ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lecture with Computer Exercises:
Modelling and Simulating Social Systems with MATLAB

Project Report

Self-Organized Criticality and Phase Transitions in a Forest Fire Model

Nishant Dogra, Michael Wild

Zurich
December 2012

Agreement for free-download

We hereby agree to make our source code for this project freely available for download from the web pages of the SOMS chair. Furthermore, we assure that all source code is written by ourselves and is not violating any copyright restrictions.

Nishant Dogra

Michael Wild

Contents

1 Abstract

In the scope of the ETH lecture "Modeling and Simulating Social Systems with Matlab", we attempt to produce a state of self-organized criticality in a forest-fire model. The concept is implemented as a cellular automata and several important data sets are extracted from the simulation. We describe the concept of self-organized criticality and cellular automata. We discuss the results found from the simulations and the implications of the input parameters on these. We try to explain what the quantitative characteristics of a SOC state in a forest fire model are and at which point such a state arises.

2 Individual contributions

The Model with finite burning time was implemented by Nishant Dogra. The Model with instantaneous burning time was implemented by Michael Wild. The respective plots and results sections are also written in that way. Both have contributed in writing the report and most of the sections are a work of collaboration and discussion.

3 Introduction and Motivations

3.1 What is Self-Organized Criticality?

Self-Organized Criticality involves 2 very important concepts: Self-Organization and Critical behavior[1]. These 2 phenomenon arouse in various fields in very different forms.In certain class of systems, the dynamics can be very well explained by a few collective degrees of freedom. This dimensional reduction is termed as "Self Organization". It leads to evolution of global patterns in the system even though all the interactions are local. Let us take an example from the field of physics.Crystallization is the process of formation of crystals. It usually requires a small nucleus for the growth of the crystal. Once the nucleus is formed, crystal can grow on this nucleus. If we consider a particle which is going to crystallize on a given site, the choice of this site is decided only by the local environment seen by the particle. It does not depend on the particle on every other site on the crystal. But this local interaction leads to the formation of an ordered crystalline phase. Bird flocking is an example of self organization from the field of Biology.

On the other hand, there is another set of dynamical systems where the individual degree of freedom keep itself more or less in a stable balance and it is not possible to describe such a system by a few collective degrees of freedom. The key point in such systems is that the interdependence of various degrees of freedom make

such systems very susceptible to noise. That's why, these systems can be called as "Critical". Critical state usually represent a point where 2 different phases occur simultaneously at the same time. A critical state has specific temporal and spatial signature. The power spectrum of the quantity representing the dynamics of the system is similar to that flicker noise. The spatial signature of a critical state is "self-similarity" which is what we find for the case of fractals.

Self-Organized criticality combines the above 2 concepts. In it, the dynamics of a physical system converge to a critical point independent of the bestowed boundary conditions. This critical point commonly is not the equilibrium point in a traditional, physical sense. We usually say that the *critical point* is an attractor of the system dynamics. The concept is best understood by example, so we shall bring up a very educative one introduced first by [1]. Imagine a sand pile in 2 dimensions. One can discretisze this pile into cells, which are denoted with the indices n. Each of those cells has an amount of sand grains on it, which is directly corresponding to the height z_n. Now we state the simple rule, that, if the difference in height between two neighboring cells becomes larger than a predefined limit, the higher cell will give grains to the lower cell (avalanche). Now every timestep, we drop a grain on cell one. Independent of the initial conditions, eventually, a straight slope will evolve. The critical point is reached if every difference between two neighboring cells is exactly the limit. What happens if we drop the next grain? Then the difference $z_1 - z_2 > p$ and one grain will drop on z_2. Subsequently, the difference $z_2 - z_3$ will be too big and the process repeats until it hits the last cell. We therefore call a state critical if a yet so small disturbance can propagate throughout the whole system. It is important to note here, that this critical state is *not* the equilibrium state of the system, since that would be a flat surface.

3.2 Forest-Fire Model (FFM) as an example of Self-Organized Criticality

As the name suggests, modeling of the ignition, propagation and extinction of fires in a forest is termed as Forest Fire Model (FFM). This model has been studied extensively as one of the dummy models to show Self-Organized Criticality. Historically, the first model for SOC was proposed by Per Bak,Kan Chen and Chao Tang [2]. But it was later on shown that this model does not exhibit SOC state [3; 4]. Later, a slightly modified version of the original model was verified to exhibit SOC [5]. After that, a lot of study has been performed on this model with various additive features.

3.3 Motivation to study Forest Fire Model

Why is it important to study Forest Fire Model? Forest fire models apply to a wide range of problems and are therefore helpful in predicting the behavior of such

systems. Self-Organized criticality is the concept that describes the dynamics of such models and the two fields are therefore closely connected. One has closely studied these effects to make better predictions about forest fires. The main problem was, that in most cases, even relatively small fires were extinguished by the authorities. Since that led to an *overcritical state*, the probability for a huge, devastating fire rose quickly. This has happened in the past and had a serious impact on the ecosystem of those areas. Since the problem of self-organized criticality has been understood, one has stopped to extinguish every little fire, therefore avoiding to reach an overcritical state at which the disturbance (a small fire) can propagate throughout the whole system (large-scale fire). The same ideas can be applied to a variety of problems such as disease spreading or urban planning.

3.4 What do we want to know from the Forest Fore Model- The QUESTIONS

First, we would like to understand what are the quantitative characteristics of a Self Organized Critical state in the Forest-Fire model? We would like to have a well defined idea regarding when does the Forest Fire model manifest SOC behavior and why. We would like to understand what are the various phases seen in a Forest Fire model besides the Self Organized Critical phase. It will be important to know what are the parameters characterizing the phase transition in the Forest Fire Model. Some of the ambitious questions will be to understand the dependence of SOC of the Forest Fire model on the dimensionality of the Lattice or on the neighborhood selection on the grid.
It will also be interesting to find out if computational restrictions like finite size grids, periodic boundary conditions and cell updating procedures have a quantitative effect on the behavior of the model?

4 Description of the Model

4.1 Cellular automata

A cellular automata is a very powerful way of simulating problems which are defined by a set of rules. It is usually simulated on a grid, but not necessarily restricted to those geometrical constraints. The main characteristics are:

- Every grid point has a state
- Grid points change their state depending on the neighbor states according to a set of rules

- Random actions may be introduced

In the example of the FFM, each cell can have one of the following three states:

- 0: A green tree
- 1: Empty site
- 2: Fire

The dynamics of the system is defined though the following rules.They are applied to every step as we go from 1 time step to the next.

1. If a tree is burning at a given time step (2), we make it an empty site during the next time step.
2. If a site is empty (1), a tree can be grown on it with a probability p. This means that if we found out a site to be empty, we generate a random number between 0 and 1. If the number is less than p, we change the state of the cell to 0, otherwise it stays an empty site.
3. If a site is a green tree (0), we burn this tree in the next step if:
    - Either of its neighbors is a burning tree.
    - We generate a random number between 0 and 1, if it is less than f (lightening probability), we burn this tree.

It is important to note that the Forest Fire Model does not only apply to forest fires as its name would suggest. A whole class of problems follows the same basic rules and can be treated accordingly. An example would be the spreading of a disease, where one can directly transform the above rules and states.

This model describes an SOC state in the regime where the ratio of p and f is very large and p itself is very small. An important question is how to characterize a state as an SOC state. This is described below.

The second approach to grid-based updating is with a second grid that stores temporary information. [6]

4.2 Understanding the SOC regime of Forest Fire Model

For FFM to be an SOC, we need to ensure 2 things. One is that the probability of growing a tree should be much larger than the probability of lightening. We can intuitively think about the ratio of p and f as the number of trees growing between 2 lightening strokes.This condition ensures long range correlation in the system (which

will lead to criticality) building out of purely local interactions (randomly growing trees). Another condition which we need to ensure is that the time needed to burn a large cluster should be much smaller than the time needed to grow a tree. This condition ensures invariance under a change of the length scale [7; 6]. The 2 conditions can be written together as:

$$T_{burn}(cluster_{max}) << p^{-1} << f^{-1} \tag{1}$$

4.3 Critical Exponents

Usually, a critical point is associated with certain power law distributions of various physical quantities characterizing a critical state. Power laws are functions of the form $P(s) \propto s^{-\tau}$. These functions are found in many data sets, such as the size distribution of cities in a certain area. Take any country, one will find one very large metropolis, a few large cities, many medium sized towns and a huge lot of small towns. Power laws have also prominently been found in data sets of earthquakes, where they apply almost perfectly, meaning that small earthquakes happen very often in respect to the frequency of big ones. Power laws imply that the frequency of a general data point is inversely proportional to its magnitude.

At a critical point, the power laws are characterized by critical exponents which are the indicators of the critical state. They usually depend only on the dimension of the system and the nature of interactions/ dynamics involved. These exponents are independent of the initial condition and the size of the system. These critical exponents can be calculated either by mean field approaches or through numerical simulations. The mean field approach becomes more and more exact in higher and higher dimensions.

In context to the FFM, it is proposed as a way of quantifying the frequency of Fires with respect to their cluster size.

4.4 Key Parameters in FFM SOC

To find out whether a basic FFM exhibits SOC behavior, we need to define a set of parameters which serves as indicators for the desired analysis.

- The time needed to burn down a forest cluster: This is heavily dependent on the form of the implementation. We can implement it in a way where we will have an instantaneous fire which essentially takes no time to burn and just resets all the connected trees. If however, we choose the form of a visible fire, meaning that the ratio $f/p \ll 1$ is not 0 in limit and that it takes the fire one time step to advance a grid cell, then the time needed to burn a forest cluster is an important variable we need to extract from the simulation.

- Number distribution of the size of clusters: Here we first need to define what a cluster is. A cluster is a set of neighboring cells (Von Neumann neighborhood) all obtaining the same state. When we talk about forest clusters, we of course mean connected trees. In the implementation where the fire spreads infinitely fast, the forest cluster containing the igniter cell is the same as the burnt area. The size distribution therefore is a very good indicator for the fire size distribution.
- The mean number of forest clusters in a unit volume $n(s)$, where s is the number of trees in a cluster.
- The radius of a cluster, it is defined as the root mean squared distance of all the elements of a cluster from the mean. It gives an estimate of the size of cluster.
- Number of fires per unit time step: This is a tuning parameter of the model.
- Correlation length.

We have tried to implement the model in a way that the measurement of these desired variables is possible with the least needed effort.

5 Implementation

As discussed before, there are two main implementation methods for the model: One way is to update the whole grid every time step, whereas the second method picks a cell randomly at every time step an only updates the chosen one. One of us worked on the first method and the other one worked on the second method.

5.1 Grid Based Updating

The algorithm for grid initialization looks like:

```
Grid(N,N)=zeros(N,N)
rand_grid=rand(N,N) // we generate a grid of random numbers
m=1;
while m<=size(Grid,1) // Loop over all y-coordinates
n=1;
while n<=size(Grid,2) // Loop over all x-coordinates
if rand_grid(m,n)<k //If the random number for a site is less than k,
//we make the site empty, else it has a tree
Grid(j,k)=1    // chossing different k will lead to different
//initial state of the grid
```

```
end
end
end
```

Basically, we initialize the grid with trees and empty sites distributed randomly on the grid. The updating of the grid based on the simple rules described above can be done in 2 different ways. One is when we make a copy of the current grid (*Grid_copy*) and then update the current grid using this *grid_copy*. The other is that we use the current grid (which is being updated) as the one which also responsible for updating. The 2nd procedure require us to pick the sites randomly and update them, but it will be hard to keep track about the sites which have been updated at a time step. In our simulations, we follow the first procedure whose algorithm is described below:

```
it=1:
while it<=Iter // Loop over all time steps
    Forest_grid_temp=Forest_grid;  //Forest_grid_temp is the copy
    //which is used to update the grid at the next time step.
// we need to take care that the neighbors of every cell are correct
//as based on the periodic boundary condition.

//This has been handled by making the size of the copy as 2 units
//larger in each of the dimensions

    Copy the first column of Forest_grid to last column of Forest_grid_copy
    Copy the last column of Forest_grid to first column of Forest_grid_copy
    Copy the first row of Forest_grid to last row of Forest_grid_copy
    Copy the last row of Forest_grid to first row of Forest_grid_copy
    m=2;  //looping over the Forest_grid and its copy,
    while m<N
        n=2;
        while n<N
            if Forest_grid_temp(m,n)==2
                Forest_grid(m,n)=1;
              //a fire becomes an empty site

            elseif Forest_grid_temp(m,n)==1
                emp=emp+1;
                if rand(1,1)<p
                    Forest_grid(m,n)=0;
                end
```

```
              //an empty site becomes a tree if a generated random no. is <p
            else
                if (rand(1,1)<f || Forest_grid_temp(m-1,n)==2 ||
                Forest_grid_temp(m+1,n)==2 || Forest_grid_temp(m,n-1)==2 ||
                 Forest_grid_temp(m,n+1)==2)
                     Forest_grid(m,n)=2;
                //if any of the neighbours of the current site is burning or a
                //random number is generated which is greater than lightening
                //probability, we burn the tree.
                end
            end
            n=n+1;
        end
        m=m+1;
    end
    it=it+1
end
```

One of the important reasons why we made another model is that the algorithm is quite slow because of all the loops and if-statements,we are using.

