Romero, Roland Albert A.
2011-36693

Demonstrating Self-Organized Criticality in 2D Abelian Sandpiles

Introduction

The Abelian sandpile model, also known as a Bak-Tang-Wiesenfeld (BTW) sandpile model, is a complex system that can demonstrate “self-organized criticality”. It is a cellular automaton (CA) model that shows that complexity and critical states can emerge spontaneously from a set of simple rules [1].

In this project I simulated an Abelian sandpile model following the basic rules for a 2D Abelian sandpile model described in [2]:

$$z(x,y) \rightarrow z(x,y) - 4 \ ,$$

$$z(x, y \pm 1) \rightarrow z(x, y \pm 1) + 1 \ ,$$

$$z(x \pm 1, y) \rightarrow z(x \pm 1, y) + 1 \quad \text{for } z(x,y) > z_c$$

where in this project $z_c = 4$.

The key result that I wanted to replicate in this project was Figure 5 from [2] which showed the cluster size distributions for a 50x50 array after adding 100 000 grains.

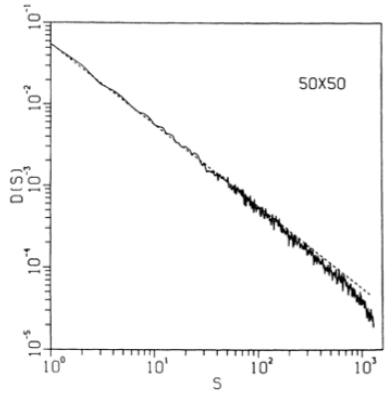

FIG. 5. Cluster size distribution for system built up from scratch according to rules (3.2) and (3.8) for a 50×50 array. The curve is indistinguishable from that in Fig. 3(a). For this system the system is in a stationary critical state and it is self-averaging. Rule (3.8) has been applied 100 000 times to the stationary critical state to obtain this curve.

Essentially, I want to replicate the results showing that the distribution of cluster sizes of avalanches for the system follows a power law i.e. in a log-log plot we should see a straight line and $D(s) \approx s^{-\tau}$.

Methodology

I wrote a module named `Sandpile` [3] containing functions to run the sandpile model.

I initialize sandpile arrays using `sandpile_init()` which creates a 2d UInt8 array that can set it up with purely 0s or random numbers from 0 to 3. The choice of having UInt8 as the array type is for performance (Abstract arrays are slower) and the fact that values in the grid will only range from 0-4 for this project.

The functions `add_grain!()`, `is_unstable()`, and `avalanche!()` are used to execute the basic rules. Grains are added individually at random locations on the input grid using `add_grain!()`. Meanwhile, `is_unstable()` is used to check if any locations on the grid have values greater than the critical value. Lastly, `avalanche!()` is used to move grains to neighbor tiles and collapse points with more grains than the threshold. I used CartesianIndex() [4] to make indexing neighbors more convenient. Finally, I made a wrapper function for the previous functions called `run_sandpile!()` to piece everything together. An example notebook running all of these can be seen here and corresponding PDF file here.

Using this module, I followed Bak and Tang's methodology [2], adding 100,000 grains to a 50 x 50 grid and getting the distribution of avalanche cluster sizes. In their paper they averaged over 200 trials and I also did comparisons with simulations running 1 and 10 trials only. I did runs starting on both empty and randomized grids. Lastly, I tried to run simulations wherein the 100,000 grains that were counted were only those that actually lead to avalanche

Performance Benchmarks

This work primarily deals with arrays for the sandpile model implementation and thus the most useful optimization done was to make sure the array types were clearly defined and not abstract. Table 1 shows the performance of running functions for arrays set as UInt8, Int64, and Abstract relative to the median benchmark times of Abstract arrays.

Table 1. Relative median benchmark time performance increases for different declared array types

	UInt8 Arrays	Int64 Arrays	Abstract
Empty Array	1.181	1.176	1.0
Random Array	1.03	1.03	1.0

From these results it is clear that setting the array types lead to considerable performance increases but the speedup of going from Int64 to UInt8 arrays is minimal.

Running times for running the models with initial conditions set to random are higher than those set to zero but this is primarily due to higher probability of having more avalanches of larger cluster sizes occur when the grid is not empty

The actual notebook for benchmarks are available here. It is also demonstrated here that all functions do not suffer from type instability.

