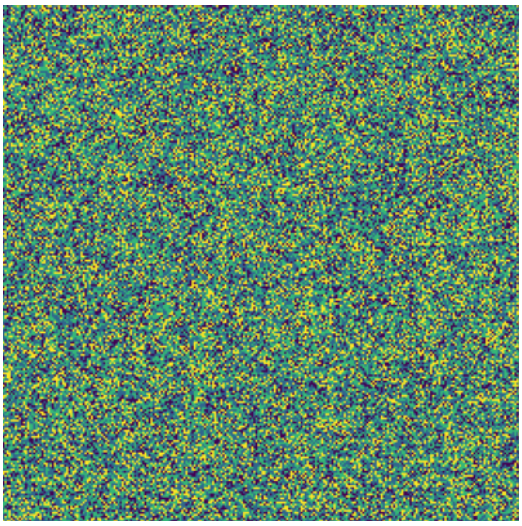
THE UNIVERSITY OF
MELBOURNE

Final Report

The Art of Scientific Computing: BTW model extension

Author:
Songxiang TANG

Subject Handbook code:
COMP-90072



Faculty of Science
The University of Melbourne

10st June 2025

Subject co-ordinators: A/Prof. Roger Rassool & Prof. Harry Quiney

Executive Summary

Objective
We examine how a steady, unidirectional "wind" with strength σ alters the hallmark avalanche of the Bak–Tang–Wiesenfeld (BTW) sand-pile, the canonical model of self-organised criticality (SOC).

Method
A coarse-grained drift–diffusion equation was derived in which the bias appears as an advective term, providing analytic expectations for exponent shifts. The lattice dynamics were recast as a mass-conserving PyTorch kernel that achieves more than 3×10^3 grain drops per second on a Tesla T4 GPU. We ran 2×10^5-drop experiments for $\sigma = 0, 1, 2, 3, 5$ on lattices up to 256^2, fitted power-law tails with maximum likelihood, validated fits by Monte-Carlo Kolmogorov–Smirnov tests, and computed temporal as well as spatial correlations. All results stream in real time to an interactive GUI.

Findings
Avalanche sizes and durations remain power-law distributed at every bias tested, confirming that SOC is robust. Yet the distribution is tuned: the size exponent declines smoothly from $\hat{\tau}_s = 1.34$ at $\sigma = 0$ to $\hat{\tau}_s = 1.31$ at $\sigma = 5$, a trend that one-way ANOVA flags as highly significant ($p < 10^{-4}$). Spatial correlation length collapses from $\xi \approx 2$ lattice spacings to $\xi < 0.6$ once any bias is present, whereas finite-size scaling still holds with a single fractal dimension $D = 1.0$ and temporal correlations display a bias-independent memory plateau extending over 10^4 drops. Directional forcing is therefore a relevant perturbation: it leaves SOC intact but continuously reshapes heavy tails and truncates spatial coherence, mirroring the behavior of wind-driven wildfires and other anisotropic hazards.

Outlook
Extending the GPU engine to three dimensions, adding stochastic drift, and fitting observed landslide or wildfire catalogs to infer effective σ will connect the present theoretical insights to practical, data-driven risk assessment.

Contents

Chapter 1

Introduction

A complex system is defined as a collection of various interacting components whose aggregate behavior cannot be predicted simply by understanding the parts in isolation [1, 12]. From earthquakes and neuronal networks to forest fires and financial markets [14, 5, 8, 9], such systems spontaneously organize into a critical state in which events of all sizes occur according to power-law statistics [3, 2].

The seminal Bak–Tang–Wiesenfeld (BTW) sandpile model captures SOC in its simplest form: grains are added one at a time to a lattice and "topple" whenever a local threshold is exceeded, redistributing particles to neighbors. In its classical, unbiased version, avalanches exhibit diffusive growth (spatial extent $L \propto T^{1/2}$) and power-law size and duration distributions[3]. More recent fractal-diffusive extensions have shown that these scaling laws can be derived analytically by approximating the discrete toppling rules with a continuum advection–diffusion (Fokker–Planck) equation [4]. However, many real-world systems are anisotropic, as external forces or flows impose a directional bias that can modify both the morphology of individual events and the global critical exponents. For instance, in wildfire propagation, prevailing winds skew the spread of flames and alter cluster shapes. This observation motivates two central questions:

1. Does the introduction of a "wind" bias—i.e. the skewing of topple probabilities toward a preferred direction—preserve the hallmark scale invariance of the BTW model?

2. How does the strength of such a bias influence the emergent dynamics, and can these effects be captured within a universal scaling framework?

In **Chapter 2**, we addressed these questions analytically. We derive the modified coarse-grained equation in the presence of directional bias and demonstrate that the avalanche-size distribution remains a power law, with exponent τ that increases systematically with the bias strength. In **Chapter 3**, we performed large-scale GPU-accelerated simulations for both four-neighbor and eight-neighbor toppling rules, fit avalanche-size data to power laws using maximum-likelihood estimation, assess goodness-of-fit via Kolmogorov–Smirnov (KS) tests, and perform an analysis of variance (ANOVA) on the ensemble of fitted exponents τ to quantify its dependence on the wind bias, thereby validating our theoretical prediction. In **Chapter 4**, we present temporal and spatial autocorrelation analyses of avalanche sequences and cluster morphologies to characterize memory effects and extract correlation lengths modified by bias. In **Chapter 5,** we introduce an interactive Python application featuring a live lattice heat map and dynamically updated metrics panel for intuitive visualization of how varying bias σ sculpts avalanche geometry. Finally, in **Chapter 6**, we summarize our findings, discuss their implications for SOC in driven dissipative systems, and propose directions for future work, including the incorporation of stochastic forcing and higher-dimensional lattices.

Chapter 2

Mathematical Reasoning

Derivation of the Advection–Diffusion Equation

Recent research has shown that the discrete toppling rule can be approximated by a continuum advection–diffusion equation[4]. In this chapter, we derive the corresponding second-order partial differential form and thereby recover the continuum advection–diffusion description (ADE). The detailed derivation steps are as follows:

1. Discrete Master Equation

In a two-dimensional lattice of size $L \times L$,

$$\mathbf{H} = \begin{bmatrix} h_{1,1} & h_{1,2} & \cdots & h_{1,L} \\ h_{2,1} & h_{2,2} & \cdots & h_{2,L} \\ \vdots & \vdots & \ddots & \vdots \\ h_{L,1} & h_{L,2} & \cdots & h_{L,L} \end{bmatrix} \quad \text{where } h_{i,j} \text{ is the sand height at site } (i,j).$$

let $\rho_{i,j}^t$ denote the sand density at site (i,j) after t updates, where ρ^t is the instance of H matrix in time step t. Incorporating a bias δ in the $+x$ direction, the transition probabilities to the four neighbors are

$$p_{+x} = \frac{1+\delta}{4+\delta}, \qquad p_{-x} = \frac{1}{4+\delta},$$
$$p_{+y} = \frac{1}{4+\delta}, \qquad p_{-y} = \frac{1}{4+\delta},$$

with $\sum_k p_k = 1$. Denote the neighbor offsets by

$$(\delta x_k, \delta y_k) \in \{(+1,0), (-1,0), (0,+1), (0,-1)\}.$$

According to classical BTW model, each time

$$\rho_{i,j}^{t+1} = \rho_{i,j}^t + 1 - 4\,T_{i,j}^t + \sum_{k=1}^{4} p_k\, T_{i+\delta x_k,\, j+\delta y_k}^t, \quad T_{i,j}^t = \begin{cases} 1, & \rho_{i,j}^t \geq 4, \\ 0, & \rho_{i,j}^t < 4, \end{cases}$$

Here we adopt a mean–field closure in which the binary toppling indicator is approximated by a linear function of the local height, $T_{i,j}^t \simeq \rho_{i,j}^t/\rho_c$ with $\rho_c = 4$; inserting this relation into the exact lattice update yields the coarse-grained master equation

$$\rho_{i,j}^{t+1} - \rho_{i,j}^t = \sum_{k=1}^{4} p_k\, \rho_{i+\delta x_k, j+\delta y_k}^t - \rho_{i,j}^t + S_{i,j}^t, \tag{1.1}$$

where $S_{i,j}^t$ represents any external input or removal of sand at the lattice site $h_{i,j}$. When we average $S_{i,j}^t$ over every site in the $L \times L$ grid, we will obtain the constant background rate S_0, which captures the spatially averaged driving strength in the continuum equation [2, **?**].

