**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. 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. (More on flicker noise later!). 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   What is Self-Organized Criticality

Self-organized criticality (SOC) is a concept applied to spacially extended dynamic physical systems. It is often quite difficult to describe the temporal and spatial evolution of such a system and a lot of research happens in that field. Usually, one uses a mean-field approach to such a problem, where the indidual coupled degrees of freedom are replaced by a field. This (external) field then acts on the DOF. SOC is a different approach to this problem. It is the concept, that a certain class of dissipative, coupled dynamic systems converge to a critical point independantly of their underlying boundary and initial conditions. What are the properties of such a critical point? First of all, it is stable with respect to small disturbances. If this was not true, the system would not evolve to such a point. Note that this does not imply that such a point *exists*. If a certain point is unstable, the system commonly will

not evolve to this point. The critical point, however, generally is not the physical equilibrium point of the system.

## 3.3 Flicker Noise

Flicker noise is something often mentioned when talking about dissipative coupled dynamical systems. Flicker Noise (or $1/f$-Noise) is the description for an often observed phenomenon of such systems; The observation, that the power spectrum $S(f)$ scales with $1/f$ or more generally, with $f^{-\beta}$. This implies that $S(f) \cdot f \propto 1$

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

A Forest Fire Model is basically a cellular automata. 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.

## 3.5 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.

## 3.6 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 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 [2]. The 2 conditions can be written together as:

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

## 4.2 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, every grid point has three states (empty, alive and burning) and four rules, the most obvious of which is change state to burning if you are alive and your neighbor is burning.

Cellular Automata can exhibit very complicated behavior when fed with very simple rules, which is the main reason why they are so powerful. There are similarities to agent-based models and networks, but it is not to be mistaken for one of these.

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

## 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
```

13

```
    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 // 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 cluste
%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

We will first present to understand the traditional model of Forest fire. Here we can consider 4 different regimes.

- small p $(< 0.001)$ and small $\frac{p}{f}$ $(\approx 1)$

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

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

- 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.
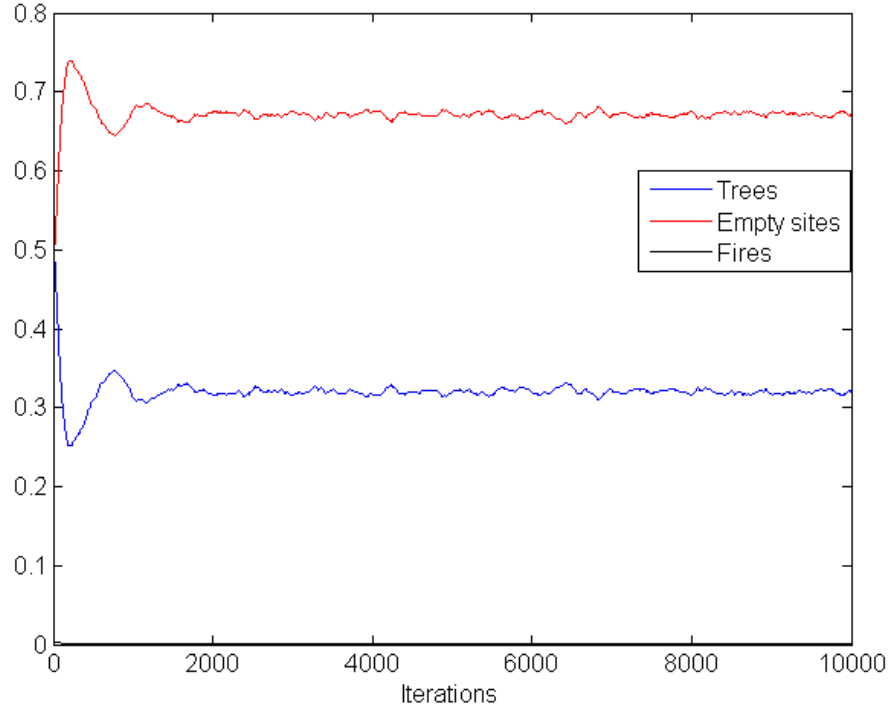


Figure 1: $N = 500$, $p = 0.001$, $\frac{p}{f} = 10$(Case1)