Results and Discussion

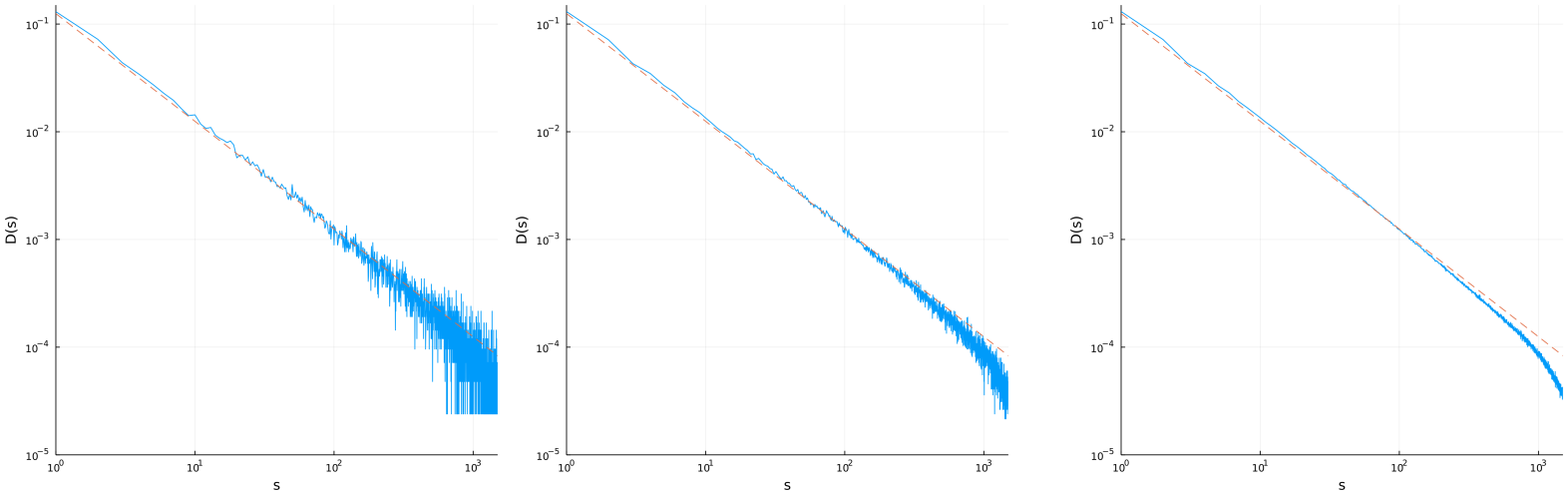

Figure 1. Cluster size distribution averages for a 50 x 50 sandpile model built from scratch after adding 100 000 grains and following BTW rules for 1 sample (left), 10 samples (middle), and 200 samples (right).

Figure 1 shows that with this module, we are able to replicate Figure 5 in [2] where they averaged 200 samples. The distributions closely follow a straight line in the log-log plots with a slope of -1 and slowly deviate at large cluster sizes. The deviation from the power law at large cluster sizes due to finite-sized effects is more apparent with lower numbers of cluster sizes.Figure 2 shows practically the same results in FIgure 1 despite initializing the sandpile model with a random distribution of grains.

Figures A1 and A2 in the appendix show similar configurations to Figures 1 and 2 respectively, but instead of just adding 100,000 grains, the model does not stop until at least 100,000 avalanches occur after adding grains. For higher number of samples (200), there is no clear difference with the results showing that with averaging, of multiple runs we still observe the same effects despite the approximately two-fold increase in the number of samples per run. The effects of ensuring 100,000 avalanches occur per run is more prominent for 1 and 10 samples; averaging over more runs is more efficient than ensuring more avalanches occur per run in terms of replicating Bak and Tang's results.

All figures can be seen here and they are generated by running generate_figures.jl.

