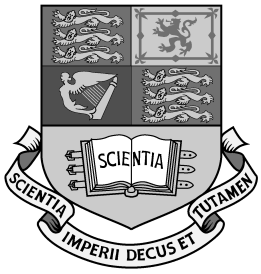
Self-Organised Criticality in Cellular Automata

MSci Project Report

Peter Dorn & David Hughes

May 2000



The Blackett Laboratory
Imperial College
London

Abstract

Self-organised critical behaviour is investigated in cellular automata. Considered are the Bak-Tang-Wiesenfeld (BTW) sandpile model and an extension to the Oslo model (Christensen *et al.*, Phys. Rev. Lett. **77**, 107 (1996)) in up to three dimensions, with numerical work focussed on the probability distribution $P(s)$ of avalanche sizes. At large system sizes, two power law regions of marginally differing scaling exponents are observable in the 2D BTW system. Possible interpretations and the consequences for the scaling behaviour of the critical exponent $\tau_{s,L}$ with L are investigated. For this model and the bulk driven 1D Oslo model with dissipative boundaries, $\tau_{s,L}$ is found to be proportional to $\log L$ and the existence of a limiting value $\tau_{s,\infty}$ for infinite system sizes cannot be confirmed numerically. Other models, including the original Oslo model, display no measurable dependence of $\tau_{s,L}$ on L. An asymptotic approach of the slope of $P(s)$ to $\tau_{s,L}$ with s is observed in all presented models. This effect is explained from the scale-dependence of the dynamics and illustrated for the 2D BTW model with the dynamically driven renormalisation group (DDRG) approach. For this, a fast approximation to the DDRG description of the system dynamics is introduced.

Contents

1 Introduction

The concept of Self-Organized Criticality (SOC) introduced in 1987 by Bak, Tang and Wiesenfeld (BTW) proposes the existence of a class of systems in which the critical state is an attractor of the dynamics [1, 2]. Such systems do not require fine tuning of external parameters to reach the state of criticality, characterised by power law distributions where the effects of small perturbations spread on all length and time scales. The example commonly used to illustrate SOC is that of a sandpile. Grains of sand dropped onto a finite size base build up a pile, which eventually reaches a critical slope. In this state, further additions will cause avalanches of all sizes, limited only by the system size.

Many systems which occur in nature are considered to be examples of SOC [3, 4]. Earthquakes [5], forest fires [6, 7], vortices in superconductors [8], evolution [9] and piles of granular material [10] have been modelled in an attempt to understand the mechanisms leading to the observed scale invariance. Physical experiments have shown power law behaviour in some systems, however these observations are not as insensitive to system parameters as was hoped [11–14].

In order to help understand the basic mechanism of SOC, attempts have been made to assign models to universality classes, where models in a class are described by the same set of exponents of the observed power law distributions. Previous studies have concentrated on the BTW sandpile model [1, 2], the Zhang model [15] and the Manna model [16]. Theoretical analyses suggest that these models should belong to the same universality class [17–21]. This corresponds with intuitive expectations that small changes in the models should not dramatically change their behaviour for sufficiently large systems. However, numerical simulations have so far failed to produce consistent results [22–25]. In this context we investigate the scaling behaviour of the BTW model and a stochastic variant based on the Oslo model [26]. These models are studied numerically in up to three dimensions and links between these results and theoretical work are identified.

2 Models

The BTW model has been examined extensively both numerically and analytically since its introduction in 1987. However, the various analysis techniques have yielded significantly different results. The Oslo model has received much less attention and results have only been published for the one dimensional case.

2.1 BTW model

At each point i on a d-dimensional hypercubic lattice an integer variable z is defined which represents the height of the sand column at that site. Starting from arbitrary initial conditions grains are added one at a time to a randomly chosen site

$$z_i \rightarrow z_i + 1$$

If this new value reaches or exceeds the height threshold parameter $z_c = 2d$ the site is unstable and topples. During a toppling the height of the sand column at the site is reduced by

$$z_i \rightarrow z_i - 2d$$

This sand is redistributed equally to the $2d$ nearest neighbours, z_{nn}, so that

$$z_{nn} \rightarrow z_{nn} + 1$$

If the height of the nearest neighbours now exceeds z_c they are toppled in turn and the avalanche continues until all sites are stable. Figure 2.1 shows the progression of a small avalanche on a 3×3 lattice. The model is slowly driven, so that all avalanches are allowed to finish before the next addition of sand. Grains crossing a boundary are removed from the system. The BTW model displays SOC only in two or more dimensions.

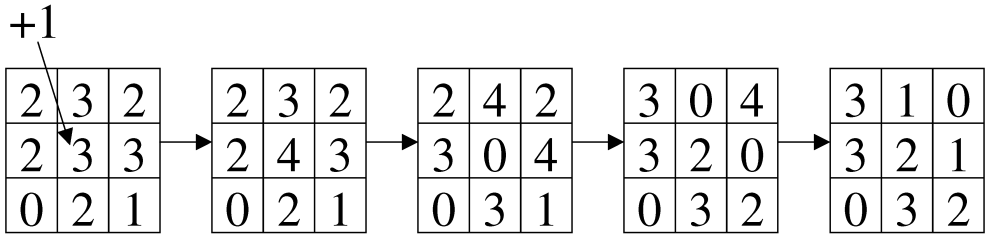


Figure 2.1: Progression of an avalanche of the BTW model. A sand grain is dropped on a stable configuration at the central site (Adapted from [27]).

2.2 Oslo model

The Oslo model is a variation on the BTW model with a dynamically varying critical height [26]. Each lattice point is now assigned its own critical height parameter $z_{c,i}$ which is determined stochastically both initially and each time that lattice site is relaxed. The critical height at each lattice point $z_{c,i}$ takes one of two values, $2d$ or $2d+1$, with equal probability. All other rules are the same as the BTW model. In addition to dissipating boundaries and bulk driving, the Oslo model in one dimension is also studied with one conserving boundary and edge driving at this boundary.

2.3 Measures

2.3.1 Avalanche size

The avalanche size s is defined as the total number of topplings during an avalanche. The corresponding probability distribution for a given system size L is given by $P(s, L)$, which is thought to follow a power law with exponent τ_s.

2.3.2 Avalanche duration

The duration of an avalanche is the total number of time steps taken from the start to the end of its propagation. The initial toppling is assigned time 0 and any of its nearest neighbours which are caused to become unstable are assigned time $t = 1$. Sites caused to become unstable by topplings from $t = 1$ sites are assigned $t = 2$ and so on. The value of t of the last site toppled is taken to be the avalanche duration.

2.3.3 Radius of gyration

The radius of gyration R_s is the mean squared deviation from the average position. If $\mathbf{r}_i$ is the position of the ith toppling, and $\mathbf{r}_0$ is the average toppling position

$$\mathbf{r}_0 = \sum_{i=1}^{s} \frac{\mathbf{r}_i}{s},$$

then the radius of gyration is calculated as

$$R_s^2 = \sum_{i=1}^{s} \frac{|\mathbf{r}_i - \mathbf{r}_0|^2}{s} \quad .$$

3 Code and Method of Analysis

3.1 Simulation code

The code is written in C++ and is designed for speed and versatility. The program is capable of simulating systems with any linear extent in any dimension, subject to physical memory constraints. Its structure is object oriented, ensuring that the program can be easily extended to incorporate new models and measures. Some platform dependent functions are used, with support for DEC Unix and Win32 environments. The most important aspects of the program are explained below. More information about the program as well as a diagram of the function dependency can be found in Appendix A. A full program listing is provided in Appendix B.

3.1.1 Determining criticality

The value of the average height $<z>$ fluctuates around a fixed value when the system is critical and heads towards this value from any arbitrary starting configuration. Once the sytem is in this critical state, the program starts recording results. Two moving averages are taken, with the faster one giving greater weighting to the last value of $<z>$. When the fast moving average drops below the value of the slow one due to fluctuations of $<z>$, the system is taken to be critical (see Figure 3.1).

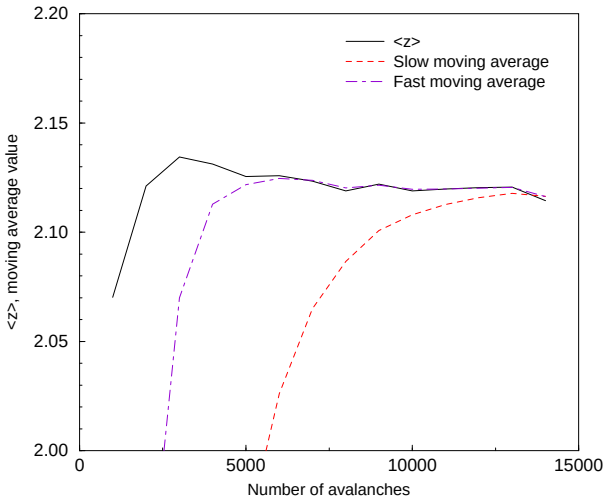


Figure 3.1: Fluctuations of $<z>$ and its moving averages. Values are calculated every 1000 avalanches and results are recorded after the two moving averages cross for the first time.

3.1.2 Queue

The queue is an array that stores the positions of unstable locations, which are stepped through and relaxed in turn. If during a relaxation process a site becomes unstable, it is added to the end of the queue, hence sites are relaxed in the order in which they become critical. It is not yet known whether the Oslo model is Abelian, so the order of relaxation may make a difference to the measured values.

The queue has a certain maximum size, determined by memory constraints. If the total size of an avalanche exceeds the queue size, the information is wrapped to the start of the queue. So for a queue of limit n, $q[n+1] = q[0], q[n+2] = q[1]...$ This method ensures that very large avalanches can be calculated while controlling memory requirements. However, as not all data is available, some measures cannot be calculated for these large avalanches.

3.1.3 Random number generation

The random number generator employed is taken from [28]. This generator is based on the simple multiplicative congruential algorithm

$$I_{j+1} = aI_j \pmod{m} \tag{3.1}$$

where a and m are carefully chosen constants. The generator uses two such sequences with different constants and uses one sequence to shuffle the order of the other. The purpose of this is to break up serial correlations that exist in equation 3.1. The generator produces uncorrelated random numbers, has a very long period ($> 2 \times 10^{18}$) and is extremely fast. These properties are necessary for simulations of the Oslo model, which requires a random number to be generated for each relaxation. Each presented result for these systems required up to 10^{10} random numbers. The generator is seeded with the system clock to prevent correlations between runs.

3.1.4 Time and memory constraints

To reduce memory requirements, the array that stores the size of avalanches is fixed to a certain maximum value. The size of any avalanche above this value is stored in a file on the hard disk. Although this is much slower than memory storage, power law behaviour ensures that only a small number of avalanches have to be treated in this manner. When the desired number of avalanches has been calculated, the file is sorted (using [29]), and appended to the results output.

Unfortunately this virtual memory method cannot be applied to the array that stores the sandpile, as all the members of the array are required frequently. This would result in continual disk access and vastly increased calculation times. The maximum system size that can be calculated is therefore limited by system memory.

3.2 Method of analysis

3.2.1 Logarithmic binning

The raw program output suffers from high statistical noise, particularly at large avalanche values. Due to power law behaviour this problem remains even for data sets with very large numbers of avalanches. To reduce this noise the data is binned, with the size of the bins increasing exponentially. The start and end of the position of the k^{th} bin are calculated by

$$x_k = (1+a)b^k$$
$$x_{k+1} = b\,x_k$$

where a is the offset and b is the base. The k^{th} bin value $B_{\text{val},k}$ is given by the average data value between x_k and x_{k+1}. The position of the bin $B_{\text{pos},k}$ is then calculated from the geometric mean of the start and end of the bin. The geometric mean provides a better approximation to power law behaviour than the arithmetic mean. Figure 3.2 shows the raw data output from the program and the data after binning.

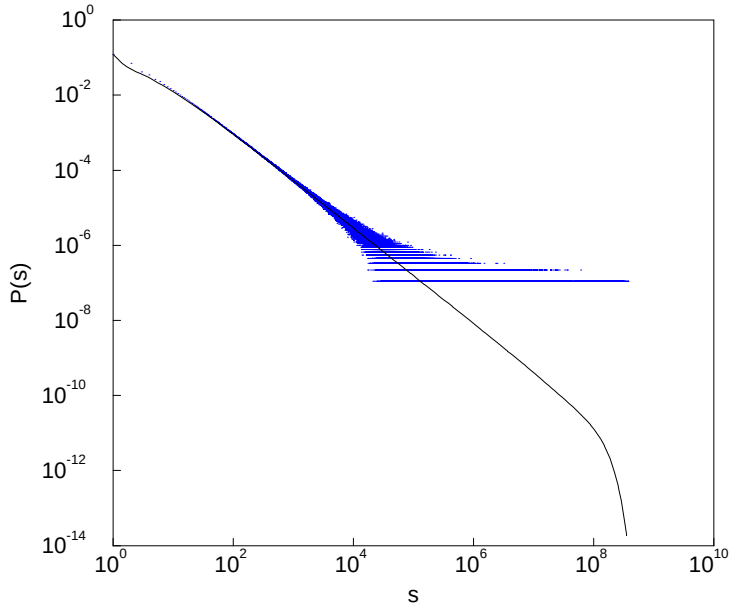


Figure 3.2: Raw and binned avalanche size distribution $P(s)$ for 2D Oslo model with $L = 2048$.

Offsets and bases of different sizes are used to prevent emphasizing spurious effects due to the binning process. Bases used in the calculations ranged from 1.1 to 1.9 in steps of 0.1, and offsets ranged from 0 to 0.8 in steps of 0.2. As bins calculated in this way are smaller than unity for small s, the end of a bin x_{k+1} is incremented until the bin contains at least one value or until the end of the input file is reached. The drawback to binning in this way is that small inaccuracies are introduced at small values of x. This is acceptable here for $P(s)$ as the small s values are outside the scaling region.

3.2.2 Determination of τ_s

The value of τ_s can be found from the slope of the binned data by

$$\frac{d \log P(s)}{d \log s} \approx \frac{\Delta \log B_{\mathrm{val}}}{\Delta \log B_{\mathrm{pos}}} \ . \tag{3.2}$$

The values of $d(\log\ P(s))/d(\log\ s)$ calculated from the different logarithmic bases are then binned to produce a histogram. This indicates the frequency of occurence of a particular local slope. Power law behaviour would result in a narrow spike centered on the value of τ_s, with the width of the peak indicating the level of statistical noise. The histogram uses the data points calculated from all the bases and offsets, with the data points weighted such that each base size makes the same overall contribution. This weighting prevents the smaller bases with relatively high statistical noise dominating the histogram.

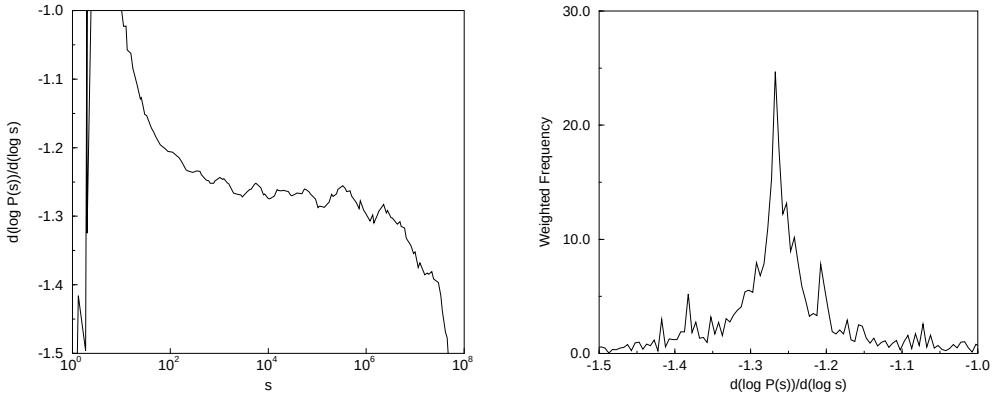


Figure 3.3: Variation in slope of $P(s)$ with s for 2D Oslo model with L=2048 (left) and histogram showing the frequency of occurance of these slope values(right)

4 Theoretical Results

A complete mathematical theory of SOC does not yet exist. However, several approaches for the calculation of one or more properties of the sandpile systems have been proposed. The 2D BTW model has been at the centre of these efforts and the discussion in this section will focus on this system to offer explanations for some of the unexpected features observed in the numerical treatment.

Dhar has shown the BTW model to be Abelian [30] and this property has been used to determine several of its characteristics [31–33]. The picture of waves of single topplings as the constituent elements of an avalanche has also been investigated [34–37]. Waves are now well understood for the 2D BTW model [38], but multiple topplings prevent a direct explanation of the scaling behaviour of avalanches in terms these results.

