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 criticallity 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 synonim 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 nerveous 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 nonexistant. 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 neighbourhood, 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 [8].

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 neighbourhood.

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 existance of power-law distributions in both spacial and temporal fluctuations. Avalanche sizes (spacial) and lifetimes (temporal), as described later in section 3.4, can both show power-law behaviour of the form $f^{-a} \approx f^{-1}$.

Unfortunately, this type of correlation doesn't necesserily imply that the system is critical, i.e. non-critical systems can also show f^{-1}-behaviour. The key idea 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 *spacial 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 adjascent 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 clasically 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-Neighbourhood*, which consists of the four nearest neighbours. A possible alternative would be e.g. the *Moore-Neighbourhood*, 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.

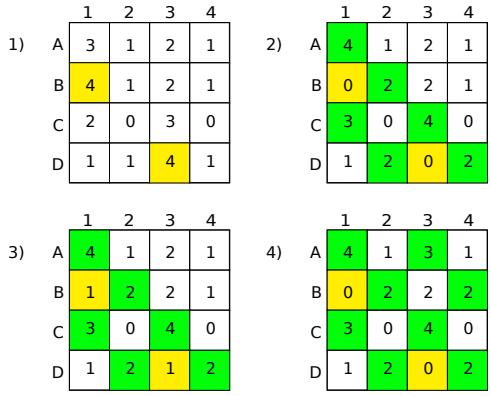

Figure 2.1: Effect of different types of boundary conditions on a sample lattice (1): open (2), closed (3) and periodic (4)

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 [9].

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.

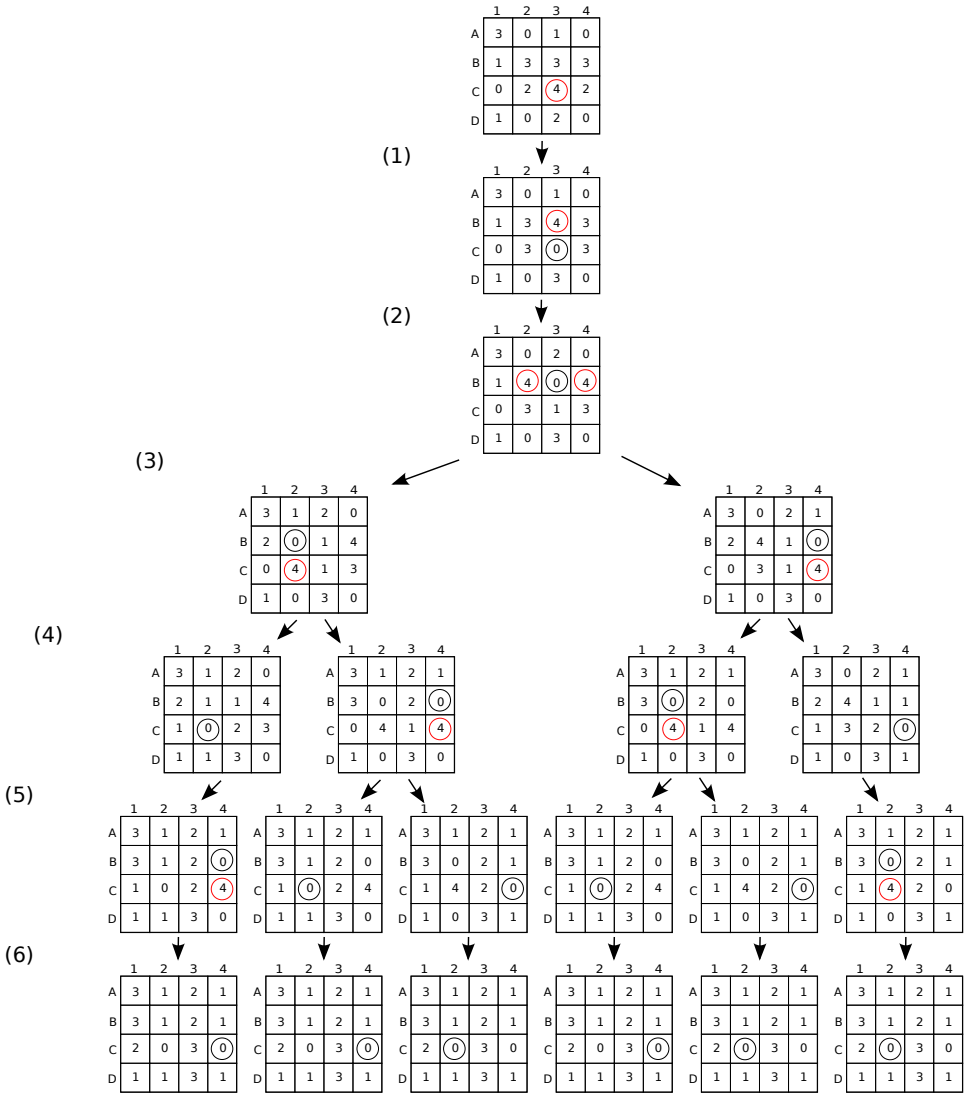

Figure 2.2: Demonstration of the abelian property: six different orders of topplings all lead to the same equilibrium state. The sites circled red indicate sites that have just become active, those circled black have just collapsed. Here, continuous boundary conditions have been used.