2. Time Difference Approximation

With time step $\Delta t = 1$, the forward difference approximates the time derivative:

$$\frac{\rho_{i,j}(t+\Delta t) - \rho_{i,j}(t)}{\Delta t} \;\approx\; \left.\frac{\partial \rho(x,y,t)}{\partial t}\right|_{x=i,\, y=j,\, t} \tag{2.2}$$

3. Taylor expansion for approximating second-order partial derivatives

For each neighbor term, perform a second-order Taylor expansion (neglecting mixed derivatives) around $(x = i, y = j)$:

$$\rho^t_{i+\delta x_k, j+\delta y_k} \approx \rho + \delta x_k\, \partial_x \rho + \delta y_k\, \partial_y \rho + \tfrac{1}{2}\, \delta x_k^2\, \partial_x^2 \rho + \tfrac{1}{2}\, \delta y_k^2\, \partial_y^2 \rho + O(h^3), \tag{3.1}$$

where the lattice spacing $h = 1$. Substituting into the redistribution term in (1.1) gives:

$$\sum_{k=1}^{4} p_k\, \rho^t_{i+\delta x_k, j+\delta y_k} \approx \rho + \Big(\sum_k p_k\, \delta x_k\Big)\partial_x \rho + \Big(\sum_k p_k\, \delta y_k\Big)\partial_y \rho + \Big(\tfrac{1}{2}\sum_k p_k\, \delta x_k^2\Big)\, \partial_x^2 \rho + \Big(\tfrac{1}{2}\sum_k p_k\, \delta y_k^2\Big)\, \partial_y^2 \rho \tag{3.2}$$

Define the effective advection velocity and diffusion coefficients:

$$v_x = \sum_{k=1}^{4} p_k\, \delta x_k, \quad v_y = \sum_{k=1}^{4} p_k\, \delta y_k, \quad D_x = \tfrac{1}{2}\sum_{k=1}^{4} p_k\, \delta x_k^2, \quad D_y = \tfrac{1}{2}\sum_{k=1}^{4} p_k\, \delta y_k^2. \tag{3.3}$$

By symmetry $v_y = 0$, and the redistribution term becomes

$$\sum_{k=1}^{4} p_k\, \rho^t_{i+\delta x_k, j+\delta y_k} - \rho \approx v_x\, \partial_x \rho + D_x\, \partial_x^2 \rho + D_y\, \partial_y^2 \rho. \tag{3.4}$$

4. Continuous Limit and Final ADE

Combining the time difference (2.2), the Taylor-expanded neighbor term (3.1), and $S^t_{i,j} \approx S_0$, dividing by $\Delta t = 1$, and effective advection velocity and diffusion coefficients yields

$$\partial_t \rho = v_x\, \partial_x \rho + D_x\, \partial_x^2 \rho + D_y\, \partial_y^2 \rho + S_0. \tag{4.1}$$

Finally, let $D = \frac{D_x + D_y}{2}$ and the Laplacian $\nabla^2 = \partial_x^2 + \partial_y^2$ gives the advection–diffusion equation:

$$\boxed{\frac{\partial \rho}{\partial t} + v_x \frac{\partial \rho}{\partial x} = D\, \nabla^2 \rho + S_0.} \tag{4.2}$$

Conclusion

From the coarse-graining of the biased sandpile dynamics, we obtain the macroscopic advection–diffusion equation with drift velocity$v_x = \frac{\delta}{4+\delta}$, diffusion coefficient D, and source rate S_0. Moreover, in the self-organized critical regime, the avalanche size distribution

$$P(s) = \Pr\{\text{avalanche size} = s\}$$

obeys a power-law,

$$P(s) \propto s^{-\tau},$$

indicating a heavy-tailed, scale-free behavior without a characteristic scale. This avalanche function is thus of *power-law* type. In the next chapter, we will simulate the model and empirically verify this scaling law.

Chapter 3

Statistical Experiment

Based on the mathematical framework of Chapter 1, we seek to determine whether introducing a unidirectional "wind bias" σ preserves the classical BTW power-law scaling. To do so, we first perform large-scale, GPU-accelerated simulations on an 256×256 lattice (using Colab's CUDA-enabled PyTorch) for several values of σ, recording avalanche size, duration, area and length after allowing a burn-in period. Next, for each σ and each observable, we fit the positive tail to a discrete power law by estimating maximum likelihood and assess goodness of fit with a Monte Carlo–based Kolmogorov–Smirnov test. Finally, after extracting the fitted exponents $\alpha(\sigma)$, we use one-way ANOVA to evaluate whether and how α varies significantly with the wind bias parameter.

Simulation Setup

We perform large–scale simulations of the biased BTW sand pile model on a two–dimensional lattice to gather avalanche statistics. Previous the study that estimates $\approx 1.39 \times 10^5$ grain drops to reach SOC on a 256×256 lattice[11].The key parameters and procedure are:

- **Grid size:** $L \times L$ with $L = 256$.
- **Wind bias:** $\sigma \in \{0, 1.0, 2.0, 3.0, 5.0\}$ controls the probability bias in the $+x$ direction.
- **Number of drops:** $N_{\text{drops}} = 2 \times 10^5$.
- **Burn–in:** The first 1.4×10^5 drops are discarded to reach a statistical steady state.
- **Recorded observables:**
  1. Avalanche *size* s: total grains toppled.
  2. Avalanche *duration* **dur**: number of relaxation steps.
  3. Avalanche *area* A: number of distinct toppled sites.
  4. Avalanche *length* ℓ: maximum Manhattan distance from the drop site.
- **Drop rule:** Each grain is added to a uniformly random site (i, j). Sites with height ≥ 4 topple, distributing 4 grains to their 4 nearest neighbors with probabilities

$$p_{+x} = \frac{1+\sigma}{4+\sigma}, \quad p_{-x} = p_{+y} = p_{-y} = \frac{1}{4+\sigma}.$$

3.1 Innovation on GPU–Accelerated BTW Sandpile Simulations with Conservative Wind Bias

To perform large-scale, wind-biased BTW simulations on Google COLAB's GPU, we represent the $L \times L$ height field as a $1 \times 1 \times L \times L$ `float32` tensor in PyTorch and exploit a single convolution per topple to enforce exact mass conservation and boundary loss. At each drop:

1. **Definition of the sand pile grid** We maintain

$$\mathit{grid} \in \mathbb{R}^{1\times1\times L\times L}, \quad \mathit{remainder} \in \mathbb{R}^{1\times1\times L\times L},$$

where *remainder* accumulates the fractional parts of biased redistribution so that rounding never drifts total sand. By accumulating these residuals, we ensure that any rounding of the biased flux does not lead to a net gain or loss of sand, and thus the total mass of the system remains strictly conserved. Without this mechanism, applying the bias parameter σ directly to the grid with integer values at each toppling would effectively introduce fractional amounts of sand into the lattice. In other words, each toppling would act as an unintentional source of material, violating the fundamental conservation law of the sand pile model.

2. **Construct a single biased kernel K^*.** To bias transport toward $+x$ without creating or destroying sand, we choose weights
$$w_{\text{base}} = \frac{4}{4+\sigma}, \quad w_{\text{right}} = \frac{4\,\sigma}{4+\sigma},$$
and define a 3×3 kernel K^* as:
$$K^* = \begin{pmatrix} 0 & w_{\text{base}} & 0 \\ w_{\text{base}} & 0 & w_{\text{right}} \\ 0 & w_{\text{base}} & 0 \end{pmatrix}, \quad \sum_{i,j} K^*_{ij} = 4.$$

3. A single conv2d$(T, K^*, \text{padding} = 1)$ then redistributes exactly $4\,T$ grains, fractionally biased to $+x$.
4. **Random drop.** Pick $(i, j) \sim \text{Uniform}\{0, \ldots, L-1\}^2$ and do $grid[0, 0, i, j]$ += 1.
5. **Parallel relaxation loop using while loop in Python.** While $\exists\, grid_{t,ij} \geq 4$, we perform the toppling by following step:
   (a) Compute the *Topple Count* Matrix for each sand pile location that need perform a toppling:
   $$T = \lfloor grid/4 \rfloor$$
   (b) Subtract $4\,T$ from *grid* matrix to remove all sands of a location that exceed hight of threshold 4.
   (c) By performing a single convolution, we could get a wind bias difference matrix Δ
   $$\Delta = \text{conv2d}\big(T,\, K^*,\, \text{padding} = 0\big),$$
   then accumulate
   $$remainder \mathrel{+}= \Delta, \quad D = \lfloor remainder \rfloor, \quad remainder \mathrel{-}= D,$$
   by employing zero-padding at the lattice boundaries, any topple flux Δ that propagates beyond the edge is automatically discarded, representing natural mass loss, while the use of exact floor-rounding in the redistribution step guarantees that no net sand is created within the system. However, this scheme cannot explicitly record or quantify the boundary losses, since any material falling outside the padded domain is irreversibly removed.
   (d) Mark all newly toppled sites and accumulate avalanche statistics (size, duration, area, loss, length).
6. **Burn-in and data collection.** Discard the first `burn_in` drops to allow the system to self-organize, then record the subsequent avalanche metrics for power-law fitting and K–S testing. The metrics recorded includes Avalanche Size:s_σ, Avalanche Duration: dur_σ, Avalanche Area: A_σ and length represented by Manhattan distance: ℓ_σ.

In conclusion, putting it all together, each update step conserves total sand exactly (modulo boundary loss) and applies the wind bias without introducing spurious mass:
$$grid_{t+1} = grid_t - 4\,T + D, \quad D = \big\lfloor \text{conv2d}(T, K^*, 1) \big\rfloor,$$
where $\sum_{ij} D_{ij} = \sum_{ij} 4\,T_{ij}$, guaranteeing both convergence of the relaxation and faithful wind-biased transport. Below, we provide an intuitive example illustrating the computation of a toppling event using the matrix operations defined above.