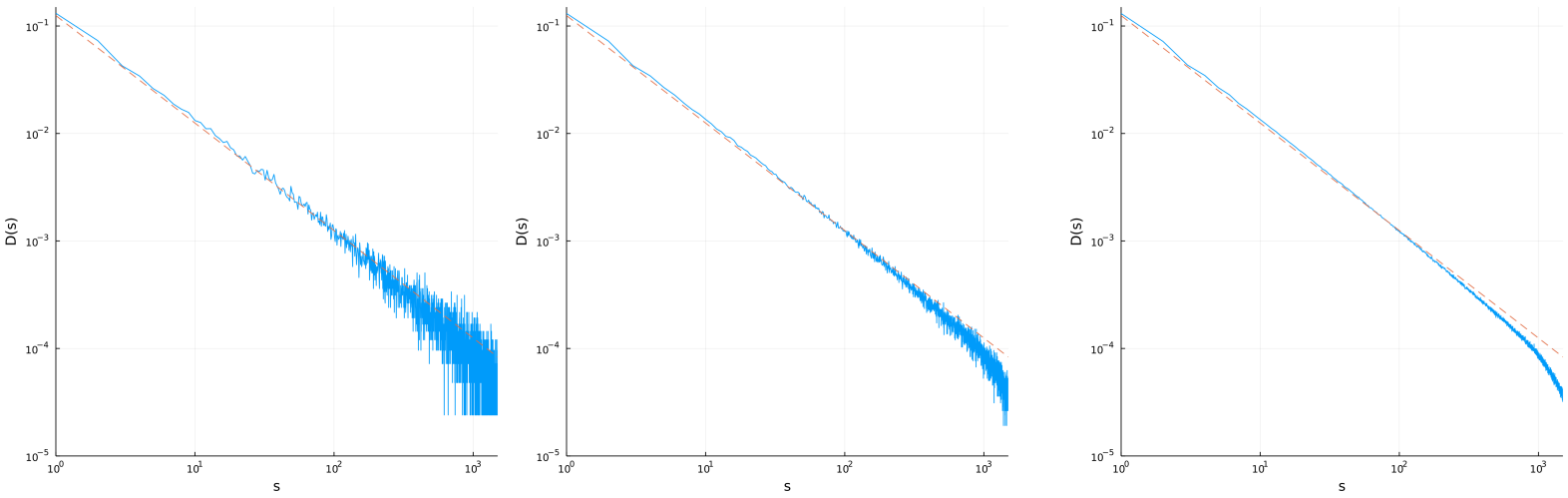

Figure 2. Cluster size distribution averages for a 50 x 50 sandpile model initialized with random conditions after adding 100 000 grains and following BTW rules for 1 sample (left), 10 samples (middle), and 200 samples (right).

Conclusions and Recommendations

The results shown in this report replicate Bak and Tang's key results using the module built in Julia. Further configurations such as changing parameters such as the array size, critical thresholds can be explored with little to no modification of the code in the module.

One limitation of the current setup is that avalanches that occur on the edges vanish since our grid is finite and it might be helpful to make the grid more adaptive but this approach might rely more on carefully-crafted structs instead of arrays.

References

[1] Weintrop, D., Tisue, S., Tinker, R., Head, B. and Wilensky, U. (2011). NetLogo Sandpile model. http://ccl.northwestern.edu/netlogo/models/Sandpile. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.
[2] Per Bak; Chao Tang; Kurt Wiesenfeld (1988). "Self-organized criticality". Physical Review A. 38 (1): 364–374.
[3] https://github.com/romeroraa/sandpile-julia/blob/main/code/sandpile.jl
[4] https://julialang.org/blog/2016/02/iteration/