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));
```

Another interesting starting point is a *uniform* critical field, where every site is either 0 or z_c.

```
f = floor(unifrnd(0,2,height,width))*critical_state;
```

When the field is ready, a global loop runs through a defined number of timesteps, 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 Simple Avalanche Code

...ASDF...

3.3 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.

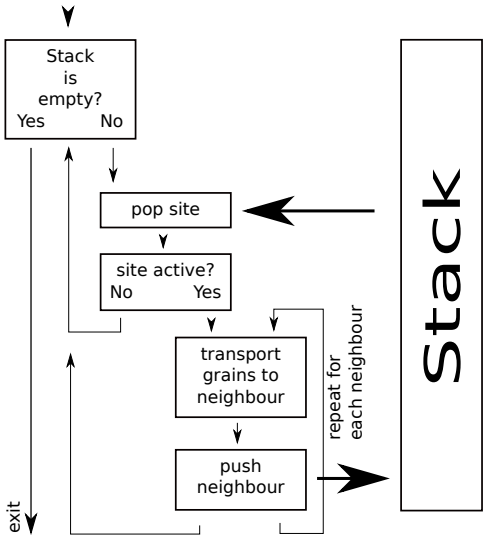

Figure 3.1: using a stack for avalanche calculation

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.

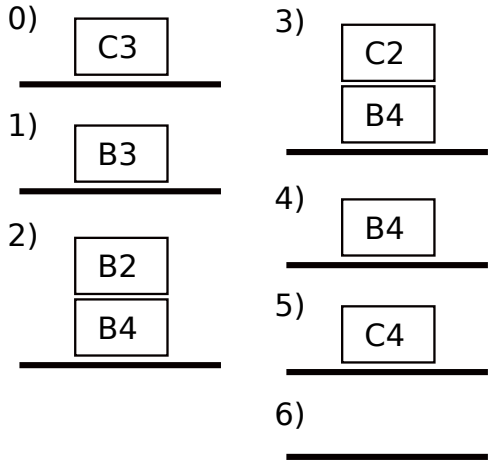

Figure 3.2: A sample sequence of stack states. The algorithm proceeds until the stack is empty.

The main loop including the stack feature looks like this:

```
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
```

3.4 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 timestep must be counted. The reason for this is that the stack algorhithm does not follow a timescale and therefore the number of timesteps 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 timestep 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;
```

Chapter 4

Simulation Results and Discussion

4.1 Fractals

As discussed in section 1.4, in order to show that a dynamical system exhibits SOC, some fractal nature must be present. As done analytically for a 1-D-model in [6], we can plot the energy of a system over time according to the definition of normalized energy

$$E(t) = \int_f h(x, y, t)^2 df$$

where h is the height of a site at position (x, y) of the field f at time t. In MATLAB/Octave, at the end of every main loop iteration the energy is calculated:

```
ee(t) = sum(sum(f.^2));
```

As the energy series in 1-dimensional case shows fractal properties (see [6]), the question arises if this is true for a 2-dimensional case.

4.2 Power-law Distributions

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

4.2.1 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 shize 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.

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 intereset, 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.

The periodic boundary in principle implies lattice size independence, which will represent a computational advantage, as a small lattice can be chosen. However, the driving time will increase in order to get enough statistics.

Results of this situation for $d = 2$, with $n = 10$, $n = 50$ and $n = 100$ are shown in figures 4.7 and 4.8. 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 shows worse statistics, which do not differ much from the case in figure 4.1.

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

From figures 4.7 and 4.8 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 4.2 and 4.3 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 4.4. 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 4.5). Furthermore, for $d = 1$, no critical behaviour is seen, in agreement with the prediction of theory presented in [7] (the situation for open boundary has also been checked to be the same). Refer to figure 4.6 for the results.

4.2.2 Finite boundary conditions

In the case of finite boundaries, grains drop automatically when boundary sites topple. However, unlike the previous case, finite size effects must be dealt with.