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.
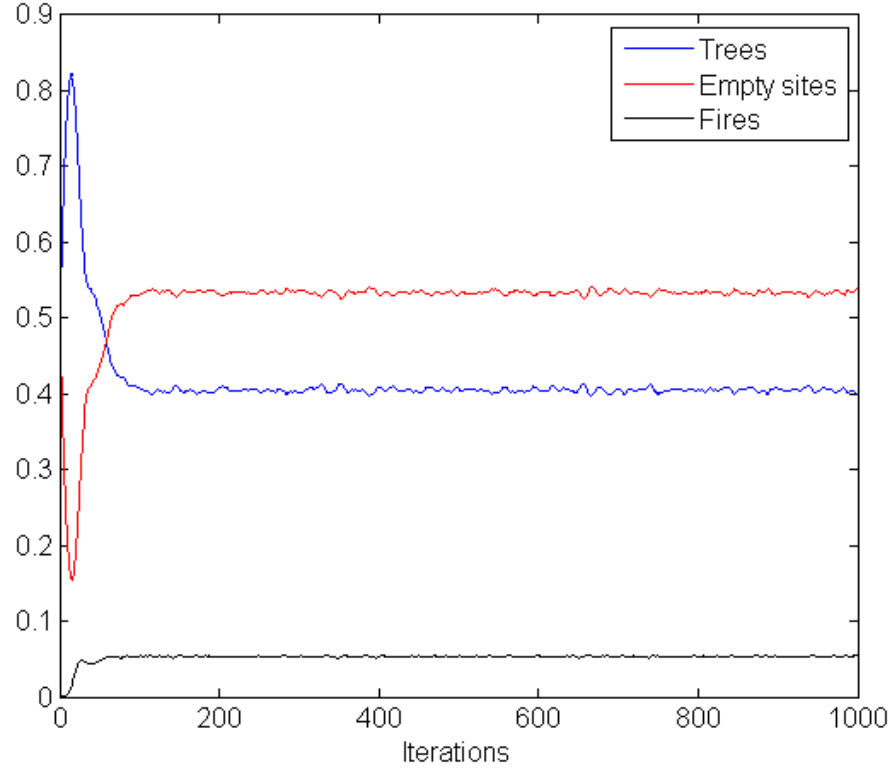
19

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.
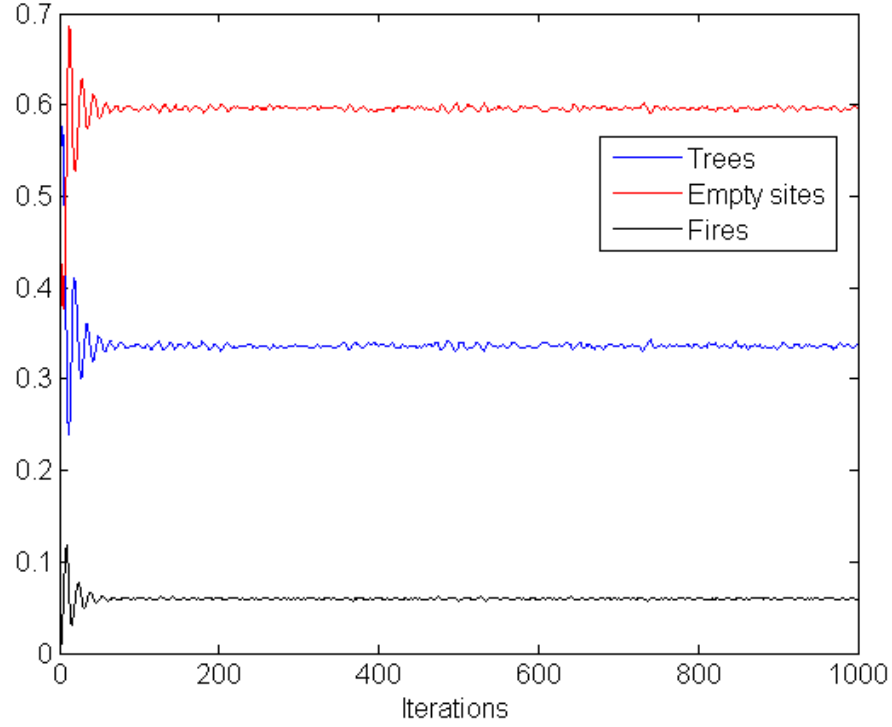
20