The next important thing which we need to find out is indexing uniquely all the clusters of trees. The algorithm is described below:

```
Index=3;
m=1;  //looping over the Forest_grid and its copy,
while m<N
   n=1;
   while n<N
    if Forest_grid(m,n) is an un-indexed tree
    we search all its neighbors to find the smallest index
    if index is found,
    Forest_grid(m,n)=index;
    We also update the index of all the nearest
    trees with this index
    if index is not found,
    Forest_grid(m,n)=Index;
    We also update the index of all the nearest
    trees with this Index
    Index=Index+1;
    end
```

```
end
end
end
//The above loop will be sufficient if we have absorbing boundary conditions.
//Periodic boundary condition requires that we should check all the indexing
//again
m=1;  //looping over the Forest_grid and its copy,
while m<N
   n=1;
   while n<N
    if Forest_grid(m,n) is a tree
    we search all its neighbors to find the smallest index
    If the index of any of the neighbors or the Forest_grid(m,n)
    is different than this smallest index, we replace all the trees
    of that index with this smallest index
end
end
end

for i=1:t // Loop over all timesteps
    for j=1:size(Grid,1) // Loop over all y-coordinates
        for k=1:size(Grid,2) // Loop over all x-coordinates
            if Grid(j,k)==0 //If the grid point is empty
                //Grid(j,k)=1 with some probability p
            end
        end
    end
    for j=1:size(Grid,1) // Loop over all y-coordinates
        for k=1:size(Grid,2) // Loop over all x-coordinates
            //if Neighbor(j,k)==3 // If grid neighbor is burning
                Grid(j,k)=2 // Set current grid point on fire
            end
        end
    end
    for j=1:size(Grid,1) // Loop over all y-coordinates
        for k=1:size(Grid,2) // Loop over all x-coordinates
            if Grid(j,k)==3 // If Grid point is burning
                Grid(j,k)=0; // Turn it into an empty site
            end
```

```
            if Grid(j,k)==2 // If grid point is ignited
                Grid(j,k)=3; // Turn it into an empty site.
            end
        end
    end
end
```

Several things are to note here:

- Commands with // or in front are not implemented exactly like that. Check the appendix for the complete source code.
- We need several spatial loops in order to successfully suppress updating fragments like infinite fire propagation speed in updating direction.
- For the same reason, we need a differention between newly ignited (state 2) and burning (state 3) trees.
- The Algorithm is quite slow because of all the loops and if-statements.

Once the indexing is done, we can simply count the number of trees corresponding to a given index and can find out the number of number of clusters of a given size (where by size, we mean the number of trees in the cluster).

Calculation of the radius of a cluster is little subtle. Naively, it just the root-mean squared distance of all the elements from the cluster mean. The formula can be simplified as:

$$R = \sqrt{\frac{1}{N}\sum x_i^2 - \frac{1}{N}(\sum x_i)^2} \tag{2}$$

This would have worked well if we have absorbing boundary conditions, but since our boundary is periodic; there can be possible a cluster which is going over the boundary. In that case, above formula can lead to a cluster mean which is outside the cluster. In such a situation, we need to rotate our grid to calculate the cluster radius properly and efficiently. The underlying idea is very simple. If you are scanning through different columns (rows) from left to right (top to bottom), you should get an element of a cluster in every column since the starting column for that cluster. In case, there is a break, it implies that you have cluster going though the edges, otherwise it cant be a cluster. Another important property of a cluster going through their boundary is that it should have a member in the leftmost (topmost) and rightmost (down most) column (row). We exploit these in the following algorithm:

```
//Column shift
define variables F and ml for every cluster index
initialize F and ml with zero for every cluster
m=1
while m<=N  %going through all the columns
n=1;
while n<=N  //going through all the rows of a given column
if the current element is a tree,store the index of current
element in variable cl
if F(cl)==0
//this is the first column where an element of this cluster appears.
if m~=1
F(cl)=3;
//this implies that this cluster cannot be connected via the boundary
else
F(cl)=1;
ml(cl)=m;
  end
elseif F(cl)==1
if m~=ml(cl)+1
//this is the case when we have atleast one column in between where no
//element of the given cluster is found and this can happen only if
//the cluster passes through the boundary.
  F(cl)=2;
end
ml(cl)=m;
end
end
end
//The column shift needed for a given cluster can be calculated as:
if F(cl)==2
shift_col(cl)=N+1-ml(cl)
else
no shift
end

//An exactly similar procedure is followed to calculate the row shift for every clust
//index where we go though all the columns for a gicen row and very all the rows.
```

5.2 Random Based Updating

In a random based updating implementation, at the beginning of every time step, one random cell is chosen. It then checks for the rules of the model and acts accordingly. In order to see fires, we need to have instantaneous burning, which is actually permitted by the model. However, it changes the dynamics of the system drastically. But more about that later. The following steps are taken for each timestep:

1. Choose a single cell
2. If cell is empty, grow a tree with probability p
3. If cell is a tree, with probabiltiy f, burn it and the connected cluster

In Matlab, this looks something like this:

```
for i=1:t // Loop over all timesteps
(j,k)=ceil(Grid_Size*rand(1,2)); // Choose a random cell
if  Grid(j,k)==0 % If the cell is empty
//With some probability p set G(j,k)==1;
end
if Grid(j,k)==1 // If the cell is alive
//with some probability f execute burn(j,k);
end
end
```

This implementation of the forest fire model has several advantages:

- There are no problems due to updating processes
- Much shorter implementation
- Easier to extract data

The last point is to be explained more extensively: Every time a tree ignites, the whole connected cluster is burned down. Since the whole process happens in a single timestep, there are no big issues. This implementation, however, also comes with quite a big disadvantage: It fails to capture dynamics. This leads to a quasi-static simulation. We are not totally sure how this affects the simulation results.

5.2.1 keeping track of the clusters

Keeping track of the different clusters was more difficult than we thought at first, but we managed to find an elegant solution. It works by keeping a second grid which stores all the cluster indices. This has several advantages:

- if a new tree grows, we can just look at the neighbor cluster index to determine the current cell index
- if a tree burns, we simply delete all the trees with the same index as the burned tree.

Problems and Solutions Problems arose quickly. Here is a short summary of all the major problems and the corresponding solutions:

Storing the indices The problem here was to find a way to store all possible cluster indices and their availability. Since the maximum number of clusters can be easily found to be $Nc_{max} = \frac{G^2}{2}$ (imagine a perfect chessboard-layout filling half of the grid; if one more is added, they combine into a bigger cluster), we already knew the size of the vector. The final format was a Matrix with Nc_{max} Columns and 2 rows. The first row would simply store all the possible indices from 1 to Nc_{max} and the second row would store the corresponding binary information about the availability.

growing a tree between two clusters It is simple to determine the cluster index of a new tree if there is just one neighbor. The big problem arises if there are more than two neighbors with *different* cluster indices. The solution to this problem was that we set the index as the lower of both neighbors and changed the whole cluster with the higher index to the lower one. This was relatively easy to do by using the matlab built-in function —ismember— which searches for all entries with a certain value and returns a matrix containing only that information.

5.3 The *neighbor* function

The *neighbor* function is the same for both implementations. It takes the current cell coordinates (j, k) and returns a vector containing the coordinates of its direct neighbors. It works like this:

```
Set all neighbors to standard
// Ex: West_X=j-1; West_Y=k;
Treat Special cases
// Ex: if j==1
// West_X=size(Grid,1);
```

Note that the *neighbor* function creates periodic boundary conditions.

5.4 Saving the data

Because we were doing a lot of parameter sweeps and thus generating a lot of data, it was crucial to implement an automatic data saving procedure. This would store all the relevant information of a single simultation in a custom matlab struct and save the important figures. We also created a custom naming concept which allows for quick search for the desired data. All simulation results are named after their input parameters:

6 Simulation Results and Discussion

6.1 Traditional Forest Fire Model

In this section, we will explain our results and understanding of the traditional model of Forest fire. Here we can consider 4 different regimes.

- small p (< 0.001) and small $\frac{p}{f}$ (≈ 1)
- large p (< 0.001) and large $\frac{p}{f}$ ($\approx 10^4$)
- large p (< 0.001) and small $\frac{p}{f}$ (≈ 1)
- small p (< 0.001) and large $\frac{p}{f}$ ($\approx 10^4$)

Let us first try to understand what we expect intuitively for these cases. In the first case, we will have growth rate of trees to be very small and comparable rate of fires.(The smaller values of usually make the time scale of variation of the system very large.) As a result, we don't expect long-range correlations for the system to build up.

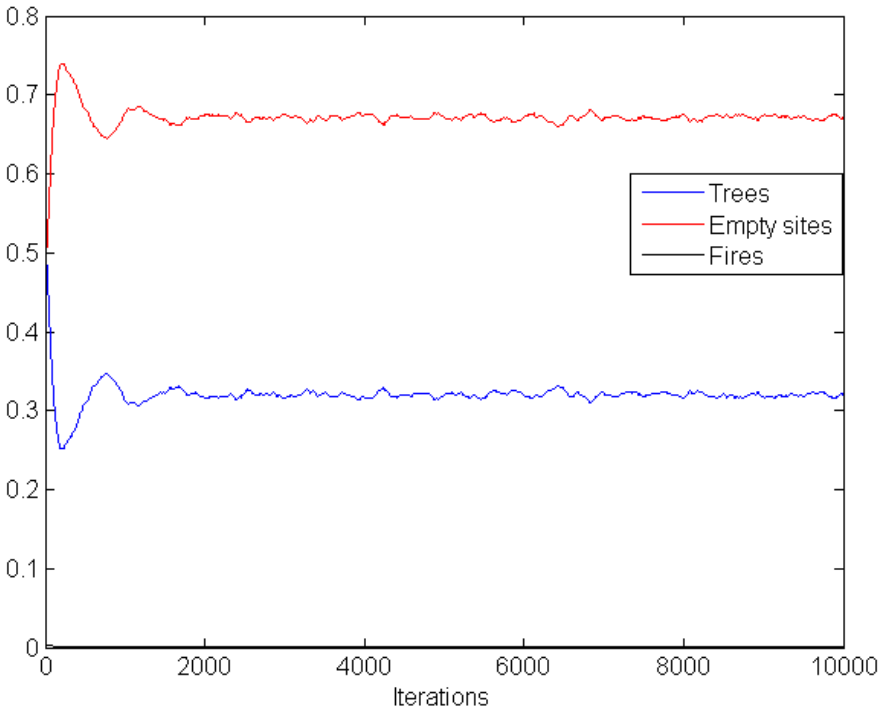

Figure 1: $N = 500,\ \ p = 0.001,\ \frac{p}{f} = 10$(Case1)

In the second case, system will try to build long correlations since the tree growth rate is large. But since p is large, it won't be possible to make the system critical because before a large cluster is burnt by the fire, trees will start growing again in the cluster and the system won't really be in a critical condition.

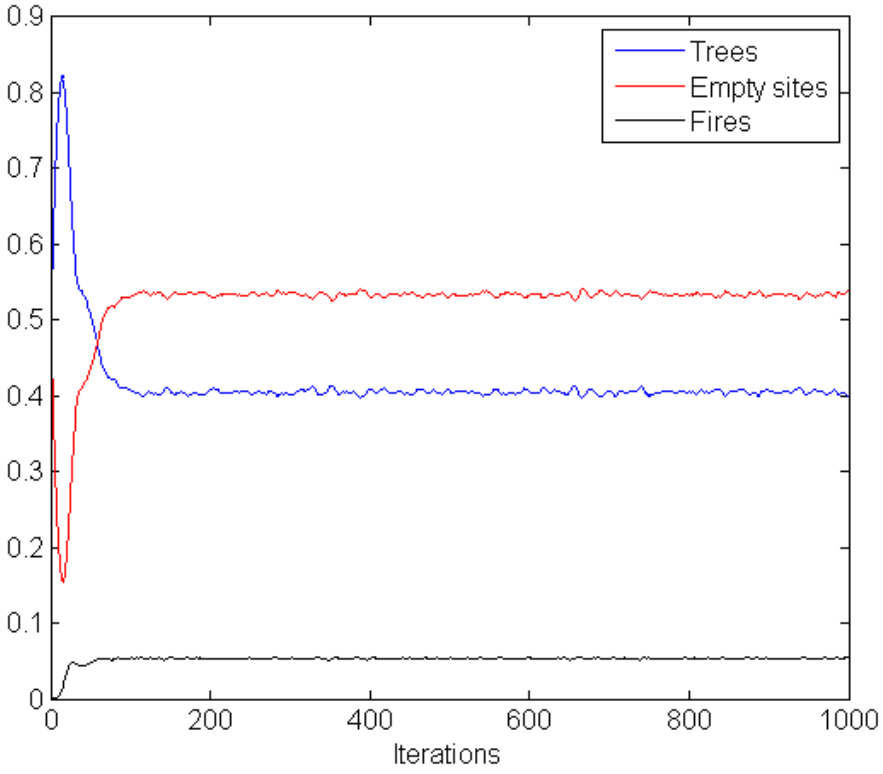

Figure 2: $N = 500,\ \ p = 0.1,\ \frac{p}{f} = 1000$(Case2)

The 3rd case takes the worst part of the above 2 cases. The large tree growth rate don't let the system to become critical. The small $\frac{p}{f}$ wont let the formation of long range correlation in the system and avoid from making the system being critical.