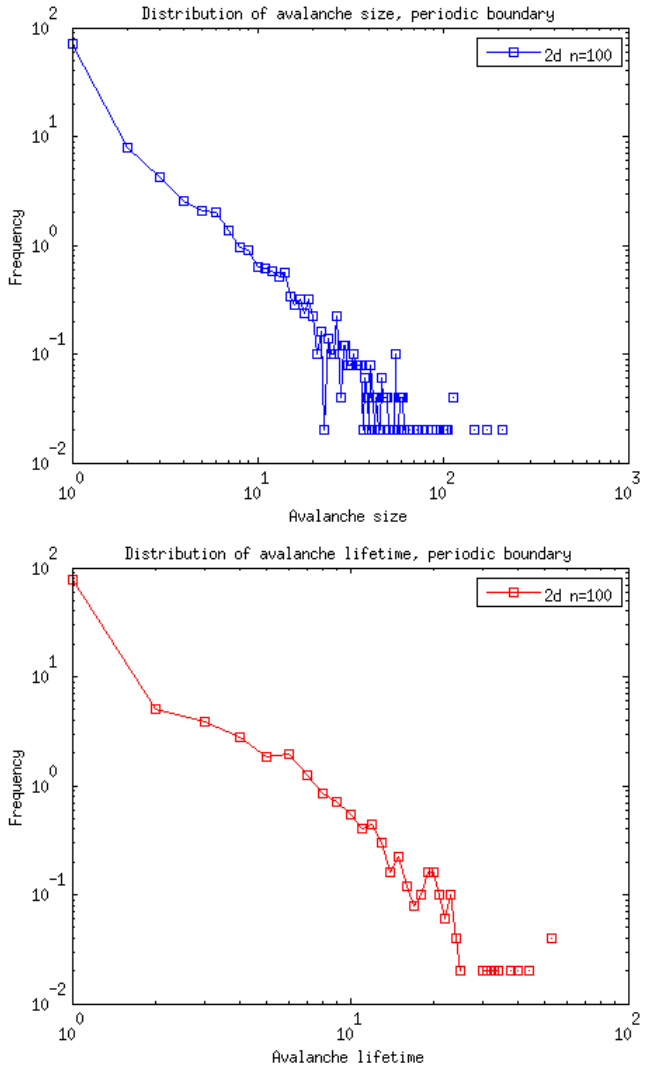

Figure 4.1: Avalanche size and lifetime distribution for a 100×100 lattice, with periodic boundary conditions. The driving time is $T = 5000$.

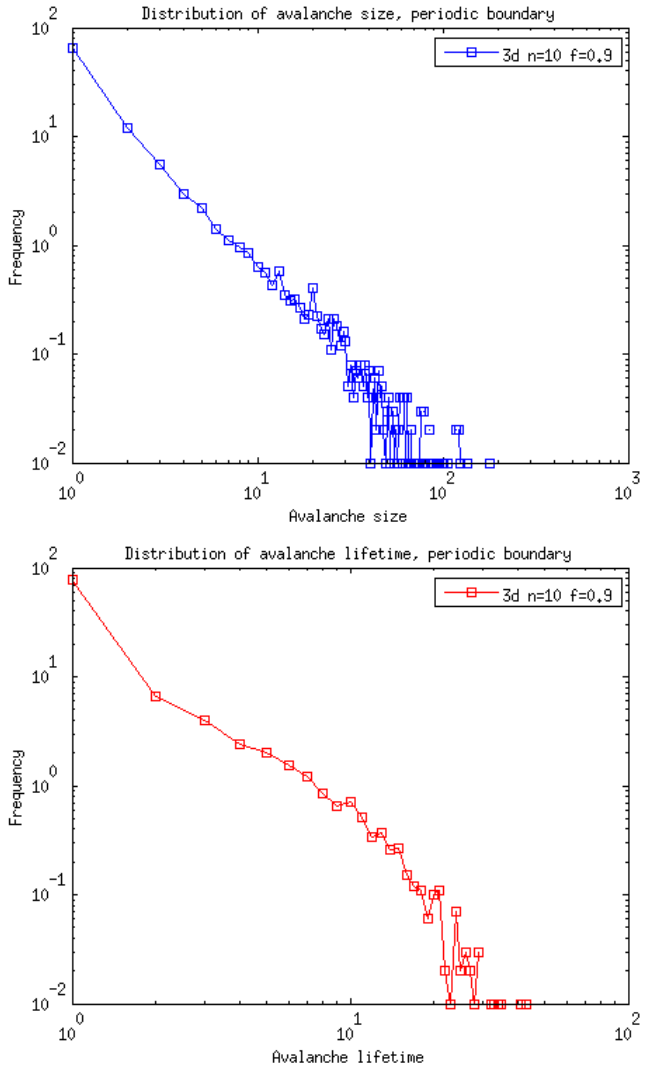

Figure 4.2: Avalanche size and lifetime distribution for a $3d$ lattice with friction and periodic boundary conditions.