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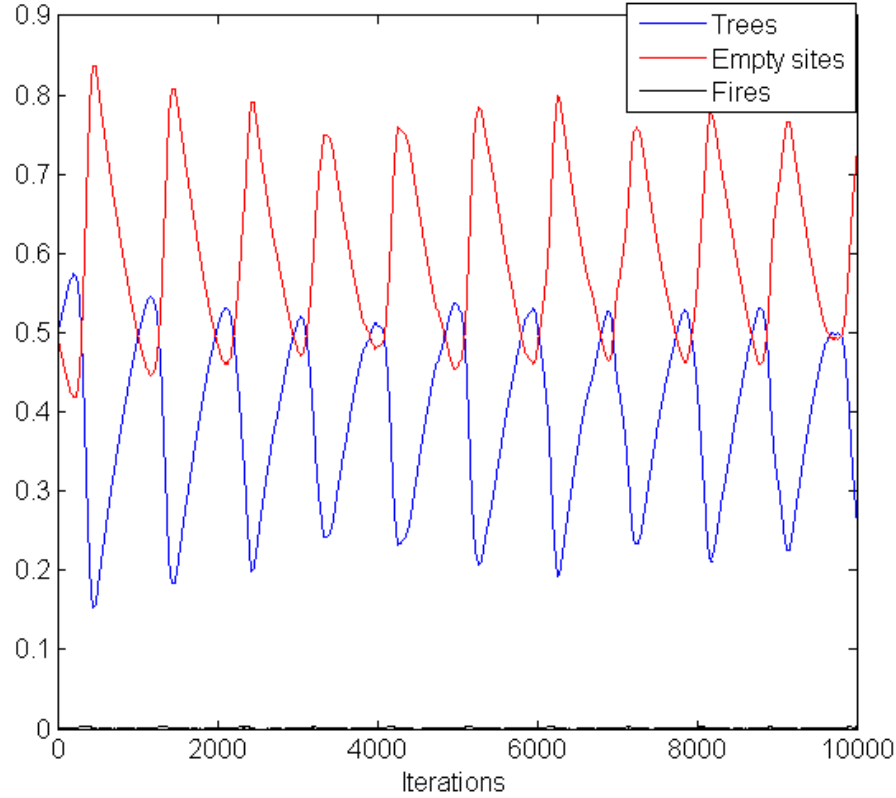
Figure 4: $N = 500, \ p = 0.001, \ \frac{p}{f} = 1000 (\text{Case4})$

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.
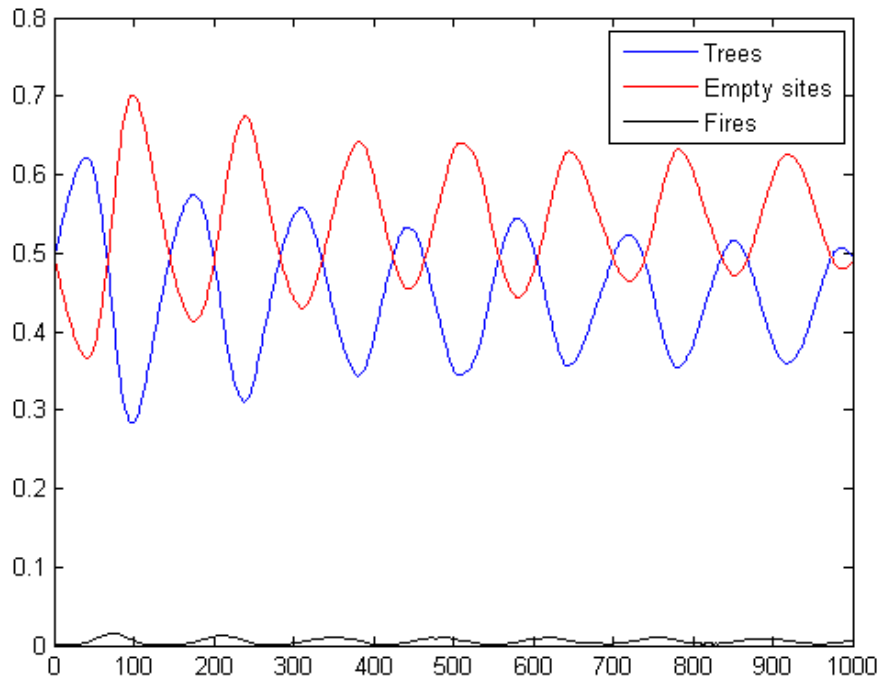