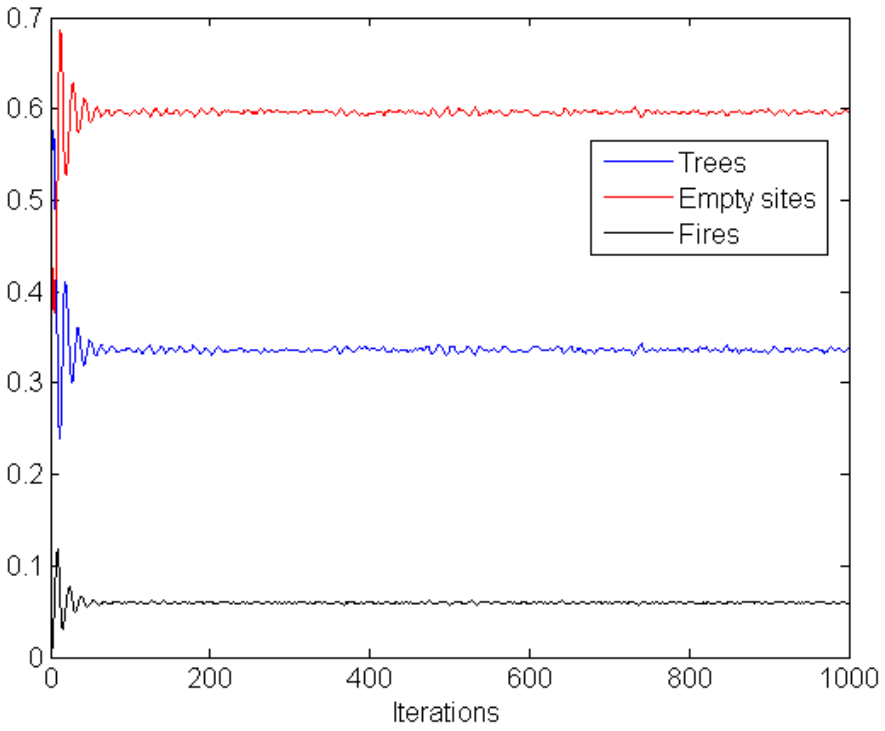

Figure 3: $N = 500,\ \ p = 0.1,\ \frac{p}{f} = 10$(Case3)

In conclusion, these 3 cases are some sort of equilibrium populations where system never becomes critical.

The final case is the one we are interested in. Since $\frac{p}{f}$ is large, large correlations can be built in the system. Since p is small, formation of large correlations make the system critical.

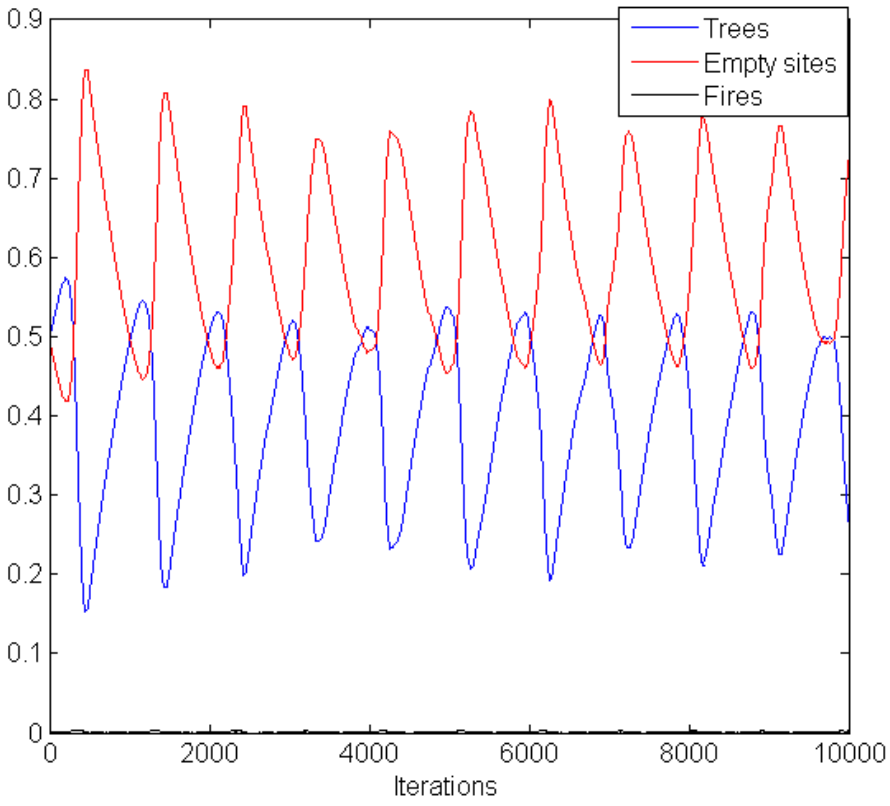

Figure 4: $N = 500,\ \ p = 0.001,\ \frac{p}{f} = 1000$(Case4)

From the way we model our system (we are considering case 4 above), it is very difficult to find out whether for a given set of parameters the time required to burn the biggest cluster in system is much smaller than the time needed to grow a tree. But there is a very simple qualitative way to see this through the variation of various fractions as the system evolves. When an event of lightening happens in the system, the fraction of empty sites grows quickly. But shortly after that when the cluster is fully burnt, the trees again start growing and the fraction of empty sites start to decrease again. If the slope of the slope of increase of the fraction of empty sites when the fire is burning the cluster is much larger than the slope of decrease of fraction of empty sites when the trees start growing, it implies that the cluster burning process is almost instantaneous in comparison to the tree growth. Hence, in the 4th case we expect the system to be in SOC state. In the following figure, we show the case when the system is critical in the sense that long range correlations are forming (a

fire is burning big cluster) but the fire is not instantaneous in comparison to the tree growth rate.

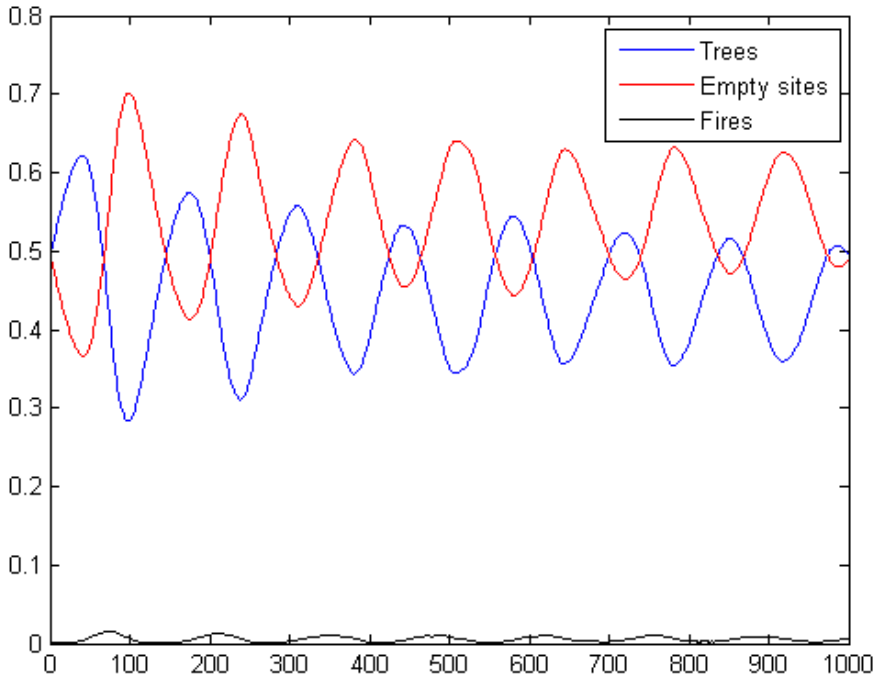

Figure 5: $N = 500,\ \ p = 0.01,\ \frac{p}{f} = 1000$(Case4)

An important point to make here is how the size of the grid affects this self-organized critical state. We see that if the grid is big, we will be having larger long range correlations. It will take larger time for a fire to burn a cluster and hence the regime where the cluster burning is almost instantaneous as compared to tree growth will require lower values of p. This has been shown in the following figure by considering a much smaller grid size as for the case 4 but same parameters.

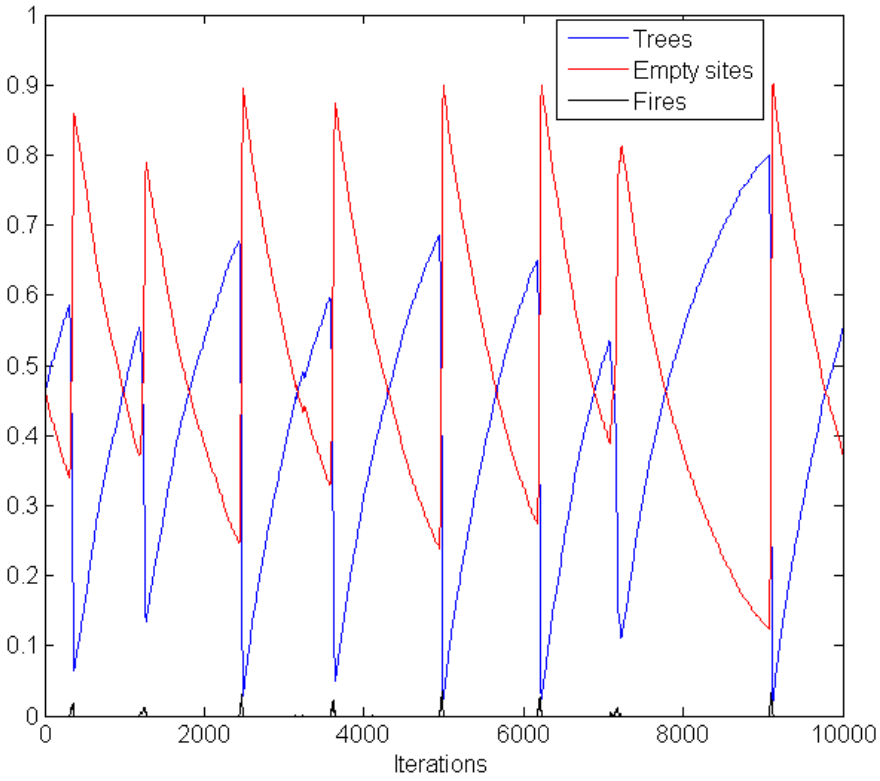

Figure 6: $N = 50,\ \ p = 0.001,\ \frac{p}{f} = 1000$(Case4)

Now, we plot the radius distribution and the frequency of clusters as a function of number of trees in the cluster. Thes eare plotted in the figures below.

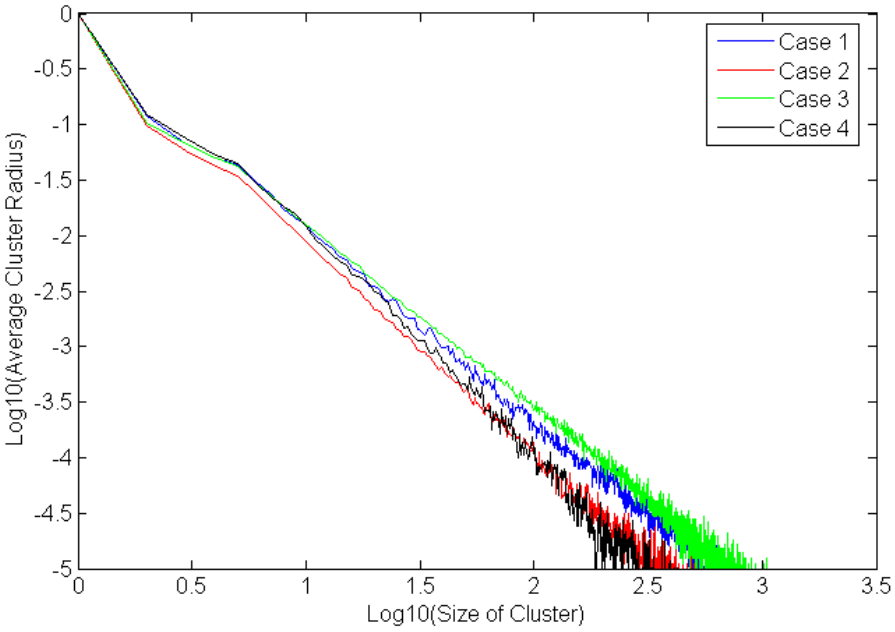

Figure 7: $N = 200$, Frequency Distribution

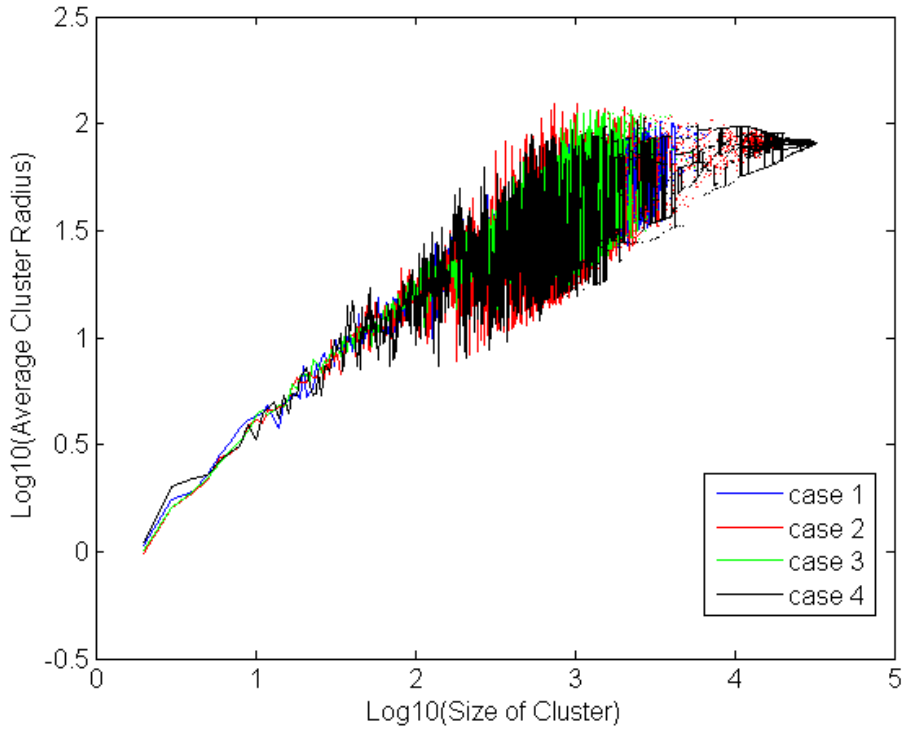

Figure 8: $N = 200$, Radius Distribution

Here we see one of the important drawbacks of this model. Since the algorithms are a bit slow, within our computational limits we cant distinguish the 4 cases from these distributions. This is the reason why we worked on another implementation of the forest fire model which is much faster than this traditional implementation.

