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 in Sandpile Models

Xinyi Chen
Artemi Egorov

Zurich
May 2012

Abstract

This paper describes the principles of self-organized criticality and their validity in cellular automation models, in particular the sandpile model. The model, its diversity, its implementation in MATLAB/Octave with different parameters and its analysis are presented in detail. Different aspects to the nature of critical systems, such as fractal structure and power-law distributions, are discussed including the effect of different system parameters, such as field size, its dimension, its boundary, presence of friction or the decoupling of the driving and the avalanche time scales.

Acknowledgements

The project group would like to thank the *Chair of Sociology, in particular of Modeling and Simulation*, for the chance to work on an interesting project and the possibility to apply simulation skills on extraordinary topics. Special thanks go to the assistants, namely Karsten Donnay and Stefano Balietti, for their all-time support during the project. Furthermore, the group would like to thank Pegah Kassraian Fard, who unfortunately had to quit the project, for her support in the early state of the project. Additional thanks go to the other groups of the class and of previous semesters.

Contents

Chapter 1

Introduction

1.1 Motivations

In Nature, most of the systems are complex, which means that their behaviour can hardly be predicted, but only partly studied within restrictive approximations. Although "Complex" might seem to be a synonym of "complicated", one frequently encounters simple power-law distributions or self-similar (fractal) patterns in a vast variety of complex systems, like the intensity of earthquakes distribution (Gutenberg-Richter Law) or our own fractal-like nervous system, respectively. This suggests the existence some simple but deep underlying laws, and understanding the mechanisms that lead to them is an exciting endeavour of science. In fact, these kinds of behaviour are also characteristic of the so-called *critical phenomena*, studied by the well-established theoretical framework of statistical physics. However, the latter deals with systems in thermodynamic equilibrium, with well-defined thermodynamic variables, such as temperature or pressure. These parameters that can be fine-tuned to obtain a critical state, i.e. a phase transition.

The concept of *Self-Organized Criticality* (SOC) was born as an appealing idea that might connect the real world of nonequilibrium physics (with self-organization) to the powerful tools of equilibrium physics (with criticality). Since its first publication in 1987, it led to many applications across almost every field of science, from astrophysics to economics. However, a comprehensive theoretical framework is still nonexistent. Hence, SOC is still merely a kind of phenomenology, mostly studied within computational simulations.

This project does not aim at providing any analytical or theoretical approach to SOC. Instead, its classical paradigm of the sandpile model is studied using cellular automation. Different possibilities are investigated in order to understand better the SOC mechanism and its characteristics. MATLAB/Octave code for n-dimensional abelian sandpile is provided, for

either discrete or continuous case, with different boundary conditions and dissipation mechanisms. The results of the discussed code are analyzed, in search of power-law distributions and possible fractal-like manifestations.

1.2 Self-Organized Criticality

The term self-organized criticality (SOC) basically consists of two properties:

- *self-organization* means that a non-equilibrium system is able to develop structures on its own, without external control or manipulation.
- *criticality* implies that a local disturbance not only influences the local neighborhood, but the whole system. In other words, all the members of a system influence each other. This term originally comes from thermodynamics and describes a state at the phase transition, where a substance (e.g. water) resides between different phases. This concept is presented in [7].

1.3 Cellular Automation

A cellular automation (CA) primarily consists of

- a finite regular d-dimensional field/lattice,
- a set of variables attached to each cell/site and
- a set of rules that specify the time evolution of the states.

A secondary property of a CA is the fact that the evolution rules are local, i.e. the updating of a certain cell only requires information about the cell itself and its finite, bounded and well defined neighborhood.

Further analysis of the above definitions show that a CA is deterministic, i.e. a given initial configuration will always evolve the same way. *Probabilistic* cellular automata imply an external probability to drive the updating rule and therefore allow to introduce a sort of continuity, even though the automation is of discrete nature.

1.4 CA and SOC

Generally it is difficult to determine whether a certain self-organized system exhibits self-organized criticality. One clue for a SOC-system is the existence of power-law distributions in both spatial and temporal fluctuations. Avalanche sizes (spatial) and lifetimes (temporal), as described later in section 3.3, can both show power-law behaviour of the form $f^{-a} \approx f^{-1}$.

Unfortunately, this type of correlation doesn't necessarily imply that the system is critical, i.e. non-critical systems can also show f^{-1}-behaviour. One idea present in literature is that the power-law behaviour is one of the consequences of the *scale-invariance* of the system. The other consequence is the presence of *spatial fractals*, which is harder to identify in a dynamical system than the presence of power-law distributions.

Chapter 2

The Sandpile Model

2.1 Bak-Tang-Wiesenfeld Model

The classical sandpile model represents a cellular automation describing a dynamical system following certain rules that can be described as follows.

The field/lattice, which is chosen to be two-dimensional, represents a sandpile. Each site on the lattice has a certain value z that intuitively represents the height or slope of the sandpile at certain position described with the coordinates x and y. At each time step, a number of grains of sand is placed on top of a random site, which increases its value by a given value, e.g. one. If the value of the site exceeds a critical value z_c (e.g. three), the site collapses/topples and its grains are evenly distributed to its neighbours.

In certain cases some of the adjacent sites will exceed the critical value too and the toppling process will continue until an equilibrium state is again reached. This series of collapsing sites is classically described as an avalanche. The next grain is not placed until the equilibrium state is reached, meaning that the time scale of the random grain placement and of the development of avalanches are decoupled.

The classical model description can mathematically be represented as follows.

Initially, the lattice is empty:

$$z(x, y) = 0 \quad \forall x, y$$

Then, the value of a random site x, y is increased:

$$z(x, y) \to z(x_r, y_r) + 1$$

If its value exceeds the critical value $z_c = 3$, then it topples and distributes its grains to its neighbours:

$$\begin{aligned} z(x, y) \overset{?}{>} 3 \Rightarrow & z(x, y) \to z(x, y) - 4 \\ & z(x \pm 1, y) \to z(x \pm 1, y) + 1 \\ & z(x, y \pm 1) \to z(x, y \pm 1) + 1 \end{aligned}$$

Here, we use the so-called *Von-Neumann-Neighborhood*, which consists of the four nearest neighbours. A possible alternative would be e.g. the *Moore-Neighborhood*, which consists of all eight nearest neighbours on a square lattice. One could also think of weighting the four corner neighbours to be "further away" from the middle cell than the four direct (Von-Neumann) neighbours.These possibilities are not discussed in this paper.

Clearly, many variations of the described model can be considered and can produce different results. The classical sandpile model, as originally described by Per Bak, Chao Tang and Kurt Wiesenfeld, represents the starting point of any further investigations considered in this paper.

2.2 Parameters

The behavior of the model is analysed dependent on different parameters such as:

- lattice size
- number of dimensions of lattice
- mass conservation, i.e. if the number of grains removed from a collapsed site is equal to the sum of grains its neighbour sites received
- boundary conditions, see below
- etc.

Different types of boundary conditions can be thought of:

- open: If a site near the border topples, some of its grains leave the system (mass is lost).
- closed: Near-border site does not fully collapse, but keeps the grains that would fall off in an open case.
- periodic: The system has no boundaries, i.e. toppling near the border is "wrapped over".
- mixed: E.g. the lattice is periodic in one dimension and has open boundaries in another dimension.

Figure 2.1 illustrates the first three basic types of boundary conditions.

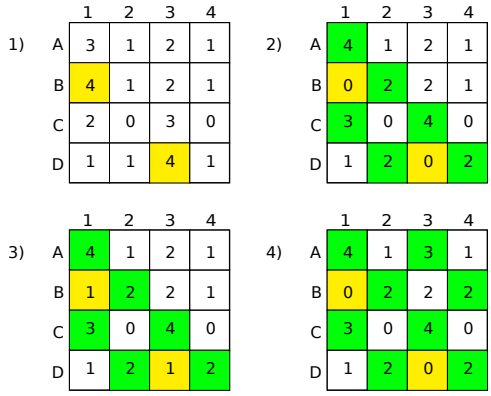

Figure 2.1: Effect of different types of boundary conditions on a sample lattice (1): open (2), closed (3) and periodic (4)

2.3 Abelian Model

One important property which can be used to categorize different sandpile models is whether they behave in a commutative or *abelian* way. In particular, this can be applied to the development of avalanches in the model described above: The question posed here is, whether an equilibrium state resulting from an avalanche depends on the way the avalanche is calculated. More precisely, it can be shown that any avalanche, being a sequence of topplings, always results in the same equilibrium state i.e. does not depend on the order, in which the topplings occur. The mathematical proof of this hypothesis is nicely presented in [8].

To illustrate this practical but not necessarily obvious fact, a sample 4x4-field with one active site is considered (see figure 2.2). At step (2), two different sites simultaneously become active, therefore creating a “choice”, which site to topple first. Depending on such choices, different sequences of topplings occur, but all lead to the same equilibrium state.

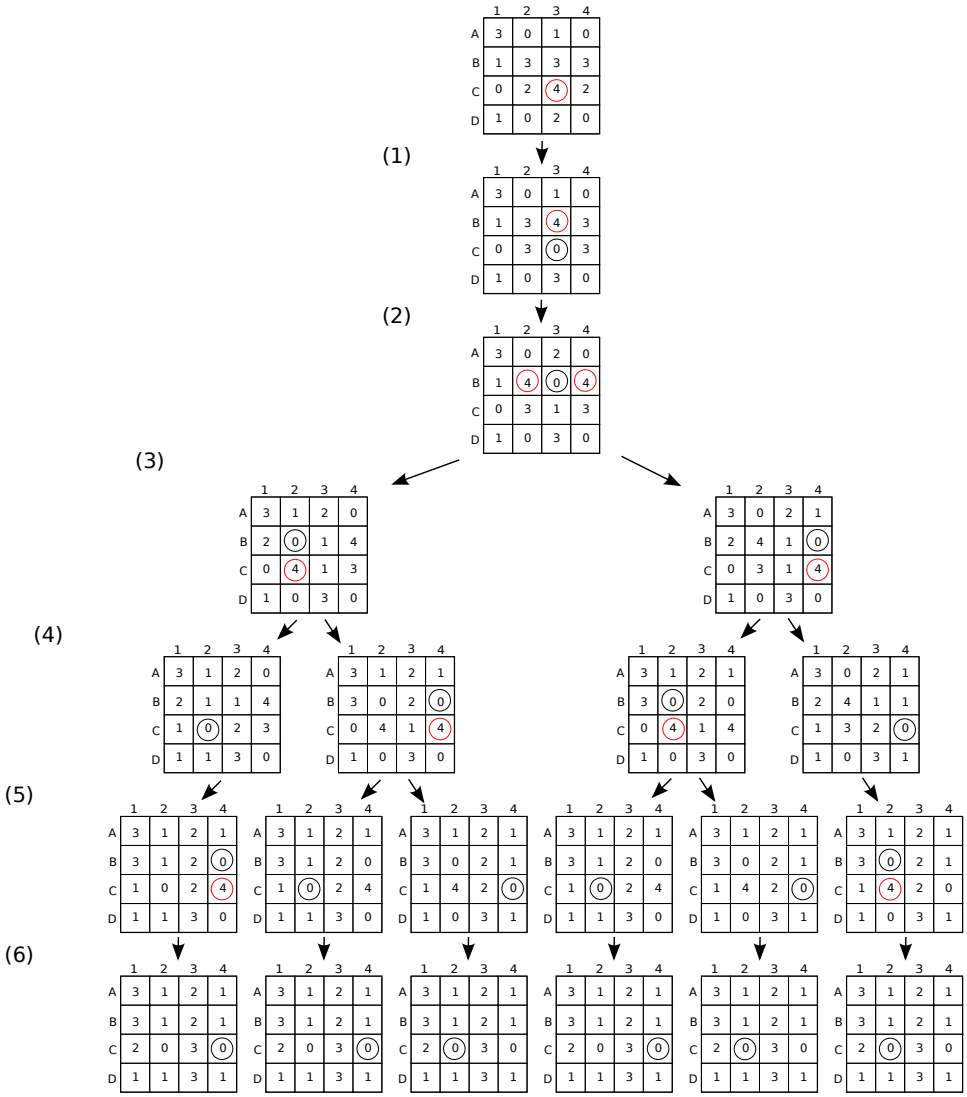

Figure 2.2: Demonstration of the abelian property: six different orders of topplings all lead to the same equilibrium state. The sites circled red indicate sites that have just become active, those circled black have just collapsed. Here, continuous boundary conditions have been used.