Figure 5: $N = 500$, $p = 0.01$, $\frac{p}{f} = 1000$(Case4)

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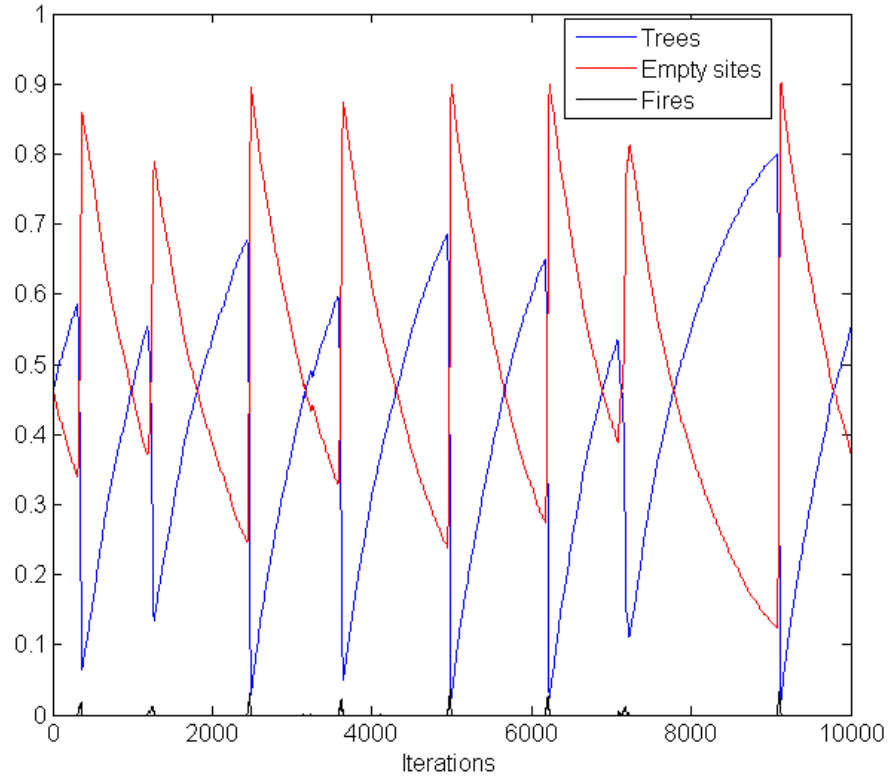
Figure 6: $N = 50$, $p = 0.001$, $\frac{p}{f} = 1000$(Case4)