6.2 The Second Implementation

6.2.1 Cluster Radius

The Cluster Radius is defined as the root mean of all the distances of all cells of a cluster to their common center. According to the literature, the cluster radius distribution should exhibit a power law behavior and therefore form a straight line in a log-log-plot. We were having some small issues with the cluster radius calculation, but ended up with something that very much resembled such a straight line, as can be seen in Fig.9. The "transient" behavior at large cluster radii is mostly due to too small simulation duration. We did longer simulations, but that constrained the grid

size since at some point, the available memory would be gone.

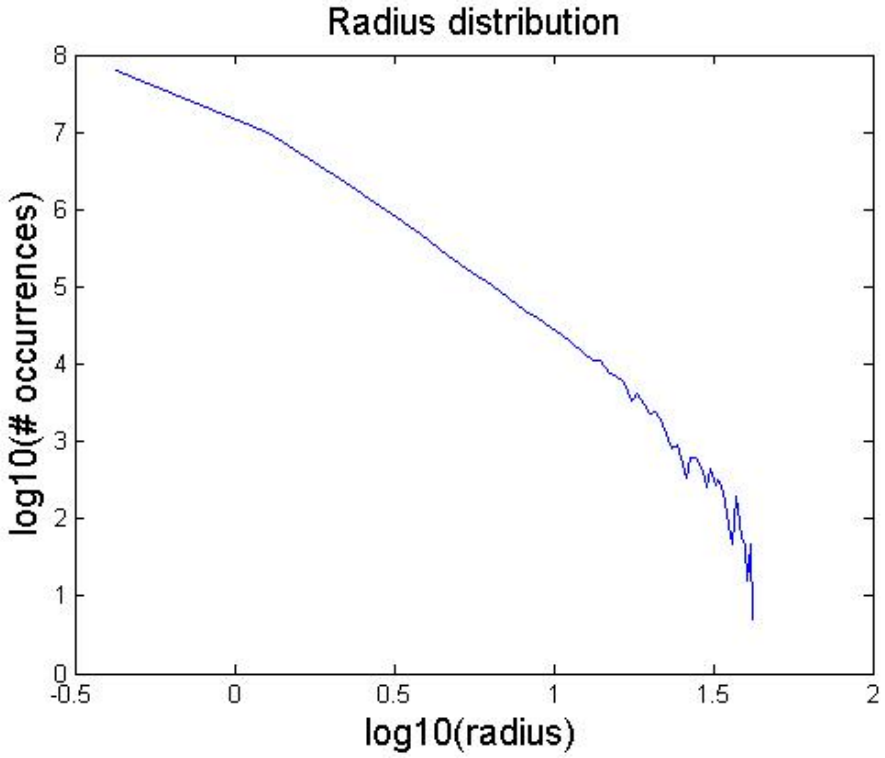

Figure 9: Cluster radius distribution ($N = 100$, $\Theta = 200$, Sim Time 1'000'000)

What we could find was a best slope estimation, which turned out to be around $\beta \approx -3$ for the parameters seen in Fig.9. Since the cluster radius distribution seems to be the most important attribute of the simulation, we thested the effects of various parameters on the simulation by the respective resulting cluster radius distributions. This will be discussed later.

6.2.2 Cluster Size

The cluster size is another variable that we can easily measure. Here is a log-log-histogram of a simulation:

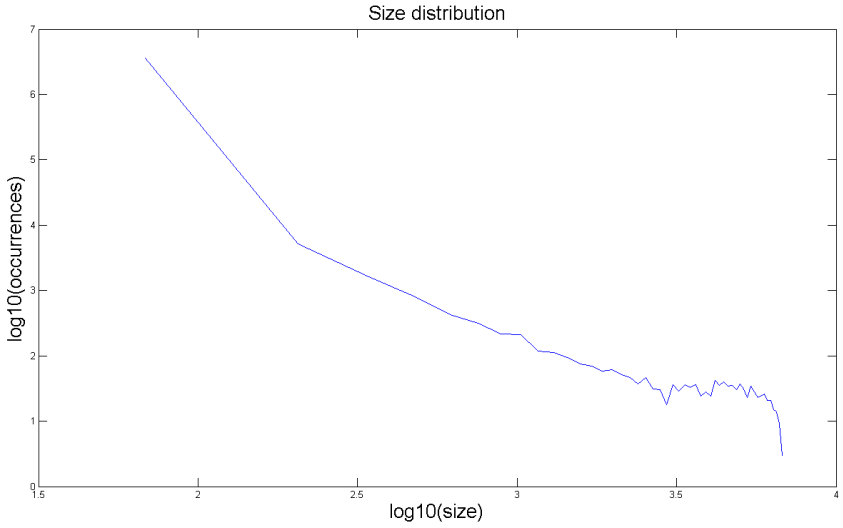

Figure 10: Cluster size distribution ($N = 100$, $\Theta = 1000$, Sim Time 500'000)

6.2.3 Effect of p and f

In the case where the time needed to burn down a cluster is significantly shorter than the time required to grow one, these effects happen on a separated timescale. Looking at fire incidents on a "growth"-timescale, fires spread instantaneously. The question was, would the values of p and f have any effect on the simulation outcome if the ratio f/p is kept constant? We tried to explain it intuitively: the larger the value of p is, the smaller the "time resolution". If it takes n timesteps to grow a certain cluster at $p = 1$, intuition says it will take $2n$ timesteps at $p = 0.5$. Given that the lightning factor f varies proportionally, it also takes twice the time for a fire to happen. Of course, we are talking about expected values here, but since the whole process is a stochastic one by nature, it all pans out in the end. We ran a bulk simulation to verify this claim and ended up with the results seen in Fig.11.

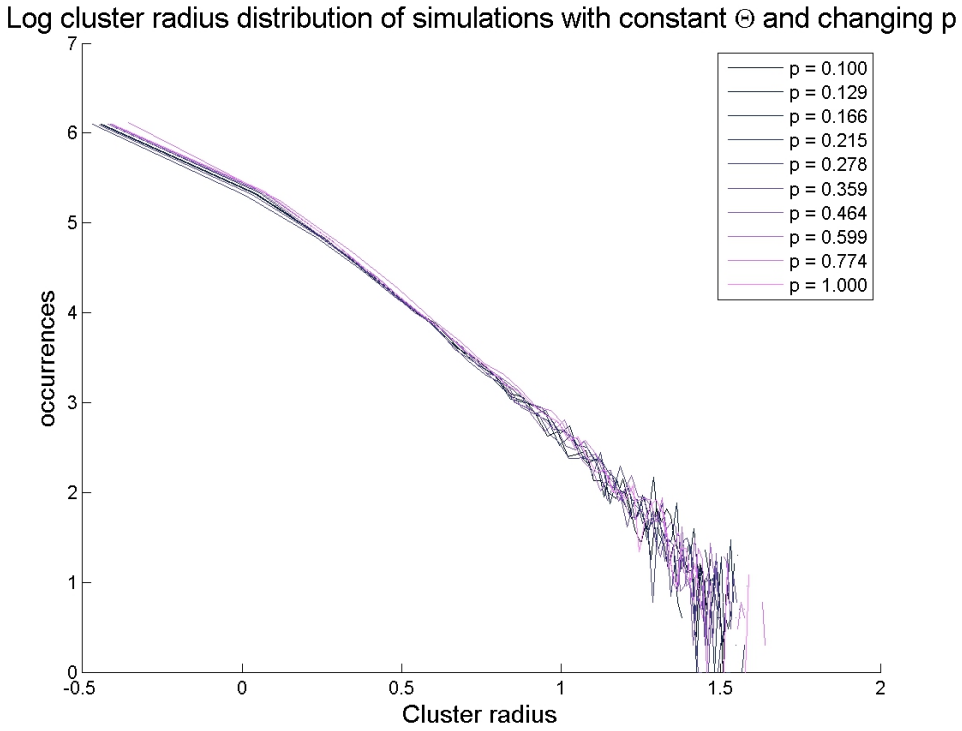

Figure 11: Log cluster radius distributions of simulations with constant Θ and changing p

The results came out as expected. Note that we are only interested in the *slope* of the curves, not the position of the curves themselves. In fact, the simulations in Fig.11 were all run for a different number of timesteps to adjust for the "time resolution", so that they fit nicely on top of each other and the resemblance is visible. We therefore conclude that at constant values of Θ, the values of p and f are not significant as long as they dont get too small. But in the case of instantaneous fire propagation, $p = 1$ is the obvious choice to maximize simulation speed.

6.2.4 Effect of Θ

Since an SOC state is also characterized in a way that Θ should be the only relevant tuning parameter, we ran some bulk simulations to test this statement. The results can be seen in Fig.12.

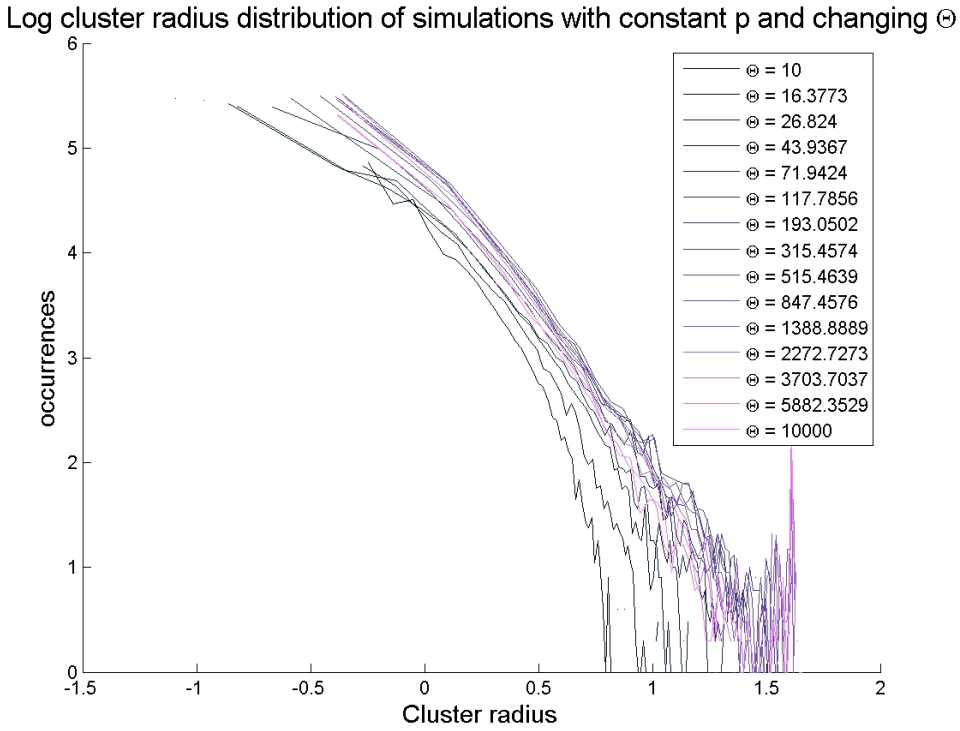

Figure 12: Log cluster radius distributions of simulations with constant p and changing Θ

For $\Theta > 70$ we start to see a very consistent slope. Having a lower value, the curves do not anymore fit the pattern and do not exhibit a straight slope. It is important to remember that we only focus on the shape and slope of the curve, not on the location.

6.2.5 Effect of the grid size

The third parameter having an impact on the simulation results was the grid size. Since we were applying periodic boundary conditions, it was not clear from the beginning as to what extent this parameter would be important. We run several simulations with all parameters except the grid size fixed. The results are shown in Fig.13.