Chapter 3

Model Implementation in MATLAB/Octave

Note on Code and Programming Sustainability

In order to produce "sustainable" code and to share the spirit of independency coming from the open-source community, the coded routines were tested in MATLAB and in Octave (one of its open-source clones). The source code can be found in Appendix A.

3.1 Basic Sandpile Code

First, a lattice/field is generated using uniformly distributed random numbers from 0 to z_c (`critical_state`). This is done in order to start with a potentially critical field and not to place single grains of sand until a site gets critical.

```
f = floor(unifrnd(0,critical_state+1,height,width));
```

When the field is ready, a global loop runs through a defined number of time steps, placing a grain on a random site, checking if the site becomes active and if so, computing the resulting avalanche.

```
for t=1:timesteps
  % choose random site
  y=floor(unifrnd(1,height));
  x=floor(unifrnd(1,width));

  % place grain
  f(y,x) = f(y,x) + 1;

  % check if overcritical/active
  if (f(y,x) > critical_state)
    % avalanche code here
    % ...
  end
end
```

3.2 Optimization of Avalanche Code

The simple avalanche code checks the whole field including the fields, that cannot possibly be affected by the avalanche. It can therefore be optimized, for example using a LIFO data structure – a *stack*. The coordinates of very site that needs to be checked are placed on the stack, so that the computation of the avalanche consists of working through the stack and toppling all the active sites in it. During their toppling, their neighbours are again put on the stack, which makes the procedure dynamical and not easily comprehensive. The algorithm is illustrated in figure 3.1.

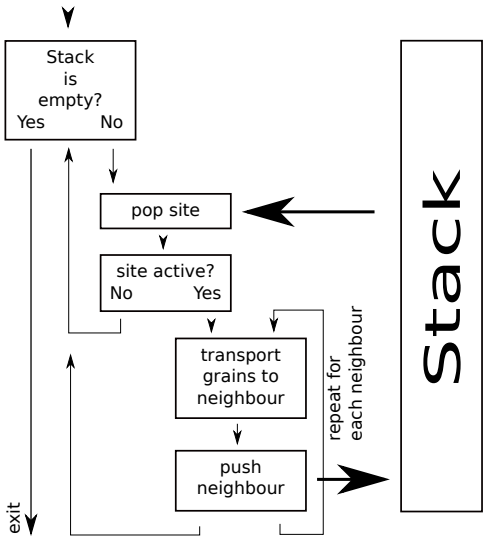

Figure 3.1: using a stack for avalanche calculation

Considering the example from figure 2.2, the stack algorithm results in the following sequence:

0. push C3
1. pop C3, topple, push its neighbours (C2,C4,B3 and D3) to stack
2. pop B3, topple, push B2, B4, A3 and C3 to stack
3. pop B2, ...
4. pop C2, ...
5. pop B4, ...
6. pop C4, ...

Figure 3.2 shows the states of the stack after each of these steps. To avoid confusion, only the active sites are shown here.

The main loop including the stack feature looks like this:

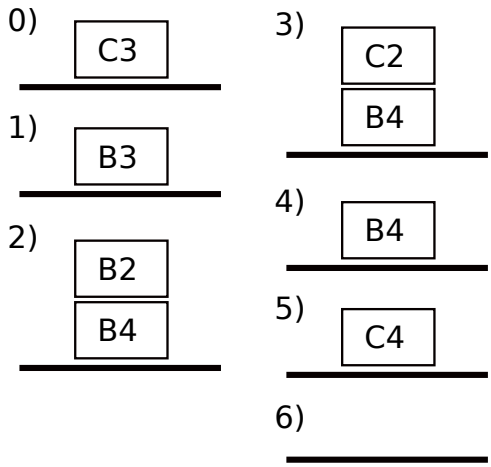

Figure 3.2: A sample sequence of stack states. The algorithm proceeds until the stack is empty.

```
for t=1:timesteps
  % choose random site
  % ...

  % place grain
  % ...

  % push site to stack
  stack_n = 1;
  stack_x(1) = x;
  stack_y(1) = y;

  % avalanche - work through stack
  while (stack_n > 0)

    % pop from stack
    x = stack_x(stack_n);
    y = stack_y(stack_n);
    stack_n = stack_n - 1;

    % check if overcritical/active
    if (f(y,x) > critical_state)
      % collapse/topple
      f(y,x) = f(y,x) - neighbours * collapse;

      % look at every neighbour
      for n=1:neighbours
        % add/transport grain to neighbour
        f(y+neighbour_offset_y(n),x+neighbour_offset_x(n)) = ...
          f(y+neighbour_offset_y(n),x+neighbour_offset_x(n)) + collapse;

        % push neighbour to stack
        stack_n = stack_n + 1;
        stack_x(stack_n) = x + neighbour_offset_x(n);
        stack_y(stack_n) = y + neighbour_offset_y(n);
      end
    end
  end
end
```

3.3 Statistics

Many different variables may be of interest for the statistical analysis of sandpile models. The easiest to implement is avalanche size:

```
...
% check if overcritical/active
if (f(y,x) > critical_state)

  % collapse/topple
  f(y,x) = f(y,x) - neighbours * collapse;

  % record statistics
  avalanche_sizes(t) = avalanche_sizes(t) + 1;

  ...
```

Here, the number of avalanches is recorded at every time step by increasing the counter after each toppling that happens during the avalanche. After the main loop, the data is sorted and the distribution is fitted into a power-law distribution given by

$$P(s) = a \cdot s^b$$

where P is the number of avalanches of size s. The coefficients a and b are determined using a simple solver that minimizes $a \cdot s^b - P(s)$.

```
% count avalanche sizes - calculate distribution
for s=1:max(avalanche_sizes)
  avalanche_count(s) = size(avalanche_sizes(avalanche_sizes==s),2);
end

% filter zero values
s = [1:max(avalanche_sizes)];
P = avalanche_count(1:end);
s = s(P>0);
P = P(P>0);

% fit into power-law
[c,fval,info,output]=fsolve(@(c)((c(1).*s.^c(2))-P),[100,1]);
a = c(1);
b = c(2);
```

In order to implement statistics of avalanche lifetime in the stack code, the number of additional (i.e. more than one) topplings per time step must be counted. The reason for this is that the stack algorithm does not follow a timescale and therefore the number of time steps taken by an avalanche cannot be counted directly. Therefore, the following equation is used:

$$s = \sum_t n = \underbrace{\sum_t (n-1)}_{a} + \sum_t \qquad \Rightarrow \sum_t = s - a$$

where s is the avalanche size, t is the avalanche lifetime, n is the number of topplings per time step and a is the total number of additional topplings. The neighbour-checking part of the stack loop looks like this:

```
% count future topplings to be caused by this toppling
future_topplings = 0;
```

```
% look at every neighbour
for n=1:neighbours

  % add/transport grain to neighbour
  % ...

  % push neighbour to stack
  % ...

  % count future topplings to be caused by this toppling
  if (f(y+neighbour_offset_y(n),x+neighbour_offset_x(n)) == (critical_state+1))
    % i.e. if neighbour site becomes active
    future_topplings = future_topplings + 1;
  end

end
% calculate additional topplings caused
if (future_topplings > 0)
  avalanche_add(t) = avalanche_add(t) + future_topplings - 1;
end
```

For each toppling, the number of further topplings is counted, summed up and subtracted by one. Then, the avalanche lifetime is calculated according to the formula $\sum_t = s - a$ as follows:

```
% return avalanche lifetimes
at = avalanche_sizes - avalanche_add;
```

3.4 The d-dimensional case

The generalization to d-dimension is quite straightforward using the linearized index, i.e. enumerating the sites with one integer number. This is done using the relation

$$v = (x_d - 1)n^d + (x_{d-1} - 1)n^{d-1} + \ldots + (x_2 - 1)n + x_1$$

which is implemented in the function `linear_index(x,d,n)`, where `x` is the array with the coordinate of the lattice site.

The inverse function is `coordinate(v,d,n)`, and is written using the Matlab modulus function `mod()`. The idea is to use a simple recurrent relation as

$$\begin{aligned} v = v_d &= (x_d - 1)n^d + v_{d-1} = (x_d - 1)n^d + (x_{d-1} - 1)n^{d-1} + v_{d-2} = \ldots \\ &= (x_d - 1)n^d + (x_{d-1} - 1)n^{d-1} + \ldots + (x_2 - 1)n + v_1 \end{aligned}$$

which leads to

$$v_d = mod(v_{d-1}, n^d)$$

and

$$(x_d - 1) = v_d - v_{d-1}/n^d; \quad x_1 = v_1.$$

The corresponding code is:

```
x=zeros(1,d); % coordinates (x(1),x(2),..., x(d))
va=zeros(1,d+1);
va(d+1)=v-1;
```

```
  for i=d:-1:1
    va(i)=mod(va(i+1),power(n,i-1));
    x(i)=1+(va(i+1)-va(i))/power(n,i-1);
  end
end
```

The nearest neighbour function for d-dimensional case is implemented in `neighbour(v,d,n,periodic_boundary)`, which gives the coordinates of the $2d$ nearest neighbour of a given site v. The option of periodic boundary (a `1` in the argument) or finite boundary (`0`) is included. Basically, the code consists of giving the coordinates of any site v using `coordinate(v,d,n)` function and distinguishing *forward* and *backward* neighbours by adding or subtracting 1 respectively. In the periodic boundary case, when the coordinate is 0 or bigger than n (in fact, equal to $n+1$), the coordinates are set to n or 1, respectively. For the finite boundary, we set all the coordinates outside the lattice equal to $-n^d$, since this will always give the corresponding $v<0$. Therefore, in the main program an `if` condition true for $v>0$ is used for excluding these cases.

3.5 Continuous sandpile field

So far discussed was the sandpile model considering a discrete field, interpreted as height, and adding value of 1 to a random site for each driving time step. This field can be thought of as a sandpile of uniform grain size. It is straightforward to generalize it to a more realistic field, where the grains can be of different sizes. Preferably, one can speak of the lattice as an energy volume and the grains as energy units. The driving is therefore modified to add a *real* random value between 0 and 1 to the field at each driving time step.

When the site is overcritical, its nearest neighbours receive a grain of random size as well. The sum of these numbers is what the toppling site loses, so the mass (or energy) is conserved. This implies closed boundary for this case, hence it is preferable to introduce some friction in order to study the system for a large driving time (as for the periodic boundary case discussed above). The friction parameter `f` means that some energy is lost during the propagation to the neighbours, i.e. for `f<1`, it corresponds to the dissipative case with an energy loss of $\Delta E=(1-f)E$, where E is the value of the field for the conservative case (`f=1`).

The basic code is rewritten here as

```
c=0;
for k=1:2*d
  r(k)=rand(1);
  E(vnn)=E(vnn)+r(k);
  c=c+r(k);
end
E(v)=(E(v)-c)*f;
```

where `c` is the sum of the energy values that each nearest neighbour site `vnn` gained during the relaxation of the site `v`.

The complete sandpile program for d-dimension is a bit more sophisticated, including the *stack* method discussed before, with one difference that there is a stack for each parallel sites dropping caused by one initial toppling site. The whole structure is therefore similar to a "tree of stacks" . The advantage is that the lifetime of an avalanche can be counted directly.