Other theoretical work includes a mean-field approach [39] and a real space renormalisation scheme. This renormalisation scheme is discussed in detail below as it provides useful insight to the scale-dependence of the dynamics and may be transferable to similar Abelian systems.

4.1 Dynamically driven renormalisation group

4.1.1 The DDRG formalism

The dynamically driven renormalisation group (DDRG) has been suggested by Pietronero et al. [17, 18] as a theoretical approach to determining the properties of the 2D BTW model in the limit of infinite system size. The method starts from describing the dynamics in finite size cells of the sandpile models, each containing four subcells. The dynamics of the compound cell are described in terms of the dynamics of its subcells. The method is iterated by then taking the resulting dynamics of the compound cell as those of each subcell at the next larger scale and reapplying the equations. Thus each iteration of the equations corresponds to doubling of the scale under consideration, so that the number of elementary cells along each edge of the compound cell L at scale k is given by $L = 2^k$. After several iterations of this map, the dynamics of the cell are found to be equivalent to those of its subcells, i.e. the system has reached a fixed point with scale-invariant behaviour.

From the properties of the fixed point, the exponent for the probability distribution of topplings τ_s can be determined. The driving and dissipation in the system must be taken into account in this iteration, so that energy, or grain number, is conserved in each cell.

Hence

$$\rho^{(k+1)} = \frac{1}{\sum_n n\, p_n^{(k+1)}} \tag{4.1}$$

must be observed, where $\rho^{(k+1)}$ is the density of critical sites at scale $k+1$ and p_n with $n = 1, 2, 3, 4$ are the elements of $\mathbf{p}$ describing the probabilities of a toppling cell to throw out grains at n sides. This nonlinear driving condition reproduces the feedback between ρ and $\mathbf{p}$ that determines the dynamics of the model.

Following [17, 18], the value of τ_s can be determined from this by considering the probability K of an avalanche stopping at scale k. The argument results in

$$\tau_s = 1 - \frac{1}{2}\,\frac{\log(1-K)}{\log 2}\ , \tag{4.2}$$

where K is also given by

$$K = \sum_n p_n^*(1-\rho^*)^n \tag{4.3}$$

and can therefore be determined from the fixed point properties ρ^* and $\mathbf{p}^*$.

4.1.2 Approximations in the DDRG approach

Equation 4.2 assumes that avalanche size s and radius r are related by $s \propto r^2$. Even though avalanches are found to be compact in 2D [40], multiple topplings within the avalanche region increase the dependence of s on r (see Figure 4.1). Multiple topplings are neglected in both the calculation of τ_s by equation 4.2 and the description of the dynamics by the $p_n^{(k+1)}$ terms. The DDRG method can thus be seen to implicitly assume that the corresponding fractal dimension approaches 2 as $L \to \infty$. This is equivalent to the assumption of one dominating wave of avalanches and an explanation of the validity of the DDRG results from the wave picture is offered by Ktitarev *et al.* [37]. A more complete description of the dynamics taking account of multiple topplings would consequently require the fractal dimension of the avalanche in r to retrieve τ_s. However, as a consequence of avalanche 'aging' [36, 41], no unique fractal dimension for the dependence of s on r would be expected and multiple topplings cannot easily form an extension of the DDRG formalism.

A further approximation is included in the classification of process series solely by the number of sides at which grains leave, rather than the number of grains leaving at each side. The $p_n^{(k+1)}$ terms entering the driving condition describe the propagation of 'excitations' rather than grains. A compound cell is assumed to receive no more than one excitation, which at the subcell level is regarded as the addition of one grain to one subcell. In contrast, the rules of the model allow for two grains to enter from a neighbouring cell and excite both subcells at the corresponding side. The stationarity criterion in equation 4.1 therefore results in the conservation of excitations rather than grains.

Additional limitations also exist, such as the necessary limitation of the range of cells considered when constructing the process series.

4.1.3 Modifications to the original DDRG

Modifications to the original DDRG scheme have been suggested, varying in the set of processes considered and in the geometry of the compound cell. Ivashkevich *et al.*

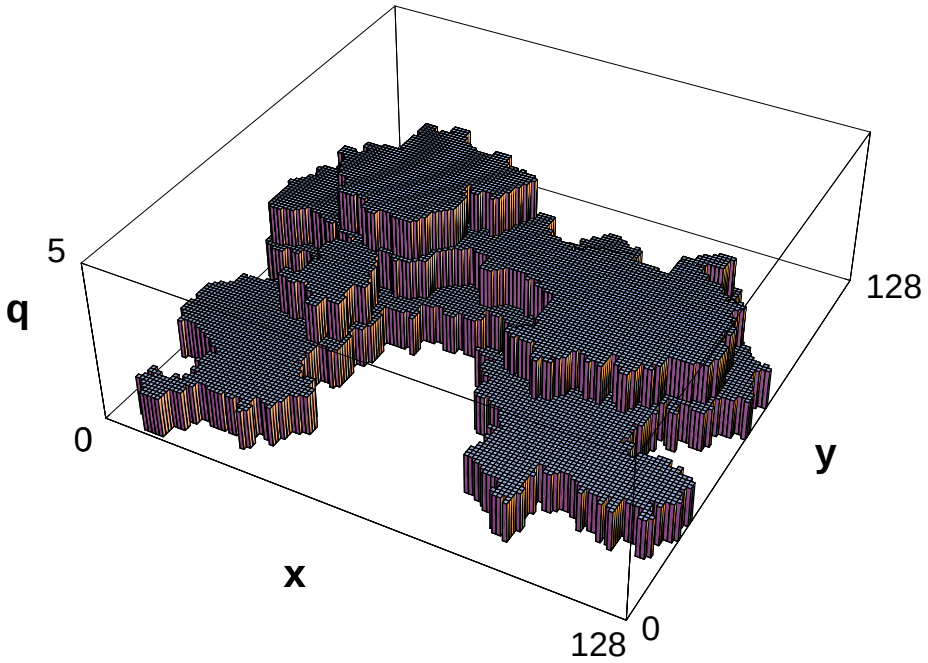


Figure 4.1: Plotting the number of topplings q for each site in the 2D BTW model illustrates that the dependence of s on r is stronger than $s \propto r^2$. The data is taken from a dissipative avalanche of size $s \approx 12000$. The plot also shows the fractality of the avalanche boundary described in [39, 42, 43]. The temporal progression of this avalanche can be seen in Figure 5.12.

extended the definition of $\mathbf{p}$ to include purely internal spanning processes, denoted by p_0 [44]. Moreno *et al.* presented an alternative geometry for the compound cell in the form of a Greek cross containing five subcells [45]. The values of τ_s resulting from these modified schemes and the original work by Pietronero *et al.* are

$\tau_s \approx 1.253$	Pietronero et al. [17, 18]	Square cell, $\mathbf{p} = \{p_1, p_2, p_3, p_4\}$
$\tau_s \approx 1.262$	Ivashkevich et al. [44]	Square cell, $\mathbf{p} = \{p_0, p_1, p_2, p_3, p_4\}$
$\tau_s \approx 1.236$	Moreno et al. [45]	Greek cross cell, $\mathbf{p} = \{p_1, p_2, p_3, p_4\}$

Table 4.1: Published values of τ_s from DDRG calculations.

A similar approach taking account of height probabilities and thereby describing the system dynamics more fully, results in $\tau_s = 1.248$ [44]. The variations of the values from these different schemes illustrate the approximative nature of the DDRG approach.

4.1.4 Fast approximation to the $p_n^{(k+1)}$ terms in the DDRG

Knowledge of the processes $p_n^{(k+1)}$ at scale $k+1$ in terms of those at the subcell scale k readily allows iteration of this map with the restriction of the driving condition. However, the lengthy construction of these $p_n^{(k+1)}$ terms makes it impractical to test the effect of small variations in the description of the dynamics on the result.

This modified approach to generating the terms involves the use of a mapping method to assign each process series at scale k to the corresponding large scale processes $p_n^{(k+1)}$. The method results in a considerable increase in ease of calculation and versatility at the expense of introducing a further approximation. The motivation for this is to obtain a tool for discussions on the dynamics of the system, for which a small deviation from previously published results is acceptable.

The properties of each process in a process series can be fully characterised by separately considering its contribution towards the propagation of the avalanche in the cell and the number of grains leaving the cell. Four ways of propagation inside the cell can be identified. In each process, a grain can be thrown to either one of its two distinct nearest neighbours, to both of them or to neither. Grouping the processes $p_n^{(k)}$ according to their probability of propagating in these ways gives

$$\begin{aligned} n_1 &= \frac{1}{4}p_1^{(k)} + \frac{1}{3}p_2^{(k)} + \frac{1}{4}p_3^{(k)} \\ n_2 &= \frac{1}{4}p_1^{(k)} + \frac{1}{3}p_2^{(k)} + \frac{1}{4}p_3^{(k)} \\ b &= \frac{1}{6}p_2^{(k)} + \frac{1}{2}p_3^{(k)} + p_4^{(k)} \\ x &= p_0^{(k)} + \frac{1}{2}p_1^{(k)} + \frac{1}{6}p_2^{(k)} \end{aligned} \tag{4.4}$$

where n_1 and n_2 denote the toppling to only one nearest neighbour, b stands for both and x for none of them. The number of grains leaving the cell can be determined by subtracting 1 from n for the processes in n_1 and n_2, 2 for those in b and 0 for x. As a result of this, the same $p_n^{(k)}$ in different groups are now distinguishable and are denoted by $p_{n,c}^{(k)}$ for a $p_n^{(k)}$ process throwing c grains outside the cell.

Each possible process series can now be written in terms of the these process groups and the density ρ of critical sites. The process series are grouped by the number m of toppling sites and these groups are denoted by t_m.

$$\begin{aligned} t_0 &= (1-\rho) \\ t_1 &= \rho[x + (n_1+n_2)(1-\rho) + b((1-\rho)^2] \\ t_2 &= \rho[\{(n_1+n_2)\rho + 2b\rho(1-\rho)\} \\ &\quad \cdot\{n_1 + n_2(1-\rho) + b(1-\rho) + x\}] \\ t_3 &= \rho[\{(n_1+n_2)\rho \\ &\quad \cdot\{(n_2+b)\rho\} \\ &\quad \cdot\{n_1 + n_2(1-\rho) + b(1-\rho) + x\} \\ &\quad +\{2b\rho(1-\rho)\} \\ &\quad \cdot\{(n_2+b)\rho\} \\ &\quad \cdot\{n_1 + n_2 + b + x\} \\ &\quad +\{b\rho^2\} \\ &\quad \cdot\{(n_1+x)^2 + (n_1+x)(n_2+b)(1-\rho)\}] \\ t_4 &= \rho[\{(n_1+n_2)\rho\} \\ &\quad \cdot\{(n_2+b)\rho\} \\ &\quad \cdot\{(n_2+b)\rho\} \\ &\quad \cdot\{n_1 + n_2 + b + x\} \\ &\quad +\{b\rho^2\} \\ &\quad \cdot\{(n_1+x)(n_2+b)\rho\} \\ &\quad \cdot\{n_1 + n_2 + b + x\}] \end{aligned} \tag{4.5}$$

Spanning processes are therefore given by the sum of t_2, t_3 and t_4. After substituting for the process groups from equation 4.5 and expanding, the sum of t_2, t_3 and t_4 includes all possible process series in terms of elementary processes $p_{n,c}^{(k)}$. Each of the resulting terms contains information on the number of topplings as well as the number of grains leaving at each toppling. This information can then be used to assign weightings for the inclusion into the larger scale processes $p_n^{(k+1)}$ to each process series. A process series consisting of two $p_{n,1}^{(k)}$ processes can be seen to have probabilities of $\frac{1}{4}$ to throw out grains to one side and $\frac{3}{4}$ to topple across its cell boundaries on two sides. The corresponding vector of weightings for $p_0^{(k+1)}$ to $p_4^{(k+1)}$ is thereby $\mathbf{m}_{1,1} = \{0, \frac{1}{4}, \frac{3}{4}, 0, 0\}$. A complete list of all 31 vectors is included in Appendix E.

The approximation contained in this approach leads to a deviation for three or four topplings. When calculating the mapping vectors $\mathbf{m}$, the processes are permutated through the available sites. All permutations enter with equal weight, disregarding the actual probability of their occurence. The term $p_{x_1,0}^{(k)} p_{x_2,1}^{(k)} p_{x_3,2}^{(k)}$, for example, corresponds to each of the three configurations shown below. These only differ in the positions of the sites relaxing with these different processes.

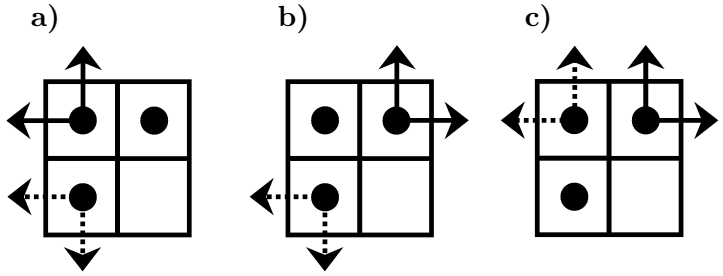


Figure 4.2: Equal weighting of all possible configurations constitutes the approximation in the mapping method. While **a)** and **c)** both have equal probabilities to topple to 2 or 3 sides, configuration **b)** always propagates to 3 sides. Hence probabilities of $\frac{1}{3}$ for 2 sides and $\frac{2}{3}$ for 3 sides are assumed for the ensemble, although the configurations would not generally have equal odds of occuring.

The initial excitation can occur at the centre subcell of the series, which has two toppling neighbours, or at either of these neighbours. For each process, the minimum value of the total number x_i of grains leaving the toppling subcell is the sum of the number of grains leaving the compound cell and the number of grains required for propagation towards the final state. Hence only configuration b) can occur for an excitation at any toppling subcell if $x_i \leq 2 \, \forall \, i$. Configuration c) can occur with this restriction if the initial excitation is not at the centre subcell, while configuration a) requires at least one $p_{4,2}^{(k)}$ process if the centre subcell is excited and at least one $p_{3,2}^{(k)}$ process otherwise. The correct weighting could thus only be obtained by having mapping vector elements dependent on the values of the relevant $p_{n,c}^{(k)}$ processes or explicitly assigning each possible process series to the correct process groups, by which equivalence to the original method should be recovered.

The value of τ_s resulting from this approximative approach is

$$\tau_s \approx 1.265 \ .$$

In the presented form, the method represents an approximation to the work of Ivashkevich *et al.* and the above result is very close to their value of $\tau_s \approx 1.262$. The presented method is therefore assumed to be a useful approximation.

Modifications to the assumptions made about the dynamics in the square cell can be made more easily with this approach and allow for some of the assumptions used in the description of the dynamics to be tested for their validity. Extra terms can be added to allow for those sites toppling only after receiving *two* grains from its neighbours to give modified versions of t_3 and t_4 in equation 4.6 (see Appendix D). This toppling of initially subcritical sites can occur within the rules of the model and corresponds to defining a density ρ' of sites becoming critical after the addition of one grain. A full treatment would need to take account of this in the driving condition, but an estimate can be obtained by assuming that $\rho' = \frac{(1-\rho)}{2}$, giving $\tau_s \approx 1.266$. Hence it can be confirmed that the effect of this approximation is small, but that nevertheless the values in [17, 18] and in the first method presented in [44] would be expected to increase slightly with the inclusion of this effect. The improved scheme in [44] does not contain this approximation.

An extension of this scheme to higher dimensions would require further approximations. In 2D, the approximation lies only in the weighting of configurations on a unique spatial arrangement of toppling cells. For three dimensions, also the arrangement of toppling cells is not unique for 4, 5, or 6 topplings, leading to further deviations.

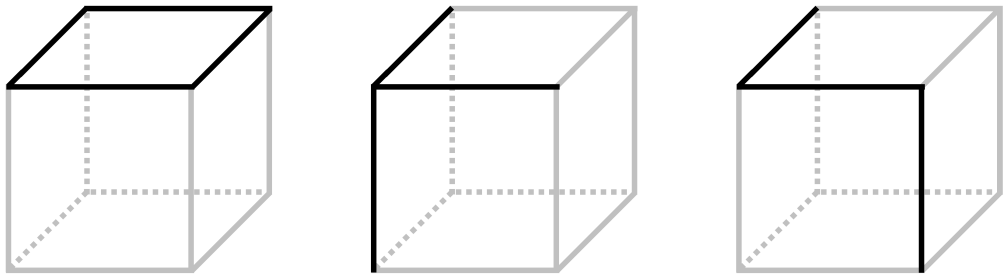


Figure 4.3: Possible arrangements of four topplings on a three dimensional elementary cell. Heavy lines mark connections between toppling sites.