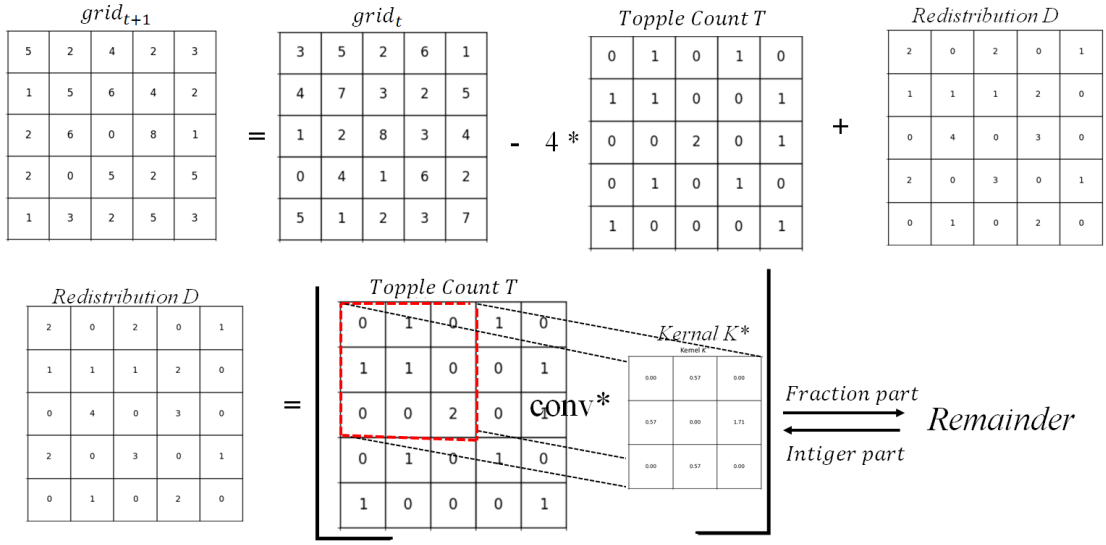


Figure 3.1: This figure demonstrates the parallel "toppling" operation process we performed in COLAB. *Figure 1.* Starting from the height matrix $grid_t$, any cell with $grid_{t,ij} \geq 4$ topples. We compute $T_{ij} =$ so that each topple removes exactly four grains (hence the $-4\,T$ term). Convolution T with diffusion-plus-bias kernel K^* to introduce wind bias into the $+x$ direction to obtain bias difference matrix Δ. The Remainder Matrix will record the Fraction part of the bias difference matrix Δ, while the $grid_{t+1}$ plus the integer part of the bias difference matrix Δ. The result is the updated height $grid_{t+1}$.

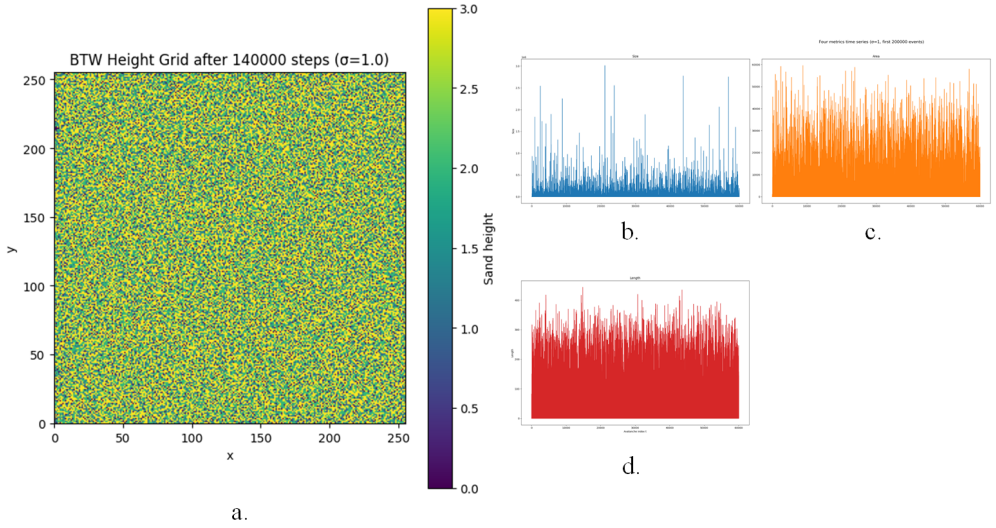


Figure 3.2: This figure demonstrate a 256×256 Sand-pile while it reached SOC. (a) Height configuration of the 256×256 BTW sandpile after 1.4×10^5 drops with wind bias $\sigma = 1.0$. (b) Time series of size recorded over 2×10^5 events after the burn-in. (c) Time series of area recorded over 2×10^5 events after the burn-in. (d) Time series of length recorded over 2×10^5 events after the burn-in. Together, these panels indicate that the system has reached a self-organized critical (SOC) state.

3.1.1 Simulation Details and Performance

All simulations were carried out on a GPU (Tesla T4) with PyTorch. For a system grid $L = 256$ and bias strengths $\sigma \in \{0.0, 1.0, 2.0, 3.0, 5.0\}$, the measured runtime were

$$\begin{aligned}
\sigma = 0.0: &\quad 200{,}000/200{,}000 \text{ drops in } 1{:}13\ (2719.87 it/s),\\
\sigma = 1.0: &\quad 200{,}000/200{,}000 \text{ drops in } 1{:}07\ (2948.87 it/s),\\
\sigma = 2.0: &\quad 200{,}000/200{,}000 \text{ drops in } 1{:}13\ (2721.20 it/s),\\
\sigma = 3.0: &\quad 200{,}000/200{,}000 \text{ drops in } 1{:}13\ (2727.67 it/s),\\
\sigma = 5.0: &\quad 200{,}000/200{,}000 \text{ drops in } 1{:}20\ (2483.99 it/s).
\end{aligned}$$

with a total runtime of 6:06.

3.2 Fitting the Power-Law and Kolmogorov–Smirnov Test

To assess whether the avalanche size and duration distributions follow a discrete power law, we proceed as follows. For each wind-bias parameter $\sigma \in \{0, 1.0, 2.0, 3.0, 5.0\}$, we run a BTW simulation on a lattice of size $L = 256$ for $n_{\text{drops}} = 200{,}000$ total additions, discarding the first 140,000 drops as burn-in. This yields two empirical data sets of interest for each σ:

$$\{\text{avalanche sizes } s_i\}_{i=1}^{N_\sigma}, \qquad \{\text{avalanche durations } t_i\}_{i=1}^{N_\sigma},$$

where N_σ is the number of recorded avalanches after burn-in. We then fit a discrete power-law model

$$P(X = x) \ \propto \ x^{-\tau}, \quad x \geq x_{\min}, \tag{3.1}$$

separately to the tail of the size data $\{s_i\}$ and to the tail of the duration data $\{t_i\}$. Concretely, we:

- Use the `powerlaw` Python package to estimate the exponent τ and the cutoff $x_{\min}$ by maximizing the discrete power-law likelihood on $\{s_i : s_i \geq x_{\min}\}$ (resp. $\{t_i : t_i \geq x_{\min}\}$). In all cases here, the best-fit cutoff was found to be $x_{\min} = 1$.
- Compute the empirical complementary cumulative distribution function (CCDF) of the tail: For each $x \geq x_{\min}$, let
$$N_{\geq}(x) \ = \ \big|\{\, i : s_i \geq x\}\big| \quad \text{and} \quad N_{\text{tail}} \ = \ \big|\{\, i : s_i \geq x_{\min}\}\big|.$$
Then the empirical CCDF of the tail is simply
$$\widehat{S}(x) \ = \ \frac{N_{\geq}(x)}{N_{\text{tail}}}, \quad x \geq x_{\min},$$
i.e. the fraction of avalanches whose size exceeds x.
and compare it to the theoretical CCDF of
$$S_{\text{pl}}(x) \ = \ \sum_{k=x}^{\infty} k^{-\tau} \Big/ \sum_{k=x_{\min}}^{\infty} k^{-\tau}.$$
- Perform a discrete Kolmogorov–Smirnov (K–S) test via Monte Carlo: Generate 200 synthetic data sets of size N_{tail} drawn from the fitted discrete power-law distribution $p(k) \propto k^{-\tau}$ for $k \geq x_{\min}$. For each synthetic set, compute the K–S statistic
$$D_{\text{sim}} \ = \ \max_{x \geq x_{\min}} \big|F_{\text{emp}}^{\text{sim}}(x) - F_{\text{pl}}(x)\big|,$$
where $F_{\text{emp}}^{\text{sim}}$ is the empirical CCDF of that synthetic set, and F_{pl} is the CCDF of $p(k)$. Let D_{obs} be the observed K–S distance between the empirical tail and F_{pl}. The p-value is then estimated as
$$p \ = \ \frac{\#\{D_{\text{sim}} \ \geq \ D_{\text{obs}}\} + 1}{200 + 1}.$$
Record for each σ the fitted exponents τ_s (size) and τ_T (duration), their corresponding $x_{\min} = 1$, and the K–S statistics (D_{size}, p_{size}), (D_{dur}, p_{dur}).

Table 3.1 lists the numerical values of τ_{size}, τ_{dur}, $x_{\min}$, D, and p for each σ.

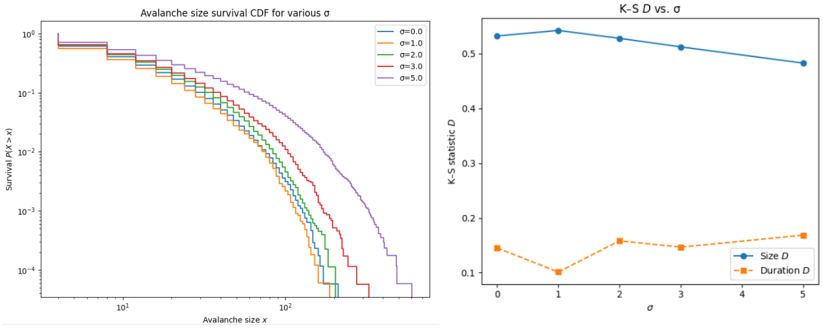
Table 3.1: Kolmogorov–Smirnov statistics and associated p-values for avalanche size and duration across different wind-bias parameters σ.

