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

to do list, beta texts

1.1 Individual contributions

1.2 Introduction and Motivations

1.3 Description of the Model

1.4 Implementation

1.5 Simulation Results and Discussion

1.6 Summary and Outlook

1.7 Sandpile model

1.7.1 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 we choose 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:

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

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.

1.7.2 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. We can pose the question, whether an equilibrium state resulting from an avalanche depends on the way we calculate the avalanche. 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 [4].

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

[5] 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)
    % define field
    f = floor(unifrnd(1,critical_state,height,width));
        % this uses uniform distribution of random numbers
end
```

A.2 sandpile

```
function [as,nc] = sandpile(f, neighbour, critical_state, collapse_per_neighbour,
                timesteps, boundary_type, make_pictures, silent)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 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
% OUTPUTS
%   as           avalanche sizes for each timestep
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % 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);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

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

        % choose random site
        y=floor(unifrnd(1,height));
        x=floor(unifrnd(1,width));  % uniform distribution rnd

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

        % 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;

                % check each 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'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);

                        % 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_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);
                            end
                        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;
end
```

A.3 avalanche distribution analysis

```
function [a,b] = avalanche_distribution_analysis(avalanche_sizes)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% analysis avalanche distribution and fits it to a power-law
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% INPUTS
%   avalanche_sizes     array of avalanche size for each timestep
% OUTPUTS
%   a,b         coefficients of power law P(s) = a*s^b
%

    % count avalanche sizes
    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

    % plot avalanche count vs size
    figure;
    subplot(1,2,1);
    plot([1:max(avalanche_sizes)],avalanche_count,'marker','s');

    % fit the curve into power law distribution (f = c1*x^c2)
    xx = [1:max(avalanche_sizes)];
    yy = avalanche_count(1:end);
    [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(1,2,2);
    loglog([1:max(avalanche_sizes)],avalanche_count,'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);
end
```

A.4 test sandpile

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% sandpile simulation
%
%

f = critical_field(200,200,3);

[s,nc] = sandpile(f, [-1 +1 0 0; 0 0 -1 +1], 3, 1, 20000, 1, false, true);

[a,b] = avalanche_distribution_analysis(s);

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