The basic approach of the method presented here, to consider propagative and dissipative topplings of a process separately, can also be used for an easier computational implementation of the generation of the process series. This would allow solutions in higher dimensions, but after first trials the required calculational cost appears to be prohibitive. In addition, the step of retrieving τ_s could not be adapted for dimensions above the critical dimension, as avalanches are no longer compact [46].

4.2 Asymptotic scaling

Rather than being an artefact of the method, the trajectory of $\mathbf{p}$ towards the fixed point can be explained as a true representation of scale-dependence in the dynamics of the model. At the smallest scale, the elementary cell has fixed toppling behaviour described by $\mathbf{p} = \{0, 0, 0, 0, 1\}$. With grains toppling to all four sides a second toppling is likely and hence the slope of $P(s)$ is expected to be low. With increasing scale, the cell dynamics approach the fixed point rules $\mathbf{p}^*$, so the average number of sides involved in a toppling decreases. A propagation of the avalanche thus becomes less likely and the slope of $P(s)$ is expected to gradually decrease towards the slope τ_s corresponding to the fixed point dynamics.

This transition can be demonstrated on the minimum number of sides that are involved in a toppling process at each of the first four scales $k = 0, 1, 2, 3$.

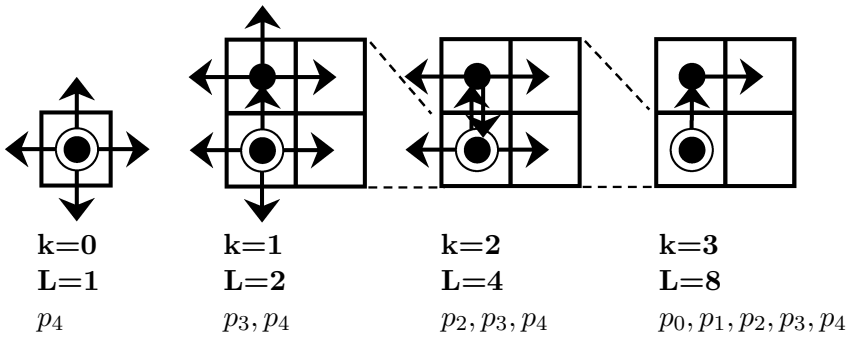


Figure 4.4: Scale-dependent dynamics at smallest scales. Shown are the minimally dissipative process series at small k. The range of non-zero elements of $\mathbf{p}$ is listed for each scale. Table 4.2 shows the corresponding numerical values of $\mathbf{p}$.

In the full DDRG calculation, the first iterations, corresponding to the first four scales shown above, correctly reproduce this behaviour. The probability mass shifts to lower p_n and a non-zero value of p_0 is first observed at $k = 3$.

k	L	p_0	p_1	p_2	p_3	p_4
0	1	0	0	0	0	1
1	2	0	0	0	0.6429	0.3571
2	4	0	0.0176	0.2136	0.5556	0.2133
3	8	0.0014	0.0602	0.3157	0.4701	0.1527

Table 4.2: Iteration of $p_n^{(k+1)}$ map from fast approximation method (rounded values).

Following this iteration to the fixed point shows that the deviation from the scale-invariant dynamics, which would be trivially expected for small s, does not fully decay until $k = 66$ within the precision of the computational iteration (see Appendix F). Such an asymptotic approach is observed in the numerical results of all presented models. In addition, the 1D Oslo and 2D BTW models display a preference for particular avalanche sizes, which also asymptotically decays with s (see Figure 4.6).

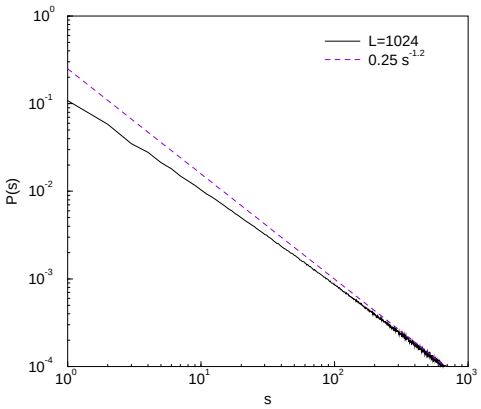


Figure 4.5: Asymptotic scaling in the 2D BTW model. Shown are unbinned data for L=1024 and a straight line for comparison. Figure 4.6 shows additional asymptotically decaying effects at very small s.

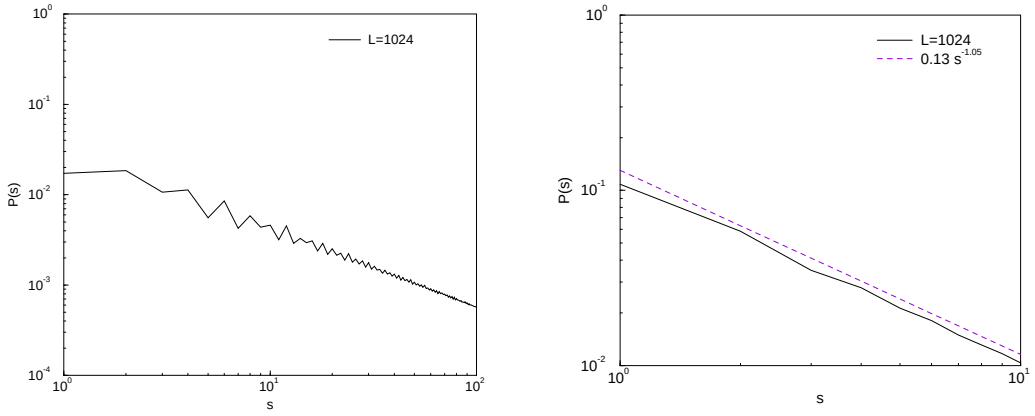


Figure 4.6: Asymptotic preference patterns for certain avalanche sizes in the 1D Oslo (left) and 2D BTW models for a total of 10^7 avalanches. Shown are unbinned data for L=1024 and a straight line for comparison for the 2D BTW model results. This effect in the 2D BTW model is the result of a weak preference for even s in non-dissipative avalanches and a stronger, more irregular pattern in dissipative avalanches. Due to the small fraction of dissipative avalanches at small s, the pattern of non-dissipative avalanches clearly dominates. The larger effect in the 1D Oslo model can also be observed when finding the slope of $P(s)$ (see Figure 5.2).

The origin of this effect is unclear and could form part of a further investigation into the system behaviour at small s. Both numerical results and DDRG calculations therefore suggest that the small scale cut-off is a significant element of the scaling behaviour for all computable system sizes. Rather than a perturbation restricted to very small s, the observations are consistent with this small scale cut-off being an asymptotic approach of the dynamics to scale-invariance.

4.3 Dissipative and non-dissipative avalanches

A distinction in the behaviour of avalanches that dissipate energy at the boundaries during their propagation and those that do not has been suggested by DeMenech *et al.* for the 2D BTW model [47] and has been investigated in more detail by Drossel [48]. Both attribute numerically observed breakdown of finite size scaling (FSS) to this, but its exact role in the scaling behaviour of the BTW model is still unclear. The observed lower slope of dissipating avalanches can be understood from a simple random walk argument describing the probability of an avalanche of size s to reach the boundary. A recent numerical study points out a deviation from FSS in in the distribution of non-dissipative avalanches [38], supporting the view that any difference in the behaviour of dissipative and non-dissipative avalanches does not represent the fundamental cause for the breakdown of FSS.

4.4 Implications of theoretical results

Due to the cut-off effects at both small and large s, the region in which $P(s)$ displays its true scaling behaviour is smaller than simple inspection suggests. In particular at small L, the asymptotic scaling effect is still significant up to the large s cut-off region,

so the true system behaviour cannot be observed. With increasing L, a region where neither effect is significant is recovered and the peak in the slope histogram is expected to narrow. Observations of the approach of $\tau_{s,L}$ to $\tau_{s,\infty}$ will be influenced by this, as the measured value for $\tau_{s,L}$ only gradually approaches the true exponent corresponding to the respective system size.

This gradual approach cannot be expected to follow FSS and hence FSS would not be expected to be observable at small system sizes, irrespective of its general validity for the model. The multifractal behaviour of the BTW model described by other investigators [47, 49] could be influenced by such a transient effect. However, their results as well as the numerical results presented here for larger systems support a more fundamental violation of FSS. The finite size scaling assumption

$$\tau_{s,L} = \tau_{s,\infty} - \frac{const.}{\log L} \tag{4.6}$$

first proposed by Manna [50] is based on an empirical argument and allows to find $\tau_{s,\infty}$ by a linear extrapolation of $\tau_{s,L}$ against $\frac{1}{\log L}$ to infinite system size, i.e. $\frac{1}{\log L} \to 0$. In the context of the above considerations, such a linear extrapolation cannot be used to reliably determine $\tau_{s,\infty}$.

5 Numerical Results

5.1 Results for $P(s)$

Numerical work was focussed on determining τ_s for the BTW and Oslo model. For each of the models $P(s)$ and the corresponding local slope is shown. The same scale is used for all plots of $P(s)$ to allow direct comparison. The plot of the local slope then shows any deviation from pure power law behaviour in more detail and allows a fast and reliable method of measuring the value of the slope. Histograms of the slope values were used to help calculate τ_s values.

5.1.1 1D Oslo

This model was examined under two sets of conditions. The first set, used by Christensen *et al.* [26], has one dissipative boundary and sand is added at the opposite, non-dissipative boundary. The plots for this system show very little variation for the different system sizes other than the large s cut-off, which scales with L^2 (see Figure 5.1). The results indicate $\tau_s = 1.55 \pm 0.02$, consistent with the values presented in [26].

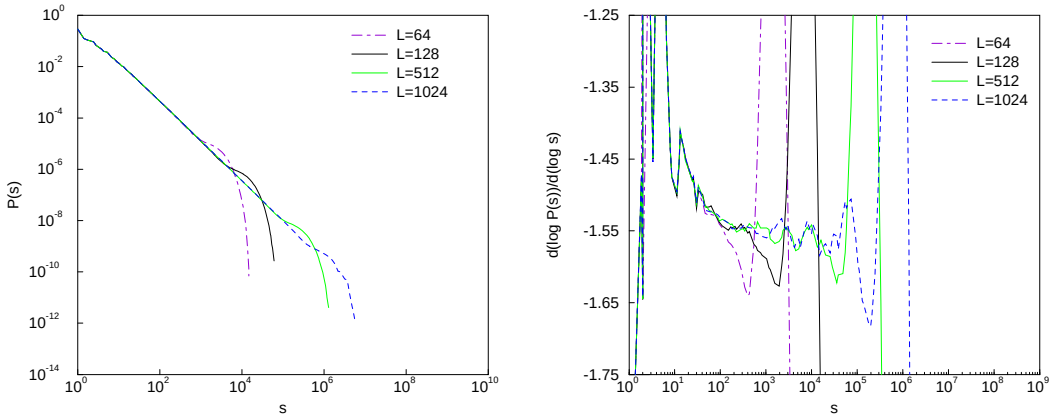


Figure 5.1: Binned size distribution and corresponding local slope for the edge driven 1D Oslo model with one dissipative and one conserving boundary.

Using dissipating boundaries and bulk driving the results obtained are markedly different. The obtained values of τ_s for different system sizes L, $\tau_{s,L}$, show strong variations with L (see Table 5.1). Figure 5.3 shows that the data do not follow FSS for the system sizes calculated and hence equation 4.6 cannot be used to extrapolate to $\tau_{s,\infty}$.

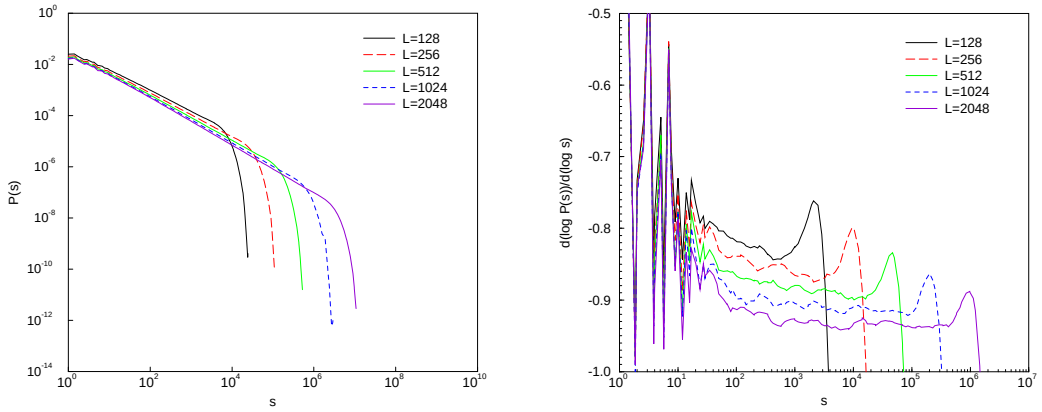


Figure 5.2: Binned size distribution and corresponding local slope for the bulk driven 1D Oslo model with dissipative boundaries.

L	$\tau_{s,L}$	No. of avalanches
64	0.790±0.020	10^7
128	0.825±0.020	10^7
256	0.860±0.015	10^7
512	0.885±0.015	10^7
1024	0.910±0.010	10^7
2048	0.935±0.005	10^7

Table 5.1: $\tau_{s,L}$ values for 1D Oslo model with dissipative boundaries and bulk driving.

Also plotted is the variation of $\tau_{s,L}$ with L on a semi-log plot which suggests a relationship of the form

$$\tau_s \propto \log L \ . \tag{5.1}$$

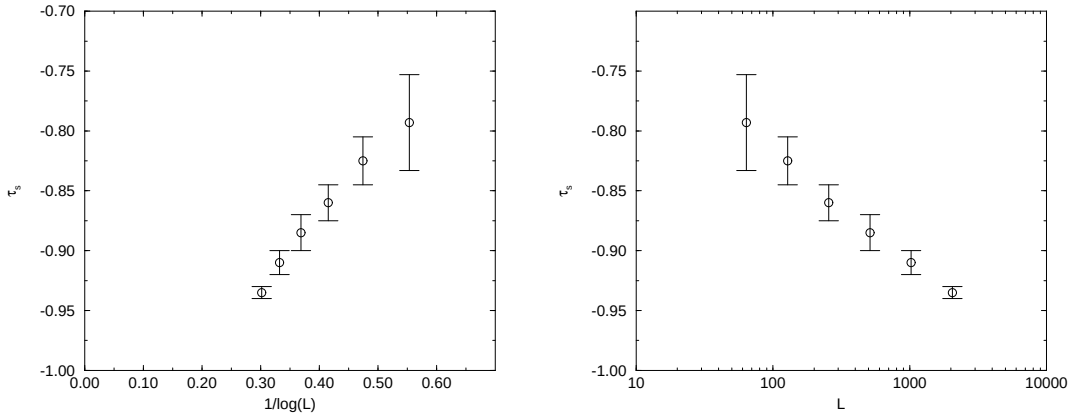


Figure 5.3: Scaling of τ_s with system size for 1D Oslo model with dissipative boundaries and bulk driving.

5.1.2 2D BTW

For the largest system sizes, the plot appears to suddenly bend, which is most clearly seen in the histogram of slope values (see Figure 5.5). The two peaks in the histogram can be seen to correspond to two separate regions in the plot of the local slopes (see Figure 5.4).

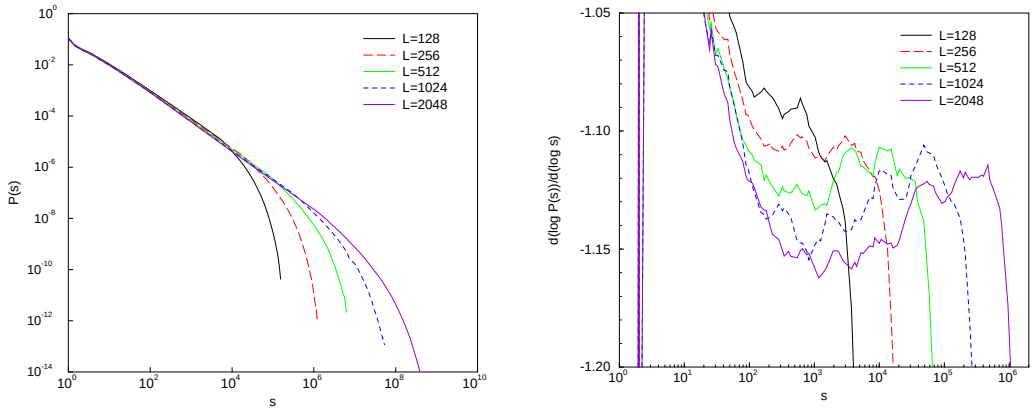


Figure 5.4: Binned size distribution and corresponding local slope for the 2D BTW model.