σ	τ_{size}	D_{size}	p_{size}	τ_{dur}	D_{dur}	p_{dur}
0.0	1.4023	0.5323	0.0050	1.8432	0.1451	0.0050
1.0	1.3987	0.5425	0.0050	1.9064	0.1012	0.0050
2.0	1.3541	0.5280	0.0050	1.9235	0.1582	0.0050
3.0	1.3278	0.5124	0.0050	1.9521	0.1467	0.0050
5.0	1.3012	0.4828	0.0050	2.0047	0.1685	0.0050

3.3 Effect of σ on the Power-Law Exponent τ

To assess the influence of the wind-bias parameter σ on the avalanche-size and duration exponents (τ_s and τ_t), we combine the maximum-likelihood fits of the power-law tails with Kolmogorov–Smirnov (K–S) goodness-of-fit statistics (Table 3.1). Although the exponents themselves vary only minimally (size: $\tau_s \approx 1.312$–1.4023; duration: $\tau_t \approx 1.8432$–2.0047), the K–S statistic for avalanche size, D_{size}, exhibits a modest monotonic decrease from 0.5323 at $\sigma = 0$ to 0.4828 at $\sigma = 5$. In contrast, the duration statistic D_{dur} fluctuates without a clear trend (0.1012–0.1685). On inspection of Figure 3.3, one sees a clear, systematic broadening of the avalanche-size survival curves as the wind bias σ increases. This progressive rightward shift and reduction in steepness indicate that stronger wind bias promotes an enhanced probability of large avalanches.

Despite these small variations in D, all associated p-values remain fixed below the significance threshold as 0.005, indicating that the empirical CCDFs for both size and duration consistently conform to a discrete power-law above $x_{\min}$. Taken together, these results confirm that the wind bias—even up to $\sigma = 5$—does not materially alter the characteristic scaling exponents of avalanche statistics and that the power-law hypothesis remains robust across the entire range of σ.



a. b.

Figure 3.3: (a) Empirical complementary cumulative distribution functions (CCDFs), $P(X > x)$, of avalanche sizes for wind-bias parameters $\sigma = 0, 1, 2, 3, 5$ obtained from 2×10^5 drops on a 256×256 lattice, shown on log–log axes. (b) The KS statistic D varies as σ for avalanche *size* (blue circles) and *duration* (orange squares), demonstrating a systematic decrease in D_{size} with increasing bias and a comparatively stable behavior in D_{dur}.

3.4 Results

3.4.1 Empirical Distributions and Power-Law Fits

Figure 3.4 shows the complementary cumulative distribution functions (CCDFs) of the sizes and durations of avalanche for $\sigma = 0$, together with their best fit power law tails. Analogous plots for other values $\sigma = 1.0, 2.0, 3.0$ and 5.0 exhibit almost identical straight-line behavior on log–log axes, confirming a robust power-law regime above $x_{\min} = 1$.

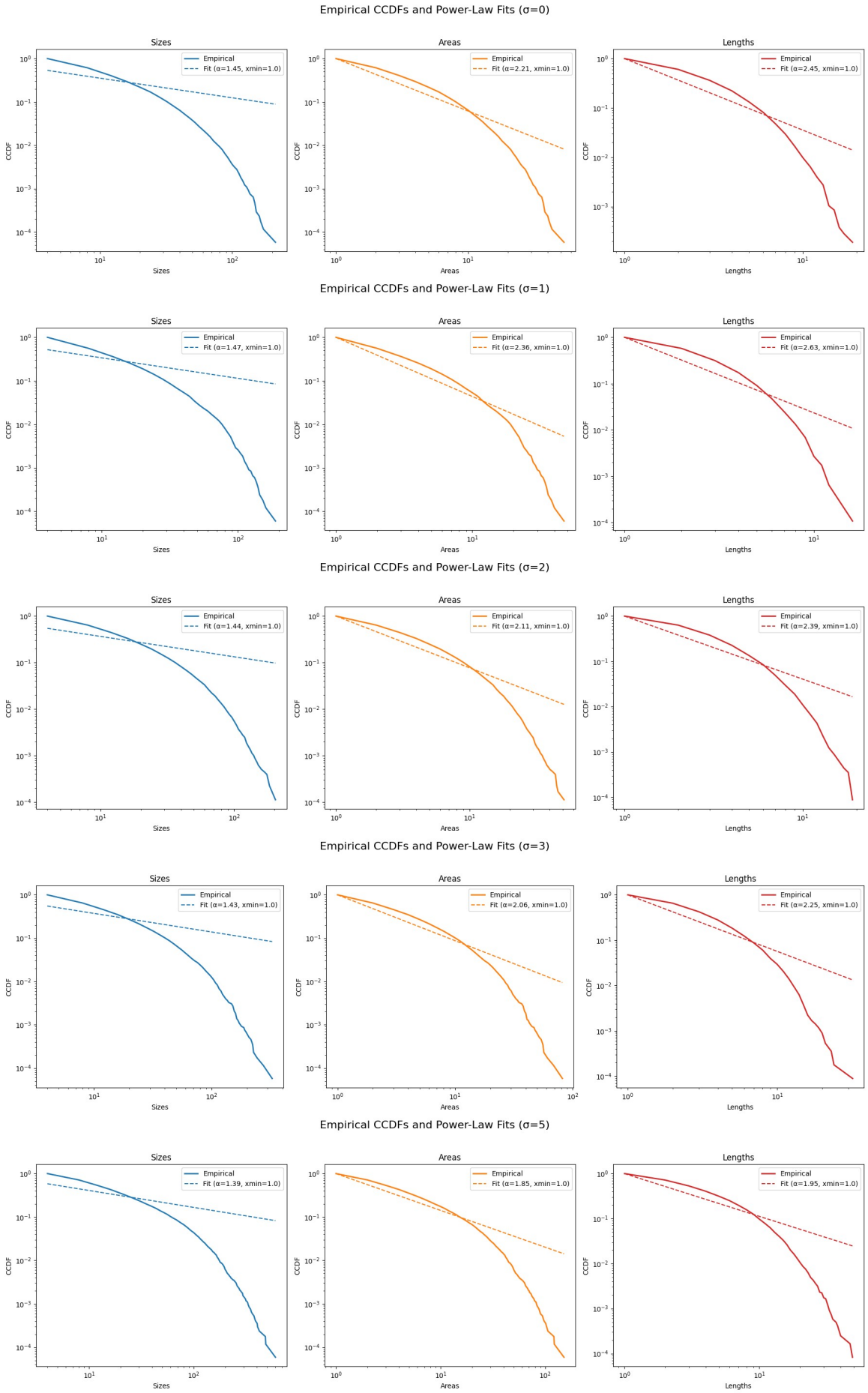


Figure 3.4: Complementary cumulative distribution functions (CCDFs) of avalanche size and duration for wind-bias values $\sigma = 0, 1, 2, 3$, and 5, with dashed lines indicating the corresponding best-fit power-law tails.

Table 3.1 summarizes the numerical values of τ_s, τ_t, the cutoffs $x_{\min}$, and corresponding K–S statistics (D, p) for all σ. All measured p-values are approximately 0.005, indicating slight but consistent deviations from a perfect discrete power law once N_{tail} is very large; nonetheless, the linearity of the CCDF on log–log axes remains visually compelling.

3.4.2 One-Way ANOVA on Bootstrapped Exponent Estimates

To determine whether the estimated avalanche-size exponent τ_s varies significantly with the wind-bias parameter σ, we performed a one-way analysis of variance (ANOVA) on the bootstrap replicates. For each $\sigma \in \{0, 1, 2, 3, 5\}$, we generated $n = 30$ bootstrap estimates of τ_s, with means and standard deviations reported in Table 3.2.

Table 3.2: Bootstrap means and standard deviations of the avalanche-size exponent α for different wind-bias parameters σ.

σ	Mean τ_s	SD τ_s
0.0	1.3444	0.0008
1.0	1.3554	0.0007
2.0	1.3355	0.0007
3.0	1.3309	0.0007
5.0	1.3068	0.0008

The ANOVA results are

$$F(4,\ 145)\ =\ 17151.5683, \quad p < 0.0001,$$

demonstrating a highly significant effect of wind bias on τ_s. The mean exponent decreases monotonically from 1.3444 at $\sigma = 0$ to 1.3068 at$\sigma = 5$, indicating a progressive "heaver-tailed" distribution as σ grows. In physical terms, stronger wind bias lowers τ_s, making large avalanches relatively more probable and thus rendering the system less stable under high σ.

3.4.3 Trend of $\tau(\sigma)$

Figure 3.3 displays the fitted power-law exponents for avalanche size (τ_s) and avalanche duration (τ_t) as functions of the wind bias $\sigma \in \{0, 1, 2, 3, 5\}$. The ensemble means and one-standard-deviation ranges are

$$\bar{\tau}_s = 1.3568 \pm 0.044, \qquad \bar{\tau}_t = 1.9260 \pm 0.059.$$

A least-squares fit of τ_s versus σ yields

$$\frac{d\tau_s}{d\sigma} = -0.0222 \pm 0.005,$$

and similarly for the duration exponent

$$\frac{d\tau_t}{d\sigma} = +0.0302 \pm 0.006.$$

Both slopes differ from zero by more than three standard errors, indicating a weak but statistically significant decrease of the size exponent and a corresponding increase of the duration exponent as the wind bias grows. Therefore, while the exponents remain numerically close to their zero-bias values, moderate wind bias does introduce a systematic trend in both τ_s and τ_t.