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.
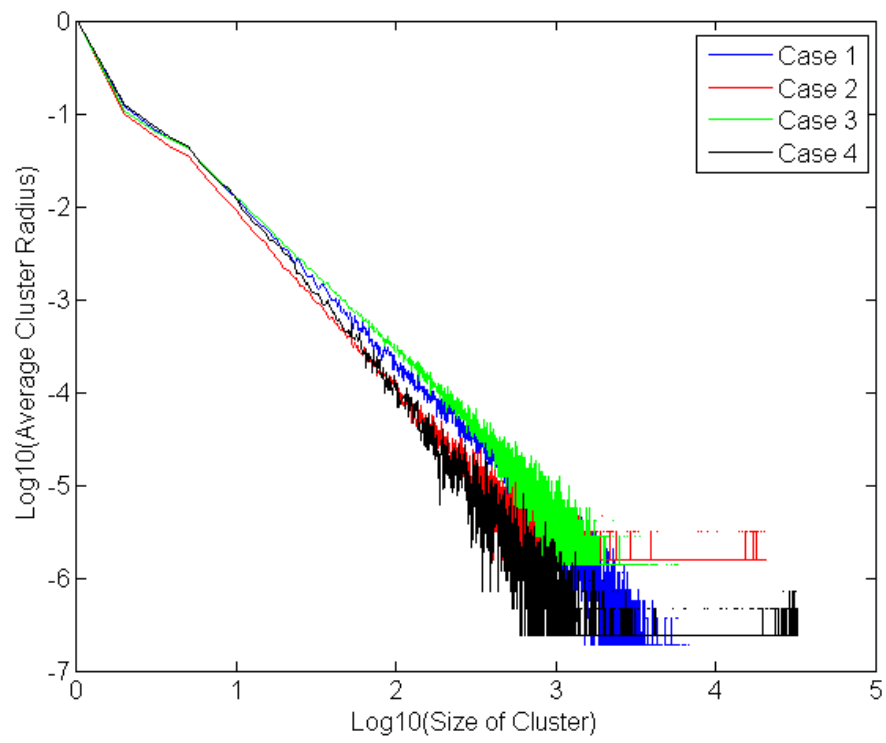
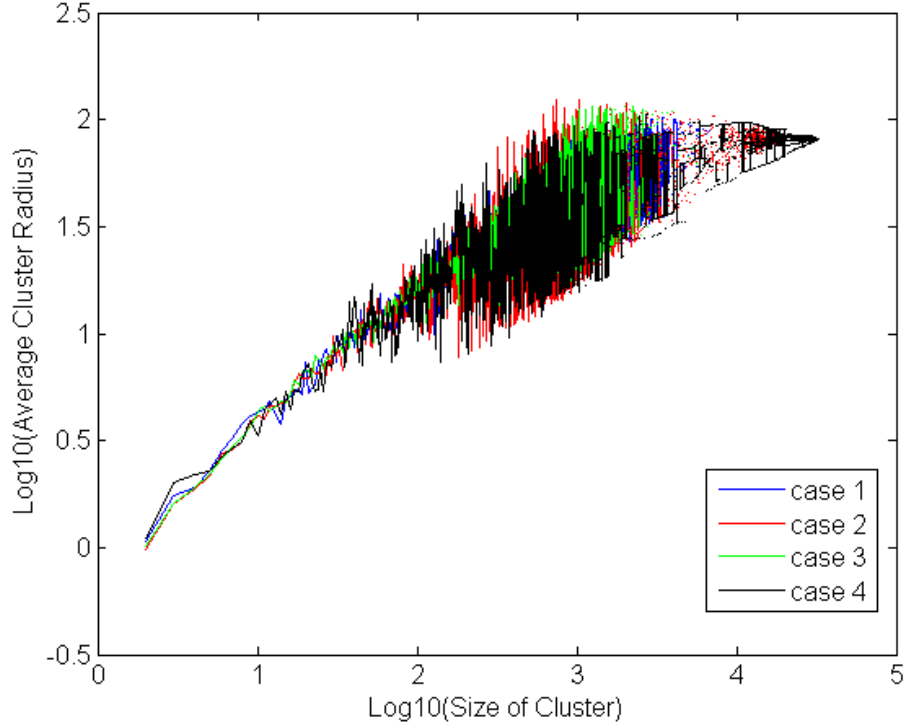Figure 7: $N = 200$, Frequency Distribution

Figure 8: $N = 200$, Radius Distribution

I don't see any difference in the 4 cases. I hope the other method will show us something. I will write some more discussion here.