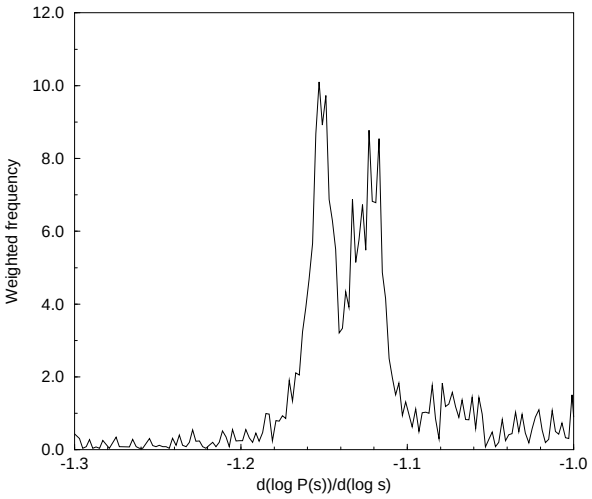


Figure 5.5: Histogram for 2D BTW model with $L = 2048$ showing two distinct slope values.

Because of this effect, two values for $\tau_{s,L}$ are listed in Table 5.2. The first is a central value, corresponding to measuring an average slope. The value for the steeper region is also given, assuming that the flatter region is due to a large s finite size effect. Thereby, the steeper region would represent the true scaling of the system.

The central values for $\tau_{s,L}$ appear to follow FSS. Extrapolation to an infinite size lattice gives $\tau_{s,\infty} \approx 1.22$ (see Figure 5.6). This value is in agreement with other published results that assume FSS [50, 51]. Performing the same analysis on the lower values yields $\tau_{s,\infty} \approx 1.25$. However, the fit for these values appears to include systematic variations (see Figure 5.7). Equation 5.1 again seems to provide a better description for the scaling of $\tau_{s,L}$.

L	$\tau_{s,L}$ (central)	$\tau_{s,L}$ (lower)	No. of avalanches
64	1.07±0.05		10^7
96	1.085±0.015		10^7
128	1.090±0.010		10^7
192	1.101±0.003		10^7
256	1.107±0.005		10^7
384	1.110±0.010	1.120 ±0.010	10^7
512	1.120±0.015	1.125 ±0.010	10^7
1024	1.135±0.015	1.140 ±0.010	10^7
1536	1.135±0.020	1.148 ±0.010	10^7
2048	1.14±0.03	1.155 ±0.010	10^7

Table 5.2: 2D BTW $\tau_{s,L}$ values.

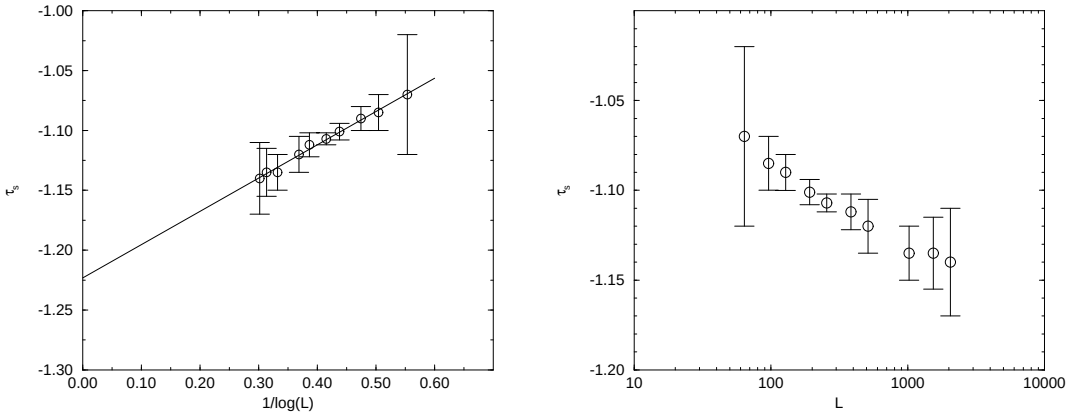


Figure 5.6: Scaling of the central values of $\tau_{s,L}$ with L for 2D BTW model.

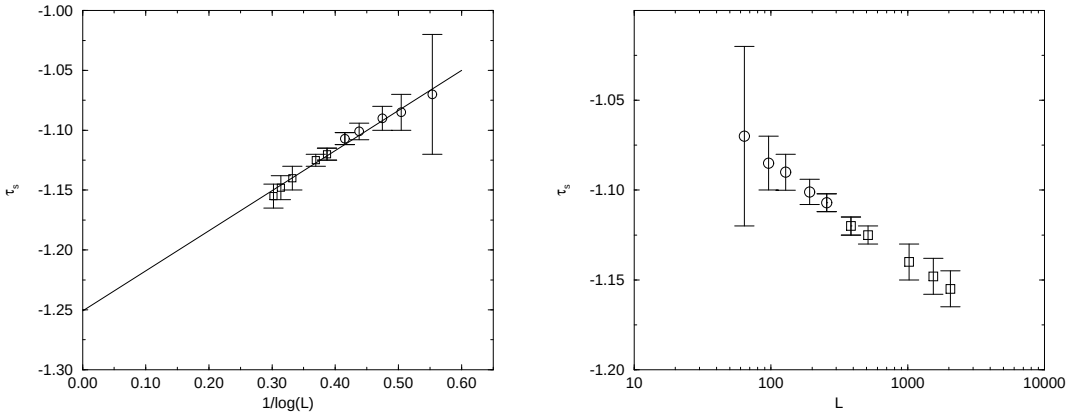


Figure 5.7: Scaling of $\tau_{s,L}$ with L for 2D BTW model using lower values where available (square symbols) or central values (circles).

Preliminary analysis has been performed on dissipative and non-dissipative avalanches. These results seem to show that, when considered separately, the values of $\tau_{s,L}$ for these avalanches do not scale as strongly as their combined values. For dissipating avalanches $\tau_{s,\infty} \approx 0.78$ and for non-dissipating $\tau_{s,\infty} \approx 1.18$. Systems of $L \leq 2048$ were evaluated, but it is stressed that these results are based on only 10^6 avalanches.

5.1.3 2D Oslo

The distribution of avalanche sizes in the Oslo model in two dimensions is similar to the edge driven 1D Oslo model with one conserving boundary. The values of $\tau_{s,L}$ are much less dependent on system size and, within error, are consistent with a constant τ_s for all system sizes.

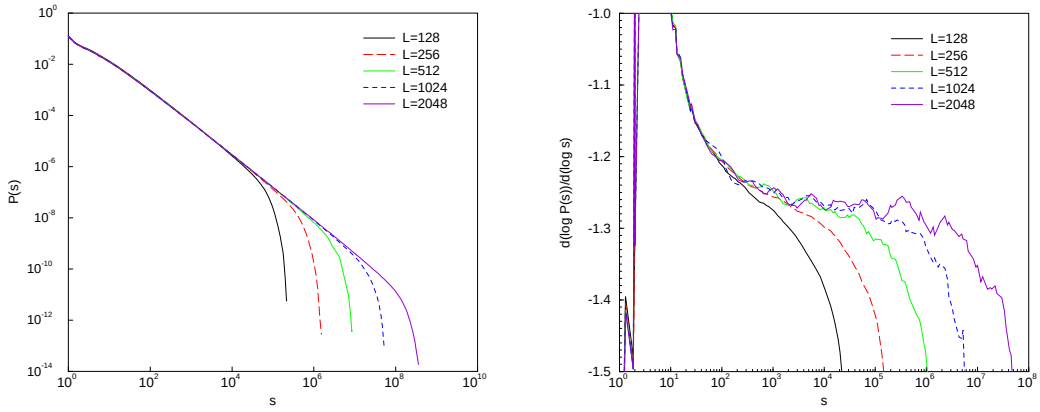


Figure 5.8: Binned size distribution and corresponding local slope for the 2D Oslo model.

L	$\tau_{s,L}$	No. of avalanches
128	1.26±0.05	10^7
256	1.26±0.02	10^7
512	1.27±0.02	10^7
768	1.26±0.02	10^7
1024	1.26±0.02	10^7
1536	1.26±0.02	7×10^6
2048	1.265±0.01	9×10^6

Table 5.3: 2D Oslo $\tau_{s,L}$ values.

5.1.4 3D BTW

Due to memory constraints, only systems with a relatively small linear extent could be calculated. Consequently the values of $\tau_{s,L}$ are difficult to determine accurately, giving the large errors listed in Table 5.4. No extrapolation to $\tau_{s,\infty}$ was attempted due to these restrictions. The values obtained are consistent with previously published values of $\tau_{s,256} = 1.33$ [52], $\tau_{s,80} = 1.34$ [23] and $\tau_{s,20} = 1.35$ [2].

L	$\tau_{s,L}$	No. of avalanches
64	1.35±0.05	10^7
128	1.34±0.04	10^7
256	1.33±0.03	10^7

Table 5.4: 3D BTW $\tau_{s,L}$ values.

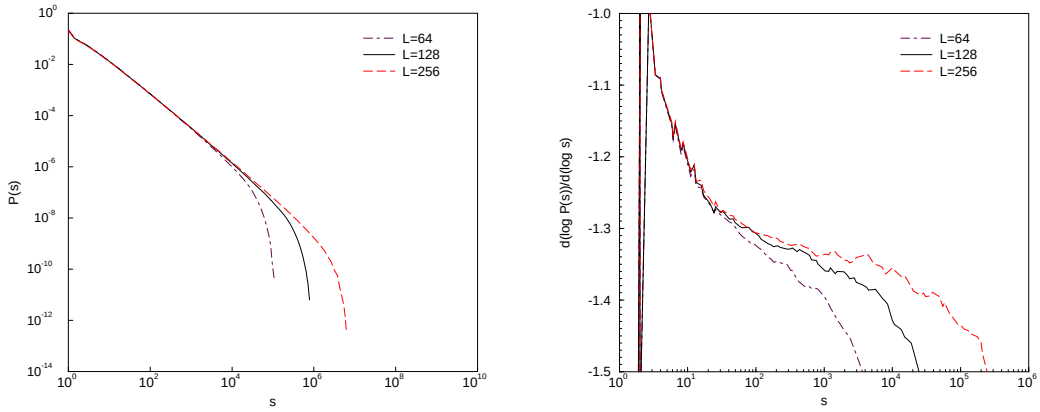


Figure 5.9: Binned size distribution and corresponding local slope for the 3D BTW model.

5.1.5 3D Oslo

The restriction on systen size again prevents an accurate measurement of $\tau_{s,L}$. Within these limits, both systems in three dimensions display very similar properties.

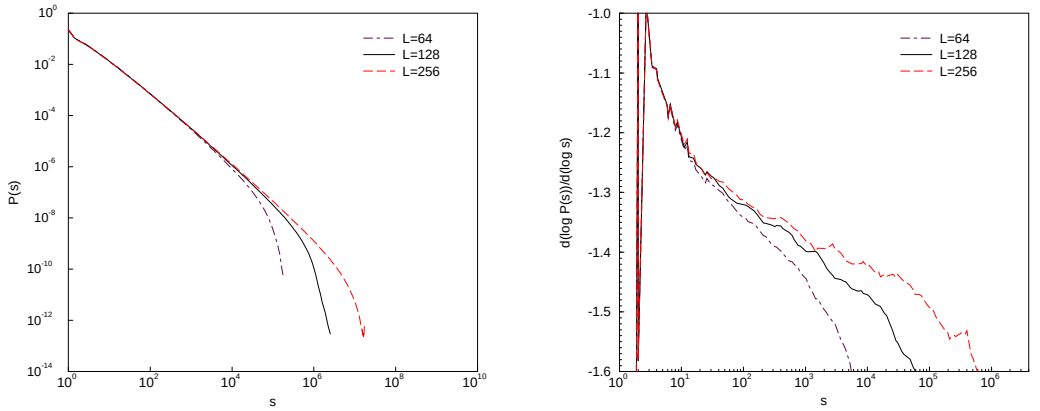


Figure 5.10: Binned size distribution and corresponding local slope for the 3D Oslo model.

L	$\tau_{s,L}$	No. of avalanches
64	1.35±0.1	10^7
128	1.40±0.1	10^7
256	1.40±0.08	10^7

Table 5.5: 3D Oslo $\tau_{s,L}$ values.

5.2 Structure of avalanches

Figures 5.11 and 5.12 show the progression of avalanches with time for the bulk driven 1D Oslo model with dissipative boundary conditions and the 2D BTW model, respectively. Figure 5.11 shows one dissipative and one non-dissipative avalanche, both with frequent multiple topplings. The plots clearly show self-similar patterns. However, the structure of avalanches in the 2D BTW is known to deviate from that of a simple fractal due the effect of 'aging' in the propagating avalanche [36, 41].

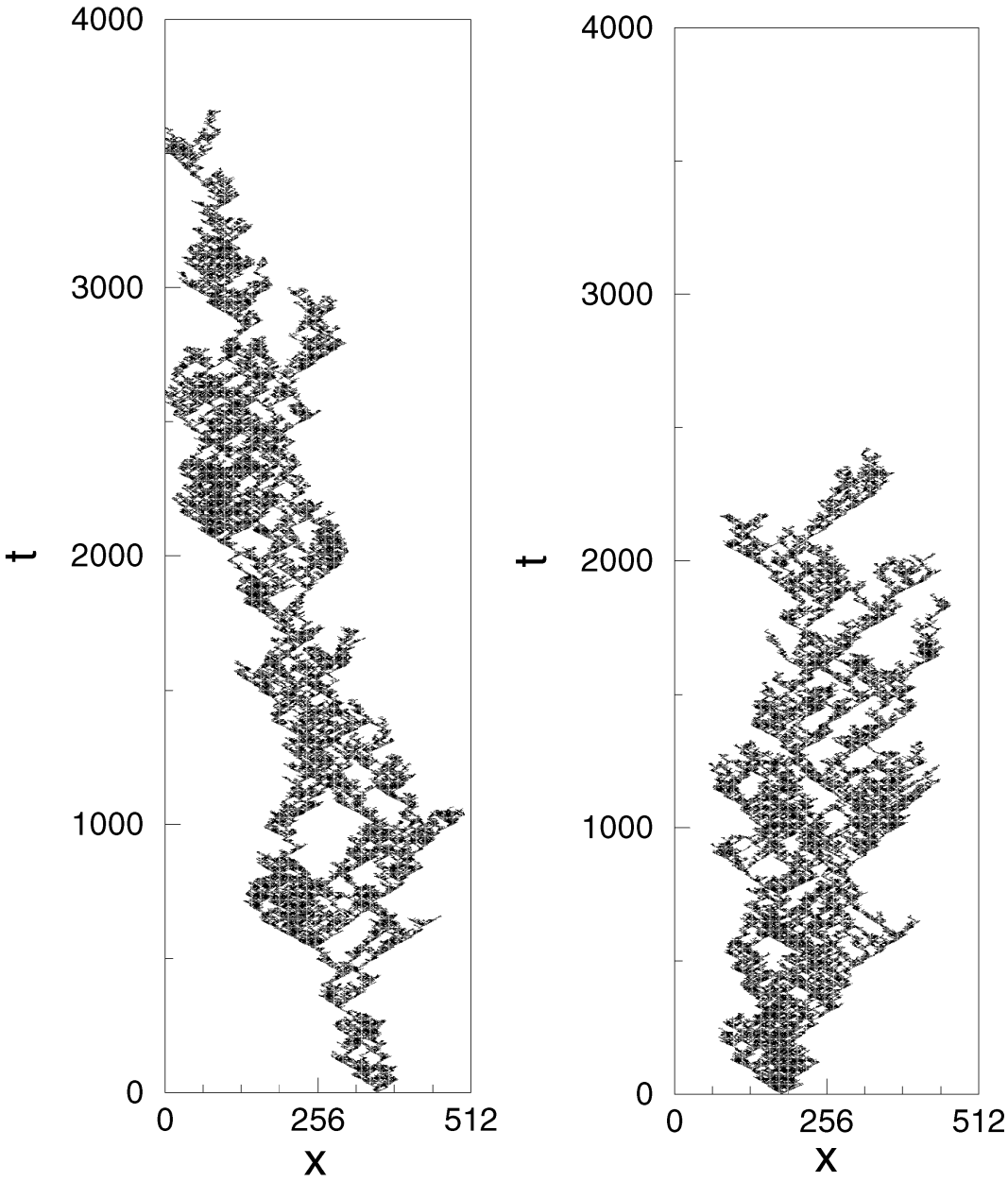


Figure 5.11: A dissipating (left) and a non-dissipating (right) avalanche of size $s \approx 100000$ in the bulk driven 1D Oslo model with dissipative boudary conditions for $L = 512$. The large number of multiple topplings can be clearly seen.