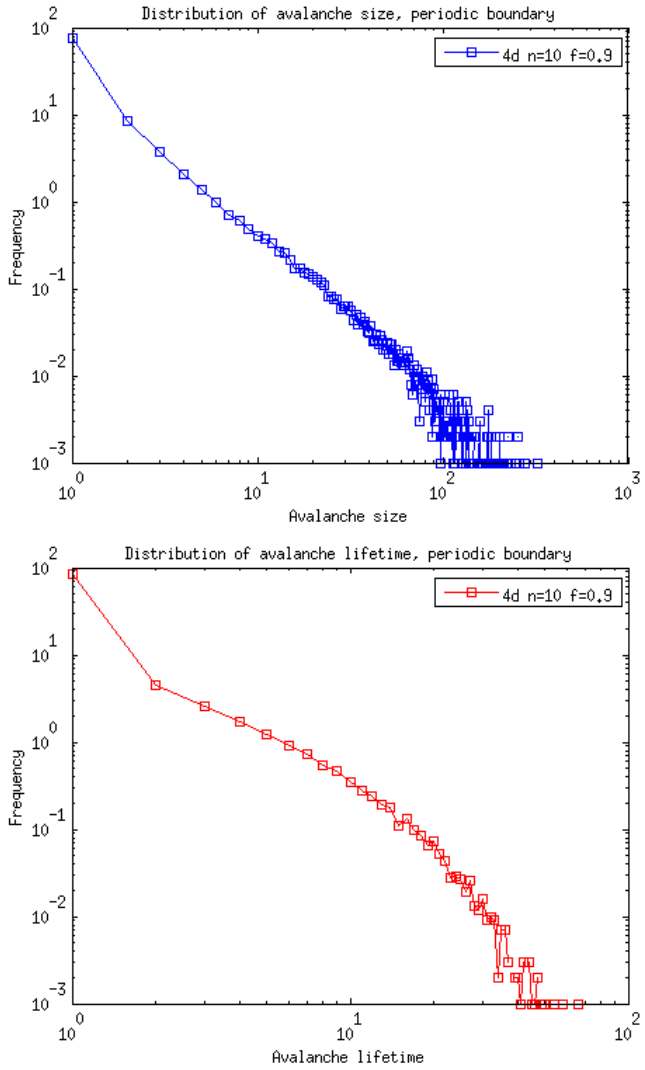

Figure 4.3: Avalanche size and lifetime distribution for a $4d$ lattice with friction and periodic boundary conditions.

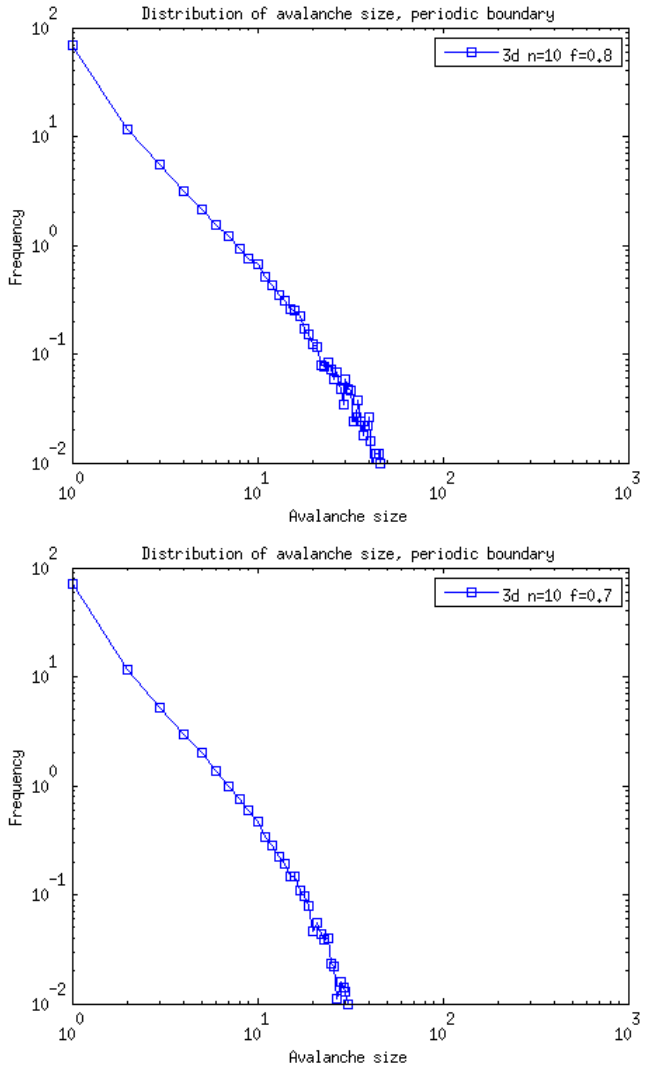

Figure 4.4: Avalanche size distribution for a $3d$ lattice with different friction parameters using periodic boundary conditions.