The generalization presented here makes the sandpile model a bit more realistic, although the number of digits is still finite. Hence, this case is equivalent to the discrete case, only with different propagation rules. Besides, the lattice sites are still discrete.

Chapter 4

Simulation Results and Discussion

4.1 Power-law Distributions

In this section, investigations of the avalanche size and lifetime distributions are shown according to different boundary conditions: periodic and open.

4.1.1 Periodic Boundary Conditions

For periodic boundary conditions, no grain gets lost as long as no friction is introduced. As a consequence, the driving time and the lattice size need to be chosen carefully, in order not to run into a situation of a never-ending avalanche. Also, to analyze the distribution, good statistics must be present - a reasonably large lattice is needed. Although, theoretically, the lattice is of infinite size (due to its periodic nature), which implies that the size should not matter.

Here, a 100×100 lattice is studied with a driving time $T = 5000$. The results are presented in the figures 4.1. The power-law behaviour is easily recognizable for the avalanche size and for the avalanche lifetime, although the latter fit is of poorer quality.

The first point of both plots correspond to the case where no avalanches occur. It is included to show that the system stays subcritical most of the time, and only sometimes becomes overcritical.

Higher dimensions are not treated here, due to high memory and simulation runtime requirements. As real systems are finite and dissipative, the periodic boundary case without grain loss is not of much interest, i.e. it is not considered a SOC phenomenon. Instead, a friction parameter is introduced and discrete grain addition is replaced by a continuous one, thus replacing the integer field by a *real* field and presenting a more realistic model.

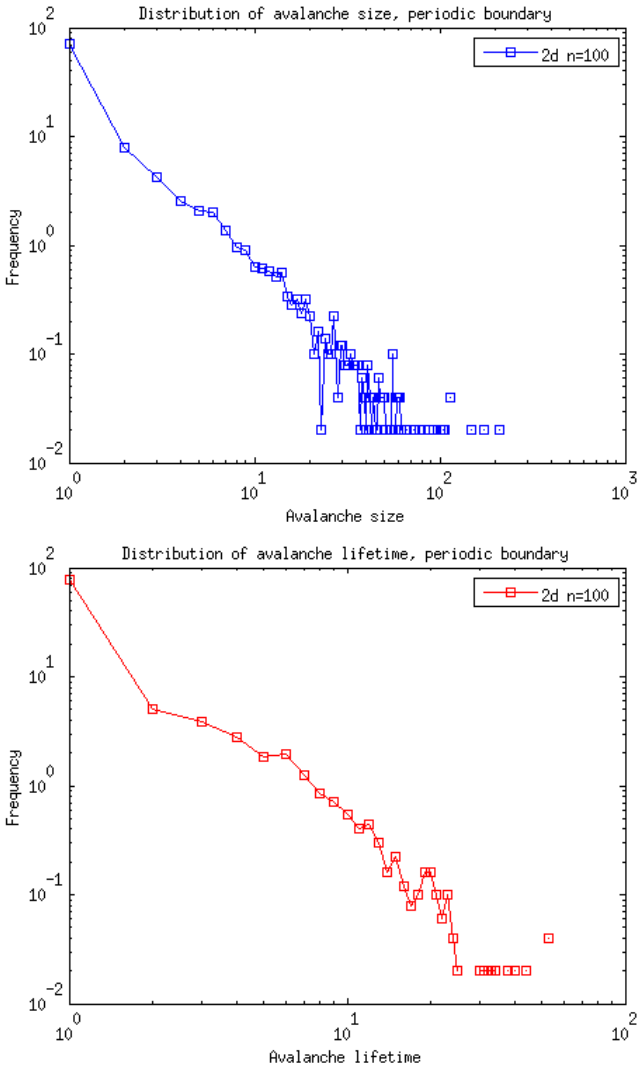

Figure 4.1: Avalanche size and lifetime distribution for a 100×100 lattice, with periodic boundary conditions. The driving time is $T = 5000$.

The periodic boundary in principle implies lattice size independence, which will represent a computational advantage, as a small lattice can be chosen.

Results of this situation for $d = 2$, with $n = 10$, $n = 50$ and $n = 100$ are shown in figures 4.2 and 4.3. A clear cut-off for large avalanche number due to friction is seen, but with a nice power-law distribution for small avalanches. The lifetime is not shown as it does not differ much from the case in figure 4.1. For the case of $n = 100$, the simulation is run ten times longer than for the other cases, resulting in a distribution different from small or medium-size ones for large avalanches.

The importance of the dissipation becomes clear as the critical exponent changes for different values of friction.

From figures 4.2 and 4.3 we can conclude that small lattices can also be used with a small driving time, saving computational capacity. Therefore, this case will be used to explore higher dimensional lattices, that are computationally more costly. Figures 4.4 and 4.5 show the results of the $3d$ and the $4d$ case respectively. The cut-off effect due to friction is much less than for the $2d$ lattice. The effect of an increase in the friction value is shown in figure 4.6. From this analysis, clearly, the dissipation plays an important role. Nevertheless, *on average*, the total energy of lattice is kept constant when the system evolves (see figure 4.7). Furthermore, for $d = 1$, no power-law behaviour is seen, in agreement with the prediction of theory presented in [6] (the situation for open boundary has also been checked to be the same). Refer to figure 4.8 for the results.

4.1.2 Finite boundary conditions

A perturbation in the boundary might cause a different avalanche distribution than a perturbation placed in the bulk. One might expect a bulk perturbation to produce bigger avalanches, as the grain has to be transported further in order to be lost at the boundaries.

Here, this effect is studied for 2-dimensional case and remarkably, different avalanche size distributions can be seen (figure 4.9). Furthermore, the high dispersion in large avalanche sizes for the bulk case causes a worse power-law fit compared to the boundary perturbation case.

For random perturbation sites, the result is closer to the bulk one than to the boundary one.

The relation of the number of sites in the volume to the number of sites at boundary should be proportional to the lattice size n, as it is basically the ratio of volume and area. This implies that on a large lattice one encounters more large avalanches than on a smaller lattice. See figure 4.10.

Adding friction (and thus going to a more realistic case), the number of large avalanches reduces as seen in figure 4.11. Again, friction is crucial for generating a power-law like distribution and for the cut-off effect discussed

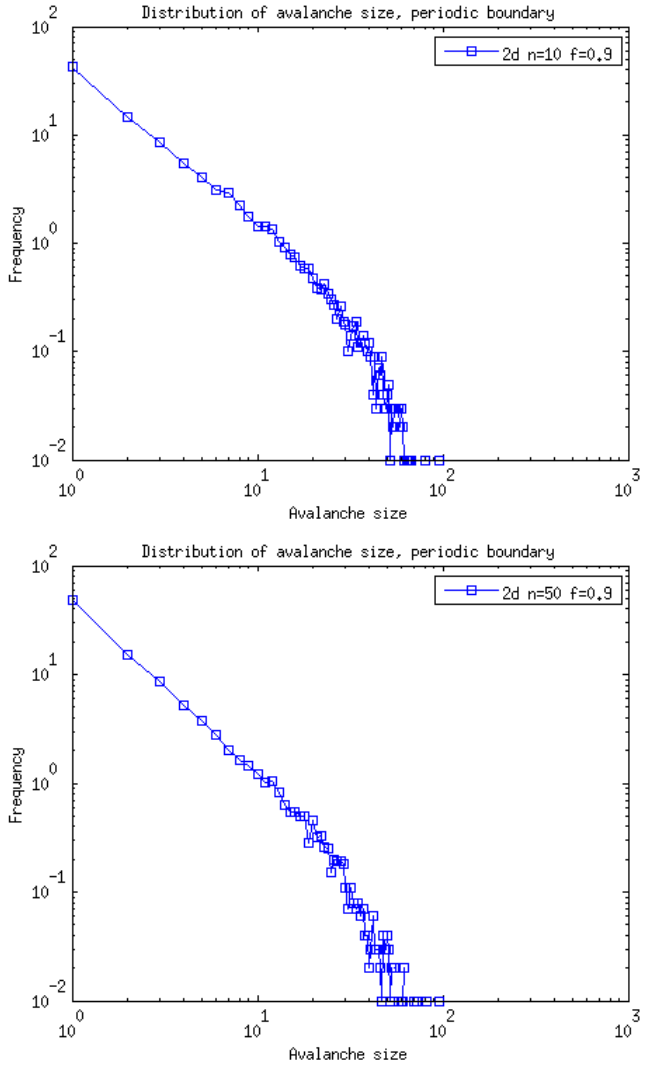

Figure 4.2: Avalanche size distribution for a $2d$ lattice with friction and periodic boundary conditions. The driving time is $T = 10\,000$ and the lattice size is $n = 10$ and $n = 50$, respectively.

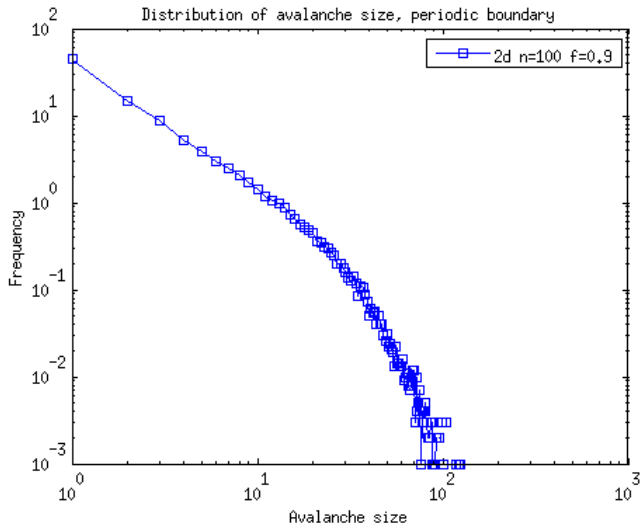

Figure 4.3: Avalanche size distribution for a $2d$ lattice with friction and periodic boundary conditions. The driving time is $T = 100\,000$.

before.

During the driving time, the energy of the system, namely the sum of the values of the field of all sites, tends to oscillate around a constant value, as the additional grain in each driving time is compensated with the dissipation during avalanche, see figures 4.11 and 4.7.

4.1.3 Distribution for small lattices

The total number of possible configuration states containing only one overcritical site is equal to

$$N_c = n^d \times (2d)^{n^d - 1}$$

A chain of size n^d linearizes the lattice. One site is fixed equal to the critical value, i.e. the minimal value that the site will topple, set as $E_c = 2d$ for convenience. The rest of the $(n^d - 1)$ sites can have any value from $\{0, 1, \ldots, 2d - 1\}$, so $2d$ values.