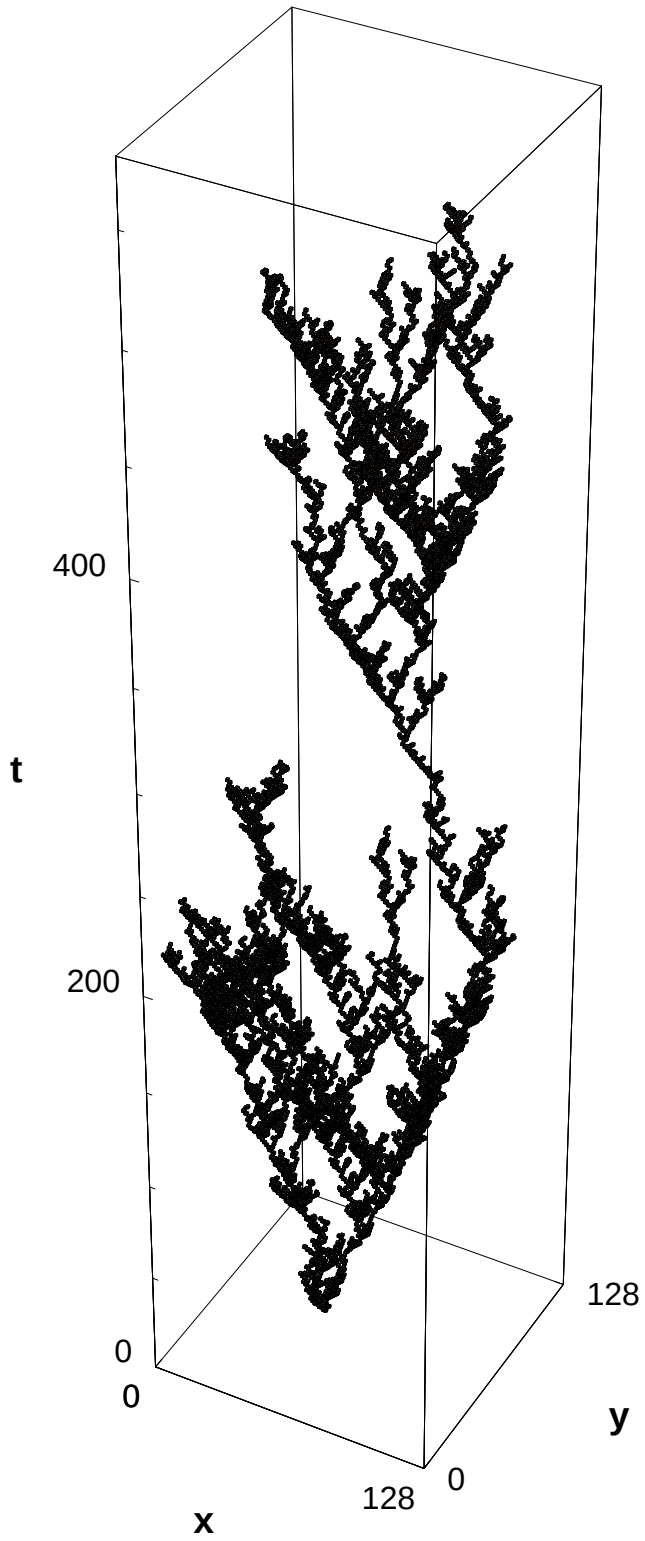


Figure 5.12: Example of the structure of an avalanche in the 2D BTW model. The avalanche contains $s \approx 12000$ topplings on a lattice of $L = 128$ and self-similar patterns are evident. Figure 4.1 shows the number of topplings at each site for this avalanche.

6 Discussion

In the analysis of $P(s)$ for the 2D BTW model, an unexpected second power law region is observed. This region is flatter and can be thought of as a cut-off, similar to the upward curved cut-off in the 1D Oslo model. This interpretation is supported by the observation that the onset of this flatter region moves to larger s proportional to L^2. Considering the contributions of dissipative and non-dissipative avalanches to $P(s)$ separately, it appears that the onset of the flatter region coincides with the point at which the fraction of dissipative avalanches becomes significant. However, the exact nature of the observed 'bend' in the distribution cannot easily be justified and will require further knowledge of the individual scaling behaviours of these two avalanche classes. For the Zhang model [15], a similar bend in $P(s)$ has been observed [53]. Here, the effect is observed as an apparent broadening of the peak in the slope histogram, apparently consistent with the multifractality proposed by Tebaldi *et al.* [49]. For large systems and good statistics, the broad peak can be resolved as a double peak (see Figure 5.5). Inspection of the plots of the local slope against s (see Figure 5.4) confirm that these peaks correspond to two distinct power law regions.

The expected asymptotic scaling is clearly observed for $P(s)$ in all models. Following the argument given in Section 4.2, a model in which the elementary dynamics are closer to the asymptotic dynamics should display much weaker deviation from $\tau_{s,L}$ at small s. It should thus be possible to construct a model in which these dynamics differ minimally using stochastic rules for the number of grains distributed in a toppling at the smallest scale. In such a model, the scaling of $P(s)$ due to other scale-dependent effects could be observed more easily and reliable extrapolation of the behaviour would be possible from smaller L and thus with significantly reduced computational effort.

The strong proportionality to $\log L$ observed for the values of $\tau_{s,L}$ in the bulk driven 1D Oslo model with dissipative boundary conditions and the 2D BTW model is unexpected and raises fundamental questions in attempts to interpret it. An indefinite drop of $\tau_{s,L}$ cannot be reconciled with SOC, as for $\tau_{s,L} > 2$, $P(s)$ would have a finite first moment for infinite system sizes. An end of the $\log L$ proportionality might thus be expected, but no indication of this is seen at the available system sizes. These findings do not support any classification of the universality of the models displaying this behaviour based on numerical results. The possible universality of the Oslo model in two dimensions and the 2D BTW model pointed to by FSS analysis can therefore not be confirmed or excluded. Any connection of this observation to a similar restricted $\log L$ proportionality of fractal dimensions in other systems [54, 55] has not been investigated.

In the 1D Oslo model, this $\log L$ dependence is found only for purely dissipative boundary conditions and bulk driving, whereas the original edge driven configuration

shows an independence of L in its scaling, as does the 2D Oslo model. This underlines the significance of the boundary and driving conditions at least for this model, although SOC occurs for both configurations and an interpretation of the conditions as 'tuning' can therefore not apply. Differences between bulk driving and edge driving have recently been discussed on the basis of changes in the power spectrum of the temporal fluctuations of the average slope [56].

The 1D Oslo model also differs from the corresponding physical experiment. SOC has been observed in a pile of long-grain rice with one open boundary [14], but the experimental exponent was determined to be slightly above 2 and thus significantly larger than the value of $\tau_s = 1.55 \pm 0.02$ determined here and in previous work [26, 57] for the model. A physically occuring effect not included in the computational Oslo model is the 'jumping' of toppling grains observed in the rice pile. In this, a grain directed 'downhill' does not fall on the nearest neighbour of the toppling site but chooses a single site in its direction at random within a certain range. Essentially, this is a variation of the nonlocal rules first proposed by Kadanoff *et al.* [58–60]. Further work on this is justified as any extension to the 1D Oslo model reconciling physical and computational results may be a useful step to understanding SOC in real systems.

7 Conclusion

In this report we have investigated the scaling behaviour of the probability distribution of the number of of topplings $P(s)$ of the BTW and Oslo models in up to three dimensions. The extent of the data gathered exceeds that of published work and allows unexpected features of the scaling to be observed in some systems. For the 2D BTW model and the edge driven 1D Oslo model with one conserving boundary, the values of $\tau_{s,\infty}$ in previous work can be recovered if analysis methods based on a finite size scaling approach are applied. However, the previously described breakdown of FSS is confirmed by the observation, invalidating these results. As part of this breakdown in the 2D BTW model, two distinct power law regions are observed for $P(s)$ and tentatively attributed to differences in the scaling behaviours of dissipative and non-dissipative avalanches.

The result of proportionality to $\log L$ of the values of $\tau_{s,L}$ in the bulk driven 1D Oslo model and steeper of the two regions in the 2D BTW model does not support the existence of a limiting value of τ_s for infinite system sizes. Further data for larger system sizes would be essential to confirm any restriction of the proposed proportionality to small L and thus attribute it to finite size effects. We also show invariance of $\tau_{s,L}$ with L for two sytems. Both the bulk driven two dimensional Oslo model with dissipative boundaries and the one dimensional version with edge driving and one conserving boundary do not display measurable scaling of $\tau_{s,L}$. The available data does not allow conclusions on the scaling behaviour for the three dimensional systems studied.

At small s, the scale-dependence of system dynamics has been observed numerically in the asymptotic approach of the slope of $P(s)$ to $\tau_{s,L}$ in all models. The presented fast approximation to the DDRG description of the dynamics in the 2D BTW model correctly reproduces this behaviour in its trajectory towards the fixed point. Further work on this effect could include the construction of a model with invariant behaviour on all scales, and thus necessarily stochastic rules.

Our results underline the influence of finite size effects at both small and large scales on the obtainable results for τ_s. It cannot be concluded whether these effects are the cause of the observed scaling behaviour or just partly obscure more fundamental mechanisms.

Acknowledgements

It is a pleasure to thank Dr. Kim Christensen for helpful discussions and encouragement throughout the project. We are grateful to Prof. Henrik Jeldtoft Jensen and Dr. Maya Paczuski for useful comments and insights. We also thank the Condensed Matter Theory Group at Imperial College for the provision of extensive computing facilities and notably Prof. Angus MacKinnon for technical advice.

Bibliography

[1] P. Bak, C. Tang, and K. Wiesenfeld, Phys. Rev. Lett. **59**, 381 (1987).

[2] P. Bak, C. Tang, and K. Wiesenfeld, Phys. Rev. A **38**, 364 (1998).

[3] P. Bak, in *How Nature Works* (Springer-Verlag, New York, 1996).

[4] H. J. Jensen, in *Self-Organized Criticality* (Cambridge University Press, Cambridge, 1998).

[5] Z. Olami, H. J. S. Feder, and K. Christensen, Phys. Rev. Lett. **68**, 1244 (1992).

[6] P. Bak, K. Chen, and C. Tang, Phys. Lett. A **147**, 297 (1990).

[7] B. Drossel and F. Schwabl, Phys. Rev. Lett. **69**, 1629 (1992).

[8] S. Field, J. Witt, F. Nori, and X. S. Ling, Phys. Rev. Lett. **74**, 1206 (1995).

[9] P. Bak and K. Sneppen, Phys. Rev. Lett. **71**, 4083 (1993).

[10] A. Malthe-Sørenssen, Ph.D. thesis, University of Oslo, Norway, 1998.

[11] H. M. Jaeger, C. Liu, and S. R. Nagel, Phys. Rev. Lett. **62**, 40 (1989).

[12] G. A. Held, D. H. Solina, H. Solina, D. T. Keane, W. J. Haag, P. M. Horn, and G. Grinstein, Phys. Rev. Lett. **65**, 1120 (1990).

[13] J. Rosendahl, M. Vekić, and J. Kelly, Phys. Rev. E **47**, 1401 (1993).

[14] V. Frette, K. Christensen, A. Malthe-Sørenssen, J. Feder, T. Jøssang, and P. Meakin, Nature **379**, 49 (1996).

[15] Y. C. Zhang, Phys. Rev. Lett. **63**, 470 (1989).

[16] S. S. Manna, J. Phys. A **24**, L363 (1991).

[17] L. Pietronero, A. Vespignani, and S. Zapperi, Phys. Rev. Lett. **72**, 1690 (1994).

[18] A. Vespignani, S. Zapperi, and L. Pietronero, Phys. Rev. E **51**, 1711 (1995).

[19] H. J. Jensen, Europhys. Lett. **35**, 397 (1996).

[20] A. Díaz-Guilera, Phys. Rev. A **45**, 8551 (1992).

[21] A. Corral and A. Díaz-Guilera, Phys. Rev. E **55**, 2434 (1997).

[22] S. Lübeck, Phys. Rev. E **56**, 1590 (1997).

[23] S. D. Edney, P. A. Robinson, and D. Chisholm, Phys. Rev. E **58**, 5395 (1998).

[24] E. Milshtein, O. Biham, and S. Solomon, Phys. Rev. E **58**, 303 (1998).

[25] A. Ben-Hur and O. Biham, Phys. Rev. E **53**, R1317 (1996).

[26] K. Christensen, A. Corral, V. Frette, J. Feder, and T. Jøssang, Phys. Rev. Lett. **77**, 107 (1996).

[27] S. S. Manna, Curr. Sci. **77**, 388 (1999).

[28] W. H. Press, S. A. Teuklosky, W. T. Vetterling, and B. P. Flannery, in *Numerical Recipes in C*, 2nd ed. (Cambridge University Press, Cambridge, 1992), Chap. 7, pp. 280–282.

[29] W. H. Press, S. A. Teuklosky, W. T. Vetterling, and B. P. Flannery, in *Numerical Recipes in C*, 2nd ed. (Cambridge University Press, Cambridge, 1992), Chap. 8, pp. 336–338.

[30] D. Dhar, Phys. Rev. Lett. **64**, 1613 (1990).

[31] S. N. Majumdar and D. Dhar, J. Phys. A **24**, L357 (1991).

[32] V. B. Priezzhev, J. Stat. Phys. **74**, 955 (1994).

[33] E. V. Ivashkevich, J. Phys. A **27**, 3643 (1994).

[34] D. Dhar and S. S. Manna, Phys. Rev. E **49**, 2684 (1994).

[35] V. B. Priezzhev, D. V. Ktitarev, and E. V. Ivashkevich, Phys. Rev. Lett. **76**, 2093 (1996).

[36] M. Paczuski and S. Boettcher, Phys. Rev. E **56**, 3745 (1997).

[37] D. V. Ktitarev and V. B. Priezzhev, Phys. Rev. E **58**, 2883 (1998).

[38] D. V. Ktitarev, S. Lübeck, P. Grassberger, and V. B. Priezzhev, Phys. Rev. E **61**, 81 (2000).

[39] K. Christensen and Z. Olami, Phys. Rev. E **48**, 3361 (1993).

[40] L. Pietronero and W. R. Schneider, Phys. Rev. Lett. **66**, 2336 (1991).

[41] S. Boettcher and M. Paczuski, Phys. Rev. Lett. **79**, 889 (1997).

[42] P. Grassberger and S. S. Manna, J. Phys. (Paris) **63**, 1077 (1990).

[43] L. Pietronero, P. Tartaglia, and Y.-C. Zhang, Physica A **173**, 22 (1991).

[44] E. V. Ivashkevich, A. M. Povolotsky, A. Vespignani, and S. Zapperi, Phys. Rev. E **60**, 1239 (1999).

[45] Y. Moreno, J. B. Gómez, and A. F. Pacheco, Phys. Rev. E **60**, 7565 (1999).

[46] D. Dhar and S. N. Majumdar, J. Phys. A **23**, 4333 (1990).

[47] M. D. Menech, A. L. Stella, and C. Tebaldi, Phys. Rev. E **58**, 2677 (1998).

[48] B. Drossel, cond-mat/9904075 (1999).

[49] C. Tebaldi, M. D. Menech, and A. L. Stella, Phys. Rev. Lett. **83**, 3952 (1999).

[50] S. S. Manna, Physica A **179**, 249 (1991).

[51] S. Lübeck and K. D. Usadel, Phys. Rev. E **55**, 4095 (1997).

[52] S. Lübeck and K. D. Usadel, Phys. Rev. E **56**, 5138 (1997).

[53] E. Milshtein, O. Biham, and S. Solomon, Phys. Rev. E **58**, 303 (1998).

[54] K. Chen and P. Bak, cond-mat/9912417 v2 (2000).

[55] P. Bak and K. Chen, astro-ph/0001443 (2000).

[56] S. D. Zhang, cond-mat/0002270 (2000).

[57] L. A. N. Amaral and K. B. Lauritsen, Physica A **231**, 608 (1996).

