \leq p \leq 0.1$, $\Delta p = 0.01$, number of runs=1000.

Figure 9: Q_∞ per P for different amounts of L.