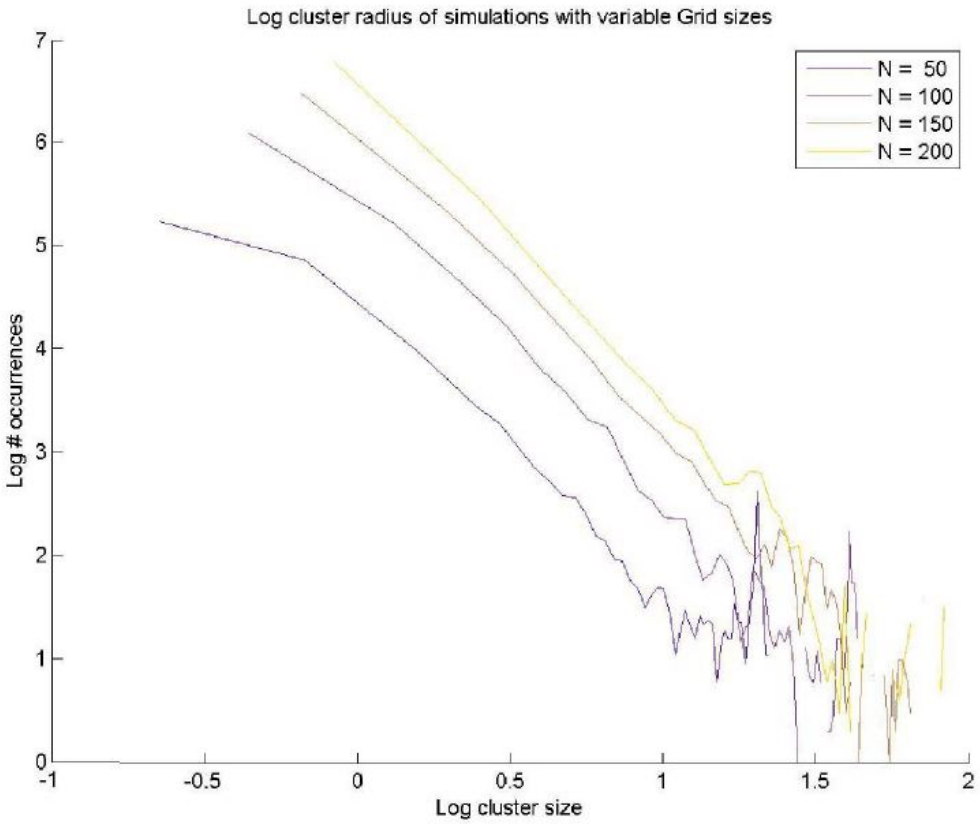

Figure 13: Log cluster radius distributions with variable grid sizes

Since all the different simulations yield the same slope, we conclude that the grid size only has a marginal impact on the qualitative simulation results.

6.2.6 Effect of the Initial conditions

One of the main properties of an SOC state is, that it is an attractor of the system dynamics, much like a minimum potential location. It is stable with respect to small disturbances. If that would not be the case, then the system would not evolve to that state in the first place. Now to check if our state can in fact be an SOC state, it needs to evolve to a certain point independant of the initial conditions. A property of the system, in the Limit $T_{burn} \to 0$, is, that it is in fact memoryless. Beside the current grid configuration, no other information is used to compute the next iteration. This basically means that if we initialize the system in any starting configuration and end up with an SOC state, we can look at all past grid configurations as initial

conditions for (smaller) simulations that have lead to an SOC state. In the course of our simulations, with the right choice of parameters, we thus have possibly created a significant fraction of all possible initial conditions (for a given grid), which have always lead to an SOC state. We therefore conclude that the simulation is *not* dependant on the initial conditions.

6.3 Research questions

To our understanding, the forest fire model exhibits SOC behavior independantly of all parameters except the p/f-ratio. Finding an appropriate value is often not too difficult. We have employed simulations with very high values $p/f > N^2$, leading the simulation to mostly filling the whole grid before a fire occurs. This does not result in an SOC behavior and is related to finite grid size effects. We have also tried to simulate with $p/f \to 1$ which has also not lead to satisfying results. We have found that choosing a value of around $p/f \approx N$ will let the system exhibit SOC behavior. A quantitative analysis of theis statement would be very time consuming because the critical threshholds of p/f are most probably dependant on the grid size and therefore have to be tested on a wide range of grid sizes.

7 Open Questions and Future Scope

There are certain other concepts which should be understood if one wants to understand Self Organized Criticality. One of the first things is from the basic ideas on SOC, we understand that the system should have certain power law behaviors. But how does one figure out which are the quantities one should be looking at, to find such power laws. One should try to make use of various mean field theories to find the critical exponents of these power laws.
The limited capability of the original model to distinguish an SOC state from other states is most probably because of limited computational capability. It would be an important question to be able to identify an SOC state from this model. Certain global aspects of SOC have to be searched by applying different boundary conditions and trying to employ different neighborhoods.
It is known[ref] that the SOC in Forest Fire model has an an upper critical dimension of 6 which means that for dimensions above 6, the critical exponents calculated from the mean field approaches become exact. In the field of physics, there is also a concept of lower critical dimension which implies that below this critical dimension, there is no phase change. It would be interesting to figure out whether there is any such thing in forest fire model as well.

8 Summary and Outlook

We were able to reproduce many of the results found in the literature. Looking at the plots, we see a very consistent picture. We have found very prominent power laws regarding the cluster radii and the cluster sizes.
Yet, after all these simulations, we are still not totally sure if we understood the concept correctly. Though we were able to reproduce the results, we have some doubts about the simulations. We were not able to completely distinguish different phases in the system or differ between equilibria and SOC states.
Programming in Matlab is a very fast approach to this sort of problem. It is incredibly convenient and offers a lot of useful built-in functions. But it obviously has its limits. The most prominent was the simulation time. We think that the simulation could have been implemented more efficiently using other programming environments like C++, but since Matlab was the topic of the course, we worked with that.
The course helped very much to make the program more efficient. We have learned a lot about how to implement efficient code, structuring data and organizing results in an efficient manner.

For lack of time, we were not able to test everything we wanted. What we feel is the most important thing that we missed, is the critical threshold at which a system turns into an SOC state. We were able to compute the critical f/p-ratio, but only for one given set of parameters. Since a single Simulation could take as long as half a day if the results had to be good, it was impossible for us to find the dependencies of the critical threshold with other parameters such as the grid size or the fire burning time. We faced similar problems in the implementation of the original model. The original model which is more dynamic and intuitive effectively helped us only in understanding the different states only qualitatively. Because of long computation time, we are unable to distinguish different phases using the original model.

The topic of self-organized criticality is an interesting and challenging one. It can explain phenomena which are not understood intuitively and is appliable to a wide variety of problems.

References

[1] Per Bak, Chao Tang, and Kurt Wiesenfeld. Self-organized criticality. *Phys. Rev. A*, 38:364–374, Jul 1988.

[2] Per Bak, Chao Tang, and Kurt Wiesenfeld. Self-organized criticality. *Phys. Rev. A*, 38:364–374, Jul 1988.

[3] Peter Grassberger and Holger Kantz. On a forest fire model with supposed self-organized criticality. *Journal of Statistical Physics*, 63:685–700, 1991.

[4] W.K. Moner, B. Drossel, and F. Schwabl. Computer simulations of the forest-fire model. *Physica A: Statistical Mechanics and its Applications*, 190(34):205 – 217, 1992.

[5] B. Drossel and F. Schwabl. Self-organized critical forest-fire model. *Phys. Rev. Lett.*, 69:1629–1632, Sep 1992.

[6] S. Clar, B. Drossel, and F. Schwabl. Scaling laws and simulation results for the self-organized critical forest-fire model. *Phys. Rev. E*, 50:1009–1018, Aug 1994.

[7] Siegfried Clar, Barbara Drossel, and Franz Schwabl. Forest fires and other examples of self-organized criticality. *Journal of Physics: Condensed Matter*, 8(37):6803, 1996.

A Forest Fire Model- Traditional Implementation

The following function is used to implement the basic Forest Fire model

```
function [Forest_grid,trees,empty,fires]=...
    basic_fire_model_smaller2(N,p,pbf,Iter,plotbit,Forest_grid,gridbit)
%Here we implement the forest fire model using the periodic boundary
%condition

% N— it defines the size of 2—dimensional grid

% p— Tree will be grown at an empty site if random number generated is
%less than p
% pbf is the ratio of p and f where f is defined below

% f— A green tree will start burning if either of its neighbours is already
%burning or a random number generated is less than the lightening parameter
%f

%Iter— it defines the number of time steps we want to evolve the system.
%If plotbit is 1, fires, trees and empty sites will be plotted as a
%function of iterations, otherwise not

%If gridbit is 1, the inital grid will be the one given as Forest_grid,
```

```
%otherwise it is generated with 50-50 prob to grow a tree or to have empty
%site

%0-tree
%1-empty site
%2-fire

f=p/pbf;
N=N+2; %We create a larger matrix, but we will update only the central
%portion of the grid
grid=rand(N);
Forest_grid(N,N)=0;

% The following double loop generates the initial forest with either empty
%sites or green trees (generated with equal probability). The probability
% can be varied

if gridbit~=1
m=2;
while m<N
    n=2;
    while n<N
        if grid(m,n)<0.5
            Forest_grid(m,n)=0;
        else
            Forest_grid(m,n)=1;
        end
        n=n+1;
    end
    m=m+1;
end
end

% colormap summer;
it=1;
trees(Iter-1)=0;   %it stores the fraction of trees for all the time steps
empty(Iter-1)=0;   %it stores the fract of empty sites for every time step
fires(Iter-1)=0;   %it stores the fraction of fires for all the time steps
burntcounter=0;
lightcounter=0;
while it<Iter
    Forest_grid_temp=Forest_grid;
    %Forest_grid_temp is used to create forest grid at the next time step.
    % we need to take care that the neighbors of every cell are correct
    %as based on the periodic boundary condition.
    Forest_grid_temp(1,2:N-1)=Forest_grid(N-1,2:N-1);
    Forest_grid_temp(N,2:N-1)=Forest_grid(2,2:N-1);
    Forest_grid_temp(2:N-1,1)=Forest_grid(2:N-1,N-1);
    Forest_grid_temp(2:N-1,N)=Forest_grid(2:N-1,2);
```

```
    grid=rand(N);
    %At every step, a matrix of random numbers of size N x N is generated.
    %It is used in 2 scenarios: either a green tree (lightening
    %probability) or an empty site (probability to grow tree)

    % Central N-1 x N-1 grid

    tre=0;      %counting trees at a times
    emp=0;      %counting empty sites at a times
    fir=0;      %counting fires at a time
    m=2;
    while m<N
        n=2;
        while n<N
%             we update cells depending on their previous state
            if Forest_grid_temp(m,n)==2
                fir=fir+1;
                Forest_grid(m,n)=1;


            elseif Forest_grid_temp(m,n)==1
                emp=emp+1;
                if grid(m,n)<p
                    Forest_grid(m,n)=0;
                end
            else
                tre=tre+1;
                if (grid(m,n)<f || Forest_grid_temp(m-1,n)==2 || ...
                        Forest_grid_temp(m+1,n)==2 ||...
                        Forest_grid_temp(m,n-1)==2 ||...
                        Forest_grid_temp(m,n+1)==2)
                    Forest_grid(m,n)=2;
                end
            end
            n=n+1;
        end
        m=m+1;
    end
%   image(25*(Forest_grid_temp(2:N-1,2:N-1)))
    trees(it)=tre;
    empty(it)=emp;
    fires(it)=fir;
%    getframe;
    it=it+1
end
if plotbit==1
    figure
    plot(trees/(N*N))
    hold on
```

```
    plot(empty/(N*N),'r')
    plot(fires/(N*N),'k')
    hleg1 = legend('Trees','Empty sites','Fires');
    set(hleg1,'Location','NorthEast')
    hold off
else
    trees=(trees(it-1))/(N*N);
    fires=(fires(it-1))/(N*N);
    empty=(empty(it-1))/(N*N);
end
end
```