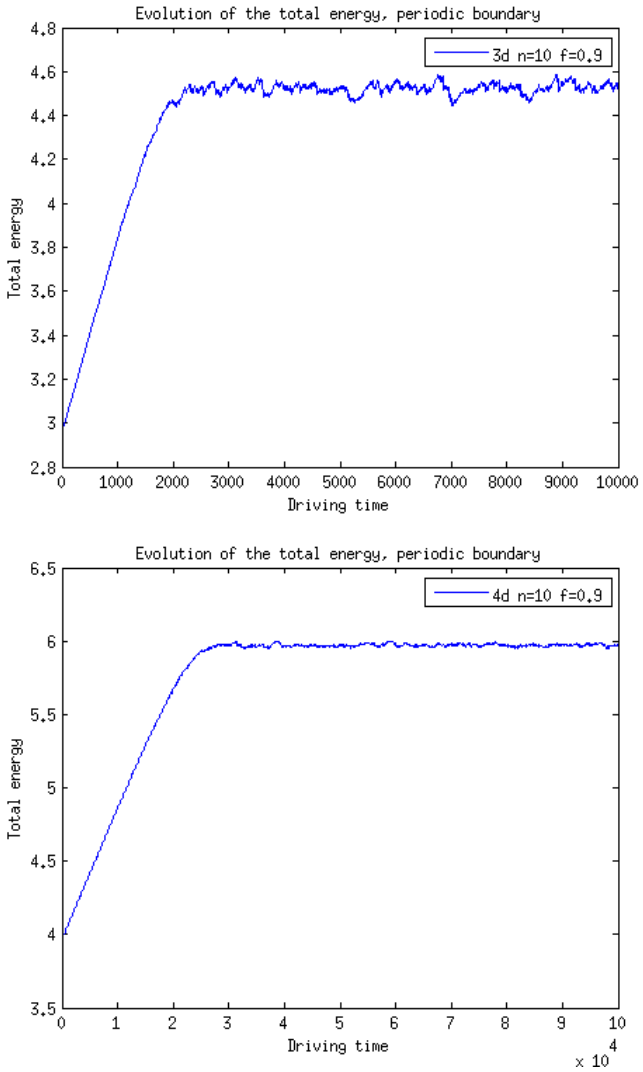

Figure 4.5: Normalized total energy evolution for $3d$ and $4d$ lattices with friction and periodic boundary conditions.

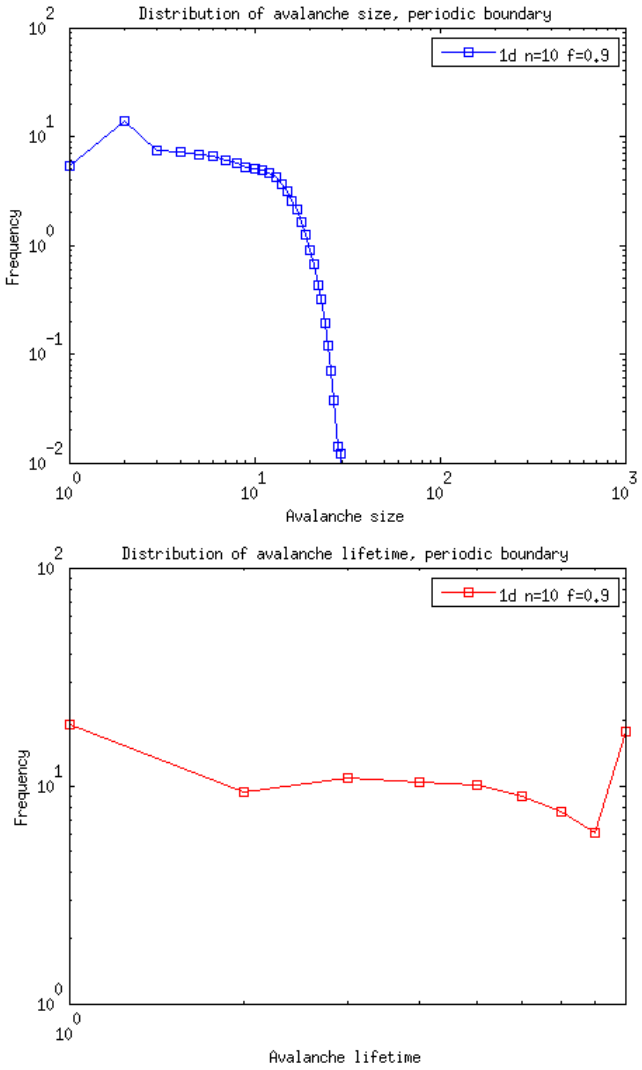

Figure 4.6: Avalanche size and lifetime distribution for $1d$ lattices with friction, periodic boundary conditions and $T = 100000$. There is no criticality for this case.

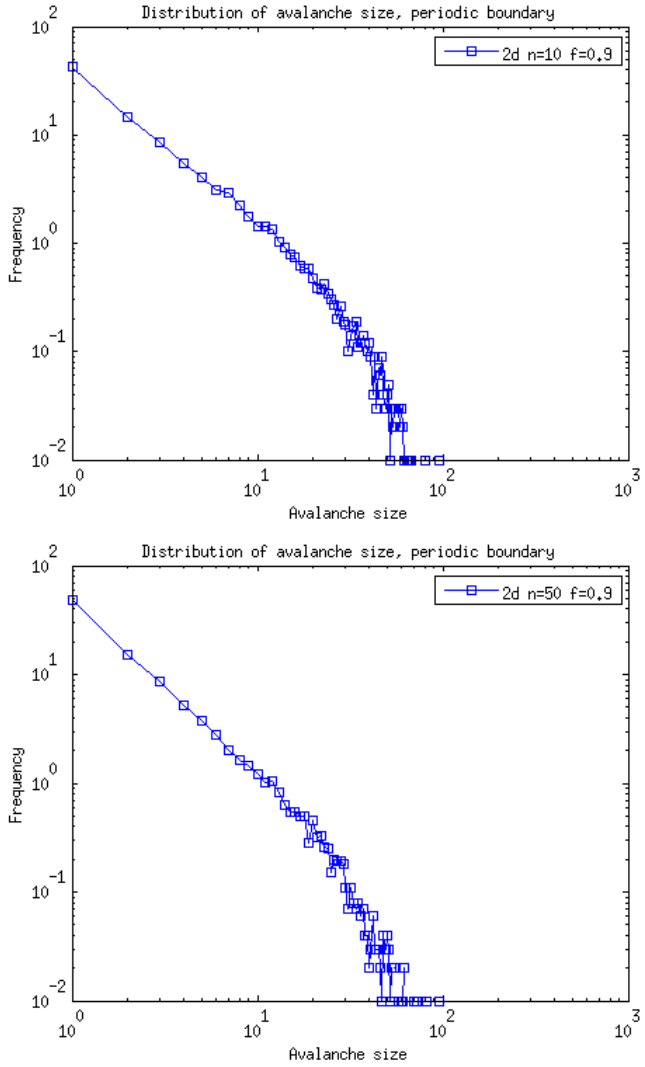

Figure 4.7: Avalanche size distribution for a $2d$ lattice with friction and periodic boundary conditions. The driving time is $T = 10000$ and the lattice size is $n = 10$ and $n = 50$, respectively.