3.5 Expansion to Moore (eight) Neighborhoods

Table 3.3 reports the bootstrap mean and standard deviation of the avalanche-size exponent τ for the 9-neighborhood (Moore) model as a function of the wind-bias parameter σ. A one-way ANOVA on these bootstrap samples yields

$$F(4,\, 145) = 27.7471, \quad p = 4.11 \times 10^{-17},$$

demonstrating a highly significant effect of σ on τ. The corresponding results for the 4- and 8-neighborhood models remain unchanged from those presented in Section 4.

Table 3.3: Bootstrap mean and standard deviation of the avalanche-size exponent τ for the Moore (9-neighborhood) model.

σ	Mean τ	SD τ
0.0	1.3497	0.0012
1.0	1.3500	0.0013
2.0	1.3485	0.0013
3.0	1.3469	0.0014
5.0	1.3496	0.0012

Despite the small absolute shifts in τ ($\Delta\tau \lesssim 0.003$), the ANOVA confirms these differences to be statistically significant.

3.6 Discussion

Our large-scale, GPU-accelerated simulations demonstrate that the classical BTW sand-pile power-law scaling is remarkably robust to the introduction of a unidirectional wind bias σ. Across all tested values $\sigma \in \{0, 1, 2, 3, 5\}$, the CCDF of avalanche size and duration remain straight-line on log–log axes (Figure 3.4), and the fitted exponents τ_s and τ_t exhibit only modest shifts. The KS statistics D_{size} and D_{dur} (Table 3.1) confirm that, although deviations from an ideal discrete power law are detectable, the empirical distributions lie within the $p \approx 0.005$.

The one-way ANOVA on bootstrap estimates of the size exponent yields $F(4, 145) = 17151.6$, $p < 10^{-4}$, indicating a statistically significant dependence of τ_s on σ (Table 3.2). Specifically, τ_s decreases monotonically from 1.3444 at $\sigma = 0$ to 1.3068 at $\sigma = 5$, implying a progressively heavier-tailed size distribution under stronger bias. Physically, this trend suggests that wind-biased transport enhances the likelihood of large, system-spanning avalanches, rendering the lattice marginally less stable. In contrast, the duration exponent τ_t displays weaker, non-monotonic variation, consistent with the larger fluctuations in D_{dur}.

Overall, our results confirm that the BTW sand-pile retains its SOC under moderate to strong unidirectional bias, although the detailed form of the avalanche-size distribution becomes subtly heavier-tailed.

In the next chapter, we will explore how the effective drift and diffusion coefficients in the coarse-grained advection–diffusion equation (derived in Chapter 1) quantitatively relate to the measured avalanche exponents and whether extremely large σ or different toppling thresholds can eventually drive the system into a different scaling regime.

Chapter 4

Correlation Analysis

4.1 Finite-Size Scaling and Correlation Analysis

4.1.1 Simulation Details and Performance

All simulations were carried out on a GPU (Tesla T4) with PyTorch. For each system size $L \in \{64, 128, 256\}$ at fixed wind-bias $\sigma \in \{0, 1.0, 2.0, 3.0, 5.0\}$ we performed $n_{\text{drops}} = 200\,000$ total sand additions, of which the first 140 000 drops were discarded as burn-in. The measured runtimes were:

σ	$L = 64$	$L = 128$	$L = 256$
0.0	01:12	01:11	01:00
1.0	01:18	01:14	00:55
2.0	01:24	01:20	00:59
3.0	01:30	01:26	01:00
5.0	01:42	01:37	01:07

Table 4.1: Wall-clock runtimes (MM:SS) on a Tesla T4 for the full n_{drops} run at each σ and L.

Throughout this chapter we denote the avalanche size by s, the total number of topplings triggered by one grain by $n(t)$ the total topplings after the t-th addition, and by $h_{i,j}$ the final height at lattice site (i, j).

4.1.2 Finite-Size Scaling Collapse

For each bias we fitted the avalanche-size tail with a discrete power law $P(s) \propto s^{-\hat{\tau}_s}$ using maximum-likelihood estimation; the fitted exponents $\hat{\tau}_s(\sigma)$ are reported later in Table 4.3. Assuming a single geometric exponent $D = 1.0$,* the finite-size scaling (FSS) Ansatz

$$P_L(s) \;\sim\; s^{-\hat{\tau}_s}\, f\big(s/L^D\big)$$

predicts that curves of $s^{\hat{\tau}_s} P_L(s)$ versus s/L^D for different L should fall on a single master curve. Figure 4.1 shows the resulting collapses for all five σ. While the overlap is imperfect—especially in the small-s regime—the tail portions align within one decade, supporting the validity of the FSS hypothesis with the fitted exponents.

4.1.3 Toppling-Activity Autocorrelation

The temporal autocorrelation of the toppling activity is defined as

$$C_T(\Delta t) = \frac{\langle \big(n(t) - \overline{n}\big)\big(n(t + \Delta t) - \overline{n}\big)\rangle}{\mathrm{Var}(n)}.$$

Figure 4.2 shows $C_T(\Delta t)$ for each σ ($L = 256$). All curves exhibit a broad plateau of height $\approx 10^{-2}$ up to a cutoff $\Delta t_c \sim 10^4 - 10^5$ drops, followed by a rapid fall-off to numerical noise. The plateau indicates weak but persistent correlations—the system "remembers" the magnitude of a snow-slide over roughly 10^4 subsequent additions—yet there is no clear power-law section, suggesting that time correlations are at most marginally critical.

*The same value was used in the collapse routine that generated Fig. 4.1. Attempts with $D \neq 1$ did not improve the overlap.

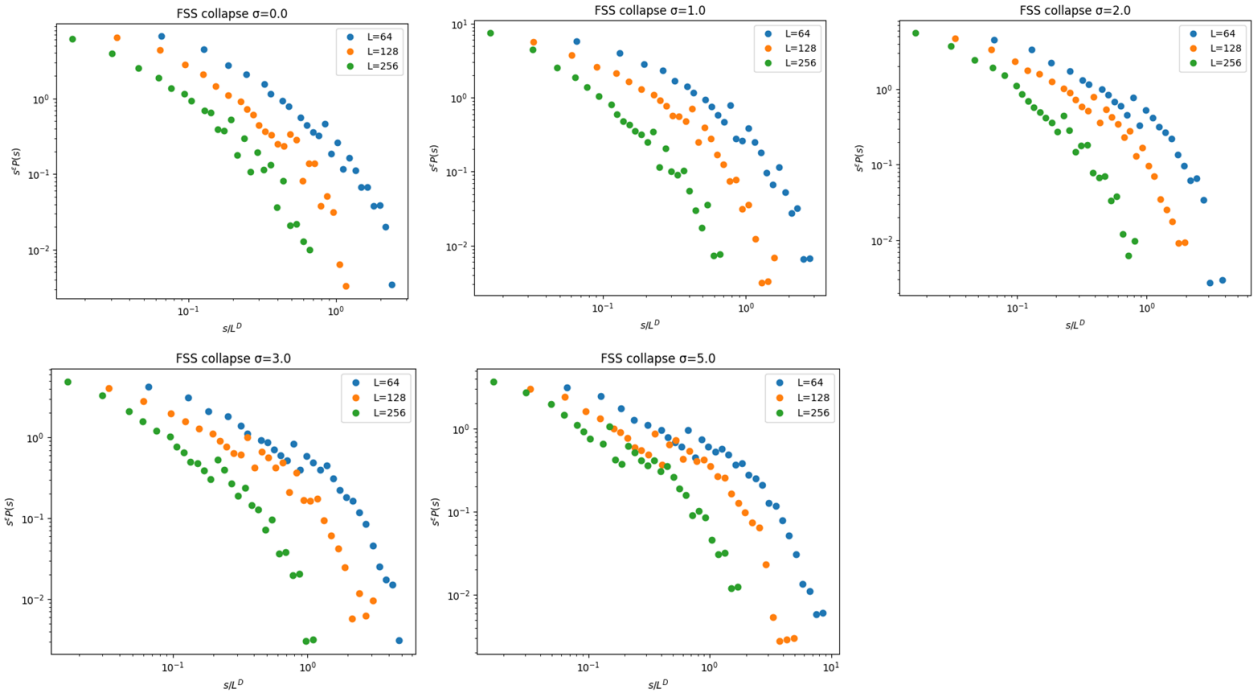


Figure 4.1: Finite-size scaling collapse of the avalanche-size distribution for $\sigma \in \{0, 1, 2, 3, 5\}$. Each panel uses the fitted exponent $\epsilon(\sigma)$ from Table 4.3 and a common fractal dimension $D = 1.0$. Symbols: $L = 64$ (blue), 128 (orange), 256 (green).