The following function gives the distribution of frequency of clusters and the radius of clusters as a function of the size of clusters.

```
function [s,Clust2,Number,Radius]=cluster_disrtibution2(Forest_grid,Nd)
%This code is written for periodic boundary conditions
%The code with periodic boundary condition works with a grid whose size
%is larger by 2 in each of the dimensions.

%Forest_grid is the grid which we want to analyze.

% Nd is size of the square lattice, but increased by 2. This is because it
%is used with the code "basic_fire_model_smaller2" which generates a
% lattice whose size is 2 larger than the expected size.
N=Nd-1;
N2=N-1;
Forest_grid=Forest_grid(2:N,2:N); %We extract the exact size of Forest-grid
pause
Index=3;
%In the following double loop, we mark every site where a green tree is
%there with a new Index if it is a unindexed green tree and the unindexed
%neighbours (which are green trees) of a green tree with the same index as
%the green tree or the minimum index among this tree and its neighbours
i=1;
while i<N
    j=1;
    while j<N
        if Forest_grid(i,j)==0 || Forest_grid(i,j)>2
           [v1,v2,v3,v4]=neighbour(i,j,N2);
           %we caclulate the indices of all the neighbours of a cell
           m=converter(Forest_grid(i,j));
           m1=converter(Forest_grid(v1(1),v1(2)));
           m2=converter(Forest_grid(v2(1),v2(2)));
           m3=converter(Forest_grid(v3(1),v3(2)));
           m4=converter(Forest_grid(v4(1),v4(2)));
           minimum=min([m,m1,m2,m3,m4]);
           %All the cells which are not green or unindexed becomes
           %infinity due to converter function
```

```
            if minimum<Inf
                %We are handeling the case when atleast one of the
                %neighbours has an index already.

                Forest_grid(i,j)=minimum;
                %The current cell is awarded this minimum index
                % In the fillowing 4 if statements, the minimum index is
                %given to each of the cells which have a green tree.
                if Forest_grid(v1(1),v1(2))==0 ||...
                        Forest_grid(v1(1),v1(2))>minimum
                    Forest_grid(v1(1),v1(2))=minimum;
                end
                if Forest_grid(v2(1),v2(2))==0 ||...
                        Forest_grid(v2(1),v2(2))>minimum
                    Forest_grid(v2(1),v2(2))=minimum;
                end
                if Forest_grid(v3(1),v3(2))==0 ||...
                        Forest_grid(v3(1),v3(2))>minimum
                    Forest_grid(v3(1),v3(2))=minimum;
                end
                if Forest_grid(v4(1),v4(2))==0 ||...
                        Forest_grid(v4(1),v4(2))>minimum
                    Forest_grid(v4(1),v4(2))=minimum;
                end
            else
                Forest_grid(i,j)=Index;
                % In the fillowing 4 if statements, the index is given to
                %each of the cells which have a green tree.
                if Forest_grid(v1(1),v1(2))==0
                    Forest_grid(v1(1),v1(2))=Index;
                end
                if Forest_grid(v2(1),v2(2))==0
                    Forest_grid(v2(1),v2(2))=Index;
                end
                if Forest_grid(v3(1),v3(2))==0
                    Forest_grid(v3(1),v3(2))=Index;
                end
                if Forest_grid(v4(1),v4(2))==0
                    Forest_grid(v4(1),v4(2))=Index;
                end
                %Index tells about total number of different indices used
                Index=Index+1;
            end
         end
        j=j+1;
    end
    i=i+1;
end
%The above code is not fullproof, since it flawed because of the periodic
%boundary condition and method of indexing goes in 1 direction along the
```

```
%grid and it does not cover all the nearest neighbours in the periodic
%boundary conditions
noter=0; %noter keeps track of the redundancy
i=1;
while i<N
    j=1;
    while j<N
        if Forest_grid(i,j)>2
            [v1,v2,v3,v4]=neighbour(i,j,N2);
            m=Forest_grid(i,j);
            m1=converter(Forest_grid(v1(1),v1(2)));
            m2=converter(Forest_grid(v2(1),v2(2)));
            m3=converter(Forest_grid(v3(1),v3(2)));
            m4=converter(Forest_grid(v4(1),v4(2)));
            minimun=min([m,m1,m2,m3,m4]);
            %"changer" helps to change replace an index by another index.
            if minimun<Inf
            if minimun==m
                [Forest_grid,noter]=...
                    changer(Forest_grid,m1,m2,m3,m4,minimun,N2,noter);
            elseif minimun==m1
                [Forest_grid,noter]=...
                    changer(Forest_grid,m,m2,m3,m4,minimun,N2,noter);
            elseif minimun==m2
                [Forest_grid,noter]=...
                    changer(Forest_grid,m1,m,m3,m4,minimun,N2,noter);
            elseif minimun==m3
                [Forest_grid,noter]=...
                    changer(Forest_grid,m1,m2,m,m4,minimun,N2,noter);
            elseif minimun==m4
                [Forest_grid,noter]=...
                    changer(Forest_grid,m1,m2,m3,m,minimun,N2,noter);
            end
            end
        end
        j=j+1;
    end
    i=i+1;
end
NofClust=Index-3;
%There is a redundency in the total number of indices which can be removed
[row_shift,col_shift]=correct_radius_shift_getter(Forest_grid,Index-1,N2);
%Now, we calculate the radius of different clusters
Clust(NofClust)=0;       %It counts the number of trees in all the clusters
FirstMoment(NofClust,2)=0; %First moment generates the mean
SecondMoment(NofClust,2)=0;
%It calsulates the second order moment which can be used to calculate the
%radius of a cluster.
%The 2nd index in above 2 variables is for 2 directions (x and y)
i=1;
```

```
while i<N
    j=1;
    while j<N
        if Forest_grid(i,j)>2
            cl=Forest_grid(i,j)-2;
            Clust(cl)=Clust(cl)+1;
            row=mod(i+row_shift(cl),N2);
            col=mod(j+col_shift(cl),N2);
            FirstMoment(cl,1)=FirstMoment(cl,1)+row;
            FirstMoment(cl,2)=FirstMoment(cl,2)+col;
            SecondMoment(cl,1)=SecondMoment(cl,1)+(row*row);
            SecondMoment(cl,2)=SecondMoment(cl,2)+(col*col);
        end
        j=j+1;
    end
    i=i+1;
end
NofClust2=Index-3-noter;%The actual number of clusters

Clust2(NofClust2)=0;
%It stores the number of trees in all the clusters without any redundancy

Size_clust(NofClust2)=0;%It stores the radius of all the clusters
i=1;
j=1;
while i<=NofClust
    if Clust(i)~=0  %Here we are excluding the cluster sizes whose frequency is zero.
        Clust2(j)=Clust(i);
        No=Clust(i);
        Size_clust(j)=(((SecondMoment(i,1)-((FirstMoment(i,1)*...
            FirstMoment(i,1))/No))+(SecondMoment(i,2)-...
            ((FirstMoment(i,2)*FirstMoment(i,2))/(No))))/No)^0.5;
        j=j+1;
    end
    i=i+1;
end
Clust2=Clust2(1:(j-1));
Size_clust=Size_clust(1:(j-1));
hist(Clust,20)
figure
hist(Clust2)
[s,Number,Radius]=radius_size_distribution(Clust2,Size_clust,j-1);
end
```

The following function is used by the *cluster_distribution2.m* and *cluster_distribution2_iter.m* function

```
function [row_shift,col_shift]=correct_radius_shift_getter(Forest_grid,K,N)
%Cluster number goes from 3 to K
```

```
%In order to find the right column shift, we scan through all the rows for
%a given column and in the external loop we change the columns.
%The value of variable F decides the structure of a cluster: whether it is
%making use of the periodic boundary or not.

%Initially, for every cluster, F is 0. If the first element of a cluster is
%not there in the first column, then this cluster cant be using the
%periodic boundary. We make F for this cluster to be 3.

%In case, it is the first row, we make F=1, because now we don't know
%whether it makes use of the bounadry or not. We also record the latest
%entry of a cluster in ml.

%If we find that F=1 for a cluster, we compare the current column with the
%last column where an element of this cluster is found. In case, it is the
%previous cluster, we go on by updating the ml.

%On the other hand, if the last column where an element of cluster was
%found is not the previous column, we know that this cluster uses boundary
%conditions. In this case,we store this entry in mr and make value of F
%for this cluster=2.

%We know now that we need to do the shift only if F=2 for a cluster and
%amount of shift is N+1-mr

F(K-2)=0;
%For a given cluster, if F is 0; no element of this cluster is yet found.
%if F=1, begining has been found, but we don't know whether it uses
%periodic boundary or not.
%if F=2, the cluster uses periodic boundary condition
%if F=3, the first element of the cluster is not in the first row and
%hence it does not use the periodic boundary

ml(K-2)=0;
%In case F=1 for a cluster, it stores the latest column where
%an element of a cluster was found

mr(K-2)=0;
%In case F=2 for a cluster, it stores the starting column where an
%element of a cluster was found near the right boundary

col_shift(K-2)=0;
%it stores the shift needed for a column
j=1;
while j<=N
    i=1;
    while i<=N
        cl=Forest_grid(i,j);
        if cl>2
            cl=cl-2;
```

```
            if F(cl)==0
                if j~=1
                    F(cl)=3;
                else
                    F(cl)=1;
                    ml(cl)=j;
                end
            elseif F(cl)==1
                if j~=ml(cl)+1
                    mr(cl)=j;
                    F(cl)=2;
                    col_shift(cl)=N+1-mr(cl);
                else
                    ml(cl)=j;
                end
            end
        end
        i=i+1;
    end
    j=j+1;
end
F(:)=0;

ml(:)=0;
%In case F=1 for a cluster, it stores the latest row where an element
%of a cluster was found

mr(:)=0;
%In case F=2 for a cluster, it stores the starting row where an
%element of a cluster was found near the lower boundary

row_shift(K-2)=0;
%it stores the shift needed for a row
i=1;
while i<=N
    j=1;
    while j<=N
        cl=Forest_grid(i,j);
        if cl>2
            cl=cl-2;
            if F(cl)==0
                if i~=1
                    F(cl)=3;
                else
                    F(cl)=1;
                    ml(cl)=i;
                end
            elseif F(cl)==1
                if i~=ml(cl)+1
                    mr(cl)=i;
```

```
                    F(cl)=2;
                    row_shift(cl)=N+1-mr(cl);
                else
                    ml(cl)=i;
                end
            end
        end
        j=j+1;
    end
    i=i+1;
end
end
```

The following function is used by the *cluster_distribution2.m* function

```
function [s,Number,Radius]=...
    radius_size_distribution(cluster,size_clust,Nofclusters)
%This function gives number of clusters and radius of clusters as a
%function of cluster-size (where cluster size means the number of trees in
%the cluster

%cluster stores the number of tree in each of the clusters
%size_clust stores the radius of various clusters
%Nofclusters is the total number of clusters
K=max(cluster);
Number(K)=0;  %it stores the number of clusters of different number of trees
Radius(K)=0;
i=1;
while i<=Nofclusters
    Number(cluster(i))=Number(cluster(i))+1;
    Radius(cluster(i))=Radius(cluster(i))+size_clust(i);
    %we will take average of the radius in case there are more
    %than 1 cluster having a given number of trees
    i=i+1;
end
Number1(K)=0;
Radius1(K)=0;
s(K)=0;
i=1;
j=1;
while i<=K
    if Number(i)~=0
        s(j)=i;
        Number1(j)=Number(i);
        Radius1(j)=Radius(i)./Number(i);
        %we are doing the averaging excluding the cases where we have
        %no clusters or 1 cluster of a given size
        j=j+1;
    end
```

```
    i=i+1;
end
j=j-1;
s=s(1:j);
Number=Number1(1:j);
Radius=Radius1(1:j);
end
```

The following function is used by the $basic\_fire\_model\_smaller2.m$, $cluster\_distribution2.m$, $basic\_fire\_model\_smaller2\_iter.m$ and $cluster\_distribution2\_iter.m$ function

```
function [v1,v2,v3,v4]=neighbour(i,j,N)
%This function calculates the neighbourhood cells of
%a cell represented by [i,j] in the periodic
%boubdary condition with the size of the grid as N x N
if i>1 && i<N
    v1=[i-1,j];
    v2=[i+1,j];
    if j>1 && j<N
        v3=[i,j-1];
        v4=[i,j+1];
    elseif j==1
        v3=[i,N];
        v4=[i,j+1];
    elseif j==N
        v3=[i,1];
        v4=[i,j-1];
    end
elseif i==1
    v1=[N,j];
    v2=[i+1,j];
    if j>1 && j<N
        v3=[i,j-1];
        v4=[i,j+1];
    elseif j==1
        v3=[i,N];
        v4=[i,j+1];
    elseif j==N
        v3=[i,1];
        v4=[i,j-1];
    end
else
    v1=[i-1,j];
    v2=[1,j];
    if j>1 && j<N
        v3=[i,j-1];
        v4=[i,j+1];
    elseif j==1
        v3=[i,N];
```

```
        v4=[i,j+1];
    elseif j==N
        v3=[i,1];
        v4=[i,j-1];
    end
end
end
```

The following function is used by the *cluster_distribution2.m*, *basic_fire_model_smaller2_iter.m* and *cluster_distribution2_iter.m* function

```
function m1=converter(m1)
%This function returns inf if a site does
%not contain unindexed green tree
if m1<3
    m1=Inf;
end
```

The following function is used by the *cluster_distribution2.m*, *basic_fire_model_smaller2_iter.m* and *cluster_distribution2_iter.m* function

```
function [Forest_grid,noter]=...
    changer(Forest_grid,m1,m2,m3,m4,minimum,N2,noter)
% This function replaces all the neighbours of a cell with the minimum
% index among these cells.
%Care has been taken that the cells which are on fire or without trees are
%not indexed through the converter function.
% Index=Forest_grid(v1(1),v1(2));
if m1<Inf && m1>minimum
     [Forest_grid]=enforcer(Forest_grid,minimum,m1,N2);
%     Forest_grid=Forest_grid-((m1-minimum)*(ismember(Forest_grid,m1)));
    %     mistake(noter)=m1;
    noter=noter+1;
end
if m2<Inf && m2>minimum && m2~=m1
     [Forest_grid]=enforcer(Forest_grid,minimum,m2,N2);
%     Forest_grid=Forest_grid-((m2-minimum)*(ismember(Forest_grid,m2)));
%     mistake(noter)=m2;
    noter=noter+1;
end
if m3<Inf && m3>minimum && m3~=m1 && m3~=m2
     [Forest_grid]=enforcer(Forest_grid,minimum,m3,N2);
%     Forest_grid=Forest_grid-((m3-minimum)*(ismember(Forest_grid,m3)));
%     mistake(noter)=m3;
    noter=noter+1;
end
if m4<Inf && m4>minimum && (m4~=m1 && m4~=m2 && m4~=m3)
     [Forest_grid]=enforcer(Forest_grid,minimum,m4,N2);
```

```
%     Forest_grid=Forest_grid-((m4-minimum)*(ismember(Forest_grid,m1)));
    %     mistake(noter)=m4;
    noter=noter+1;
end
end
```

The following function is used by the *cluster_distribution2.m*, *basic_fire_model_smaller2_iter.m* and *cluster_distribution2_iter.m* function

```
function [Forest_grid]=enforcer(Forest_grid,In_small,In_large,N)
%It replaces all the cells with index In_large with the index In_small
%This is the inefficient guy!!!!
%It takes an index (In_large) and replaces all the cells with that index by
%a smaller index (In_small)
i=1;
    while i<=N
        j=1;
        while j<=N
            if Forest_grid(i,j)==In_large
                Forest_grid(i,j)=In_small;
            end
            j=j+1;
        end
        i=i+1;
    end
% end
end
```