However, not all of these configurations will appear during the driving time. As only one energy grain is randomly added to the system for each driving time, and as the system can dissipate grains, this will soon lead to a characteristic average energy value per site, as discussed before. For example, configurations such that all sites are zero except the overcritical one will never occur, unless it is started with. Thus, only a finite subset of the possible set A will occur during one simulation (subset of A). The avalanche phenomenon, a set of configurations with one or more overcritical site(s), indeed creates a relation of the subset of A with a subset of subcritical

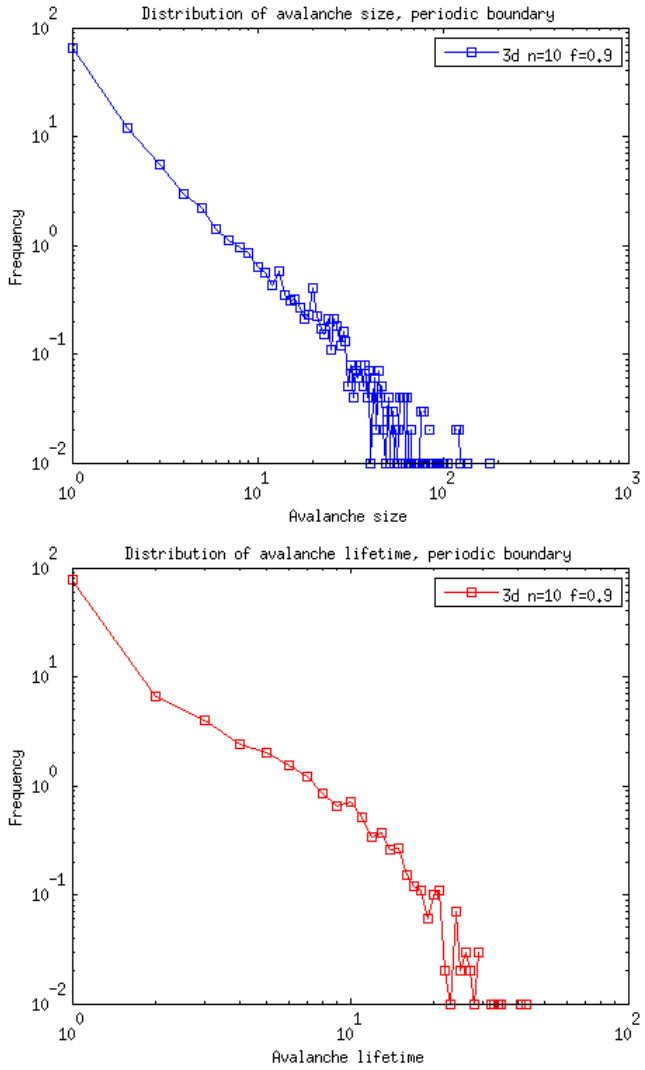

Figure 4.4: Avalanche size and lifetime distribution for a $3d$ lattice with friction and periodic boundary conditions.

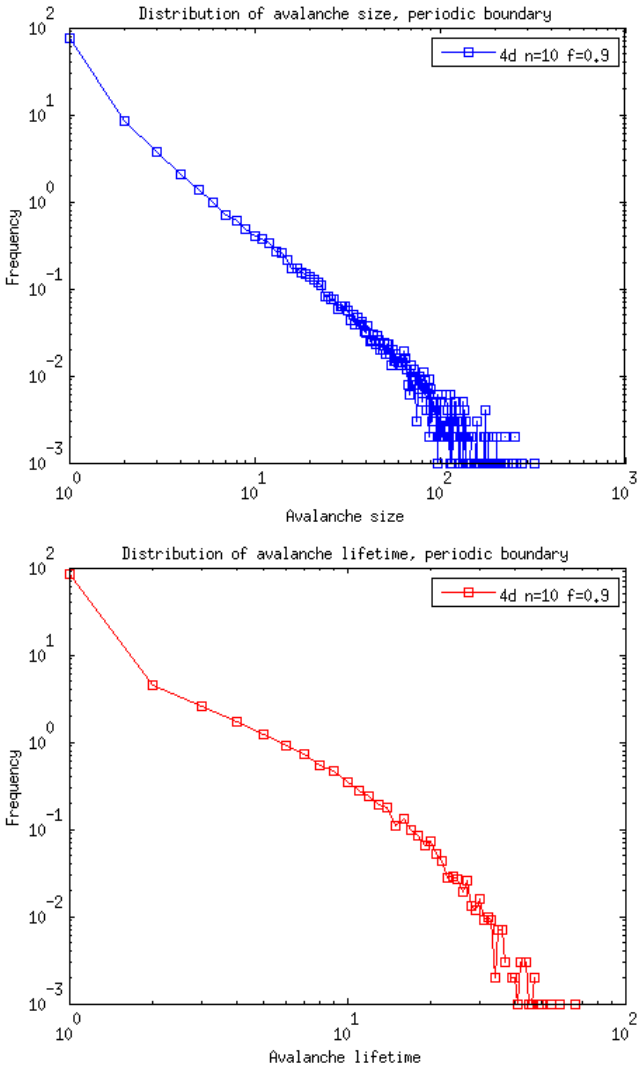

Figure 4.5: Avalanche size and lifetime distribution for a $4d$ lattice with friction and periodic boundary conditions.

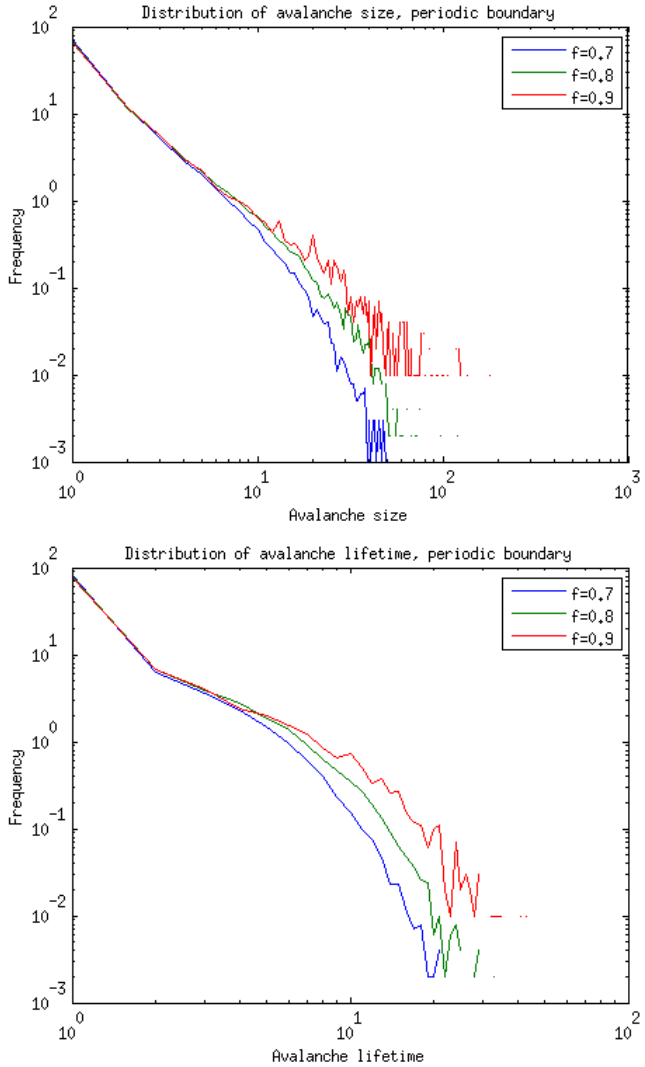

Figure 4.6: Avalanche size and lifetime distribution for a $3d$ lattice with different friction parameters using periodic boundary conditions. The smaller the friction parameter, the larger the dissipation. The first point in the lifetime distribution corresponds to "zero avalanches" during the lifetime.

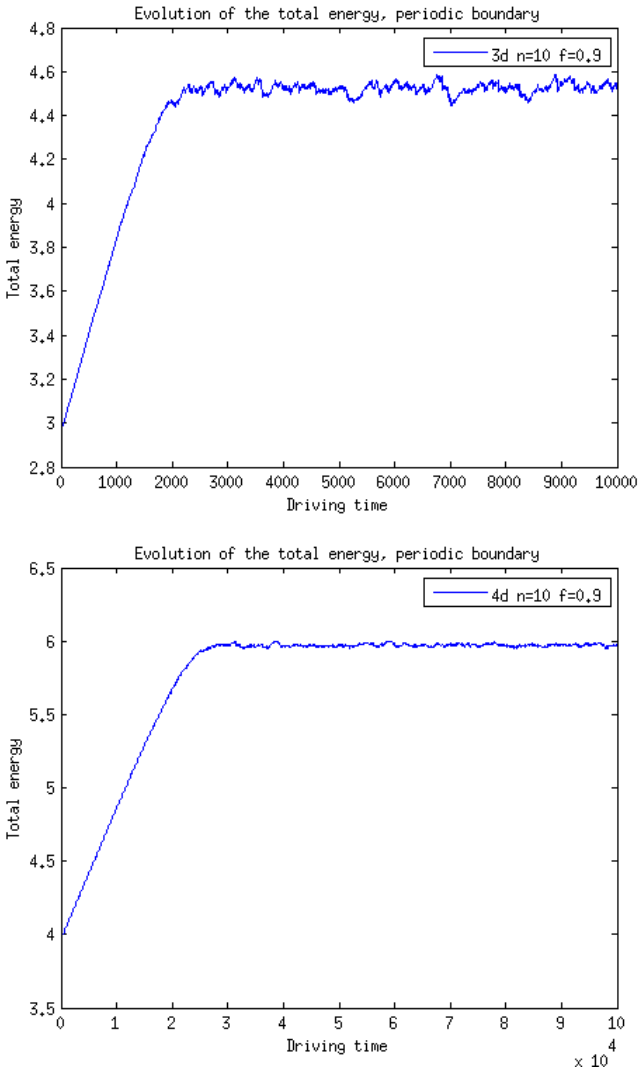

Figure 4.7: Normalized total energy evolution for $3d$ and $4d$ lattices with friction and periodic boundary conditions.

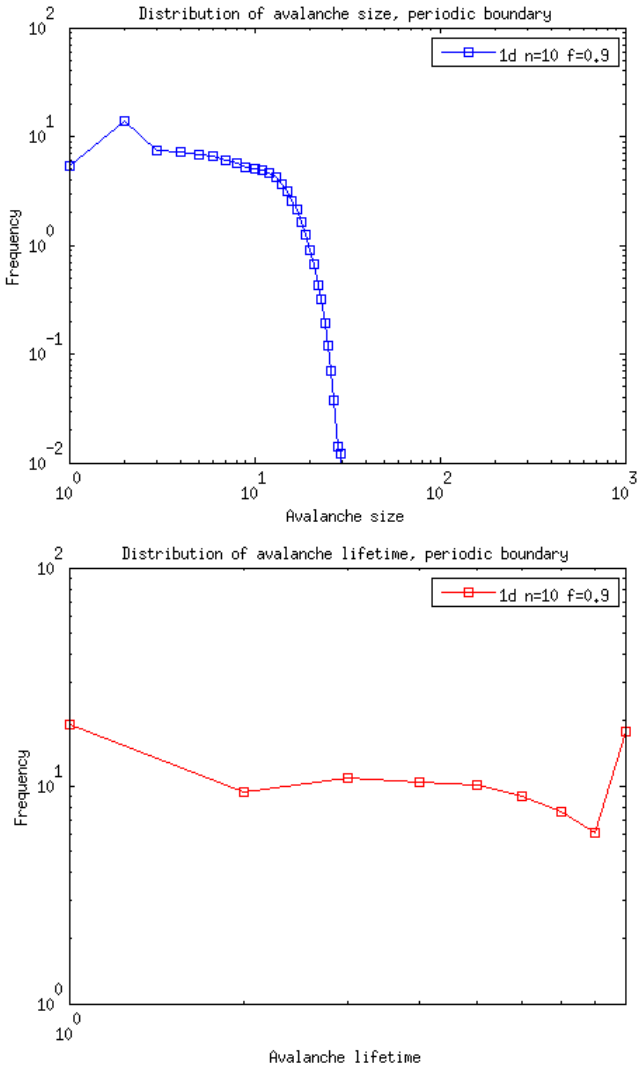

Figure 4.8: Avalanche size and lifetime distribution for $1d$ lattices with friction, periodic boundary conditions and $T = 100\,000$. There is no criticality for this case.