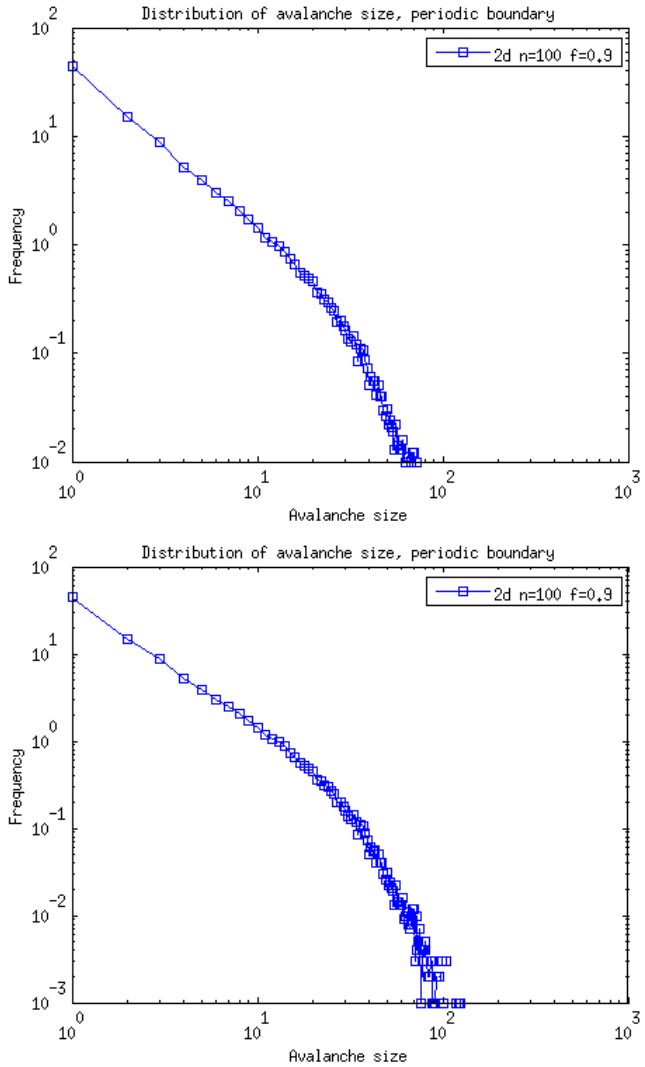

Figure 4.8: Avalanche size distribution for a $2d$ lattice with friction and periodic boundary conditions. The driving time is $T = 100000$. The first plot is restricted in the y-direction for a good comparison with the $n = 10$ and $n = 50$ cases. The second one shows the complete statistics, where a cut-off due to friction is seen clearly.

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. Again, friction is cruxial for generating a power-law like distribution and for the cut-off effect discussed before.

4.2.3 Slowly Driven Test

Apart from friction, additional grain addition during the avalanche process can be studied. For this, a probability h of an additional grain placement during an avalanche is introduced. The results in figure 4.12 show that the system indeed fits less into a power-law curve. The relative amount of large avalanches rises, as there is a probability of a new avalanche to happen during another one and these two would be counted as one, but a bigger one.

In *mean field theory*, one can consider this h parameter as a fine-tuning parameter, as when $h \longrightarrow 0$, the systems turns to exhibit more power-law behaviour.

From this analysis, we see that it is important that dissipation exists during the avalanche time, rather losing energy in the boundary or by energy dissipation during the propagation. (???)

During the driving time the energy of the system, defined as the sum of the values of the field of all sites, tends to oscillate around a constant value. Each additional grain is compensated with the dissipation during avalanche. For examples see figures 4.11 and 4.5.