The following function is used to calculate the various distributions by summing the distributions of all the steps (this is required for SOC).

```
function [K,Number,Radius,Forest_grid,trees,empty,fires]=...
    basic_fire_model_smaller2_iter(N,p,pbf,Iter,Forest_grid,gridbit)
%Here we implement the forest fire model using the periodic boundary cond.
% N- it defines the size of 2-dimensional grid
% p- Tree will be grown at an empty site if random number generated is less
%than p
% pbf is the ratio of p and f where f is defined below
% f- A green tree will start burning if either of its neighbours is already
%burning or a random number generated is less than the lightening parameter f
%Iter- it defines the number of time steps we want to evolve the system.
%If gridbit is 1, the inital grid will be the one given as Forest_grid,
%otherwise it is generated with 50-50 prob to grow a tree or to have empty
%site

%0-tree
%1-empty site
%2-fire
```

```
%This code is specifically written to calculate various distributions at
%every step and then average over all of them.
%This is important since in the SOC state, the distributions change in every step.

%It_steps store is a vector which stores the correct time for measuring the
%various distributions (associated index is "me")
It_steps=linspace(201,Iter,Iter-200);
f=p/pbf;
N2=N;
N=N+2;
%We create a larger matrix, but we will update only the central portion of the grid
grid=rand(N);
Forest_grid(N,N)=0;

% The following double loop generates the initial forest with either empty
%sites or green trees (generated with equal probability). The probability
% can be varied

if gridbit~=1
pause
m=2;
while m<N
    n=2;
    while n<N
        if grid(m,n)<0.5
            Forest_grid(m,n)=0;
        else
            Forest_grid(m,n)=1;
        end
        n=n+1;
    end
    m=m+1;
end
end
K=0;
Number=0;
Radius=0;
it=1;
me=1;
Index_grid(N-2,N-2)=0;
trees(Iter-1)=0; %it stores the fraction of trees for all the time steps
empty(Iter-1)=0; %it stores the fraction of empty sites for every time steps
fires(Iter-1)=0; %it stores the fraction of fires for all the time steps
while it<Iter
    tic
    Forest_grid_temp=Forest_grid;
    %Forest_grid_temp is used to create the forest grid at next time step.
    Forest_grid_temp(1,2:N-1)=Forest_grid(N-1,2:N-1);
    Forest_grid_temp(N,2:N-1)=Forest_grid(2,2:N-1);
```

```
    Forest_grid_temp(2:N-1,1)=Forest_grid(2:N-1,N-1);
    Forest_grid_temp(2:N-1,N)=Forest_grid(2:N-1,2);

    grid=rand(N);
    %At every step, a matrix of random numbers of size N x N is generated.
    %It is used in 2 scenarios: either a green tree (lightening
    %probability) or an empty site (probability to grow tree)

    if It_steps(me)==it%measuring various distributions
        Index=3;
        Index_grid(:,:)=0;
    end
    tre=0;      %counting trees at a time step
    emp=0;      %counting empty sites at a time step
    fir=0;      %counting fires at a time
    i=2;
    while i<N
        j=2;
        while j<N
%             we update cells depending on their previous state
            if Forest_grid_temp(i,j)==2
                fir=fir+1;
                Forest_grid(i,j)=1;
            elseif Forest_grid_temp(i,j)==1
                emp=emp+1;
                if grid(i,j)<p
                    Forest_grid(i,j)=0;
                end
            else
                tre=tre+1;
                if (grid(i,j)<f || Forest_grid_temp(i-1,j)==2 || ...
                        Forest_grid_temp(i+1,j)==2 ||...
                   Forest_grid_temp(i,j-1)==2 || Forest_grid_temp(i,j+1)==2)

                    Forest_grid(i,j)=2;

                end
                if It_steps(me)==it
                    i1=i-1;
                    j1=j-1;
                    if Forest_grid_temp(i,j)==0 || Index_grid(i1,j1)>2
                        [v1,v2,v3,v4]=neighbour(i1,j1,N2);
                        m=converter(Forest_grid_temp(i1,j1));
                        m1=converter(Forest_grid_temp(v1(1),v1(2)));
                        m2=converter(Forest_grid_temp(v2(1),v2(2)));
                        m3=converter(Forest_grid_temp(v3(1),v3(2)));
                        m4=converter(Forest_grid_temp(v4(1),v4(2)));
                        minimum=min([m,m1,m2,m3,m4]);
                        %All the cells which are not green or unindexed
                        %becomes infinity due to converter function
```

```
                if minimum<Inf
                    %We are handeling the case when atleast one of
                    %the neighbours has an index already.
                    Index_grid(i1,j1)=minimum;
                    %The current cell is awarded this minimum index
                    % In the fillowing 4 if statements, the minimum
                    %index is given to cell which has a green tree.
                    if Forest_grid_temp(v1(1),v1(2))==0 || ...
                            Index_grid(v1(1),v1(2))>minimum

                        Index_grid(v1(1),v1(2))=minimum;
                    end
                    if Forest_grid_temp(v2(1),v2(2))==0 || ...
                            Index_grid(v2(1),v2(2))>minimum

                        Index_grid(v2(1),v2(2))=minimum;
                    end
                    if Forest_grid_temp(v3(1),v3(2))==0 || ...
                            Index_grid(v3(1),v3(2))>minimum

                        Index_grid(v3(1),v3(2))=minimum;
                    end
                    if Forest_grid_temp(v4(1),v4(2))==0 || ...
                            Index_grid(v4(1),v4(2))>minimum

                        Index_grid(v4(1),v4(2))=minimum;
                    end
                else
                    Index_grid(i1,j1)=Index;
                    % In the fillowing 4 if statements,the index
                    %is given to each cell which has green tree.
                   if Forest_grid_temp(v1(1),v1(2))==0
                        Index_grid(v1(1),v1(2))=Index;
                   end
                    if Forest_grid_temp(v2(1),v2(2))==0
                        Index_grid(v2(1),v2(2))=Index;
                    end
                    if Forest_grid_temp(v3(1),v3(2))==0
                        Index_grid(v3(1),v3(2))=Index;
                    end
                    if Forest_grid_temp(v4(1),v4(2))==0
                        Index_grid(v4(1),v4(2))=Index;
                    end
                    %Index tells about the total number of
                    %different indices used
                    Index=Index+1;
                end
            end
        end
    end
```

```
            j=j+1;
        end
        i=i+1;
    end
    if It_steps(me)==it
        [K1,Number1,Radius1]=cluster_disrtibution2_iter(Index_grid,Index,N);
%         K1
        if K<K1
            K=K1;
            Number(K)=0;
            Radius(K)=0;
        else
            Number1(K)=0;
            Radius1(K)=0;
        end
        Number=Number+Number1;
        Radius=Radius+Radius1;
        me=me+1;
    end
    trees(it)=tre;
    empty(it)=emp;
    fires(it)=fir;
    it=it+1
    toc
end
    figure
    plot(trees/(N*N))
    hold on
    plot(empty/(N*N),'r')
    plot(fires/(N*N),'k')
    hleg1 = legend('Trees','Empty sites','Fires');
    set(hleg1,'Location','NorthEast')
    hold off
s=1:K;
figure
plot(log10(s),log10(Radius./Number))
figure
plot(log10(s),log10(Number/(Number(1))))
end
```

The following function is used by the *cluster_distribution2_iter.m* function

```
function [K,Number,Radius]=cluster_disrtibution2_iter(Forest_grid,Index,Nd)
%This code does the right cluster indexing of the grid which has been once
%indexing, but there are some redundencies in it.
%The code with periodic boundary condition works with a grid whose size is
%larger by 2 in each of the dimensions.

%Forest_grid is the grid which we want to analyze.
```

```
% Nd is size of the square lattice, but increased by 2. This is because it
%is used with the code "basic_fire_model_smaller2" which generates a
% lattice whose size is 2 larger than the expected size.
N=Nd-1;
N2=N-1;
noter=0; %noter keeps track of the redundancy
i=1;
while i<N
    j=1;
    while j<N
        if Forest_grid(i,j)>2
            [v1,v2,v3,v4]=neighbour(i,j,N2); %we calculate all the neighbors
            m=Forest_grid(i,j);
            m1=converter(Forest_grid(v1(1),v1(2)));
            m2=converter(Forest_grid(v2(1),v2(2)));
            m3=converter(Forest_grid(v3(1),v3(2)));
            m4=converter(Forest_grid(v4(1),v4(2)));
            minimun=min([m,m1,m2,m3,m4]);
            %"changer" helps to change replace an index by another index.
            if minimun<Inf
            if minimun==m
                [Forest_grid,noter]=changer(Forest_grid,m1,...
                m2,m3,m4,minimun,N2,noter);
            elseif minimun==m1
                [Forest_grid,noter]=changer(Forest_grid,m,...
                m2,m3,m4,minimun,N2,noter);
            elseif minimun==m2
                [Forest_grid,noter]=changer(Forest_grid,m1,...
                m,m3,m4,minimun,N2,noter);
            elseif minimun==m3
                [Forest_grid,noter]=changer(Forest_grid,m1,...
                m2,m,m4,minimun,N2,noter);
            elseif minimun==m4
                [Forest_grid,noter]=changer(Forest_grid,m1,...
                m2,m3,m,minimun,N2,noter);
            end
            end
        end
        j=j+1;
    end
    i=i+1;
end
NofClust=Index-3;
%There is a redundency in the total number of indices which can be removed

[row_shift,col_shift]=correct_radius_shift_getter(Forest_grid,Index-1,N2);
%this calculates the correct shift required for evert cluster to find the
%cluster radius appropriately.
```

```
%Now, we calculate the radius of different clusters
Clust(NofClust)=0; %It counts the number of trees in all the clusters
FirstMoment(NofClust,2)=0; %First moment generates the mean
SecondMoment(NofClust,2)=0;
%It calsulates the second order moment which can
%be used to calculate the radius of a cluster.
%The 2nd index in above 2 variables is for 2 directions (x and y)
i=1;
while i<N
    j=1;
    while j<N
        if Forest_grid(i,j)>2
            cl=Forest_grid(i,j)-2;
            row=mod(i+row_shift(cl),N2);
            col=mod(j+col_shift(cl),N2);
            Clust(cl)=Clust(cl)+1;
            FirstMoment(cl,1)=FirstMoment(cl,1)+row;
            FirstMoment(cl,2)=FirstMoment(cl,2)+col;
            SecondMoment(cl,1)=SecondMoment(cl,1)+(row*row);
            SecondMoment(cl,2)=SecondMoment(cl,2)+(col*col);
        end
        j=j+1;
    end
    i=i+1;
end
NofClust2=Index-3-noter;            %The actual number of clusters
Clust2(NofClust2)=0;
%It stores the number of trees in all clusters without any redundancy
Size_clust(NofClust2)=0;  %It stores the radius of all the clusters
i=1;
j=1;
while i<=NofClust
    if Clust(i)~=0
        %Here we are excluding the cluster sizes whose frequency is zero.
        Clust2(j)=Clust(i);
        No=Clust(i);
        Size_clust(j)=(((SecondMoment(i,1)-((FirstMoment(i,1)*...
        FirstMoment(i,1))/No))+(SecondMoment(i,2)-((FirstMoment(i,2)*...
        FirstMoment(i,2))/(No))))/No)^0.5;
        j=j+1;
    end
    i=i+1;
end
Clust2=Clust2(1:(j-1));
[K,Number,Radius]=radius_size_distribution_iter(Clust2,Size_clust,j-1);
end
```

The following function is used by the *cluster_distribution2_iter.m* function

```
function [K,Number,Radius]=...
    radius_size_distribution_iter(cluster,size_clust,Nofclusters)
%This function gives number of clusters and radius of clusters as a
%function of cluster-size (where cluster size means the number of trees in
%the cluster)

%cluster stores the number of tree in each of the clusters
%size_clust stores the radius of various clusters
%Nofclusters is the total number of clusters
K=max(cluster);
Number(K)=0;%it stores the number of clusters of different number of trees
Radius(K)=0;
i=1;
while i<=Nofclusters
    Number(cluster(i))=Number(cluster(i))+1;
    Radius(cluster(i))=Radius(cluster(i))+size_clust(i);
    %If we have more than 1 cluster of a given size, we just add all such
    %radii and in the end, we will divide with it by the total number of
    %clusters of that size
    i=i+1;
end
end
```