[58] L. P. Kadanoff, S. R. Nagel, L. Wu, and S. Zhou, Phys. Rev. A **39**, 6524 (1989).

[59] L. A. N. Amaral and K. B. Lauritsen, Phys. Rev. E **56**, 231 (1997).

[60] M. Markošová, Phys. Rev. E **61**, 253 (2000).

Appendices

A Overview of Simulation Code

A.1 Program structure

The program uses two classes to implement the sandpile and the queue of critical sites. The data within the two classes are kept as private variables, with member functions providing access to the required variables. The structure of the program is shown in Figure A.1.

The first stage of the program involves generating the classes and their required variables. `Globals.h` contains all variables with global scope. `Sandpile.h` and `Queue.h` contain variables which can be accessed by any function in the classes `CSandpile` and `CQueue`, respectively. Finally, any variables which are only used within a particular function are declared within that function.

The sandpile array is initialised with random slopes between 0 and $z_c - 1$. The slope is then increased by 1 at a random location and any resulting avalanche is allowed to finish propagating before the next addition. The avalanche propagation is dealt with by a queuing system, explained in detail in section 3.1.2. Results are only recorded after the system reaches the stationary state (see section 3.1.1).

As the queue size is restricted, there is the possibility of active data being overwritten when 'wrapping around', i.e. sites that have not yet relaxed being replaced in the queue by new sites. There is no easy way to calculate the size of queue required to prevent this from happening, so a check is performed such that if active data is about to be overwritten, an error message is given and the program stops. A complete description is available only for those avalanches with a size up to the length of the queue.

The program uses virtual memory to store the size and duration of very large avalanches. This is described in section 3.1.4.

A.2 Known limitations

The code cannot address arrays larger than ≈ 100 Mbytes, which is required for a 3D system with $L \geq 512$, or a 2D system with $L \geq 8192$. A possible workaround would be to split the arrays into two or more smaller sections. A better solution would be to write a class to handle large arrays using a different memory addressing technique.

If the code is compiled on a machine using an OS other than UNIX or Win32, the code will default to UNIX file handling commands which may result in errors.