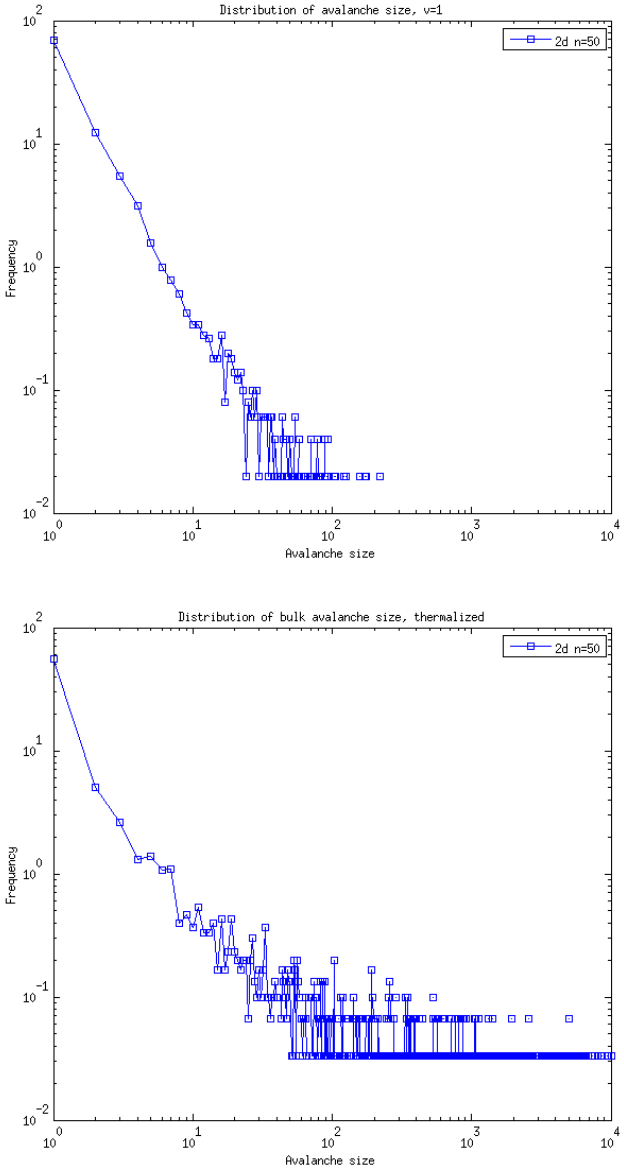

Figure 4.9: Avalanche size distribution for 50×50 lattice for different perturbation sites (first plot: site $(1, 1)$, upper left the corner; second plot: bulk site, $(25, 25)$). The driving time is $T = 5000$.

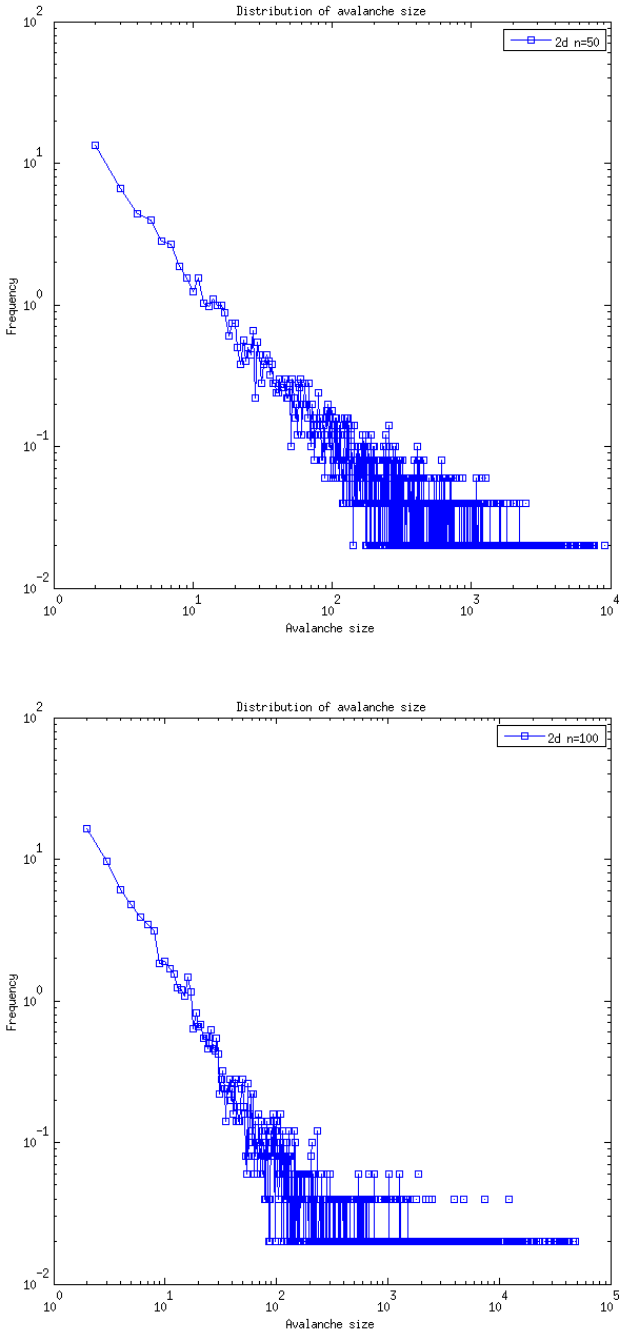

Figure 4.10: Avalanche size distribution for $2d$ lattices with different sizes and open boundary conditions.