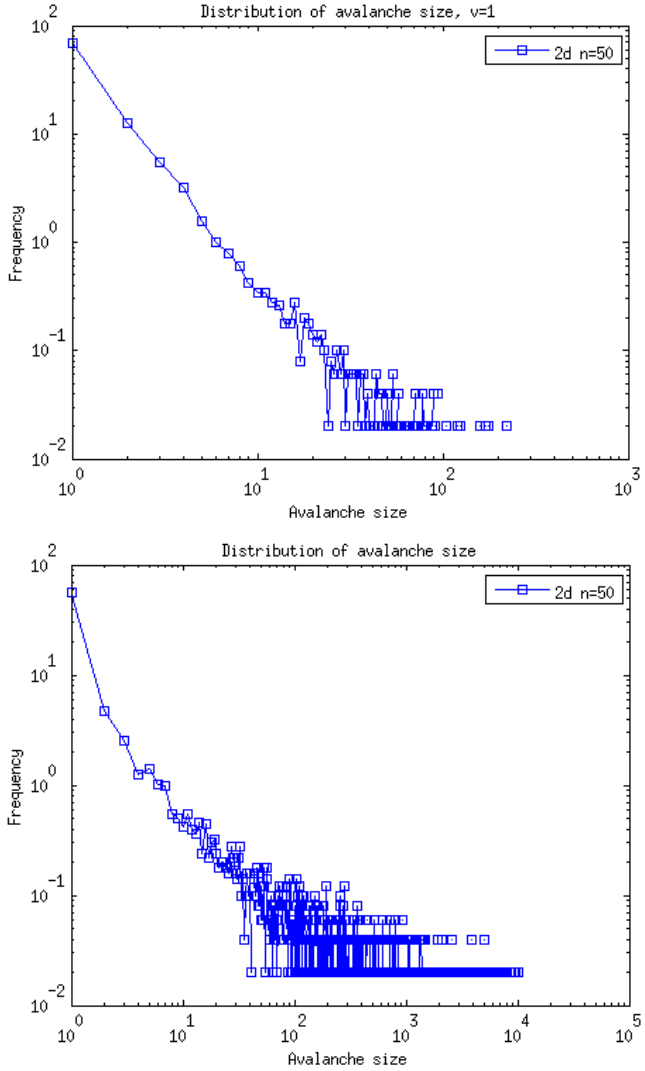

Figure 4.9: Avalanche size distribution for a 50×50 lattice for different perturbation sites (first plot: site $(1, 1)$, upper left the corner; second plot: bulk site, $(25, 25)$). The driving time is $T = 5000$.

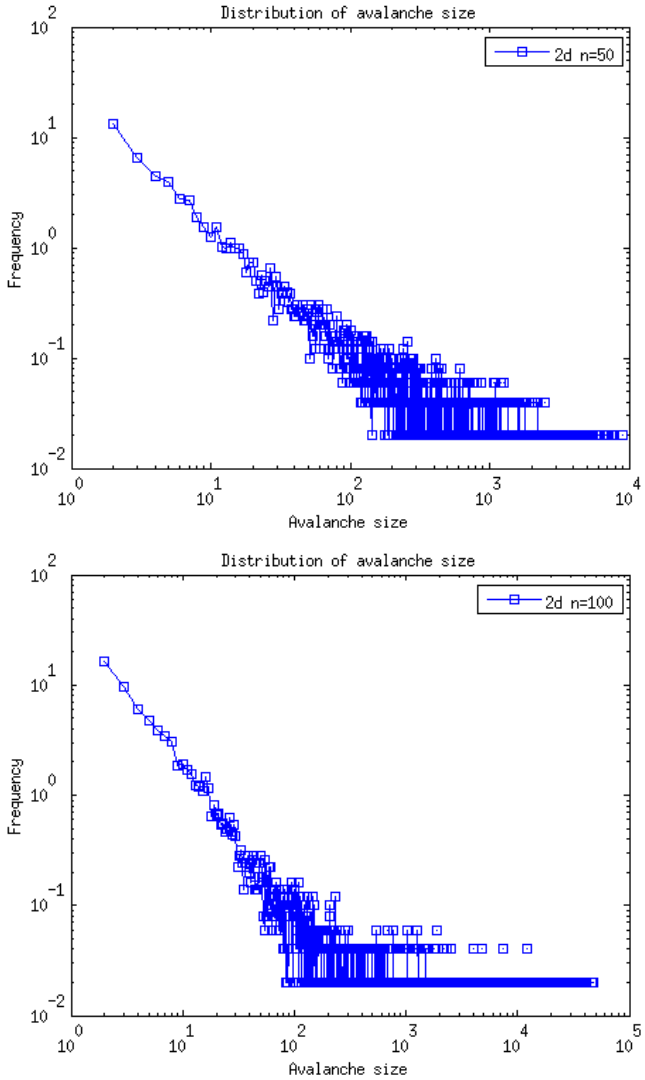

Figure 4.10: Avalanche size distribution for $2d$ lattices with different sizes and open boundary conditions.

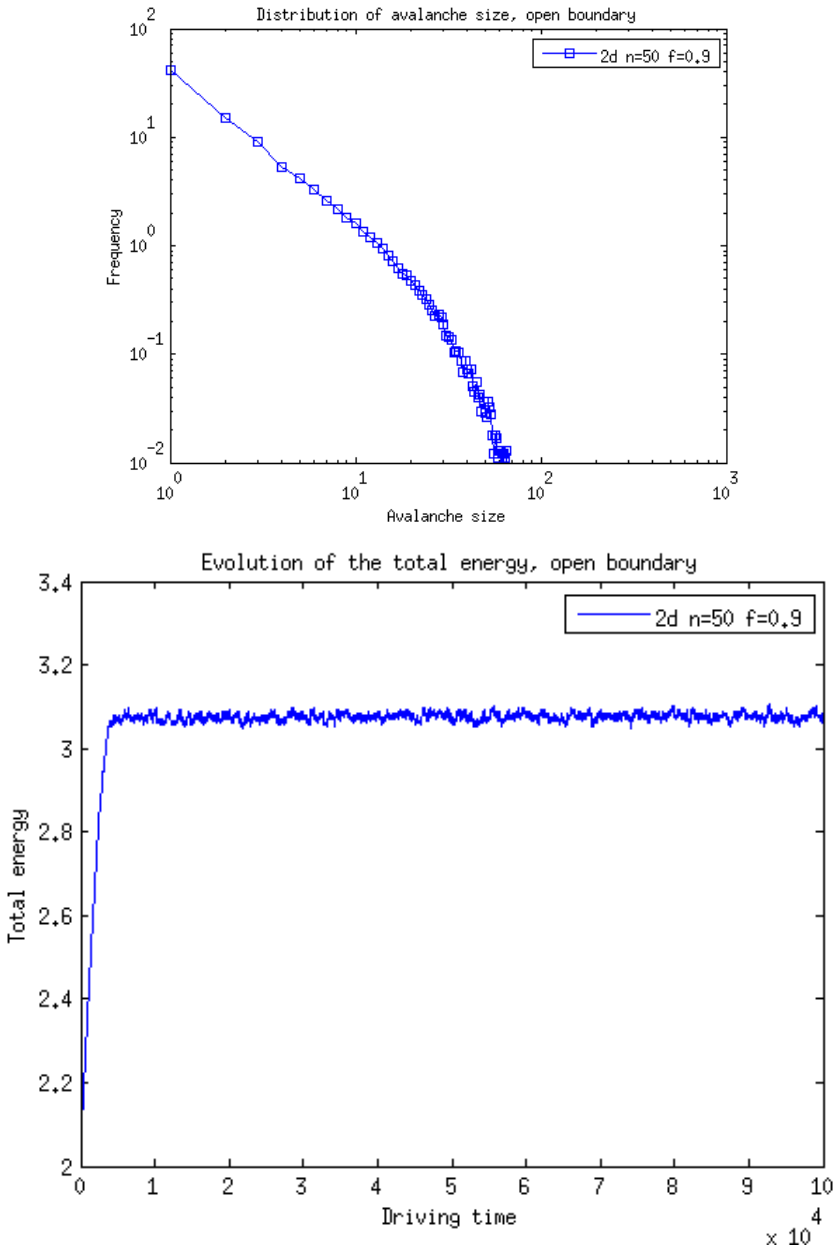

Figure 4.11: Avalanche size distribution and energy evolution for a $2d$ lattice with friction and open boundary conditions.

configurations B. In contrast, the driving time (the grain adding) makes the system critical, therefore creating the relation $B \to A$.

Because of the discrete nature of the lattice, all these relations between the discussed two sets A and B exist independently if the system is run or not. Although running the system restricts the results to only a certain part of the full distribution.

The avalanches can be characterized by their size (the total number of sites that became overcritical) or by their lifetime (the number of different configurations with some sites overcritical). By definition, the values that the size can have are bigger than the values of the lifetime (as the former refers to the number of *sites*, and the latter, to the intermediate *configurations*).

For clarification, the avalanche size and lifetime distribution of a simple case of a small lattice with finite boundary is shown in figure 4.12. The initial configuration is **minimally stable**, i.e. all sites have a value E_c-1, meaning that one grain addition in any of the site will lead to a *catastrophic* avalanche that will affect the whole system. We might consider these distributions as *exact* (the subset of 1 state overcritical configurations is almost the whole set, as the system is small enough), as they do not change after running the simulation for even longer time. These distributions are more exponetial-like ($y = c^{-x}$) than power-law-like.

What happens for larger lattices is that the size of the avalanche is restricted to the small and middle ones, so in some sense the power-law distribution is an approximation for this regime. Catastrophic avalanches that affect to the whole system never happen, except one is imposed as an initial configuration. A zero configuration will neither evolve to the minimally stable state (that leads to catastrophic events).

To check this idea, a bigger lattice of 50×50 is run in 2 dimensions with a total driving time of $T = 1\,000\,000$. The result of avalanche size distribution is shown in figure 4.13, with a respectable power-law fitting over two decades.

The catastrophic avalanches would have a size of order 10^4, occurring for minimally stable states, which will have an average energy per site of

$$< E >= (n^d(E_c - 1) + 1)/n^d = E_c - 1 + 1/n^d$$

For a large lattice, it is $< E > \sim E_c - 1$, so 3 for this case. The system evolves to an average energy per site around 2 (see figure 4.14, which starts with a random initial configuration. Starting with a minimally stable state would converge to the same value). Therefore, many theoretically possible states must be strongly suppressed. A small remnant which is not power-law-like stays. One might first consider it as a finite size effect, but it is indeed a discrete effect, as it will remain for any lattice size if the simulation is run long enough. This is due to the fact that the system is intrinsically discrete.

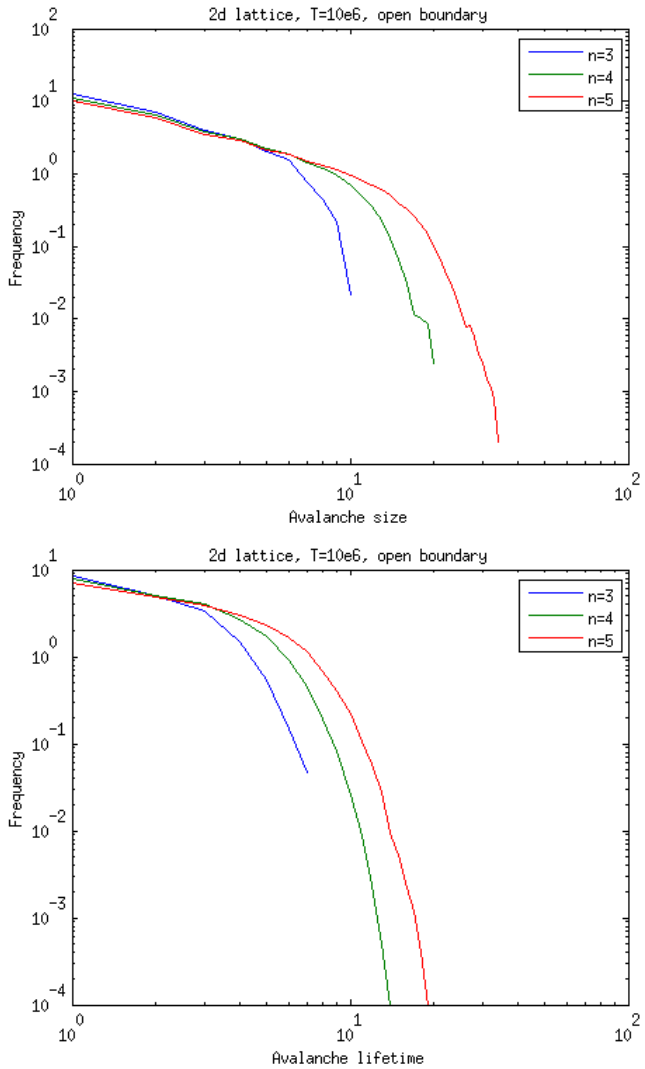

Figure 4.12: Avalanche size and lifetime distribution for small $2d$ lattices, which started with all the sites critical. These distributions are checked to fit quite well with exponential decaying distributions. The fitting parameters are not shown, as their values are not meaningful for the discussion.