B Instantaneous fire implementation

```
%% FORFIRE

%%
function [data]= ForFire (GSize, f, p, t, d_rate,output)
% Syntax: ForFire(Grid Size, Lightning parameter, Growth parameter,
% Simulation time, Diagnostics rate, Output option)
%% Input variables
% The input variables are:
% GSize (for creating a grid GSize x GSize
% f, the fire parameter (chance of spontaneous fire)
% p, the growth parameter
% t, the simulation time
% d_rate, the diagnostics rate (set it to 50 or more for performance
% issues)
% output, the option of having a command window output (set it to 1) which
% tells the current state of the simulation, or having no output (set it to
% 0) which is much more convenient when doing parameter sweeps. Note that
% the output does not have a relevant effect on the simulation duration.
if output
fprintf('Initializing Simulation...\n');
tic
```

```
end
aviObj=avifile('movie2','compression','Cinepak','fps',20,'keyframe',1);
handle=figure;
hold on;
colormap(hsv);
cmap=colormap;
cmap=[cmap;cmap;cmap];
cmap(1,:)=[189/256 183/256 107/256];
cmap(2,:)=[0 0.6 0];
colormap(cmap);

%% Initialization
% Both the original and the shadow grid are initialized empty and global.
% The original grid contains the information about the state of a cell
% The shadow grid contains the information about the cluster Index of a
% cell.
global grid
grid=zeros(GSize);
global shgrid
shgrid=grid;
% The index matrix contains all the possible indexes and their availability
% where 1 means that the index is already in use and 0 means that the index
% is currently not in use
% There are GSize^2/2 possible clusters (in a perfect
% chessboard-configuration)
% Index(1,k) returns the k-th index
% Index(2,k) returns the availability
indexes=1:ceil(GSize^2/2);
availability=zeros(1,ceil(GSize^2/2));
global Index
Index=[indexes;availability];

% initialize the vector which holds all the sizes of all the clusters over
% all timesteps. Since we do not know the amount of clusters we are going
% to get, it is impossible to initialize it to the correct size. The
% Performance issues are marginal.
sizevec=0;
% initialize the vector which holds all the radii of all the clusters over
% all timesteps. Same problem as above applies.
radvec=0;
% Initialize the vector which will store the amount of alive trees over all
% timesteps. Since we only measure this property at distinct intervals, its
% length is well known.
Ntrees=zeros(1,t/d_rate);
% Ugly help variable for later...
op=0;


%% Loop over all timesteps
for i=1:t
```

```
% Pick a random cell to work on
x=ceil(GSize*rand());
y=ceil(GSize*rand());
% Generate random number for growth and fire possibilities
valrand=rand();
% Check if site is empty
if grid(x,y)==0
    % Check if growth is possible
    if valrand<p
        % Grow a tree
        grid(x,y)=1;
        % Updating the shadowgrid
        [a,b,c,d]=GetNcIndex(x,y);
        % if all neighbors are empty, set the index to the smallest
        % possible
        if ((a+b+c+d)==0)
            shgrid(x,y)=lowIndex();
            %fprintf('cell at %d,%d has no indexed neighbors and has
            %been set to %d \n',x,y,shgrid(x,y));
        end
        % if only one neighbor is already indexed
        % note that the program does NOT need to know which one it is,
        % since the condition is satisfied iff exactly one of them is
        % different from zero which automatically gives the max value
        if ((max([a,b,c,d])==(a+b+c+d) )&& max([a,b,c,d])~=0)
            shgrid(x,y)=max([a,b,c,d]);
            %fprintf('cell at %d, %d has one indexed neighbor and has
            %been set to %d \n',x,y,shgrid(x,y));
        end
        % if more than one neighbors are already indexed, the lowest
        % will be chosen and the other indexed cluster will be changed
        % to this one too
        if (max([a,b,c,d])<(a+b+c+d))
            % check for empty ones
            % the resulting vector cInd holds all the relevant indexes.
            % meaning that the empty ones are thrown out
   %fprintf('cell at %d,%d has more than two indexed neighbors ', x,y);
            cInd=0;
            if a>0
                cInd=a;
            end
            if b>0
                cInd=[cInd b];
            end
            if c>0
                cInd=[cInd c];
            end
            if d>0
                cInd=[cInd d];
```

```
                end
                % ugly workaround to check if a was zero --> gives nasty
                % errors if not checked.
                if cInd(1)==0
                    cInd=cInd(2:size(cInd,2));
                end
                % set the current grid cell to the lowest available Index
                shgrid(x,y)=min(cInd);


                % for all the other clusters, the indices are changed to
                % the new, common index.
                for k=1:size(cInd,2)
                    changeIndex(cInd(k),min(cInd));
                end
            end
        end
    end
    %check if Grid point is a tree
    if grid(x,y)==1
        % if the condition for fire is satisfied
        if valrand<f
            % burn the cluster. See description in the function for closer
            % insight.
        clusterburn(x,y);
        end
    end

    % Since the diagnostics function is by far the most time-consuming part
    % of the simulation, it is only called at distinct timesteps with a
    % period of d_rate.
    if mod(i,d_rate)==0
        % run the diagnostics and retrieve the data.
    [cSize,cRadius,N]=diagnostics();
    % store the data in a bigger vector. It is not necessary to sort this,
    % since it will only be used in histograms later on. Note that it is
    % impossible to determine the size of these vectors at the beginning of
    % the program, therefore they change their size in every loop. The
    % effects on the overall performance of the program are though
    % marginal.
sizevec=[sizevec; cSize];
radvec=[radvec; cRadius];
Ntrees(i/d_rate)=N;
% for every percent the simulation advances, there is a print at the
% command window. this can be suppressed with setting the output option
% to zero.
op=op+1;
if mod(op,t/d_rate/100)==0
    if output
fprintf('Simulation is at %d percent, estimated time left: %d seconds\n',...
```

```
   round(i/t*100), round((t-i)/i*toc));
       end
   op=0;
   end
    end


  % commented out section for live-plotting of the grid, the shadow grid
  % and histograms of the radius and size distribution
  if i>5000
    subplot(1,2,1);
    image(shgrid);
    axis square;
    subplot(1,2,2);
    image(grid+1);
    axis square;
%    subplot(2,2,3);
%    hist(cRadius);
%    subplot(2,2,4);
%    hist(cSize);
    M=getframe(handle);
    aviObj=addframe(aviObj,M);
  end
%    end
  %
  %


end
aviObj=close(aviObj);
% applying a low-pass filter to the tree evolution data for smoother
% visualization
fourierTrees=fft(Ntrees);
for i=50:size(fourierTrees,2)
    fourierTrees(i)=0;
end
cleanTrees=ifft(fourierTrees);

%% Data Preview
if output
fprintf('preparing data preview...\n');
    figure('Name','Data Preview');

    subplot(2,2,1);
    hist(radvec,20);
    title('Radius distribution');
    xlabel('Radius');
    ylabel('# of measurments');
    subplot(2,2,2);
    hist(sizevec,20);
```

```
    title('Size distribution');
    xlabel('Cluster Size');
    ylabel('# of measurments');
    subplot(2,2,3);
    plot(Ntrees);
    title('# trees evolution');
    xlabel('time'); % note that this is not the simulation time but
    %the time divided by the diagnostics period.
    ylabel('Alive Trees');
    subplot(2,2,4);
    plot(real(cleanTrees));
    title('# trees evolution (denoised)')
    xlabel('time');
    ylabel('Alive Trees');

    fprintf('saving data...\n');

end

%% Saving Data in a custom struct
data=struct('rad_dist',radvec,'size_dist',sizevec,'trees',Ntrees,'d_period',d_rate);
Theta=p/f;
a=num2str(GSize);
b=num2str(Theta);
c=num2str(t);
i=0;
cd Results
% making a custom folder with information about the grid size, theta and
% the simulation time
folder_name=[a '_' b '_' c];
% if such a simulation has already been run, it just adds increasing
% numbers to the end of the folder name.
while exist(folder_name)==7
    d=num2str(i);
    folder_name=[a '_' b '_' c '_' d];
    i=i+1;
end
mkdir (folder_name)
cd (folder_name)
save(folder_name,'data');
% save the plots
[a,b]=hist(data.rad_dist,50);
a=log10(a);
b=log10(b);
radplot=figure;
plot(b,a);
title('Radius distribution','FontSize',15);
xlabel('log10(radius)','FontSize',15);
ylabel('log10(# occurrences)','FontSize',15);
hgsave(radplot,'RadiusDist','-v6');
```

```
close all

[a,b]=hist(data.size_dist,50);
a=log10(a);
b=log10(b);
sizeplot=figure;
plot(b,a);
title('Size distribution','FontSize',15);
xlabel('log10(size)','FontSize',15);
ylabel('log10(# occurrences)','FontSize',15);
hgsave(sizeplot,'SizeDist','-v6');
close all
cd ..
cd ..
if output
fprintf('simulation finished. \n');
end




end
```

```
function [cSize,cRadius,N] = diagnostics ()
% The diagnostics function can work independantly of the simulation, it
% does not in any way alter the grid or the shadow grid nor the Index
% Matrix.
% It computes the current Number of alive trees, a vector containing the
% cluster sizes and a vector containing the cluster radii.
global grid;
global shgrid;
global Index;
% retrieve number of trees alive
N=sum(sum(grid));
% retrieve all used indices
cIndex=find(Index(2,:));

% initialize size vector
cSize=zeros(size(cIndex,2),1);
% initialize radius vector
cRadius=zeros(size(cIndex,2),1);
%counter variable for the output

for i=1:size(cIndex,2);

    % retrieve cluster index
    ind=Index(1,cIndex(i));
    % The matrix cluster is a Matrix with the same size as the grid, but is
```

```
    % zero everywhere except the grid points where trees with the current
    % index are (value=1);
    cluster=ismember(shgrid,ind);
    % find cluster size

    cSize(i)=sum(sum(cluster));
    % x and y give the coordinates of the cluster cells.
    [x,y]=find(cluster);
    % x_center and y_center are the center of the cluster
    x_center=mean(x);
    y_center=mean(y);
    % initialize the distance vector as zero
    dist=zeros(size(x,1),1);
    % for every cluster cell, compute its distance to the cluster center
    for h=1:size(x)
        dist(h)=sqrt((x(h)-x_center)^2+(y(h)-y_center)^2);
    end

    % compute the cluster radius as the mean of all radii.
    cRadius(i)=round(sqrt((mean(dist.^2))));



end
% This step is necessary to make the above loop suitable for parallel
% execution. Otherwise, one could just have written for i=cIndex.
% Here, we find all non-zero entries in the cRadius and cSize vectors and
% condense them into smaller vectors which hold no zero-elements.
temp=find(cIndex);
cRadius=cRadius(temp);
cSize=cSize(temp);
```

```
function [] = changeIndex(from, to)
% the changeIndex function changes all cell indices "from" on the
% shadowgrid to "to".
global Index;
global shgrid;
% check if the change is not senseless
if (from~=to)
   % make Index available again
   Index(2,from)=0;
   % update shadow grid
   % the ismember-function returns a matrix the same size as shgrid with
   % everything being zero except the entries where cells had the value
   % "from" (value=1). Multiplied by the difference from-to, we can
   % subtract it from the original grid and therefore change all the values
   % "from" to "to".
   shgrid=shgrid-ismember(shgrid,from)*(from-to);
```

```
end
```

```
function [] = clusterburn(x,y)
% clusterburn takes the grid coordinates to be burned and resets the entire
% connected cluster
global Index
global grid
global shgrid
% retrieve the local cluster index
locind=shgrid(x,y);
% if the current local index is zero, then an error must have occurred,
% since it should always be greater than one. Otherwise, no tree should be
% at this position and the function should not have been called in the
% first place!
if locind==0
fprintf('current cell is %d,%d.An error must have occurred.Cell Index was 0. \n',x,y);
end
% resetting the index
Index(2,locind)=0;
% update the grid: simply subtract the cluster from the grid
grid=grid-ismember(shgrid,locind);
% update the shadow grid: simply subtract the cluster*its value from the
% grid.
shgrid=shgrid-ismember(shgrid,locind)*locind;


end
```

```
function [ind_a,ind_b,ind_c,ind_d]= GetNcIndex (x,y)
% The function GetNcIndex takes the local coordinates and returns the
% cluster indices of the neighboring cells

global shgrid;
% call the neighbor function to retrieve the neighboring cell coordinates
% with periodic boundary conditions
[a,b,c,d,e,f,g,h]=neighbor(x,y,size(shgrid,1),size(shgrid,2));
% the neighbor indices are simply the entries of the shadow grid at the
% neighboring coordinates.
% Note that special cases such as some of the indices being zero is treated
% in the main program.
ind_a=shgrid(a,b);
ind_b=shgrid(c,d);
ind_c=shgrid(e,f);
ind_d=shgrid(g,h);
end
```

```
function [ind] = lowIndex()
% The function lowIndex returns the lowest available index.

global Index;

% the find function returns the coordinate of the first non-zero-element of
% a vector or matrix. since we used 1 for unavailable and 0 for available,
% we had to subtract 1 to make 0 unavailable and -1 available. so the first
% time, the find-function hits a value -1, it returns the lowest available
% index.
ind=find(Index(2,:)-1);
% in fact, find returns all the available indexes, so we just take the
% first one (being the lowest since the index matrix is sorted by
% construction)
ind=ind(1);
% Since the function is only called if the index is being used, we set it
% as unavailable.
Index(2,ind)=1;
end
```

```
function [a,b,c,d,e,f,g,h] = neighbor(j,k,m,n)
% This sub-function returns the 1st order moore neighbourhood for a grid
% cell [j,k] on a mxn grid with periodic bc. [a,b]: West; [c,d]: North;
% [e,f]: East; [g,h]: South;
% Treat the standard case
a=j-1;
b=k;
c=j;
d=k-1;
e=j+1;
f=k;
g=j;
h=k+1;
% treat special cases is the current grid cell is a boundary cell.
if j==1
    a=m;
end
if j==m
    e=1;
end
if k==1
    d=n;
end
if k==n
    h=1;
end
```

```
function [Rad] = computeradius(x,y)
global grid;
Rad=0;

    %% New Implementation
    % just check for the condition that the cluster is touching both
    % boundaries; if it does not, stop the loop!
    % check if the cluster stretches from the far left to the far right
    iter=0;
    while(max(x)-min(x)+1==size(grid,1)&& iter<size(grid,1))
        % "flip" the lowest coordinates. They are the boundary cells (check
        % in the while-statement)
        x(ismember(x,min(x)))=x(ismember(x,min(x)))+size(grid,1);
        iter=iter+1;
    end
    % check if the cluster stretches from the top to the bottom
    iter=0;
    while(max(y)-min(y)+1==size(grid,2)&&iter<size(grid,1))
        %flip the lowest coordinates
        y(ismember(y,min(y)))=y(ismember(y,min(y)))+size(grid,2);
        iter=iter+1;
    end
    %do the computation
    x_center=mean(x);
    y_center=mean(y);
    dist=zeros(size(x,1),1);
    for h=1:size(x,1)
        dist(h)=sqrt((x(h)-x_center)^2+(y(h)-y_center)^2);
    end
    Rad=sqrt(mean(dist.^2));
end
```