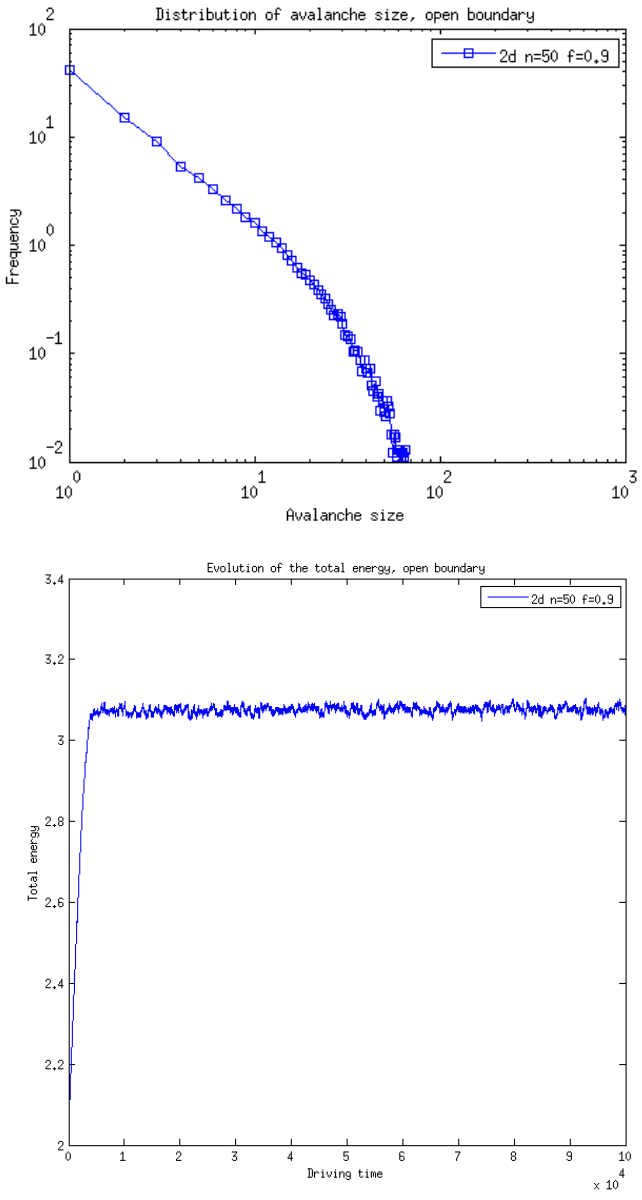

Figure 4.11: Avalanche size distribution and energy evolution for a $2d$ lattice with friction and open boundary conditions.

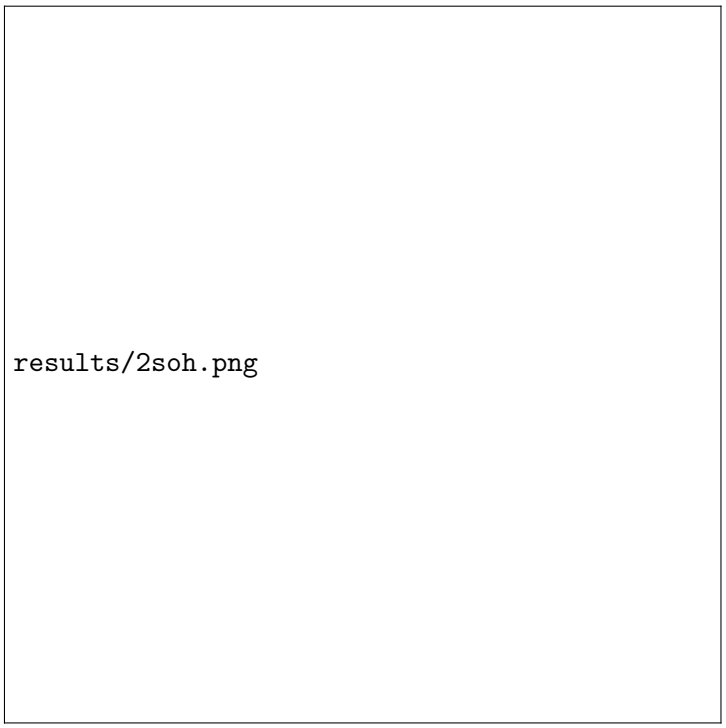

Figure 4.12: Avalanche size distribution for a $2d$ lattice with open boundary conditions and $h = 1$.

Chapter 5

Summary and Outlook

5.1 Conclusion

SOC is still a vague phenomenology, involving many attempts to use it for explaining power-like distributions in real systems using computational models.

From the studies of different cases of the abelian sandpile, it can be concluded that the continuous case, with a realistic field and friction in propagating the grains, the system bears a nice power-law like distribution for avalanche sizes, with a little worse fits for avalanche lifetimes. This case, independent of the boundary conditions, is also the more realistic one, therefore one might identify it with SOC.

In summary, **slowly driven** systems are associated with **dissipation** and a **local threshold** that leads to avalanche phenomena and SOC. The computational models provided in this paper bear these characteristics. Distinct specific laws might change only the critical exponent of the power-law distribution.

The cellullar automation models of sandpile is a convenient way of modeling real systems with the properties discussed.

A deeper understanding of SOC, if it exists, definitely needs a mathematical framework (???).

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] R. O. Dendy G. Rowlands P. Helander, S. C. Chapman and N. W. Watkins. Exactly solvable sandpile with fractal avalanching. October 1998.

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

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

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

[10] 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

```
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

```
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_topplings = future_topplings + 1;
      end
```

```
              end
            end
          end

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

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

          % communicate additional topplings to come
          if (silent==false)
            disp(['this collapse generates ' ...
              num2str(future_topplings - 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

```
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

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