## 6.1 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.
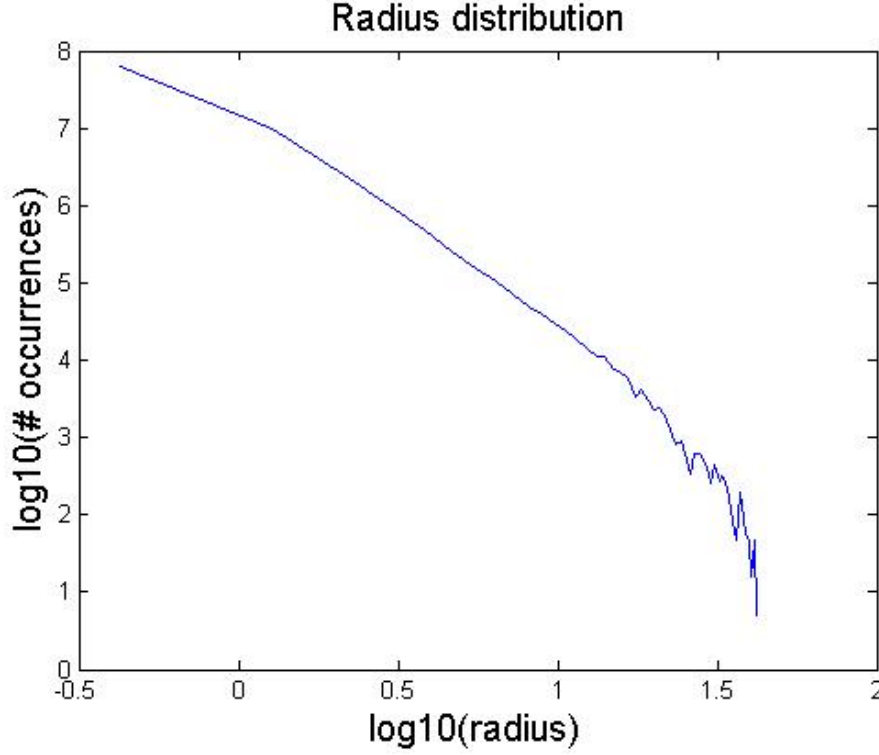
Figure 9: Cluster radius distribution ($N = 100$, $\Theta = 200$, Sim Time 1'000'000)

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   Cluster Size

The cluster size is another variable that we can easily measure. Here is a log-log-histogram of a simulation:
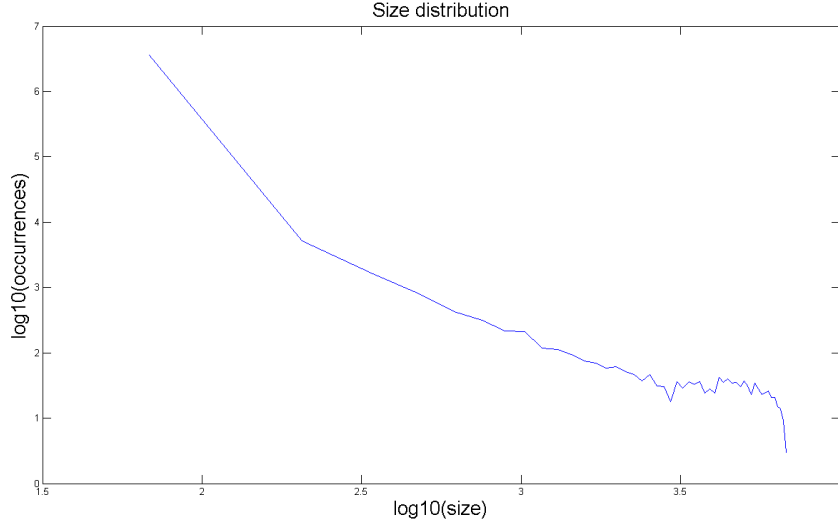
Figure 10: Cluster size distribution ($N = 100$, $\Theta = 1000$, Sim Time 500'000 )

## 6.3 Effect of different parameters

### 6.3.1 Effect of p,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.
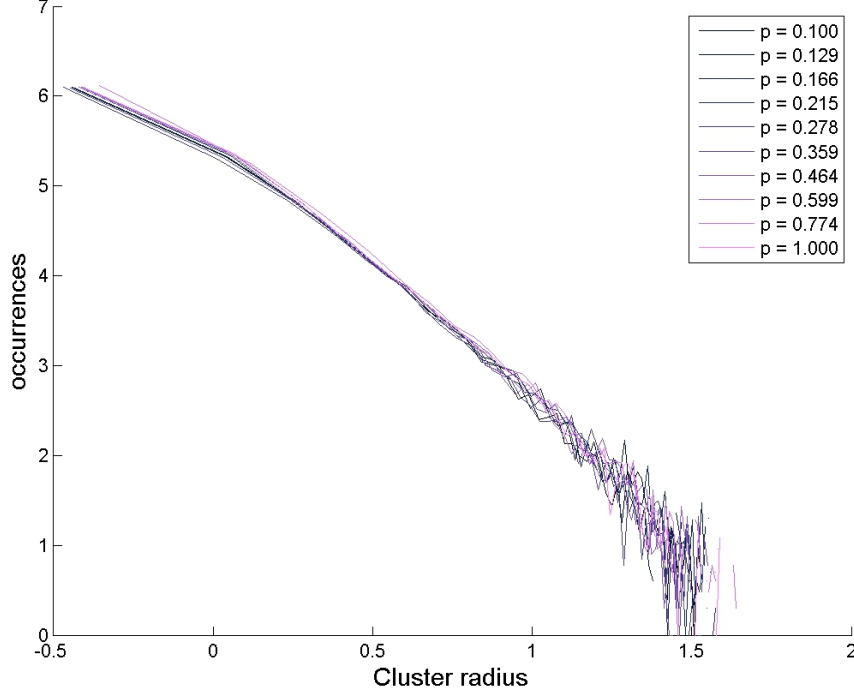
Figure 11: Log cluster radius distributions of simulations with constant Θ and changing p