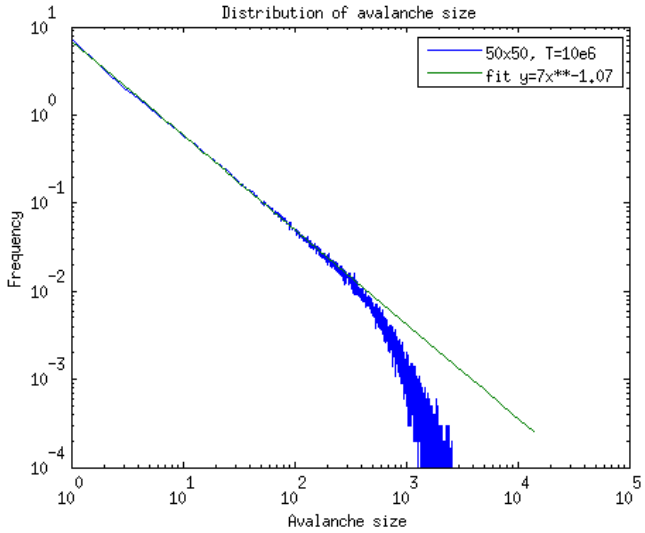

Figure 4.13: Avalanche size distribution for a $2d$ lattice.

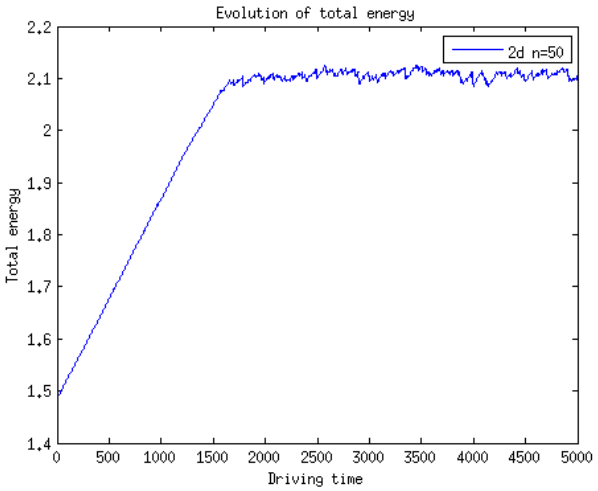

Figure 4.14: The evolution of the total energy per site for $2d$ lattice, starting with a random configuration.

n	site	av. size	av. lifetime
3	(1,1)	9	5
	(1,2)	9	4
	(2,2)	10	3
4	(1,1)	16	7
	(1,2)	16	6
	(2,2)	21	5
5	(1,1)	25	9
	(1,2)	25	8
	(1,3)	25	7
	(2,2)	34	7
	(2,3)	34	6
	(3,3)	35	5

Table 4.1: Table with catastrophic avalanches, where the *site* refers to the critical site, while other sites are minimally stable (with value $E_c - 1$).

Nevertheless, if the system is continuous, which cannot exactly be reproduced with a computer simulation, then an infinite number of possible configurations would occur while running the simulation for an infinitely large time and the result should give a power-law distribution. Of course, this is a hypothesis and should be proven mathematically, which is not part of this project. Our discussion is at the qualitative level, aiming to help understanding why sandpile models generate power-law-like distributions.

Previous plots showed an accumulated large amount of large events, but also with a lot of dispersion which is mainly statistical fluctuations. Running for long enough time, this effect should disappear and a distribution like the one in figure 4.13 should come out, comparing with the plots in figure 4.10, where T s much smaller.

As seen in all the different cases studies here, the lifetime distribution always keeps the same shape. The catastrophic avalanches do not have the largest lifetime and normally, the largest lifetime value is much less than the total number of sites. The avalanche size can in contrast outnumber it. Table 4.1 shows catastrophic avalanches in a 2-dimensional lattice. The maximum avalanche size is of the order n^d, where the lifetime is of the order $n \cdot d$. Therefore, lifetime should converge faster than size.

If one grain is always placed onto the same site at each driving time, this will lead to a *limit cycle*, where a finite set of configurations will keep repeating themselves. These states are called *recurrent states*.

The existence of limit cycles is proven analytically using the abelian

property in [4], and is also checked it for small lattices, for example

$$\begin{bmatrix} 3 & 2 & 2 \\ 3 & 4 & 3 \\ 0 & 1 & 3 \end{bmatrix}.$$

Adding a grain to the center constantly leads to a repeating but finite number of configurations, which can be checked by hand or using the the `abelian_sandpile.m` program. In fact, this happens to any initial configuration attracted by the corresponding limit cycle (many different limit cycles exist depending on where the grain is placed). Furthermore, for the periodic boundary condition, with only one grain the system will go into an infinite loop between these configurations. Using randomized grain adding avoids such situations.

Chapter 5

Conclusion and Outlook

5.1 Summary

Self-Organized Criticality (SOC) was born as a new mechanism to produce power-law behaviour, as it is common in many complex systems. During the work on this paper, the aim was to understand the underlying idea behind the concept of SOC and the focus was on the Abelian Sandpile Model (ASM), as this is the simplest and the most popular model in the literature.

A simulation environment of ASM has been created for the investigation of its properties and the simulation of the avalanche behaviour for different parameters of the model, e.g. the boundary conditions or the dimension of the cellular automation lattice. The observables of the avalanche size and lifetime were measured, which are claimed to obey power-law-like statistics in literature.

The results were, most of the time, close to power-law, especially for the avalanche size distribution. Nevertheless, these results do not necessarily give a good insight into the meaning of either self-organization nor criticality. In fact, despite the large amount of publication in this field, SOC is still a vague concept with many controversies. For example, many authors relate the power-law behaviour to fractal structure which is not proven, and seemingly, wrong.

5.2 Interpretation of SOC in ASM model

The cellular automaton of the ASM is always a discrete system, even implementing a continuous field is still limited by the size of decimal numbers. Therefore, there is a finite number of possible configurations. The critical behaviour is implemented by including the so-called *avalanche effect*, meaning that there exists a critical value to the field that associated with each lattice point, where beyond it, the site gets relaxed obeying certain *local* rules. In this case, the toppling site distributes grains to its nearest neigh-

bours, which themselves might also be overcritical and relax. In general, we can say that the avalanche effect comes down to the existence of a critical value and a local propagation rule. Note: *Local* does not necessarily imply relaxing a site to its nearest neighbours.

The many different cases shown in this work, either open or close or periodic boundary condition with friction or not, all can be understood conceptually in the same way.

The conclusion made here is that avalanches connect the set of configurations with one overcritical site to a set of subcritical configurations. As we have a finite number of configurations in total, an exact distribution of avalanche size or lifetime exists. The driving time, by adding grains (randomly or not) brings a subcritical state to an overcritical state. Hence, running the simulation for long enough, all finite configurations could be passed, and the measured distribution would be accurate.

Simulating a small lattice with a driving time comparable with the total number of possible critical configurations in 2, 3 and 4 dimensions leads to a striking result: the distribution seems to fit less a power-law, but more a kind of an exponential law.

How can this result relate to the power-law results obtained for the many other cases that were dealt with? In all these cases, either large lattices or friction or continuous grain size were introduced. Indeed, all these small changes to the model are equivalent to a discrete lattice with different local rules, but they share a common thing: an increase in the number of critical and also subcritical or *metastable* configurations. As the random energy addition and dissipation evolves and oscillates to some constant average value, the total *exact* distribution cannot be reproduced, unless using a large total driving time comparable with all possible configurations of the system, which is a combinatorial number. Hence, for large lattices it is computationally impossible and the result is limited to small or medium size avalanches. The power-law distribution is a good approximation for this part. The different cases just specify a particular way to connect the set of critical configurations to the subcritical set.

It should be possible to treat the problem analytically exploiting the symmetry of the lattice, and to deduce a general large lattice law, probably tending to a power-law approximation, as this will correspond to reducing the 'finite size' effect, i.e. the 'continuous' limit.

5.3 Criticality

The meaning of criticality is a concept borrowed from thermodynamic systems that undergo a second order phase transition (e.g. spontaneous magnetization below Curie temperature), which imply the correlation length to diverge, i.e. the whole system, independent of size, behaves in the same way,

hence there is no characteristic scale during the phase transition.

For the ASM, no such phenomenon could be observed. What has been seen here is a metastable configuration that goes critical after receiving an external input and then relaxes to another metastable configuration, keeping in mind that there is a large number of such configurations. The avalanches connect to each other, as we discussed, so no criticality is concluded here in the sense of thermodynamic systems.

5.4 Self-organization

The system does not self-organize itself to a particularly interesting state, as in the case of magnetization, where all the pins point to one direction. In fact, if one always inputs the grains in the same site, it can be proven that there exists a limit cycle, i.e. there is a finite number of configurations that repeat themselves infinitely. Hence, we might also refute the self-organization hypothesis, as the system needs external input to organize to a critical state, its relaxation just giving one of many metastable subcritical states.

5.5 Final word on SOC and simulation

All in all, the term SOC might not seem very appropriate. Nevertheless, this model is useful to understand many real systems in nonequilibrium. Systems in nonequilibrium are open systems, i.e. they receive external input. Because of this, they might undergo different metastable states, especially for large but slowly driven systems with possibilities of dissipation (avalanche). Consider for example the earthquake model, where the earthquake is the avalanche of the system, that releases energy when it finishes and brings the system back to a metastable state; then the system will be slowly driven again to a critical state, and so on and so forth. The statistics of earthquakes size will follow a power-law distribution. Note as well that this system would be considered continuous.

In conclusion, slowly driven systems with avalanche phenomena can be modelled by computational sandpile models, and we can refer to this kind of systems as SOC systems. However, the name of SOC is not necessarily appropriate, as the system is slowly driven to a critical state that relaxes itself with an avalanche phenomenon, and the process is repeated again. The statistics of avalanche for such a continuous system seem to exhibit power-laws, as it is checked by computer simulations.

Bibliography

[1] Alessandro Vespignani Alain Barrat and Stefano Zapperi. Fluctuations and correlations in sandpile models. September 1999.

[2] Kim Christensen. Self-organization in models of sandpiles, earthquakes, and flashing fireflies. August 2002.

[3] Michael Creutz. Cellular automata and self-organized criticality. November 1996.

[4] Deepak Dhar. Studying self-organized criticality with exactly solved models.

[5] Monwhea Jeng. The abelian sandpile model.

[6] Kurt Wiesenfeld Per Bak, Chao Tang. Self-organized criticality. August 1987.

[7] Kurt Wiesenfeld Per Bak, Chao Tang. Self-organized criticality: An explanation of 1/f noise. July 1987.

[8] Frank Redig Ronald Meester and Dmitri Znamenski. The abelian sandpile; a mathematical introduction. April 2001.

[9] Alessandro Vespignani and Alessandro Vespignani. How self-organized criticality works: A unified mean-field picture. June 1998.