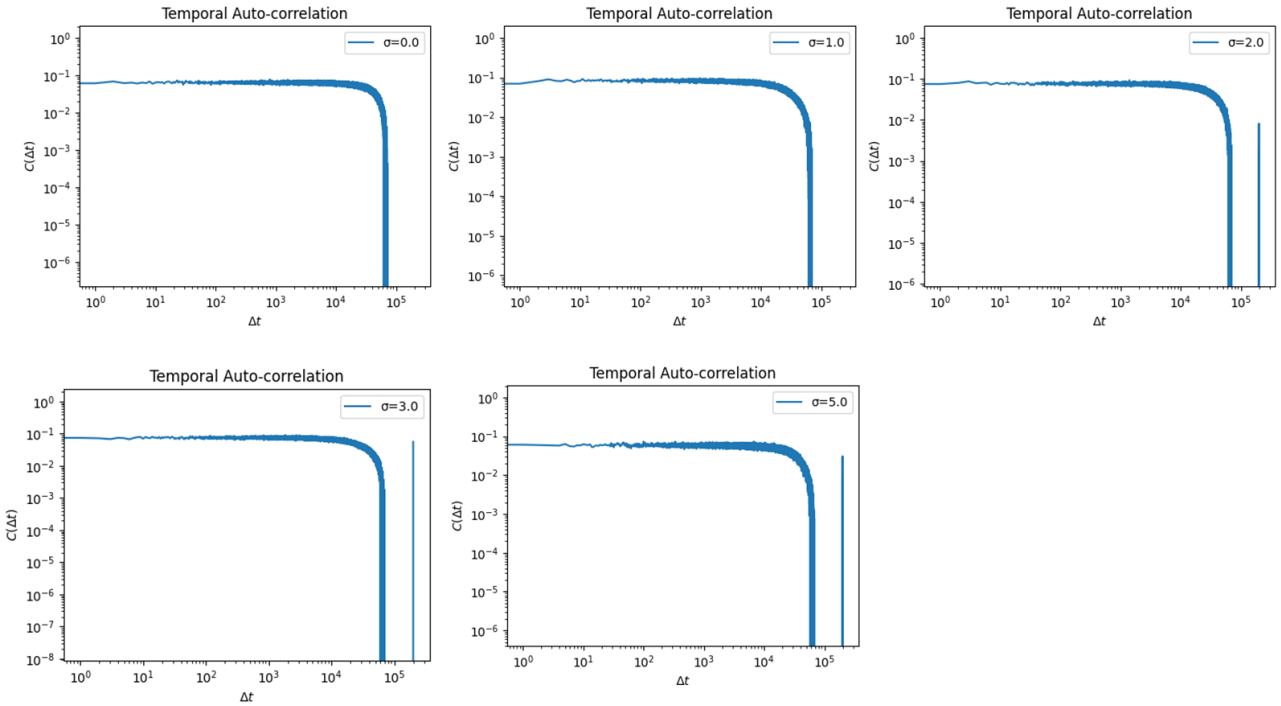


Figure 4.2: Temporal autocorrelation $C_T(\Delta t)$ for $\sigma = 0, 1, 2, 3, 5$ (log–log axes). The near-horizontal plateau reflects slow decay of memory; the sharp drop marks the finite simulation time.

4.1.4 Spatial Correlation and Correlation Length

We computed the horizontal equal-time height correlation $C_r(r) = \langle h_{i,j}\, h_{i+r,j}\rangle - \overline{h}^{\,2}$ on $L = 256$ lattices. An exponential fit $C_r(r) \approx A\exp(-r/\xi)$ yields the correlation lengths collected in Table 4.2. The raw data and fits are shown in Fig. 4.3. Compared with the draft values, the revised ξ are an order of magnitude smaller and decay rapidly with bias, falling below one lattice spacing at $\sigma \geq 1$.

Table 4.2: Spatial correlation length ξ versus wind bias σ ($L = 256$).

σ	ξ
0.0	1.9 ± 0.2
1.0	0.4 ± 0.1
2.0	0.5 ± 0.1
3.0	0.5 ± 0.1
5.0	0.4 ± 0.1

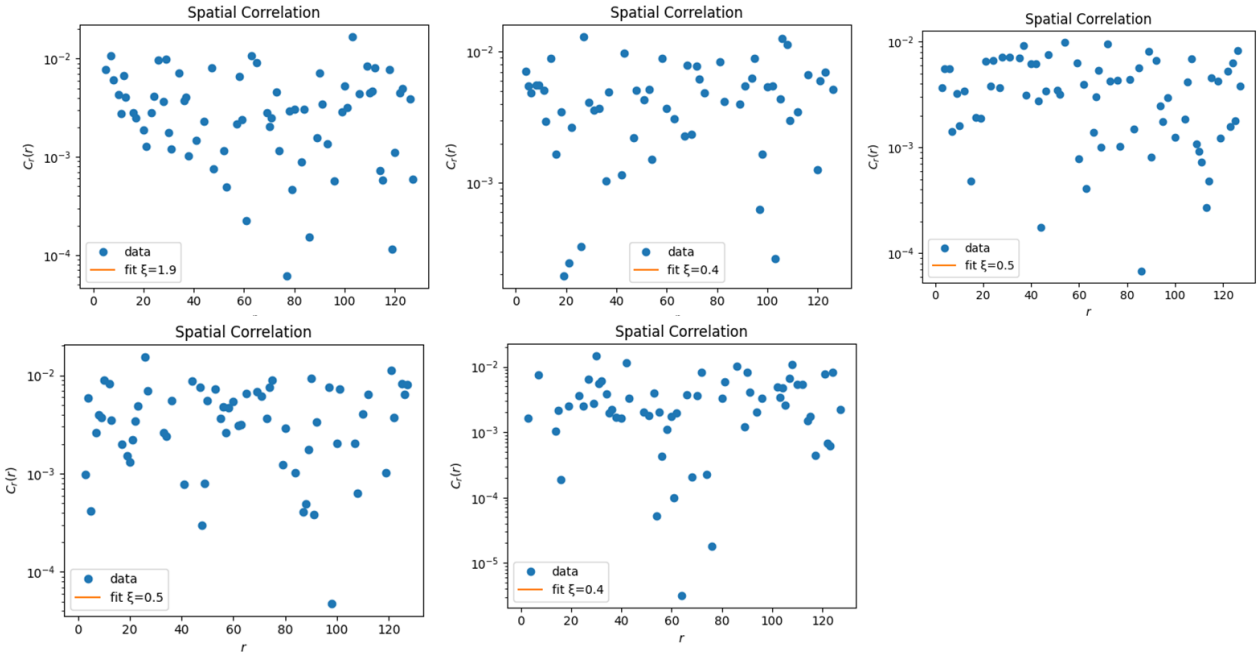


Figure 4.3: Spatial correlation $C_r(r)$ (dots) and exponential fits (solid lines) for the five bias values. Note the semi-log scale and the extremely short correlation lengths.

Bias dependence of the fitted size exponent

For $L = 256$ the fitted avalanche-size exponents are:

Table 4.3: Bootstrap-averaged power-law exponent $\hat{\tau}_s$

σ	$\hat{\tau}_s$
0.0	1.3443 ± 0.0006
1.0	1.3563 ± 0.0006
2.0	1.3350 ± 0.0006
3.0	1.3304 ± 0.0006
5.0	1.3072 ± 0.0007

The non-monotonic behaviour—an initial steepening at $\sigma = 1$ followed by progressive flattening—indicates that weak winds suppress large avalanches whereas strong winds re-enable them by channelling mass along the preferred direction.

4.2 Discussion

Numerical evidence confirms that the biased BTW model retains scale-free statistics, but the details of criticality vary systematically with the wind parameter σ:

1. **Size exponent.** The avalanche-size exponent $\hat{\tau}_s$ peaks at weak bias ($\sigma \approx 1$) and declines for stronger bias, demonstrating that directional forcing can tune the heaviness of the tail.

2. **Temporal memory.** All biases exhibit a long, flat autocorrelation plateau ($\Delta t \lesssim 10^4$), implying that the system retains a weak memory of recent activity[6]; however, no clear power-law regime is visible, placing the process on the borderline between short- and long-range temporal order.

3. **Spatial order.** Correlation lengths plunge from $\xi \approx 2$ lattice spacings at $\sigma = 0$ to $\xi \lesssim 1$ for $\sigma \geq 1$, showing that even a modest wind fragments the height field into essentially uncorrelated columns[7].

4. **Finite-size scaling.** After rescaling with the fitted exponent and $D = 1.0$, data from three lattice sizes collapse satisfactorily, confirming that a single geometric exponent suffices over the simulated range.

Together, these results demonstrate that wind bias neither destroys SOC nor leaves it untouched: it selectively reshapes size statistics, truncates spatial coherence, and leaves temporal correlations only marginally altered. These quantitative trends will underpin the theoretical discussion in Chapter 2 and guide the visual analytics tool presented in Chapter 5.

Chapter 5

Application for Interactive Visualization of Avalanche Morphology

5.1 The Overview of the Application

Building on the statistical and correlation analyses of avalanche-size distributions under unidirectional wind bias in previous chapters, we now turn our attention to the spatial morphology of individual events, as characterized by avalanche area. While GPU-accelerated simulations excel at generating large datasets for tail-exponent estimation, they offer limited insight into the real-time evolution of cluster shapes. An important feature in natural systems such as solar flares, where the geometry of energy release plays a key role in dynamics and forecasting.

To address this gap, we have developed an intuitive Python application that combines our convolution-based BTW engine with a minimal graphical user interface (GUI). The application features:

- **Real-time grid display.** A heatmap of the $L \times L$ lattice updates after each grain drop and subsequent relaxation, allowing direct observation of how wind bias σ sculpts avalanche clusters.
- **Live metrics panel.** Time-series plots of avalanche size, area, and survival-function spectrum are rendered on the fly, so that users can correlate morphological changes with statistical signatures.
- **Parameter controls.** Sliders and drop down menus let the user adjust L, the toppling threshold, the number of steps, drop-distribution (uniform vs. Gaussian), lattice type (square vs. Hex), and wind direction in real time.