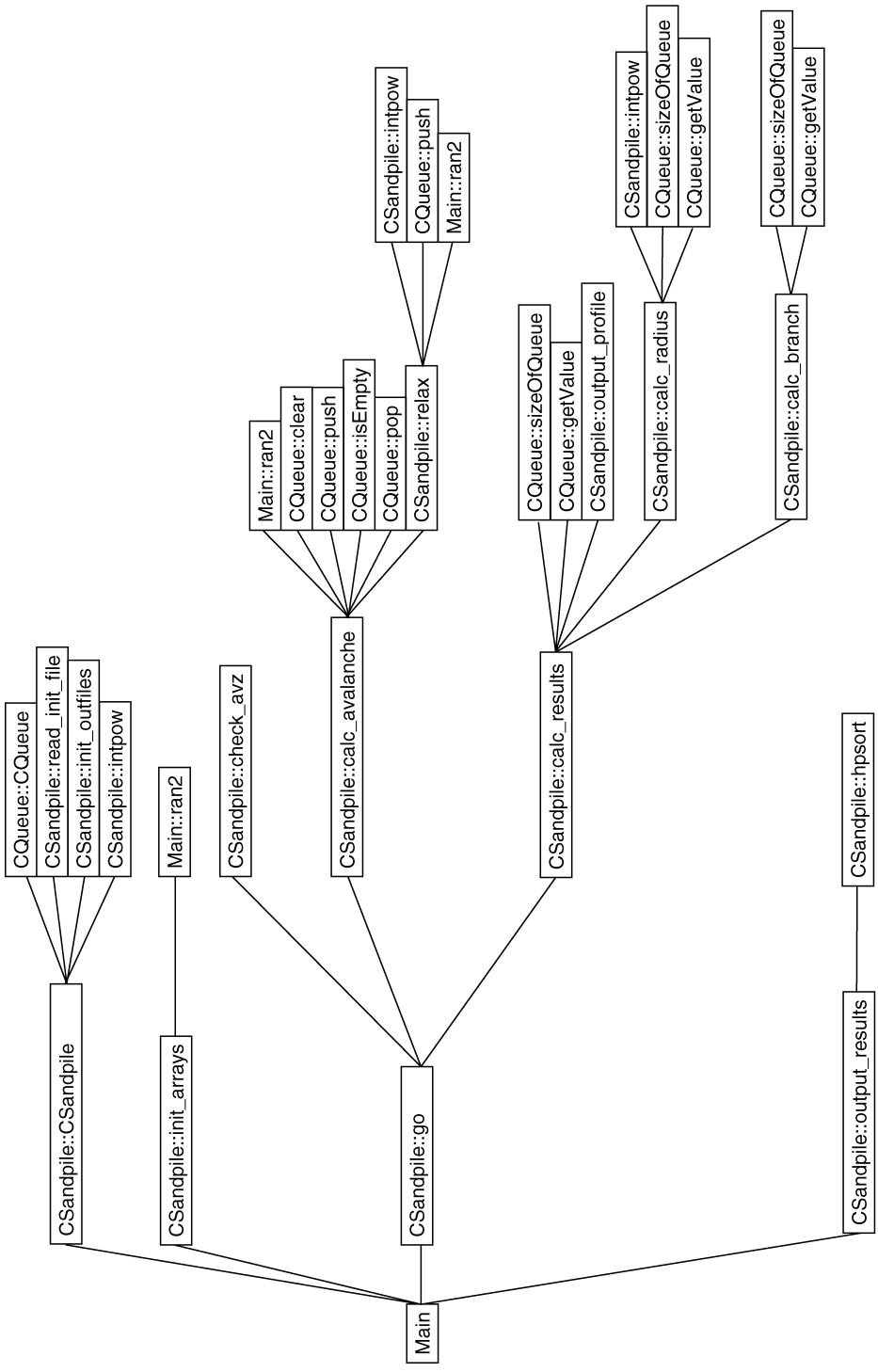


Figure A.1: Function dependency

B Simulation Code

```
#ifndef __GLOBALS_HEADER
#define __GLOBALS_HEADER

// required header files
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <sys/timeb.h>
#include <sys/types.h>
#include <sys/stat.h>


// define boolean type and UNIX include files
#ifdef WIN32
#include <direct.h>
#include <windows.h>
#include <conio.h>
#else
#include <unistd.h>
#define BOOL int
#define TRUE 1
#define FALSE 0
#endif

// Required definitions for ran2 random number generator
#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 0.00000012 // 1.2e-7
#define RNMX (1.0-EPS)

// define classes
typedef class CQueue *PQueue;
typedef class CSandpile *PSandpile;

// define data structures
struct SAvInfo
{
   int pos;
   int time;
};

// define global functions
```

```
double ran2(long *);

const int sampleTime=10000;
// header files required for queue and sandpile classes
#include "Queue.h"
#include "Sandpile.h"


#endif // __GLOBALS_HEADER

#include "Globals.h"

int main()
{

   PSandpile sandpile = new CSandpile();  // create sandpile

   sandpile->init_arrays();  // initialise all arrays

   sandpile->go();      // start calculation loop

   if (sandpile)
   {
      delete sandpile;  // tidy up
      sandpile = NULL;
   }
   return 0;
}

// Random number generator ran2
// from 'Numerical Recipes in C (2nd ed.)', p. 282
// Calculates a float from 0 to 1 exclusive
double ran2(long *idum)
{
   int j;
   long k;
   static long idum2=123456789;
   static long iy=0;
   static long iv[NTAB];
   double temp;

   if (*idum<=0)
   {
      if (-(*idum)<1) *idum=1;
         else *idum=-(*idum);
      idum2=(*idum);
      for (j=NTAB+7;j>=0;j--)
      {
         k=(*idum)/IQ1;
         *idum=IA1*(*idum-k*IQ1)-k*IR1;
         if (*idum<0) *idum+=IM1;
         if (j<NTAB) iv[j]=*idum;
      }
      iy=iv[0];
   }
   k=(*idum)/IQ1;
   *idum=IA1*(*idum-k*IQ1)-k*IR1;
   if (*idum<0) *idum+=IM1;
   k=idum2/IQ2;
   idum2=IA2*(idum2-k*IQ2)-k*IR2;
   if (idum2<0) idum2+=IM2;
   j=iy/NDIV;
   iy=iv[j]-idum2;
   iv[j]=*idum;
   if (iy<1) iy+=IMM1;
   if ( (temp=AM*iy) >RNMX)
      return RNMX;
   else return temp;
}
```

```
#ifndef __SANDPILE_HEADER
#define __SANDPILE_HEADER

#include "Globals.h"

// Declares all variables with scope of CSandpile, and
// declares function prototypes

class CSandpile
{
private:
   int maxRad, *rad_data;
   int maxAvSize, *dur_data, *size_data, *prf_data, *largeav_data;
   int *dur_data_d, *dur_data_nd, *size_data_d, *size_data_nd, *largeav_data_d, *largeav_data_nd;
   SFracInfo *boxInfo, *planeInfo;
   CQueue      *queue;
   char     *zc, *z;
   int *coord;
   int maxGrains, *remainder;
   int L;
   int dimension;
   int zcMinRelax;
   int lPow;
   int systemSize;
   int maxProfiles, sizeToProfile;
   int totalAv, maxAv;
   int output_percent;
   unsigned long largeav_count, largeav_count_d, largeav_count_nd;
   BOOL btw, oslo, rad_flag, dur_flag, branch_flag, corneradd_flag, open_corner_flag, avz_flag;
   BOOL dissipating;

   unsigned int startTime, endTime;
   double totalBranch, *branch_ratio_d, *branch_ratio_nd;
   int totalTime;
   int timeCounter;

   long seed;
   ofstream size_out, dur_out, rad_out, avz_out, prf_out, misc_out, pile_out;
   ofstream dur_out_d, dur_out_nd, largeav_out_d, largeav_out_nd, size_out_d, size_out_nd;

public:
   CSandpile();
   ~CSandpile();

   void read_init_file();
   void init_outfiles();
   void init_arrays();

   inline BOOL calc_avalanche();
   inline void relax(SAvInfo &aI);

   inline BOOL check_avz();

   inline void calc_results();
   inline double calc_radius();
   inline void calc_branch();
   inline int intpow(int, int);

   void output_profile(SAvInfo &aI);
   void combine_results(int);
   void output_results(int);
   void hpsort(unsigned long, int []);
   inline void components(int x);

   void go();

};
```

```
#endif // __SANDPILE_HEADER

#include "Sandpile.h"

//  CSandpile constructor. Initialises variables and arrays used
CSandpile::CSandpile()
{
   output_percent=0;
   cout << "Reading parameter file" << endl;
   read_init_file();
   cout << dimension << "D\t L=" << L << endl;
   seed = -(long)time(NULL);
   startTime = (unsigned)time(NULL);
   zcMinRelax=2*dimension;
   maxAvSize=1000000;
   systemSize=intpow(L,dimension);
   largeav_count=largeav_count_d=largeav_count_nd=0;
   slicesCount=0;
   timeCounter=0;

   cout << "Initialising output files" << endl;
   init_outfiles();
}

// CSandpile destructor. Deletes arrays and returns memory to heap
CSandpile::~CSandpile()
{
   if (z)
   {
      delete [] z;
      z = NULL;
   }
   if (rad_flag)
   {
      delete [] rad_data;
      rad_data = NULL;
   }
   if (size_data)
   {
      delete [] size_data_d;
      delete [] size_data_nd;
      size_data = NULL;
   }
   if (dur_flag)
   {
      delete [] dur_data;
      dur_data = NULL;
   }
   if (branch_flag)
   {
      delete [] prf_data;
      prf_data = NULL;
   }
   if (queue)
   {
      delete [] queue;
      queue = NULL;
   }
   if (zc)
   {
      delete [] zc;
      zc = NULL;
   }
   if (remainder)
   {
      delete [] remainder;
      remainder = NULL;
   }
}
```

```
// Reads in run parameters from initialisation file
void CSandpile::read_init_file()
{
   char descriptor[30];

   ifstream in("parameters.txt", ios::nocreate);
#ifdef WIN32
   if(!in.is_open())    // not available in UNIX c++
   {
      cerr << "Input file not found" << endl;
      exit(1);
   }
#endif

   if (!in.eof())
      in >> descriptor >> dimension;
   if (!in.eof())
      in >> descriptor >> L;
   if (!in.eof())
      in >> descriptor >> maxAv;
   if (!in.eof())
      in >> descriptor >> btw;
   if (!in.eof())
      in >> descriptor >> oslo;
   if (!in.eof())
      in >> descriptor >> rad_flag;
   if (!in.eof())
      in >> descriptor >> dur_flag;
   if (!in.eof())
      in >> descriptor >> branch_flag;
   if (!in.eof())
      in >> descriptor >> open_corner_flag;
   if (!in.eof())
      in >> descriptor >> corneradd_flag;
   if (!in.eof())
      in >> descriptor >> maxProfiles;
   if (!in.eof())
      in >> descriptor >> sizeToProfile;
   if (!in.eof())
      in >> descriptor >> avz_flag;
   if (!in.eof())
      in >> descriptor >> output_percent;
   if ((output_percent<0)||(output_profile>100))
      output_percent=100;
}

// Opens output files ready for results. Also creates a new folder
// to store the files in
void CSandpile::init_outfiles()
{
   time_t timer;

#ifdef WIN32
   struct _stat dummybuf;
   int n=0, sign, decptr;
   char descriptor[30];

   strcpy(descriptor,ecvt(dimension,1,&sign,&decptr));
   strcat(descriptor,"D");
   if (_stat(descriptor,&dummybuf)!=0)
      _mkdir(descriptor);
   _chdir(descriptor);

   if (btw)
      strcpy(descriptor,"BTW");
   else
   {
      if (oslo)
         strcpy(descriptor,"Oslo");
      else
```

```
         strcpy(descriptor,"MaxRandom");
   }
   if (_stat(descriptor,&dummybuf)!=0)
      _mkdir(descriptor);
   _chdir(descriptor);

   do
   {
      n++;
      strcpy(descriptor,"Run_");
      strcat(descriptor,ecvt(n,1,&sign,&decptr));
   }while(_stat(descriptor,&dummybuf)==0);

   strcpy(descriptor,"Run_");
   // create output directory
   mkdir(strcat(descriptor,ecvt(n,1,&sign,&decptr)));
   // change working directory
   chdir(descriptor);

   cout << descriptor << " Folder created" << endl;
#else

   struct stat dummybuf;
   int n=0, sign, decptr;
   char descriptor[30];

   strcpy(descriptor,ecvt(dimension,1,&sign,&decptr));
   strcat(descriptor,"D");
   if (stat(descriptor,&dummybuf)!=0)
      mkdir(descriptor,0777);
   chdir(descriptor);

   if (btw)
      strcpy(descriptor,"BTW");
   else
   {
      if (oslo)
         strcpy(descriptor,"Oslo");
      else
         strcpy(descriptor,"MaxRandom");
   }
   if (stat(descriptor,&dummybuf)!=0)
      mkdir(descriptor,0777);
   chdir(descriptor);

   do
   {
      n++;
      strcpy(descriptor,"Run_");
      strcat(descriptor,ecvt(n,1,&sign,&decptr));
   }while(stat(descriptor,&dummybuf)==0);

   strcpy(descriptor,"Run_");
   // create output directory
   mkdir(strcat(descriptor,ecvt(n,1,&sign,&decptr)),0777);
   // change working directory
   chdir(descriptor);

   cout << descriptor << " Folder created" << endl;
#endif

   if(branch_flag)
      prf_out.open("profiles");
   avz_out.open("avz");
// pile_out.open("pile");
   misc_out.open("misc");
   largeav_out_d.open("largeavD");
   largeav_out_nd.open("largeavND");

   timer=time(NULL);
```

```
   misc_out << ctime(&timer) << endl;
   misc_out << "Dimension = " << dimension << endl;
   misc_out << "L = " << L << endl;
   misc_out << "maxAv = " << maxAv << endl;
   misc_out << "btw = " << btw << endl;
   misc_out << "oslo = " << oslo << endl;
   misc_out << "rad_flag = " << rad_flag << endl;
   misc_out << "dur_flag = " << dur_flag << endl;
   misc_out << "branch_flag  = " << branch_flag << endl;
   misc_out << "open_corner_flag  = " << open_corner_flag << endl;
   misc_out << "corneradd_flag  = " << corneradd_flag << endl;
   misc_out << "maxProfiles = " << maxProfiles << endl;
   misc_out << "sizeToProfile = " << sizeToProfile << endl;
   misc_out << "avz_flag = " << avz_flag << endl;
}

// sets initial values for array elements
void CSandpile::init_arrays()
{
   cout << "Initialising arrays" << endl;

   z = new char[systemSize];
   zc = new char[systemSize];
   queue     = new CQueue(maxAvSize);
   size_data_d = new int[maxAvSize];
   size_data_nd = new int[maxAvSize];
   remainder = new int[dimension+1];
   coord = new int[dimension];

   int i;

   if(dur_flag)
      dur_data = new int[maxAvSize];

   if(rad_flag)
   {
      maxRad = L*dimension;
      rad_data  = new int[maxRad];
   }

   if(branch_flag)
   {
      prf_data  = new int[maxAvSize];
      branch_ratio_d = new double[20];
      branch_ratio_nd = new double[20];

      totalTime=0;
      totalBranch=0.0;
   }

   for (i=0; i<maxAvSize; i++)
   {
      size_data_d[i] = 0;
      size_data_nd[i] = 0;
   }

   if(dur_flag)
   {
      for (i=0; i<maxAvSize; i++)
         dur_data[i] = 0;
   }

   if(rad_flag)
   {
      for (i=0; i<maxRad; i++)
         rad_data[i] = 0;
   }
   for (i=0; i<systemSize; i++)
   {
      if(btw)
```

```
         zc[i]=zcMinRelax;
      else
      {
         if(oslo)
            zc[i]= int(ran2(&seed)*2)+zcMinRelax;
         else
            zc[i]= int(ran2(&seed)*dimension*2)+zcMinRelax;
      }
      z[i] = int(ran2(&seed)*(zc[i]-1)+1);
   }
   cout << "Initialising sandpile..." << endl;
}

// main calculation loop
void CSandpile::go()
{
   BOOL takeData = FALSE, avalanche;
   totalAv=1;
   static int initCtr=0;
   static int outnum=maxAv/(100/output_percent);
   static int percent=maxAv/100;

   while(totalAv<=maxAv)
   {
      dissipating = FALSE;
      avalanche = calc_avalanche();
      initCtr++;
      if(!takeData)
      {
         if(initCtr%1000==0)
         {
            cout << initCtr << " grains calculated" << endl;  // output progression to screen
            takeData = check_avz();  // checks whether to take data
         }
      }

      if (takeData && avalanche)
      {
         if(avz_flag)
            takeData = check_avz();
         calc_results();
         totalAv++;
         if(totalAv%percent==0)
            cout << totalAv/percent << " % Done" << endl;
         if(totalAv%outnum==0)
         {
            output_results(totalAv);
            misc_out << totalAv/percent << "% Completed" << endl;
         }
      }
   }
}

// calculates vector position from array position
void CSandpile::components(int x)
{
   remainder[0]=x;
   for(int n=0; n<dimension; n++)
   {
      lPow = intpow(L, (dimension-(n+1)));
      coord[dimension-(n+1)]=remainder[n]/lPow;
      remainder[n+1]=remainder[n]%lPow;
   }

}

// calc_avalanche() together with relax() add sand to the pile and
// calculate the progression of any resultant avalanche
BOOL CSandpile::calc_avalanche()
{
```

```
   SAvInfo avInfo;
   queue->clear();
   BOOL avalanche = FALSE;
   if(corneradd_flag)
      avInfo.pos = 0;
   else
      avInfo.pos = int(ran2(&seed) * systemSize);
   avInfo.time = 0;
   queue->push(avInfo);
   z[avInfo.pos]++;
   timeCounter++;
   while (!queue->isEmpty())
   {
      avInfo = queue->pop();
      if (z[avInfo.pos] >= zc[avInfo.pos])
      {
         relax(avInfo);
         timeCounter++;
         avalanche = TRUE;
      }
   }
   return avalanche;
}

// calculates additions to nearest neighbours and pushes
// critical sites onto queue
void CSandpile::relax(SAvInfo &aI)
{
   SAvInfo avInfo = aI;
   int x=avInfo.pos;
   avInfo.time++;
   remainder[0]=x;
   z[x]-=2*dimension;
   for(int n=0; n<dimension; n++)
   {
      lPow = intpow(L, (dimension-(n+1)));
      if(remainder[n]/lPow > 0 )
      {
         z[x-lPow]++;
         if (z[x-lPow]==zc[x-lPow])
         {
            avInfo.pos = x-lPow;
            queue->push(avInfo);
         }
      }
      else
         dissipating=TRUE;

      if(remainder[n]/lPow < L-1)
      {
         z[x+lPow]++;
         if (z[x+lPow]==zc[x+lPow])
         {
            avInfo.pos=x+lPow;
            queue->push(avInfo);
         }
      }
      else
         dissipating=TRUE;

      if(open_corner_flag)
      {
         if((remainder[n]/lPow)==0)
            z[x]++;
      }

      remainder[n+1]=remainder[n]%lPow;
   }

   if(oslo)
```

```
      zc[x]=int(ran2(&seed)*2)+zcMinRelax;
   else
   {
      if(!btw)
         zc[x]=int(ran2(&seed)*2*dimension)+zcMinRelax;
   }
   if(z[x]>=zc[x])
   {
      avInfo.pos=x;
      queue->push(avInfo);
   }
}

// Checks whether the sandpile has reached a critical state
BOOL CSandpile::check_avz(void)
{
   int zSum = 0;
   static BOOL takeData = FALSE;
   for (int i=0; i<systemSize; i++)
      zSum += z[i];
   double avz = zSum/double(systemSize);
   static double movavz_fast=0, movavz_slow=0;

   if (!takeData)
   {
      movavz_fast = (0.3 * movavz_fast) + (0.7 * avz);
      movavz_slow = (0.6 * movavz_slow) + (0.4 * avz);

      if (movavz_fast < movavz_slow)
      {
         takeData = TRUE;
         cout << "Starting calculations..." << endl;
      }
      cout << avz << "\t" << movavz_slow << "\t" << movavz_fast << endl;
      avz_out   << avz << "\t" << movavz_slow << "\t" << movavz_fast << endl;
   }
   else
      avz_out << avz << endl;

   return takeData;
}

// Calculates and stores required avalanche features
void CSandpile::calc_results()
{
   static int prf_count=0;
   static int fractal_count=0;
   int qSize = queue->sizeOfQueue();
   if(qSize>=maxAvSize)
   {
      if(dissipating)
      {
         largeav_out_d << qSize << endl;
         largeav_count_d++;
      }
      else
      {
         largeav_out_nd << qSize << endl;
         largeav_count_nd++;
      }
   }
   else
   {
      SAvInfo avInfo = queue->getValue(qSize-1);
      if(dur_flag)
         dur_data[avInfo.time]++;
      if(dissipating)
         size_data_d[qSize]++;
      else
         size_data_nd[qSize]++;
```

```
      if(rad_flag)
         rad_data[int(calc_radius() + 0.5)]++;
      if ((qSize > 1)&&(branch_flag))
      {
         calc_branch();
         if ((qSize == sizeToProfile)&&(prf_count<maxProfiles))
         {
            output_profile(avInfo);
            prf_count++;
         }
      }
   }
}

// calculates radius of gyration of avalanche
// (average r.m.s. distance of elements from avalanche c.m)
double CSandpile::calc_radius()
{
   double result;
   double rbar=0, rdiffsq=0, rsum=0, rsumsq=0;
   int qSize=queue->sizeOfQueue(), i, n;
   SAvInfo avInfo;

   for(i=0;i<qSize;i++)
   {
      avInfo=queue->getValue(i);
      remainder[0]=avInfo.pos;
      rsumsq=0;
      for(n=0; n<dimension; n++)
      {
         lPow = intpow(L, (dimension-(n+1)));
         rsumsq+=intpow((remainder[n]/lPow),2);
         remainder[n+1]=remainder[n]%lPow;
      }
      rsum=sqrt(rsumsq);
   }
   rbar=rsum/double(qSize);

   for(i=0;i<qSize;i++)
   {
      avInfo=queue->getValue(i);
      remainder[0]=avInfo.pos;

      for(n=0; n<dimension; n++)
      {
         lPow=intpow(L, (dimension-(n+1)));
         rsumsq+=intpow((remainder[n]/lPow),2);
         remainder[n+1]=remainder[n]%lPow;
      }
      rdiffsq += pow( (sqrt(rsumsq)-rbar), 2 );
   }
   result=sqrt(rdiffsq/qSize);
   return result;
}

// calculates branching ratio
void CSandpile::calc_branch()
{
   int i, bin;
   SAvInfo avInfo;
   int qSize = queue->sizeOfQueue();
// for (i=0; i<qSize; i++)
//    prf_data[i] = 0;
   avInfo = queue->getValue(qSize-1);
   int maxTime = avInfo.time;
   for (i=0; i<qSize; i++)
   {
      avInfo=queue->getValue(i);
      prf_data[avInfo.time]++;
   }
```

```
   bin = int(log(qSize)/log(2));
   for (i=1; i<=maxTime; i++)
   {
      if(dissipating)
         branch_ratio_d[bin]+=(prf_data[i]/double(prf_data[i-1]));
      else
         branch_ratio_nd[bin]+=(prf_data[i]/double(prf_data[i-1]));
      totalBranch += (prf_data[i]/double(prf_data[i-1]));
   }
   if(dissipating)
      branch_ratio_d[bin]=branch_ratio_d[bin]/double(avInfo.time);
   else
      branch_ratio_nd[bin]=branch_ratio_d[bin]/double(avInfo.time);
   totalTime += avInfo.time;
}

// outputs the avalanche profile
void CSandpile::output_profile(SAvInfo &aI)
{
   SAvInfo avInfo=aI;
   for (int i=0; i<=avInfo.time; i++)
      prf_out << i << "\t" << prf_data[i] << endl;
   prf_out << endl;
}

// combine dissipating and non-dissipating size results
void CSandpile::combine_results(int totalAv)
{
   int i=0, comb_res;
   unsigned int j;
   size_out.open("size");
   for(i=0;i<maxAvSize;i++)
   {
      comb_res = size_data_d[i]+size_data_nd[i];
      if(comb_res)
         size_out << i << "\t" << comb_res/double(totalAv) << endl;
   }

   largeav_count=largeav_count_d+largeav_count_nd;
   largeav_data = new int[largeav_count];
   ifstream largeav_in_d("largeavD", ios::in, ios::nocreate);
   ifstream largeav_in_nd("largeavND", ios::in, ios::nocreate);

   for(j=0;j<largeav_count_d;j++)
   {
      if(largeav_in_d.eof())
         break;
      largeav_in_d >> largeav_data[j];
   }
   for(j=largeav_count_d;j<(largeav_count_d+largeav_count_nd);j++)
   {
      if(largeav_in_nd.eof())
         break;
      largeav_in_nd >> largeav_data[j];
   }
   hpsort(largeav_count, largeav_data); // sort results

   int thisAvSize, prevAvSize, avNumber=1;
   if(largeav_count>0)
   {
      prevAvSize=largeav_data[0];
      for (j=1;j<largeav_count;j++)
      {
         thisAvSize=largeav_data[j];
         if(thisAvSize==prevAvSize)
            avNumber++;
         else
         {
            size_out << prevAvSize << "\t" << avNumber/double(totalAv) << endl;
            avNumber=1;
```

```
         }
         prevAvSize=thisAvSize;
      }
      size_out << thisAvSize << "\t" << avNumber/double(totalAv) << endl;
   }
   delete [] largeav_data;
   size_out.close();
}


// writes results arrays to disk
void CSandpile::output_results(int totalAv)
{
   cout << "Writing results" << endl;
   combine_results(totalAv);  // output combined results
   int i;
   unsigned int j;

   //Output dissipating avalanches
   size_out_d.open("sizeD");
   for (i=0; i<maxAvSize; i++)
      if (size_data_d[i])
         size_out_d << i << "\t" << size_data_d[i]/double(totalAv) << endl;

   ifstream largeav_in_d("largeavD", ios::in, ios::nocreate);
   largeav_data_d = new int[largeav_count_d];
   for(j=0;j<largeav_count_d;j++)
      if(!largeav_in_d.eof())
         largeav_in_d >> largeav_data_d[j];

   cout << "Sorting..." << endl;
   hpsort(largeav_count_d, largeav_data_d); // sort results
   ofstream large_sort_d("lsortD");

   for(i=0;i<int(largeav_count_d);i++)
      large_sort_d << largeav_data_d[i] << endl;

   int thisAvSize, prevAvSize, avNumber=1;
   if(largeav_count_d>0)
   {
      prevAvSize=largeav_data_d[0];
      for (i=1;(unsigned)i<largeav_count_d;i++)
      {
         thisAvSize=largeav_data_d[i];
         if(thisAvSize==prevAvSize)
            avNumber++;
         else
         {
            size_out_d << prevAvSize << "\t" << avNumber/double(totalAv) << endl;
            avNumber=1;
         }
         prevAvSize=thisAvSize;
      }
      size_out_d << thisAvSize << "\t" << avNumber/double(totalAv) << endl;
   }
   delete [] largeav_data_d;
   size_out_d.close();

   // Output non dissipating avalanches
   size_out_nd.open("sizeND");
   for (i=0; i<maxAvSize; i++)
      if(size_data_nd[i])
         size_out_nd << i << "\t" << size_data_nd[i]/double(totalAv) << endl;

   ifstream largeav_in_nd("largeavND", ios::in, ios::nocreate);
   largeav_data_nd = new int[largeav_count_nd];
   for(j=0;j<largeav_count_nd;j++)
      if(!largeav_in_nd.eof())
         largeav_in_nd >> largeav_data_nd[j];
   hpsort(largeav_count_nd, largeav_data_nd); // sort results
```

```
   ofstream large_sort_nd("lsortND");
   for(i=0;i<int(largeav_count_nd);i++)
      large_sort_nd << largeav_data_nd[i] << endl;

   if(largeav_count_nd>0)
   {
      prevAvSize=largeav_data_nd[0];
      for (i=1;(unsigned)i<largeav_count_nd;i++)
      {
         thisAvSize=largeav_data_nd[i];
         if(thisAvSize==prevAvSize)
            avNumber++;
         else
         {
            size_out_nd << prevAvSize << "\t" << avNumber/double(totalAv) << endl;
            avNumber=1;
         }
         prevAvSize=thisAvSize;
      }
      size_out_nd << thisAvSize << "\t" << avNumber/double(totalAv) << endl;
   }
   delete [] largeav_data_nd;
   size_out_nd.close();

   if(dur_flag)      // output duration
   {
      dur_out.open("duration");
      for (i=0; i<maxAvSize; i++)
      {
         if (dur_data[i])
         dur_out << i+1 << "\t" << dur_data[i] << endl;
      }
      dur_out.close();
   }

   if(rad_flag)      // output radius
   {
      rad_out.open("radius");
      for (i=0; i<maxRad; i++)
         if (rad_data[i])
            rad_out << i << "\t" << rad_data[i] << endl;
      rad_out.close();
   }
   if(branch_flag)
   {
      prf_out.close();
      prf_out.open("profiles");
      for (int i=0; i<=maxAvSize; i++)
      {
         if(prf_data[i]!=0)
            prf_out << i << "\t" << prf_data[i] << endl;
      }
   }

//Writes sandpile configuration to pile_out
//WARNING: can produce a VERY large file
// for(i=0;i<L*L;i++)
// {
//    if(z[i]==3)  // output location of critical sites only
//       pile_out << i%L << "\t" << (i/L) << endl;
// }


   if(totalAv==maxAv)
   {
      endTime = (unsigned)time(NULL);

      if(branch_flag&&(totalTime>0))
      {
         cout << "Average branching ratio = " << totalBranch / totalTime << endl;
```

```
         misc_out << "Average branching ratio = " << totalBranch / totalTime << endl;
      }
      cout << "Calculations completed in " << endTime-startTime
          << " seconds for " << totalAv << " avalanches" << endl;
      misc_out << "Calculations completed in " << endTime-startTime
             << " seconds for " << totalAv << " avalanches" << endl;
   }
}


// quick integer power calculator x^y
int CSandpile::intpow(int x, int y)
{
   int result=1;
   for (int i=0;i<y;i++)
      result*=x;
   return result;
}

// adapted from Numerical Recipes in C
// heapsort routine pg 337
void CSandpile::hpsort(unsigned long n, int ra[])
// n is the size of ra[]
{
   unsigned long i,ir,j,l;
   int rra;

   if (n < 2) return;
   l=(n >> 1);
   ir=n-1;
   for(;;)
   {
      if (l >0)
         rra=ra[--l];
      else
      {
         rra=ra[ir];
         ra[ir]=ra[0];
         if (--ir ==0)
         {
            ra[0]=rra;
            break;
         }
      }
      i=l;
      j=l+1;
      while (j<=ir)
      {
         if (j<ir && ra[j]<ra[j+1])
            j++;
         if (rra < ra[j])
         {
            ra[i]=ra[j];
            i=j;
            j <<=1;
         }
         else
            break;
      }
      ra[i]=rra;
   }
}



#ifndef __QUEUE_HEADER
#define __QUEUE_HEADER

#include "Globals.h"
```

```
class CQueue
{
private:
   int size;
   int front, back, wrap;
   SAvInfo *queue;
public:
   CQueue();
   CQueue(int s);
   ~CQueue();

   void push(const SAvInfo &x);
   SAvInfo pop();

   BOOL isEmpty();
   void clear();
   int sizeOfQueue();

   SAvInfo& getValue(int n);
};


#endif // __QUEUE_HEADER

#include "Queue.h"

// constructor
CQueue::CQueue(int s)
{
   size = s;
   front = 0;
   back = 0;
   wrap = 0;
   queue = new SAvInfo[size];
}

// destructor
CQueue::~CQueue()
{
   if (queue)
   {
      delete queue;
      queue = NULL;
   }
}

// pushes information on to end of queue
// and updates position of queue end
void CQueue::push(const SAvInfo &x)
{
#ifdef _DEBUG
   if (back == front-1)
   {
      cerr << "Trying to push onto full queue" << '\n';
      int i;
      cin >> i;
      exit(0);
   }
#endif
   queue[back] = x;
   if (back==(size-1))
   {
      back=0;
      wrap++;
   }
   else
      back++;

}
```

```
// return information at front of queue
// and advance front of queue
SAvInfo CQueue::pop()
{
#ifdef _DEBUG
   if (front == back)
   {
      cerr << "Trying to pop off empty queue" << '\n';
      int i;
      cin >> i;
      exit(0);
   }
#endif

   SAvInfo result = queue[front];
   if (front==(size-1))
      front=0;
   else
      front++;
   return result;
}

//check to see if there are any elements in the queue
BOOL CQueue::isEmpty()
{
   if (front == back)
      return TRUE;
   return FALSE;
}

// reset queue
void CQueue::clear()
{
   front = 0;
   back = 0;
   wrap = 0;
}

// return number of elements in queue
int CQueue::sizeOfQueue()
{
   int qsize = back + (size*wrap);
   return qsize;
}

// get value at a given location in queue
SAvInfo& CQueue::getValue(int n)
{
   SAvInfo& result = queue[n];
   return result;
}
```