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.

### 6.3.2  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 Θ > 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.

### 6.3.3  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.
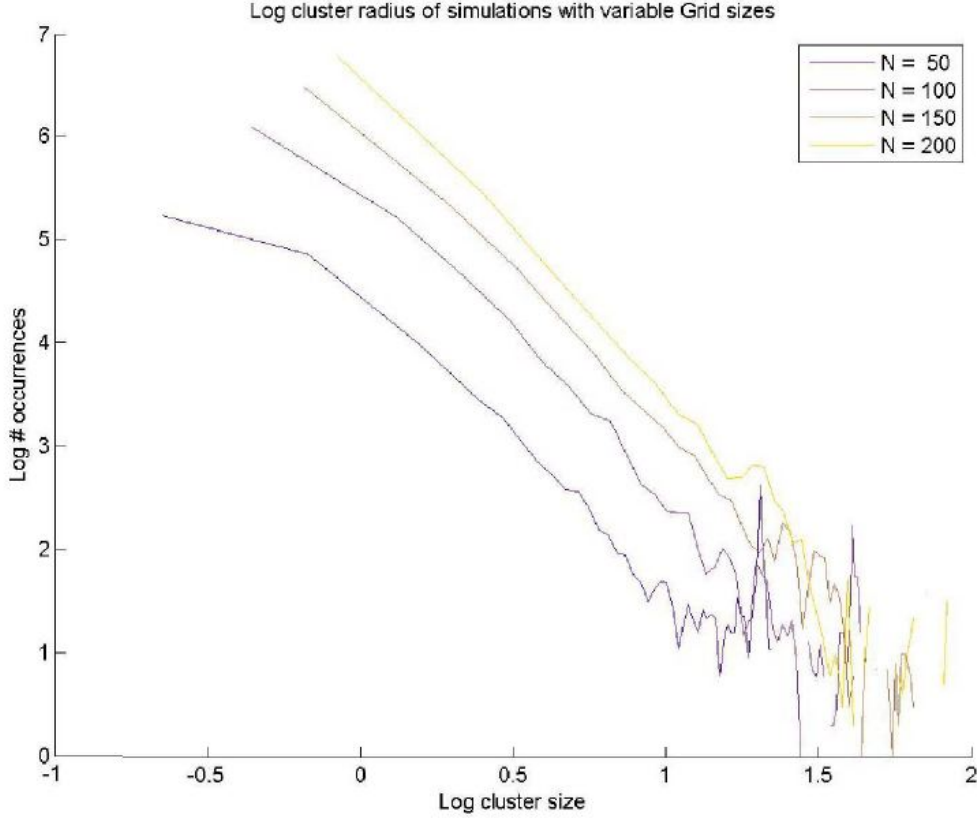
Figure 13: Log cluster radius distributions with variable grid sizes

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.

### 6.3.4 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.

## 6.4 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

- How does the dimensionality of the lattice influence the results?

- How does the choice of computational neighborhood influence the results?

# 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 complety 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 threshhold 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 dependancies of the critical threshhold with other parameters such as the grid size or the fire burning time. 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] 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  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
```

```matlab
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;
```

```matlab
%% 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,s
            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,
            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];
```

35

```matlab
                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), roun
```

```matlab
        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');
```

```matlab
    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 diagn
    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
```

```matlab
[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
```

```matlab
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);
```

```matlab
        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);
```

```matlab
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
```

```matlab
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
```

```matlab
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
```

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

```matlab
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
```

```matlab
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
end
```

```matlab
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
```