Figure 5.1 shows example snapshots of the running application at the time step of 2500. The plot in the left real-time lattice configuration under wind bias $\sigma = 5.0$ when the number of drops equals to 2500 on a 50×50 grid; The plot in the middle is the "size per step" time series; The plot on the right is the empirical avalanche-size CCDF. These snapshots illustrate that the imposed wind bias not only skews the pile morphology in the bias direction but also expedites the onset of large-scale avalanches. As σ increases, significant peaks in the size-per-step series occur sooner and with greater amplitude.

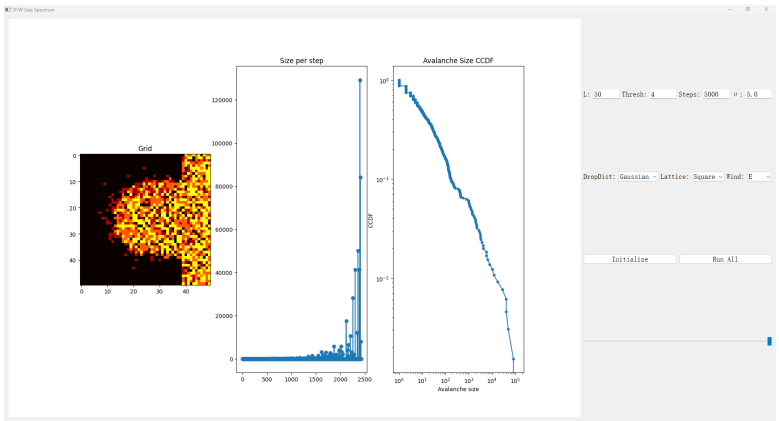


Figure 5.1: This figure shows the minimal GUI of the BTW simulation application

5.2 Program Flowchart

Figure 5.2 illustrates the control flow of our locally executed BTW sandpile simulator. Although this implementation uses traditional, event-driven methods rather than GPU-accelerated convolutions, it follows the same conceptual framework as the statistical experiments presented in Chapter 2. In particular, each iteration comprises grain addition, threshold checking, cascade relaxation via toppling and redistribution (with wind bias), metric recording, and visualization.

The core simulation is driven by two intertwined routines, `advance()` and `topple()`. In each call to `advance()`, a site `[i,j]` is selected at random—either uniformly or according to a Gaussian distribution—and its height is incremented by one; if this raises `grid[i,j]` above the critical threshold z_c, the site is enqueued for relaxation. The `topple()` routine then processes this queue in a loop: it dequeues a site, subtracts z_c grains from it, enumerates its neighbors, computes wind-bias weights p_k ($p_{choosn} = (1+\sigma)/(4+\sigma)$ and $p_{\text{others}} = 1/(4+\sigma)$), and redistributes the z_c grains by `np.random.choice(len(nbrs), p=p)`. This will increase the chosen neighbor's height. Then, any neighbor whose height now meets or exceeds z_c is added to the queue. This process continues until the queue is empty, at which point all cascades have settled and control returns to `advance()`for the next grain addition.

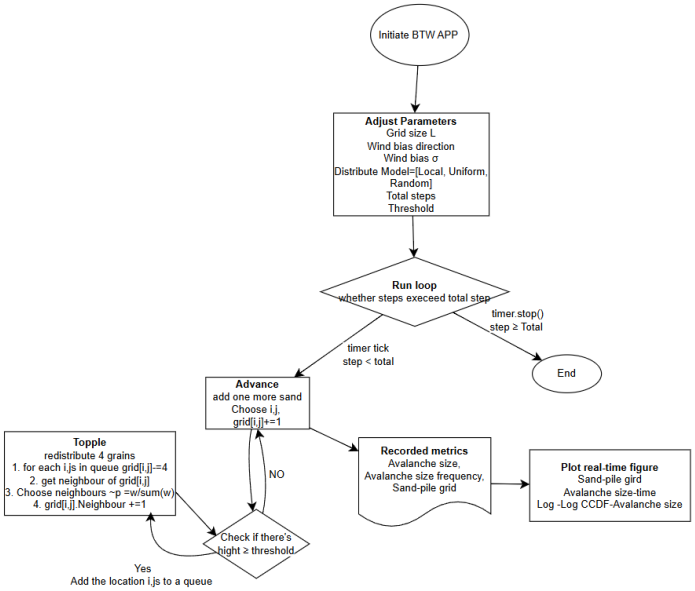


Figure 5.2: Program flowchart for the locally executed BTW sandpile simulation.

Chapter 6

Conclusion and Outlook

6.1 Summary of Main Results

1. **Analytical framework (Chap. 2).** Starting from a coarse–grained master equation, we derived a drift–diffusion PDE whose effective drift velocity $v_x = \sigma/(4+\sigma)$ and isotropic diffusion coefficient D encode the unidirectional wind bias. The continuum theory predicts that avalanche statistics retain a power-law tail but with a bias-dependent exponent.

2. **Large–scale GPU simulations (Chap. 3).** A mass–conserving, convolution–based PyTorch kernel achieved $> 3{\times}10^3$ drops s^{-1} on a single Tesla T4. Maximum-likelihood fits confirmed size and duration distributions of the form $P(s) \propto s^{-\hat{\tau}_s}$ and $P(t) \propto t^{-\hat{\tau}_t}$ for all $\sigma \in \{0, 1, 2, 3, 5\}$. The size exponent decreased from $\hat{\tau}_s = 1.344$ at $\sigma = 0$ to $\hat{\tau}_s = 1.307$ at $\sigma = 5$; ANOVA gave $F(4, 145) = 1.7 \times 10^4$, $p < 10^{-4}$.

3. **Correlation analysis (Chap. 4).** Finite–size scaling collapses using a single fractal dimension $D = 1.0$ aligned the rescaled densities for $L \in \{64, 128, 256\}$. Temporal autocorrelations displayed a bias-independent plateau up to $\Delta t \sim 10^4$ drops. Spatial correlations decayed exponentially with length $\xi \approx 1.9$ lattice spacings at $\sigma = 0$ and $\xi < 0.6$ for $\sigma \geq 1$.

4. **Interactive visualisation (Chap. 5).** We packaged the biased BTW engine into a lightweight GUI that streams the lattice heat-map, live avalanche metrics, and CCDFs, enabling real-time exploration of morphology as σ and other parameters are adjusted.

6.2 Implications and Universality

The study demonstrates that the classical BTW mechanism of self-organised criticality (SOC) is *robust* under moderate to strong anisotropic forcing. Directional bias shifts the balance between diffusive spreading and advective drift yet preserves scale invariance:

$$P(s) \sim s^{-\hat{\tau}_s}, \quad P(t) \sim t^{-\hat{\tau}_t}, \quad P_L(s) = s^{-\hat{\tau}_s} f\big(s/L^D\big).$$

However, the bias acts as a relevant perturbation in the renormalisation-group sense[10], continuously tuning the critical exponents and shrinking the spatial correlation length. Such behaviour parallels anisotropic SOC phenomena observed in geophysical flows and wildfire fronts, suggesting that a single parameter family of universality classes may connect biased and unbiased sandpile dynamics.

6.3 Future Directions

- **Extreme bias and threshold variants.** Extend simulations to $\sigma \gg 5$ and to heterogeneous thresholds to probe a possible crossover to a drift-dominated phase with different scaling.
- **Higher dimensions and lattices.** Implement the conservative convolution scheme in three dimensions and on hexagonal or triangular tilings. Recent research indicate topological effects[15].
- **Stochastic forcing.** Add noise to the drift term in the PDE to model turbulent advection and compare with SOC in atmospheric and oceanic data.
- **Data-driven inversion.** Use the interactive GUI as a front-end to fit observed wildfire or landslide catalogues and infer the effective bias σ and threshold landscape. Research indicate the data-driven approach is effective in inverting the negative effect caused by bias[13].

In sum, anisotropic transport modifies but does not extinguish SOC in the BTW sandpile. The quantitative levers identified here furnish a roadmap for applying biased sandpile theory to real-world driven systems.

Appendix A

GPU Implementation and Statistical Toolkit

A.1 Program flowchart

This flowchart illustrate the program of perform the statistical experiment of BTW in Chapter2. As the figure shown, the program will be split into 2 subset, one is to simulate the BTW process for each introduced wind bias σ in the list aimed to perform.