C Logarithmic Binning Code

```
#include <stdlib.h>
#include <fstream.h>
#include <iostream.h>
#include <stdio.h>
#include <math.h>
#include <string.h>

int main ()
{
   const long avnum=5;
   int sign, decptr=1, ocount;

   long s, oldpos, endpos, j;
   double Ns, base, offset, bin, newpos, binpos, binval;
   double store_binpos, store_binval, slope_val;
   char infilename[20], binoutname[20], slopeoutname[20], histoutname[20];
   double *hist_data, *hist_data_weight;
   double hist_binsize;

   ifstream in;
   ofstream bin_out;
   ofstream slope_out;
   ofstream hist_out;

   cout << "File to be processed ?" << '\n';
   cin >> infilename;
   cout << "Enter histogram bin size" << '\n';
   cin >> hist_binsize;
   cout << "Working..." << '\n';

   hist_data = new double[int(3/hist_binsize)];
   hist_data_weight = new double[int(3/hist_binsize)];
   for(j=0;j<int(3/hist_binsize);j++)
   {
      hist_data[j]=0;
      hist_data_weight[j]=0;
   }

   for(j=1;j<10;j++)
   {
      base=1+0.1*j;
      cout << base << endl;
      strcpy (binoutname,"bin");
      strcat(binoutname,ecvt(base,2,&decptr,&sign));
      strcat(binoutname,"_");
      strcat(binoutname,infilename);

      strcpy (slopeoutname,"slope");
      strcat(slopeoutname,ecvt(base,2,&decptr,&sign));
      strcat(slopeoutname,"_");
      strcat(slopeoutname,infilename);

      bin_out.open(binoutname);

      slope_out.open(slopeoutname);
```

```
   for(ocount=0;ocount<avnum;ocount++)
   {
      offset=ocount*(base*base-base)/avnum;
      in.open(infilename);

      oldpos=0;
      newpos=1.0+offset;
      store_binpos=0.0;

      // read first set of data
      in >> s >> Ns;

      do
      {
         bin=0;
         while ((s<=long(newpos))&&(!in.eof()))
         {
            if (!in.eof())
               bin+=Ns;
            in >> s >> Ns;
         }
         if(bin>0.0)
         {
            // calculate appropriate range of s
            if (!in.eof())
               endpos=long(newpos);
            else
               endpos=s;

            binpos=sqrt( ((oldpos==0)?1:oldpos) * double(endpos) );
            binval=bin/(endpos-double(oldpos));
            if(store_binpos!=0)
               slope_val =log(binval/store_binval)/log(binpos/store_binpos);
            if((slope_val<0)&&(slope_val>-3))
            {
               hist_data[abs(int(slope_val/hist_binsize))]++;
               hist_data_weight[abs(int(slope_val/hist_binsize))]+=log10(base);
            }

            // write binned data to file using geometric mean as binposition
            // condition prevents writing empty last bin
            if ((!in.eof())||(bin>0))
               bin_out << binpos << "\t" << binval << '\n';
            if (store_binpos>0)
            {
               slope_out << sqrt(binpos*store_binpos) << "\t"
                  << slope_val << "\t" << bin << '\n';
            }

            oldpos=long(newpos);
         }

         store_binpos=binpos;
         store_binval=binval;

         // find next newpos
         // do-loop to avoid zero-size bins
         do
         {
            newpos*=base;
         }while((long(newpos)-oldpos)==0);

      }while (!in.eof());
      in.close();
   }
   bin_out.close();
   slope_out.close();
}

strcpy (histoutname,"hist_");
```

```
   strcat(binoutname,ecvt(hist_binsize,5,&decptr,&sign));
   strcat(histoutname,infilename);

   hist_out.open(histoutname);
   hist_out << "Bin \t weighted value \t frequency \n";
   for(j=0;j<int(3/hist_binsize);j++)
   {
      if(hist_data[j]!=0)
         hist_out << -((hist_binsize*j)+(hist_binsize/2))
             << "\t" << hist_data_weight[j]
             << "\t" << hist_data[j] << '\n';

   }
   delete [] hist_data;
   delete [] hist_data_weight;
   hist_out.close();

   cout << "Done!" << endl;
   return 0;
}
```

D Modified t_n after inclusion of ρ'

$$
\begin{aligned}
t_0 &= (1-\rho) \\
t_1 &= \rho[x + (n_1+n_2)(1-\rho) + b((1-\rho)^2] \\
t_2 &= \rho[\{(n_1+n_2)\rho + 2b\rho(1-\rho)\} \\
&\quad \cdot\{n_1 + n_2(1-\rho) + b(1-\rho) + x\}] \\
t_3 &= \rho[\{(n_1+n_2)\rho \\
&\quad \cdot\{(n_2+b)\rho\} \\
&\quad \cdot\{n_1 + n_2(1-\rho) + b(1-\rho) + x\} \\
&\quad +\{2b\rho(1-\rho)\} \\
&\quad \cdot\{(n_2+b)\rho\} \\
&\quad \cdot\{n_1 + n_2 + b + x\} \\
&\quad +\{b\rho^2\} \\
&\quad \cdot\{(n_1+x)^2 + (n_1+x)(n_2+b)(1-\rho) + (n_2+b)^2(1-\rho')\}] \\
t_4 &= \rho[\{(n_1+n_2)\rho\} \\
&\quad \cdot\{(n_2+b)\rho\} \\
&\quad \cdot\{(n_2+b)\rho\} \\
&\quad \cdot\{n_1 + n_2 + b + x\} \\
&\quad +\{b\rho^2\} \\
&\quad \cdot\{(n_1+x)(n_2+b)\rho + (n_2+b)^2\rho'\} \\
&\quad \cdot\{n_1 + n_2 + b + x\}]
\end{aligned}
$$

E Mapping for approximative DDRG

No. of processes	No. of grains leaving	Processes with $c = 0$	Processes with $c = 1$	Processes with $c = 2$	**m**
2	0	2	0	0	$\{1, 0, 0, 0, 0\}$
2	1	1	1	0	$\{0, 1, 0, 0, 0\}$
2	2	1	0	1	$\{0, 0, 1, 0, 0\}$
2	2	0	2	0	$\{0, \frac{1}{4}, \frac{3}{4}, 0, 0\}$
2	3	0	1	1	$\{0, 0, \frac{1}{2}, \frac{1}{2}, 0\}$
2	4	0	0	2	$\{0, 0, 0, 1, 0\}$
3	0	3	0	0	$\{1, 0, 0, 0, 0\}$
3	1	2	1	0	$\{0, 1, 0, 0, 0\}$
3	2	2	0	1	$\{0, 0, 1, 0, 0\}$
3	2	1	2	0	$\{0, \frac{1}{6}, \frac{5}{6}, 0, 0\}$
3	3	1	1	1	$\{0, 0, \frac{1}{3}, \frac{2}{3}, 0\}$
3	3	0	3	0	$\{0, 0, \frac{1}{2}, \frac{1}{2}, 0\}$
3	4	1	0	2	$\{0, 0, 0, \frac{2}{3}, \frac{1}{3}\}$
3	4	0	2	1	$\{0, 0, \frac{1}{12}, \frac{2}{3}, \frac{1}{4}\}$
3	5	0	1	2	$\{0, 0, 0, \frac{1}{3}, \frac{2}{3}\}$
3	6	0	0	3	$\{0, 0, 0, 0, 1\}$
4	0	4	0	0	$\{1, 0, 0, 0, 0\}$
4	1	3	1	0	$\{0, 1, 0, 0, 0\}$
4	2	3	0	1	$\{0, 0, 1, 0, 0\}$
4	2	2	2	0	$\{0, \frac{1}{6}, \frac{5}{6}, 0, 0\}$
4	3	2	1	1	$\{0, 0, \frac{1}{3}, \frac{2}{3}, 0\}$
4	3	1	3	0	$\{0, 0, \frac{1}{2}, \frac{1}{2}, 0\}$
4	4	2	0	2	$\{0, 0, 0, \frac{2}{3}, \frac{1}{3}\}$
4	4	1	2	1	$\{0, 0, \frac{1}{12}, \frac{2}{3}, \frac{1}{4}\}$
4	4	0	4	0	$\{0, 0, \frac{1}{8}, \frac{3}{4}, \frac{1}{8}\}$
4	5	1	1	2	$\{0, 0, 0, \frac{1}{3}, \frac{2}{3}\}$
4	5	0	3	1	$\{0, 0, 0, \frac{1}{2}, \frac{1}{2}\}$
4	6	1	0	3	$\{0, 0, 0, 0, 1\}$
4	6	0	2	2	$\{0, 0, 0, \frac{1}{6}, \frac{5}{6}\}$
4	7	0	1	3	$\{0, 0, 0, 0, 1\}$
4	8	0	0	4	$\{0, 0, 0, 0, 1\}$

F Iteration of approximative DDRG

k	ρ	p_0	p_1	p_2	p_3	p_4
0	0.25	0	0	0	0	1
1	0.297872	0	0	0	0.642857	0.357143
2	0.337321	0	0.017573	0.213582	0.555577	0.213267
3	0.368667	0.001402	0.060171	0.315661	0.470082	0.152683
4	0.393740	0.005182	0.100835	0.363134	0.410753	0.120095
5	0.413857	0.010039	0.133864	0.386196	0.369567	0.100333
6	0.430025	0.015024	0.159587	0.397738	0.340217	0.087433
7	0.443031	0.019658	0.179436	0.403554	0.318775	0.078577
8	0.453501	0.023748	0.194771	0.406421	0.302788	0.072272
9	0.461935	0.027254	0.206674	0.407738	0.290675	0.067658
10	0.468729	0.030205	0.215965	0.408240	0.281378	0.064212
11	0.474205	0.032659	0.223256	0.408318	0.274171	0.061596
12	0.478620	0.034683	0.229003	0.408189	0.268539	0.059586
13	0.482178	0.036344	0.233553	0.407967	0.264110	0.058026
14	0.485048	0.037701	0.237167	0.407718	0.260610	0.056805
15	0.487362	0.038806	0.240046	0.407472	0.257832	0.055845
16	0.489228	0.039705	0.242344	0.407246	0.255620	0.055085
17	0.490733	0.040434	0.244183	0.407046	0.253855	0.054482
18	0.491947	0.041025	0.245657	0.406873	0.252442	0.054002
19	0.492926	0.041504	0.246839	0.406727	0.251311	0.053618
20	0.493716	0.041891	0.247789	0.406605	0.250403	0.053312
21	0.494353	0.042204	0.248552	0.406504	0.249675	0.053066
22	0.494867	0.042457	0.249166	0.406420	0.249089	0.052868
23	0.495282	0.042662	0.249660	0.406351	0.248617	0.052710
24	0.495616	0.042827	0.250058	0.406295	0.248238	0.052583
25	0.495886	0.042960	0.250378	0.406249	0.247933	0.052480
26	0.496103	0.043068	0.250637	0.406211	0.247687	0.052398
27	0.496279	0.043155	0.250845	0.406181	0.247488	0.052331
28	0.496420	0.043225	0.251012	0.406156	0.247329	0.052278
29	0.496535	0.043282	0.251148	0.406137	0.247200	0.052235
30	0.496627	0.043327	0.251257	0.406120	0.247096	0.052200
31	0.496701	0.043364	0.251345	0.406107	0.247012	0.052172
32	0.496761	0.043394	0.251416	0.406097	0.246945	0.052149
33	0.496810	0.043418	0.251473	0.406088	0.246890	0.052131
34	0.496849	0.043437	0.251519	0.406081	0.246846	0.052116
35	0.496880	0.043453	0.251556	0.406076	0.246811	0.052105
36	0.496905	0.043465	0.251586	0.406071	0.246782	0.052095
37	0.496926	0.043476	0.251610	0.406068	0.246759	0.052087
38	0.496942	0.043484	0.251630	0.406065	0.246741	0.052081
39	0.496956	0.043490	0.251646	0.406062	0.246726	0.052076
40	0.496967	0.043496	0.251658	0.406060	0.246714	0.052072
41	0.496975	0.043500	0.251668	0.406059	0.246704	0.052069
42	0.496982	0.043504	0.251677	0.406058	0.246696	0.052066
43	0.496988	0.043506	0.251683	0.406057	0.246690	0.052064
44	0.496992	0.043509	0.251689	0.406056	0.246685	0.052062
45	0.496996	0.043510	0.251693	0.406055	0.246681	0.052061
46	0.496999	0.043512	0.251697	0.406055	0.246677	0.052060
47	0.497001	0.043513	0.251699	0.406054	0.246674	0.052059
48	0.497003	0.043514	0.251702	0.406054	0.246672	0.052058
49	0.497005	0.043515	0.251704	0.406054	0.246671	0.052058
50	0.497006	0.043515	0.251705	0.406053	0.246669	0.052057
51	0.497007	0.043516	0.251706	0.406053	0.246668	0.052057
52	0.497008	0.043516	0.251707	0.406053	0.246667	0.052056
53	0.497009	0.043517	0.251708	0.406053	0.246666	0.052056
54	0.497009	0.043517	0.251709	0.406053	0.246666	0.052056
55	0.497010	0.043517	0.251709	0.406053	0.246665	0.052056
56	0.497010	0.043517	0.251710	0.406053	0.246665	0.052056
57	0.497010	0.043518	0.251710	0.406053	0.246665	0.052056
58	0.497010	0.043518	0.251710	0.406053	0.246664	0.052055
59	0.497011	0.043518	0.251710	0.406053	0.246664	0.052055
60	0.497011	0.043518	0.251710	0.406052	0.246664	0.052055
61	0.497011	0.043518	0.251711	0.406052	0.246664	0.052055
62	0.497011	0.043518	0.251711	0.406052	0.246664	0.052055
63	0.497011	0.043518	0.251711	0.406052	0.246664	0.052055
64	0.497011	0.043518	0.251711	0.406052	0.246664	0.052055
65	0.497011	0.043518	0.251711	0.406052	0.246664	0.052055
66	0.497011	0.043518	0.251711	0.406052	0.246663	0.052055
∞	0.497011	0.043518	0.251711	0.406052	0.246663	0.052055