Appendix A

MATLAB/Octave-Code

A.1 “critical field.m” - random field generation

```
function f = critical_field(width,height,critical_state,uniform)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% generates a random field/lattice for sandpile simulation
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PARAMETERS:
% width, height = size of lattice/field to be created
% critical_state = maximum/critical state of a site, usually = 3
% uniform = true will generate a field of e.g. 0's and 3's only
% uniform = false will generate a field e.g. with numbers 0 to 3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % define field using uniform distribution
  if (uniform)
    f = floor(unifrnd(0,2,height,width))*critical_state;
  else
    f = floor(unifrnd(0,critical_state+1,height,width));
  end
end
```

A.2 “sandpile.m” - 2D sandpile model simulation, parametrized

```
function [as,nc,at,final,energy] = sandpile(f, neighbour, critical_state, ...
  collapse_per_neighbour, timesteps, boundary_type, make_pictures, ...
  silent, driving_plane_reduction, var_grain, same_place)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% sandpile simulation using stack algorithm for avalanche generation
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% INPUTS
% f         field matrix
% neighbour       2xN matrix with x & y offsets of neighbours
% critical_state      critical/max. number of grains before collapse
```

```
% collapse_per_neighbour   number of grains to collapse
% timesteps      simulation duration in steps (excl. avalanches)
% boundary_type      type of boundary condition
%          1 - infinite/continuous, like pac-man
%          2 - energy loss at boundaries, table-like
%          3 - mixed. continuous in x-direction and
%           energy loss in y-direction
% make_pictures      draw and export all frames or not
%        >0 means save a pic for each t,
%        >1 means avalanches too
% silent       produces no output (except time progress) if true
% driving_plane_reduction  percentage of field close to the boundary
%        not to be affected by driving (putting grains)
%        = 0   => use whole field (default)
%        = 0.2 => put grains at least 0.2*width
%                 and 0.2*height far away from boundary
%        > 0.5 => invalid [!]
% var_grain   true/false - use random grain size [0...1]

% OUTPUTS
% as      avalanche sizes (topplings count) for each timestep
% nc      size at avalanche-starting-site for eacg t
% at      avalanche lifetime for each t
% final     final field
% energy      array of energy states for each timestep

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % translate parameters

  width = size(f,2);
  height = size(f,1);
  neighbours = size(neighbour,2);   % number of neighbours to collapse to
  neighbour_offset_x = neighbour(1,:);
  neighbour_offset_y = neighbour(2,:);
  collapse = collapse_per_neighbour;
  boundary = boundary_type;

  picture_counter = 0;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % define stack for avalanches
  stack_x = 0;
  stack_y = 0;
  stack_n = 0;

  % avalanche statistics
  avalanche_sizes = zeros(1, timesteps);
  av_begin_t = zeros(1,timesteps);
  avalanche_add = zeros(1,timesteps); % = av_size - av_ltime

  % energy
  ee = zeros(1,timesteps);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % show starting field
  if (silent==false)
    disp('starting from this field:');
    disp(f);
  end

  for t=1:timesteps
    % display time progress
    disp(['time: ' num2str(t) ' / ' num2str(timesteps)]);

    % choose random site
    if (same_place)
      x=floor(width/2);
      y=floor(height/2);
    else
      y=floor(unifrnd(1,height*(1-2*driving_plane_reduction)) + ...
        height*driving_plane_reduction);
      x=floor(unifrnd(1,width*(1-2*driving_plane_reduction)) + ...
        width*driving_plane_reduction); % uniform distribution rnd
    end

    % place grain
    f(y,x) = f(y,x) + 1;

    % communicate
    if (silent==false)
      disp(['new grain on x' num2str(x) ',y' num2str(y)]);
    end
```

```
% save picture of field before collapsing (incl. active field)
if (make_pictures>0)
  draw_field(f,2);
  title(['random grain on x' num2str(x) ',y' num2str(y)]);
  picture_counter=picture_counter+1;
  print(['field' sprintf('%04.0f', picture_counter) '.png'],'-dpng');
end

% push site to stack
stack_n = 1;
stack_x(1) = x;
stack_y(1) = y;

% save avalanche starting site
av_begin_x = x;
av_begin_y = y;
av_begin_t(t) = 0; % # topplings at avalanche starting site

% avalanche - work through stack
while (stack_n > 0)

  % pop from stack
  x = stack_x(stack_n);
  y = stack_y(stack_n);
  stack_n = stack_n - 1;

  % display current site
  if (silent==false)
    disp(['current site: x ' num2str(x) '; y ' num2str(y)]);
  end

  % check if overcritical/active
  if (f(y,x) > critical_state)

    % communicate collapsing
    if (silent==false)
      disp('collapse!');
    end

    % save avalanche size for statistics
    avalanche_sizes(t) = avalanche_sizes(t) + 1;
    if ((x==av_begin_x) && (y==av_begin_y))
      % save # topplings at av starting site
      av_begin_t(t) = av_begin_t(t) + 1;
    end

    % collapse/topple
    f(y,x) = f(y,x) - neighbours * collapse;

    % count future topplings to be caused by this toppling
    future_topplings = 0;

    % look at every neighbour
    for n=1:neighbours

      % communicate
      if (silent==false)
        disp(['neighbour ' num2str(n)]);
      end

      %%%%%% check boundary %%%%%%

        % 1) no-boundary conditions (continuous field, pack-man style)
        if (boundary == 1)

          % modify neighbour offsets
          if (y+neighbour_offset_y(n) < 1)
            neighbour_offset_y(n) = neighbour_offset_y(n) + height;
          end
          if (y+neighbour_offset_y(n) > height)
            neighbour_offset_y(n) = neighbour_offset_y(n) - height;
          end
          if (x+neighbour_offset_x(n) < 1)
            neighbour_offset_x(n) = neighbour_offset_x(n) + width;
          end
          if (x+neighbour_offset_x(n) > width)
            neighbour_offset_x(n) = neighbour_offset_x(n) - width;
          end

          % add/transport grain to neighbour
          f(y+neighbour_offset_y(n),x+neighbour_offset_x(n)) = ...
            f(y+neighbour_offset_y(n),x+neighbour_offset_x(n)) + collapse;

          % push neighbour to stack
          stack_n = stack_n + 1;
          stack_x(stack_n) = x + neighbour_offset_x(n);
```

```
    stack_y(stack_n) = y + neighbour_offset_y(n);

    % count future topplings to be caused by this toppling
    if (f(y+neighbour_offset_y(n),x+neighbour_offset_x(n)) == (critical_state+1))
      future_topplings = future_topplings + 1;
    end

  % 2) energy loss at boundary (table style)
  elseif (boundary == 2)

    % keep offsets, but check if outside of boundary
    if ((y+neighbour_offset_y(n) < 1) || ...
       (y+neighbour_offset_y(n) > height) || ...
       (x+neighbour_offset_x(n) < 1) || ...
       (x+neighbour_offset_x(n) > width))
      % outside of boundary...do nothing =)
    else
      % add/transport grain to neighbour
      f(y+neighbour_offset_y(n),x+neighbour_offset_x(n)) = ...
        f(y+neighbour_offset_y(n),x+neighbour_offset_x(n)) + collapse;

      % push neighbour's neighbours to stack
      stack_n = stack_n + 1;
      stack_x(stack_n) = x + neighbour_offset_x(n);
      stack_y(stack_n) = y + neighbour_offset_y(n);

      % count future topplings to be caused by this toppling
      if (f(y+neighbour_offset_y(n),x+neighbour_offset_x(n)) == (critical_state+1))
        future_topplings = future_topplings + 1;
      end
    end
  % 3) mixed (1 for x and 2 for y)
  elseif (boundary == 3)

    % for x-direction -> continuous boundary
    if (neighbour_offset_y(n)==0)

      % modify neighbour offsets
      if (y+neighbour_offset_y(n) < 1)
        neighbour_offset_y(n) = neighbour_offset_y(n) + height;
      end
      if (y+neighbour_offset_y(n) > height)
        neighbour_offset_y(n) = neighbour_offset_y(n) - height;
      end
      if (x+neighbour_offset_x(n) < 1)
        neighbour_offset_x(n) = neighbour_offset_x(n) + width;
      end
      if (x+neighbour_offset_x(n) > width)
        neighbour_offset_x(n) = neighbour_offset_x(n) - width;
      end

      % add/transport grain to neighbour
      f(y+neighbour_offset_y(n),x+neighbour_offset_x(n)) = ...
        f(y+neighbour_offset_y(n),x+neighbour_offset_x(n)) + collapse;

      % push neighbour to stack
      stack_n = stack_n + 1;
      stack_x(stack_n) = x + neighbour_offset_x(n);
      stack_y(stack_n) = y + neighbour_offset_y(n);

      % count future topplings to be caused by this toppling
      if (f(y+neighbour_offset_y(n),x+neighbour_offset_x(n)) == (critical_state+1))
        future_topplings = future_topplings + 1;
      end

    % for y-direction -> open boundary
    else

      % keep offsets, but check if outside of boundary
      if ((y+neighbour_offset_y(n) < 1) || ...
         (y+neighbour_offset_y(n) > height) || ...
         (x+neighbour_offset_x(n) < 1) || ...
         (x+neighbour_offset_x(n) > width))
        % outside of boundary...do nothing =)
      else
        % add/transport grain to neighbour
        f(y+neighbour_offset_y(n),x+neighbour_offset_x(n)) = ...
          f(y+neighbour_offset_y(n),x+neighbour_offset_x(n)) + collapse;

        % push neighbour's neighbours to stack
        stack_n = stack_n + 1;
        stack_x(stack_n) = x + neighbour_offset_x(n);
        stack_y(stack_n) = y + neighbour_offset_y(n);

        % count future topplings to be caused by this toppling
```

```
                if (f(y+neighbour_offset_y(n),x+neighbour_offset_x(n)) == (critical_state+1))
                  future_toppling = future_toppling + 1;
                end
              end
            end
          end

          %%%%%%%%%%%%%%%%%%%%%%%%%%%%
        end

        % calculate additional topplings caused
        if (future_toppling > 0)
          avalanche_add(t) = avalanche_add(t) + future_toppling - 1;

          % communicate additional topplings to come
          if (silent==false)
            disp(['this collapse generates ' ...
              num2str(future_toppling - 1) ...
              ' additional toppling(s)']);
          end
        else
          % communicate additional topplings to come
          if (silent==false)
            disp(['this collapse generates no additional topplings']);
          end
        end

        % save picture of avalanche timestep
        if (make_pictures>1)
          draw_field(f,2);
          title(['avalanche...']);
          picture_counter=picture_counter+1;
          print(['field' sprintf('%04.0f', picture_counter) '.png'],'-dpng');
        end
    end
      end

    % display field after collapsing
    if (silent==false)
      disp(f);
      disp('');
    end

    % calculate energy
%   for fx=1:width
%     for fy=1:height
%       ee(t)=ee(t)+f(fy,fx)^2;
%     end
%   end
    ee(t) = sum(sum(f.^2));
  end

  % return avalanche sizes
  as = avalanche_sizes;

  % return number of topplings at avalanche starting site
  nc = av_begin_t;

  % return final state
  final = f;

  % return avalanche lifetimes
  at = avalanche_sizes - avalanche_add;

  % return energy
  energy = ee;
end
```

