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
Pegah Kassraian Fard

Zurich
April 2012

Abstract

blah blah

Acknowledgements

blah blah

Contents

Chapter 1

Introduction and Motivations

1.1 to do list

Individual contributions Introduction and Motivations Description of the Model Implementation Simulation Results and Discussion Summary and Outlook Sandpile Model

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:

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

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

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

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.

ASDF ILLUSTRATE DIFFERENT BOUNDARY CONDITIONS

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 [7].

To illustrate this practical but not necessarily obvious fact, a sample 4x4-field with one active site is considered (see figure 2.1). 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.

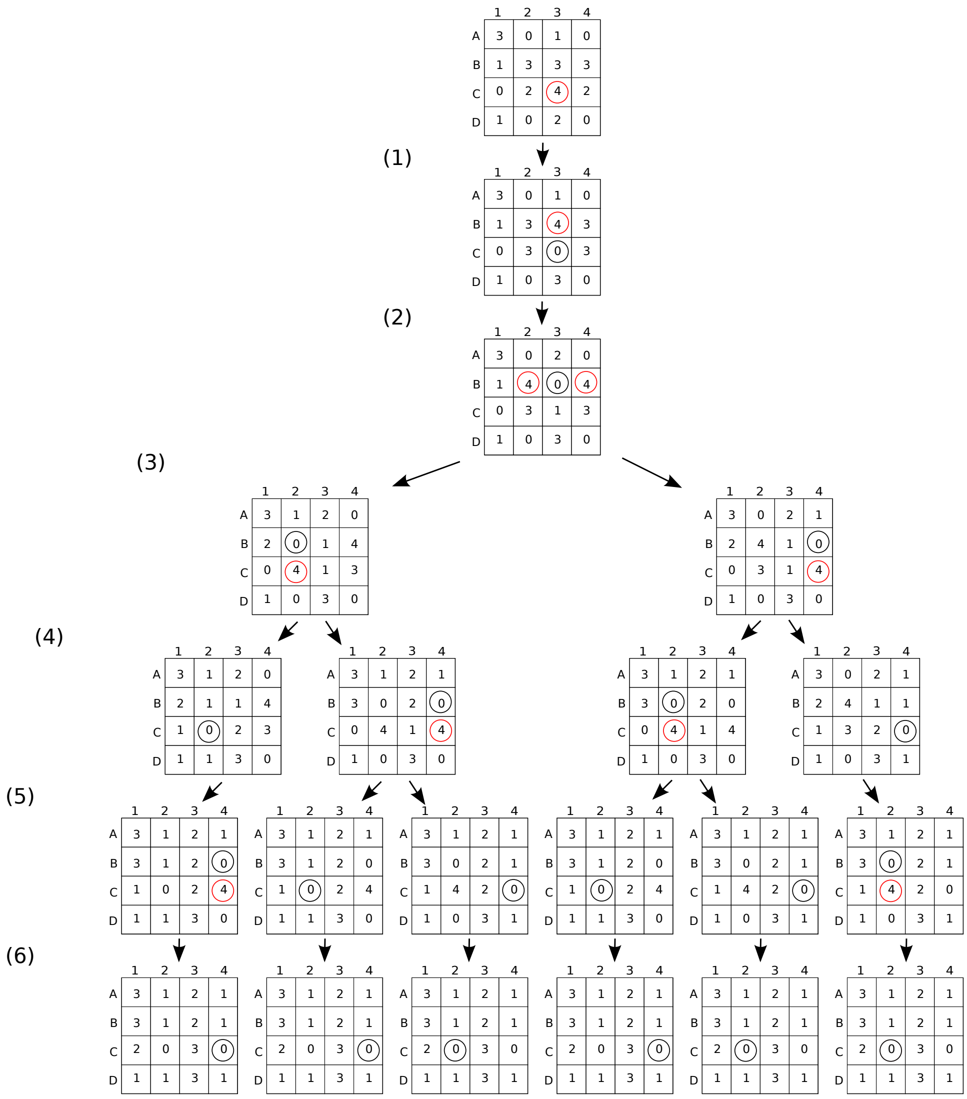

Figure 2.1: Demonstration of the abelian property: six different orders of topplings all lead to the same equilibrium state. The sites circled red indicate sites that have just become active, those circled black have just collapsed. Here, continuous boundary conditions have been used.

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.

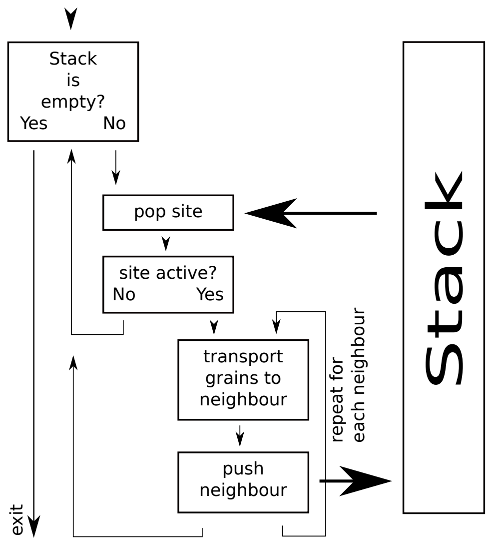

Figure 3.1: using a stack for avalanche calculation

Considering the example from figure 2.1, 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.

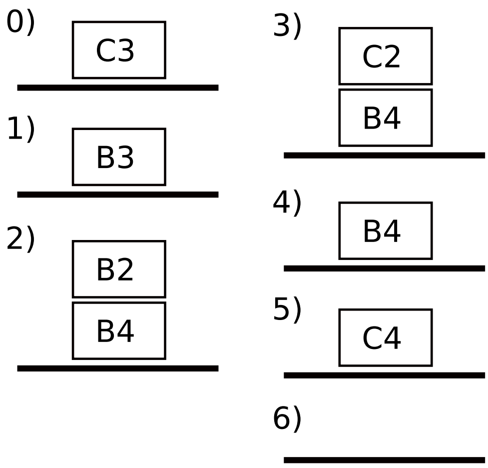

Figure 3.2: A sample sequence of stack states. The algorithm proceeds until the stack is empty.

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);
```

Chapter 4

Simulation Results

Chapter 5

Summary and Outlook

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

[8] 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)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 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] = sandpile(f, neighbour, critical_state, collapse_per_neighbour, timesteps
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 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 - ...
%   make_pictures         draw and export all frames or not
%   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 [!]

% 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

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

    % translate parameters

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

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

    % 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

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

    % 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
        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);  % un

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

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

        % save picture of field before collapsing (with active field)
        if (make_pictures)
            draw_field(f,2);
            print(['field' num2str(t) '.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_offse

                    % 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_stat
                        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) |
                        % 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_o

                        % 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_
                            future_topplings = future_topplings + 1;
                        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
                    end
                else
                    % communicate additional topplings to come
                    if (silent==false)
                        disp(['this collapse generates no additional topplings']);
                    end
                end

        end
        end

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

f = critical_field(200,200,3,true);

[s,nc,ts,f] = sandpile(f, [-1 +1 0 0; 0 0 -1 +1], 3, 1, 5000, 2, false, true, 0.2);

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

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

% to do: continuous grain placing (0...1) Ã  la grain size
%
%
```