Appendix

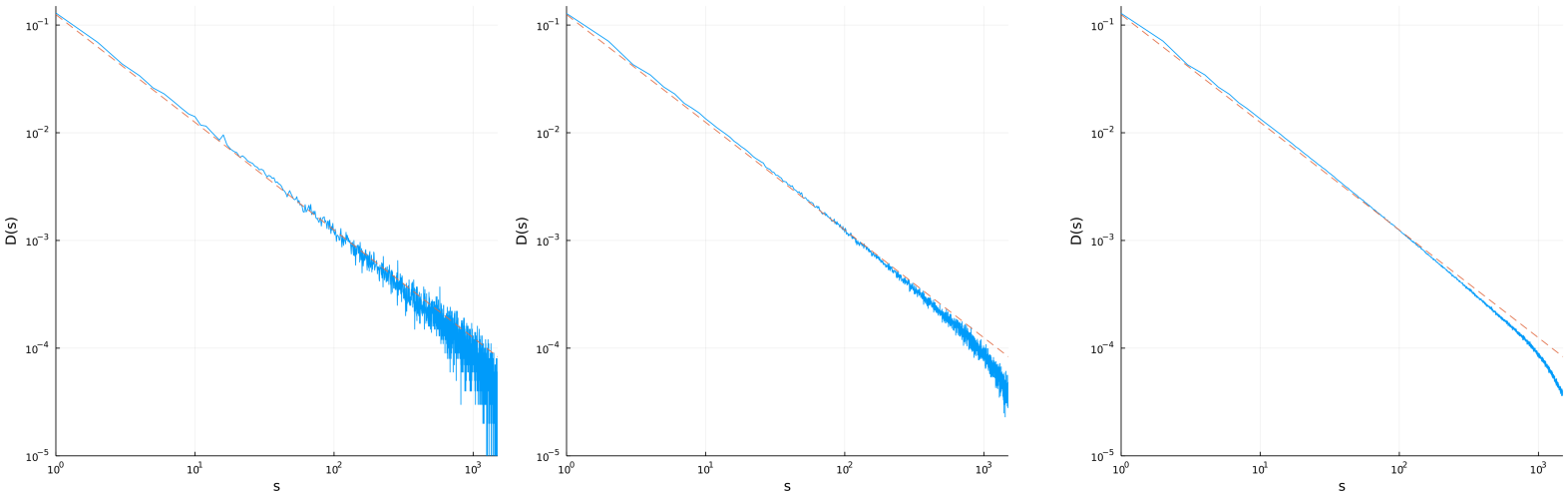

Figure A1. Cluster size distribution averages for a 50 x 50 sandpile model built from scratch after adding 100 000 grains and following BTW rules for 1 sample (left), 10 samples (middle), and 200 samples (right).

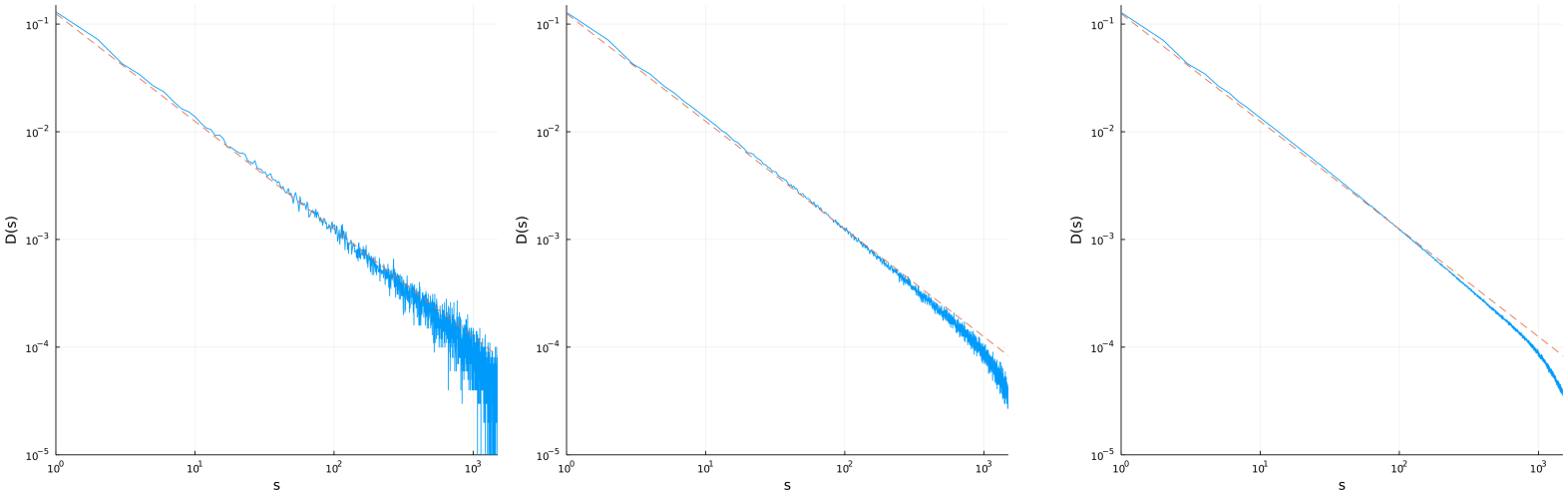

Figure A2. Cluster size distribution averages for a 50 x 50 sandpile model initialized with random conditions after adding 100 000 grains and following BTW rules for 1 sample (left), 10 samples (middle), and 200 samples (right).