A.3 “avalanche distribution analysis.m”

```
function [a,b,a2,b2] = avalanche_distribution_analysis( ...
  avalanche_sizes,avalanche_lifetimes)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% analysis avalanche distribution and fits it to a power-law
%
%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% INPUTS
% avalanche_sizes   array of avalanche size for each timestep
% avalanche_lifetimes same for av. lifetime

% OUTPUTS
% a,b     coefficients of power law P(s) = a*s^b
% a2,b2     coefficients of power law P(t) = a2*t^b2


  % count avalanche sizes/lifetimes
  avalanche_count = zeros(1,max(avalanche_sizes)); % init
  for s=1:max(avalanche_sizes)
    avalanche_count(s) = size(avalanche_sizes(avalanche_sizes==s),2);
  end
  avalanche_count2 = zeros(1,max(avalanche_lifetimes)); % init
  for t=1:max(avalanche_lifetimes)
    avalanche_count2(t) = size(avalanche_lifetimes(avalanche_lifetimes==t),2);
  end

  % non-zero filter
  xx = [1:max(avalanche_sizes)];
  yy = avalanche_count(1:end);
  xx = xx(yy>0);
  yy = yy(yy>0);

  xx2 = [1:max(avalanche_lifetimes)];
  yy2 = avalanche_count2(1:end);
  xx2 = xx2(yy2>0);
  yy2 = yy2(yy2>0);

  % plot avalanche count vs size
  figure;
  subplot(2,2,1);
  plot(xx,yy,'marker','s');

  % fit the curve into power law distribution (f = c1*x^c2)
  [c,fval,info,output]=fsolve(@(c)((c(1).*xx.^c(2))-yy),[100,1]);
  hold on;
  plot(xx,c(1).*xx.^c(2),'r');
  xlabel('avalanche size s');
  ylabel('avalanche count P(s)');
  title(['avalanche distribution and power-law-fit P(s)=' ...
    num2str(c(1)) '*s^ ' num2str(c(2))]);

  % same on a log-log-scale plot
  subplot(2,2,2);
  loglog(xx,yy,'marker','s');
  hold on;
  loglog(xx,c(1).*xx.^c(2),'r');
  xlabel('avalanche size s');
  ylabel('avalanche count P(s)');
  title(['avalanche distribution and power-law-fit P(s)=' ...
    num2str(c(1)) '*s^ ' num2str(c(2))]);

  % return coefficients
  a = c(1);
  b = c(2);

  % plot avalanche count vs lifetime
  subplot(2,2,3);
  plot(xx2,yy2,'marker','s');

  % fit the curve into power law distribution (f = c1*x^c2)
  [c,fval,info,output]=fsolve(@(c)((c(1).*xx2.^c(2))-yy2),[100,1]);
  hold on;
  plot(xx2,c(1).*xx2.^c(2),'r');
  xlabel('avalanche lifetime t');
  ylabel('avalanche count P(t)');
  title(['avalanche distribution and power-law-fit P(t)=' ...
    num2str(c(1)) '*t^ ' num2str(c(2))]);

  % same on a log-log-scale plot
  subplot(2,2,4);
  loglog(xx2,yy2,'marker','s');
  hold on;
  loglog(xx2,c(1).*xx2.^c(2),'r');
  xlabel('avalanche lifetime t');
  ylabel('avalanche count P(s)');
  title(['avalanche distribution and power-law-fit P(t)=' ...
    num2str(c(1)) '*t^ ' num2str(c(2))]);

  % return coefficients
  a2 = c(1);
```

```
  b2 = c(2);
end
```

A.4 “test sandpile.m” - sandpile simulation test environment

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% sandpile simulation environment
%
%

%f = critical_field(50,50,3,false);
%f = 3*ones(50,50);
f = zeros(30,30);

%f, neighbour, critical_state, ...
% collapse_per_neighbour, timesteps, boundary_type, make_pictures, ...
% silent, driving_plane_reduction, var_grain, same_place
[s,nc,ts,f,energy] = sandpile(f, [-1 +1 0 0; 0 0 -1 +1], 3, ...
      1, 5000, 3, 0, ...
      true, 0, false, true);

[a,b,c,d] = avalanche_distribution_analysis(s,ts)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% not yet implemented:
% - continuous grain placing with grain size (0...1)
% - h parameter
```

A.5 “abelian sandpile.m” - n-dimensional sandpile simulation w. friction

```
% By Xinyi Chen

% Abelian Sandpile Model (ASM)
% This program studies d-dimensional ASM,
% for either discrete case or continuous case:
%    *Discrete ¬ uniform grain size, i.e. 1
%    *Real ¬ random grain size, (0,1)
% with periodic boundary conditions in all directions
% or finite boundary conditions, where we have 2 possibilities:
%    *open boundary (default for the discrete case)
%    *closed boundary (default for the real case)
% Either periodic or closed boundary will accumulate the driving grain
% and make the system be overcritical with large driving time, for these
% cases, it is better to introduce some friction in the propagation.
% For the discrete case


clear all
clc

% Parameters-------------------------------------------------------------
d=2;             % Lattice dimension
n=100;            % size per dimension
T=1;          % Driving time (¬ one grain addition/time)
periodic_boundary=0; % 0=false (finite boundary with grain lost)
real=0;    % 1=true -> real field
f=1;             %'friction', f=1->no grain lost
%-----------------------------------------------------------------------

N=power(n,d);    % Lattice size
```

```
Ec=2*d;          % Critical value (>¬2*d)

size=zeros(T,1);      % avalanche size = total number of sites that toppled
lifetime=zeros(T,1); % avalanche lifetime
sumE=zeros(T,1);            % total energy of the system

stack=zeros(N,1);
stack_site=zeros(N,N);% to save the coordinates of critical neighbours
r=zeros(2*d,1);
%critical_site=zeros(max,2*d,1); % save critical sites

% Initial configuration
if (real==1)
    E=rand(N,1)*Ec;
else
 %  E=randi([0,Ec-1],N,1); % Random integer numbers from 0 to Ec-1


   E=ones(N,1)*(Ec-1); % to test
end


for time=1:T

  % v=5; E(v)=Ec; % to test

  % Select a random site to be critical
    v=randi(N,1,1); E(v)=E(v)+1;


    a=0; % Counter of avalanche size
    t=1; % Counter of timesteps
    stack(t)=1;
    stack_site(stack(t),t)=v;

    critical=1;
    while(critical==1)
        u=t;

        while(stack(t)>0)

            v=stack_site(stack(t),t);
            stack(t)=stack(t)-1;
            Enew=E(v);

            if (Enew≥Ec)
                c=0;
                a=a+1;
                nn=neighbour(v,d,n, periodic_boundary);

                for k=1:2*d
                    vnn=linear_index(nn(k,:),d,n);
                    if(vnn>0) % vnn=0 -> outside lattice (see neighbour)

                        if (real==1)
                            r(k)=rand(1);
                            c=c+r(k); %closed boundary
                        else
                            r(k)=1;
                            %c=c+r(k); %closed boundary
                            c=2*d; %open boundary
                        end

                        E(vnn)=E(vnn)+r(k);

                        if (E(vnn)≥Ec)
                            critical=1;
                            u=t+1;
                            stack(u)=stack(u)+1;
                            stack_site(stack(u),u)=vnn;
                        end
                        E(v)=(Enew-c)*f; % no lost in the boundary for the real case, unless f<1
                    end
                end
 %               r
            end
        end
% E
% u
        if (u==t || u==N) % u==N to cut off very large avalanches
            critical=0;
        else
            t=u;
        end

    end
```

```
     size(time)=a;
    lifetime(time)=t;
%      sumE(time)=sum(E);
% E
end

% Saving data
% sE=sprintf('e%dd%dn%gf.dat', d,n,f);
% ss=sprintf('xs%dd%dn%gf.dat', d,n,f);
% st=sprintf('xt%dd%dn%gf.dat', d,n,f);
% save(sE, '-ascii', 'sumE')
% save(st, '-ascii', 'lifetime')
% save(ss, '-ascii', 'size')


% Obtain the statistics of avalanche distribution
% tabs=tabulate(size);
% tabt=tabulate(lifetime);
%
% % Plot figures
%
% namelegend=sprintf('%dd n=%d', d,n);
%
% figure(1);
% loglog(tabs(:,1), tabs(:,3),'marker', 's', 'color', 'b')
% xlabel('Avalanche size')
% ylabel('Frequency')
% title('Distribution of avalanche size, open boundary')
% legend(namelegend)
% %xlim([0 1000])
% %ylim([0.01 100])
%
% figure(2);
% loglog(tabt(:,1), tabt(:,3),'marker', 's', 'color', 'r')
% xlabel('Avalanche lifetime')
% ylabel('Frequency')
% title('Distribution of avalanche lifetime, open boundary')
% legend(namelegend)
% xlim([0 100])

%
% figure(3)
% plot((1:T), sumE/N)
% xlabel('Driving time')
% ylabel('Total energy')
% title('Evolution of the total energy, open boundary')
% legend(namelegend)
%
```

A.6 “coordinate.m”

```
% This function gives the coordinate of a given site
% with the linearized index v for a d-dimension array.
% It is the inverse function for linear_index function.
% n=size per one dimension (suppose ni=n for all i=1,...,d)

function[x]=coordinate(v,d,n)

x=zeros(1,d);    % coordinates (x(1),x(2),..., x(d))
va=zeros(1,d+1); % an auxiliary parameter
va(d+1)=v-1;
  for i=d:-1:1
    va(i)=mod(va(i+1),power(n,i-1));
    x(i)=1+(va(i+1)-va(i))/power(n,i-1);
  end
end
```

A.7 “linear index.m”

```
% v is the linearized index for a d-dimension array
% Example for d=3: v=(x3-1)*n*n+(x2-1)*n+x1
% n=size per one dimension (suppose ni=n for all i=1,...,d)

function[v]=linear_index(x,d,n)
v=1;
for i=1:d
    v=v+(x(i)-1)*power(n,i-1);
end
end
```

A.8 “neighbour.m”

```
% By Xinyi Chen

% This function gives the nearest neighbours coordinates
% of a given site v (the linealized index), in d-dimension.
% n=size per one dimension, n_i=n for all i=1,...,d (hypercube)

% It needs coordinate(v,d,n) function.
% The inverse function of coordinate(v,d,n) is linear_index(v,d,n), but not used here.

% Example: lattice sites for d=2, n=3
%
%     (x(1),x(2))                    v
%
% (1,1)+--(1,2)+--(1,3)+        1+--4+--7+
%      |       |       |         |   |   |
% (2,1)+--(2,2)+--(2,3)+   ->   2+--5+--8+
%      |       |       |         |   |   |
% (3,1)+--(3,2)+--(3,3)+        3+--6+--9+

% Periodic boundary conditions
% The nearest neighbours of v=8 are:
% direction dir=1: forward  (or +) v=9 or (3,3)
%                  backward (or -) v=7 or (1,3)
% direction dir=2: forward  (or +) v=2 or (2,1) (periodic boundary)
%                  backward (or -) v=5 or (2,2)

% periodic_boundary = 1 true
%                     0 false (finite)

function[nn]=neighbour(v,d,n,periodic_boundary)

x=coordinate(v,d,n);
nn=zeros(2*d,d);    % 2d nearest neighbours coordinates
b=zeros(1,d);       % for backward neighbours (-)
f=zeros(1,d);       % for forward neighbours (+)
N=power(n,d);


for dir=1:d % space direction
  for i=1:d % coordinate components

      if (i==dir)
          p=0; q=1;
      else
          p=1; q=0;
      end


    b(i)=p*x(i)+q*(x(i)-1);
    f(i)=p*x(i)+q*(x(i)+1);

    % Periodic boundary conditions
    if (periodic_boundary==1)
    if (b(i)==0)
      b(i)=n;
    end

    if (f(i)==0)
      f(i)=n;
```

```
    end

    if (b(i)>n)
      b(i)=1;
    end

    if (f(i)>n)
      f(i)=1;
    end
    end
    %-----------------------------------


    % Finite boundary conditions
    % Sites outside of the lattice will give negative linear index
    if (periodic_boundary==0)

    if (b(i)==0)
      b(i)=-N;
    end

    if (f(i)==0)
      f(i)=-N;
    end

    if (b(i)>n)
      b(i)=-N;
    end

    if (f(i)>n)
      f(i)=-N;
    end

    end
    %-----------------------------------


  end

  nn(dir,:)=b;
  nn(d+dir,:)=f;

end
end
```