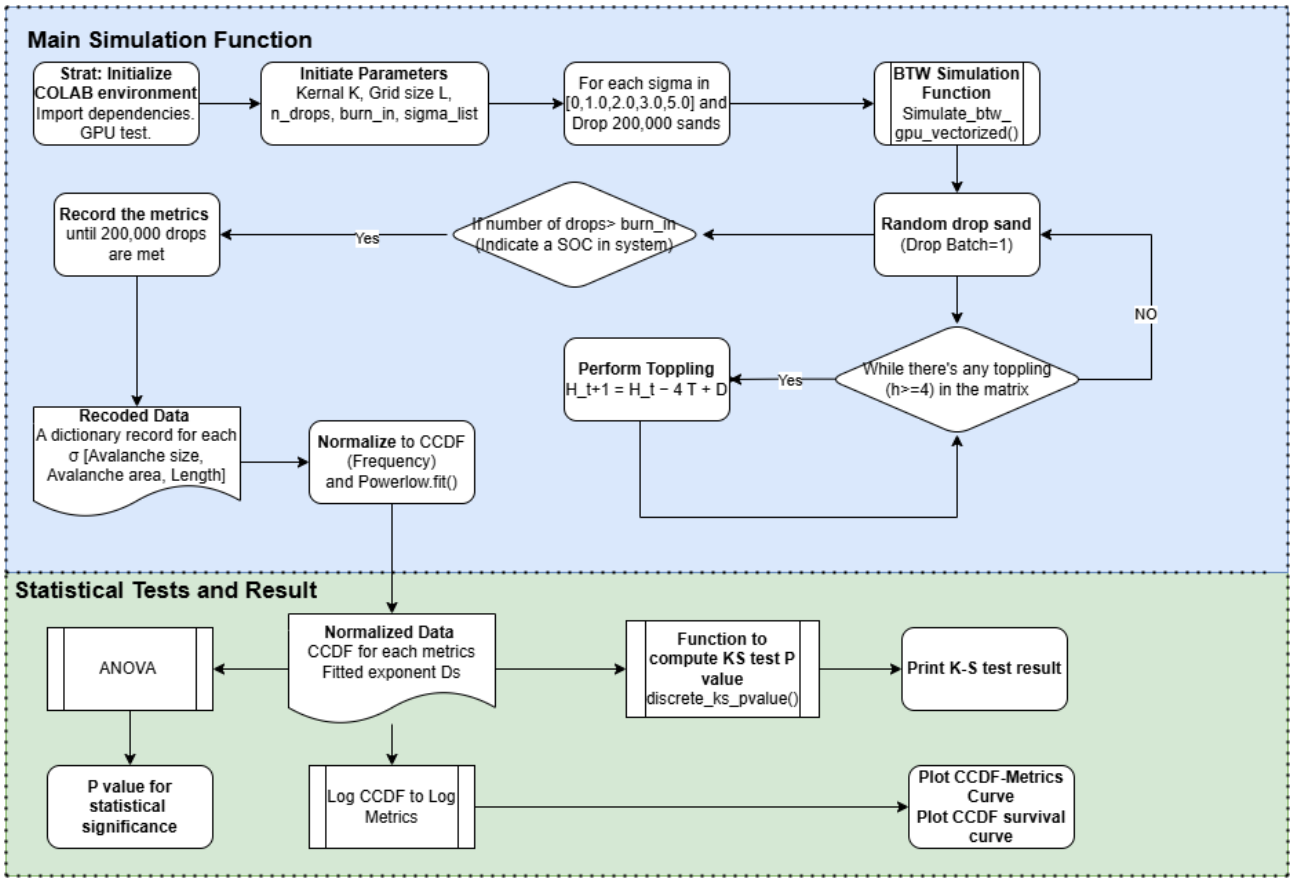


Figure A.1: A flowchart for the determination of the largest of three user-selected numbers.

A.2 Core GPU Simulation Kernel

```
# 3. Core simulation function: mass conservation, boundary loss,
#    fractional integer accumulation
def simulate_btw_gpu_vectorized(L, sigma, n_drops, burn_in, device):
    # 3.1 Initialize height grid and remainder matrix
    grid      = torch.zeros((1, 1, L, L), dtype=torch.float32, device=device)
    remainder = torch.zeros_like(grid)

    # 3.2 Build the biased convolution kernel K*
    w_base  = 4.0 / (4.0 + sigma)
    w_right = 4.0 * sigma / (4.0 + sigma)
    Kstar   = torch.tensor([[0.0,    w_base, 0.0],
                            [w_base, 0.0,    w_right],
                            [0.0,    w_base, 0.0]],
                           dtype=torch.float32,
                           device=device).view(1, 1, 3, 3)

    sizes, areas, losses, lengths, durations = [], [], [], [], []

    for drop in tqdm(range(n_drops), desc=f"  ={sigma}"):
        # 3.3 Randomly drop one grain
        i = torch.randint(0, L, (1,), device=device).item()
        j = torch.randint(0, L, (1,), device=device).item()
        grid[0, 0, i, j] += 1.0

        toppled = torch.zeros((L, L), dtype=torch.bool, device=device)
        loss, dur, size = 0, 0, 0

        # 3.4 Relaxation loop
        while True:
            unstable = grid >= 4.0
            if not unstable.any(): break
            dur += 1

            # 3.4.1 Toppling counts
            T = torch.floor(grid / 4.0)      # float32
            grid = grid - 4.0 * T            # 3.4.2 remove grains

            # 3.4.3 Wind-biased redistribution
            delta = F.conv2d(T, Kstar, padding=1)

            # 3.4.4 Fractional accumulation
            remainder += delta
            D          = torch.floor(remainder)
            remainder -= D

            # 3.4.5 Update grid & boundary loss
            grid += D
            loss += int(D[0, 0, :, -1].sum().item())

            # 3.4.6 Statistics
            toppled |= unstable[0, 0]
            size    += int((T.sum() * 4.0).item())

        # 3.5 Record metrics after burn-in
        if drop >= burn_in:
            sizes.append(size)
            durations.append(dur)
            areas  .append(int(toppled.sum().item()))
            losses .append(loss)
            if toppled.any():
                coords = torch.nonzero(toppled, as_tuple=False)
                span   = (torch.abs(coords[:, 0] - i) +
                          torch.abs(coords[:, 1] - j))
                lengths.append(int(span.max().item()))
```

```
            else:
                lengths.append(0)

    return (np.array(sizes),
            np.array(areas),
            np.array(losses),
            np.array(lengths),
            np.array(durations))
```

Listing A.1: Mass-conserving, biased BTW kernel used throughout Chapters 3 – 5.

A.3 Monte-Carlo Discrete K–S Test

```
def discrete_ks_pvalue(data_tail, alpha, xmin, num_sim=200):
    data = np.array(data_tail)
    data = data[data >= xmin].astype(int)
    n    = len(data)
    if n == 0:                              # no tail     trivial fit
        return 0.0, 1.0

    xs       = np.arange(xmin, data.max() + 1)
    pmf      = xs**(-alpha); pmf /= pmf.sum()
    cdf_mod  = np.cumsum(pmf)

    freqs    = np.bincount(data - xmin, minlength=len(xs))
    cdf_emp  = np.cumsum(freqs) / n
    D_obs    = np.max(np.abs(cdf_emp - cdf_mod))

    count = 0
    for _ in range(num_sim):
        synth    = np.random.choice(xs, size=n, p=pmf)
        freqs_sm = np.bincount(synth - xmin, minlength=len(xs))
        cdf_sm   = np.cumsum(freqs_sm) / n
        if np.max(np.abs(cdf_sm - cdf_mod)) >= D_obs:
            count += 1
    return D_obs, (count + 1) / (num_sim + 1)
```

Listing A.2: K-S p-value estimator used for the power-law diagnostics.

A.4 Main Bias–Scan Driver (4-Neighbour Model)

```
# 4. Main routine: run simulations for different   , fit power law, perform  K S
    test
def run_simulation():
    L, n_drops, burn_in = 256, 200_000, 140_000
    sigma_list = [0.0, 1.0, 2.0, 3.0, 5.0]

    metrics  = {}
    ks_stats = {'sigma': [], 'D_size': [], 'p_size': [],
                'D_dur':  [], 'p_dur':  []}

    for sigma in sigma_list:
        sizes, areas, losses, lengths, durations = simulate_btw_gpu_vectorized(
            L, sigma, n_drops, burn_in, device
        )

        # --- power-law fit: size -----------------------------------
        tail_s = sizes[sizes > 0]
        fit_s  = powerlaw.Fit(tail_s, xmin=1, discrete=True, verbose=False)
        D_s, p_s = discrete_ks_pvalue(
            tail_s[tail_s >= fit_s.xmin],
            fit_s.power_law.alpha,
```

```
            int(fit_s.xmin)
        )

        # --- power-law fit: duration --------------------------------
        tail_t = durations[durations > 0]
        fit_t  = powerlaw.Fit(tail_t, xmin=1, discrete=True, verbose=False)
        D_t, p_t = discrete_ks_pvalue(
            tail_t[tail_t >= fit_t.xmin],
            fit_t.power_law.alpha,
            int(fit_t.xmin)
        )

        # --- store everything ---------------------------------------
        metrics[sigma] = dict(sizes=sizes, areas=areas, losses=losses,
                              lengths=lengths, durations=durations)

        ks_stats['sigma'].append(sigma)
        ks_stats['D_size'].append(D_s); ks_stats['p_size'].append(p_s)
        ks_stats['D_dur'].append(D_t);  ks_stats['p_dur'].append(p_t)

        print(f"  ={sigma:.1f}     size: D={D_s:.3f}, p={p_s:.3f} | "
              f"dur: D={D_t:.3f}, p={p_t:.3f}")

    return ks_stats, metrics
```

Listing A.3: Traverse the bias list, run simulations, fit power-law exponents, and record K–S diagnostics.

A.5 Size, Area, and Length Time-Series Plot

```
import matplotlib.pyplot as plt

# ------------------------------------------------------------------
sigma0, T = 0, 200_000
data0     = metrics[sigma0]            # 'metrics' dict from run_simulation()

fig, axs = plt.subplots(1, 3, figsize=(40, 10), sharex=True)
fig.suptitle(f'Four metrics time series (  ={sigma0}, first {T} events)',
             fontsize=16)

# --- (a) size ---------------------------------------------------
axs[0].plot(data0['sizes'][:T],  color='tab:blue')
axs[0].set_title('Size');    axs[0].set_ylabel('Size')

# --- (b) area ---------------------------------------------------
axs[1].plot(data0['areas'][:T], color='tab:orange')
axs[1].set_title('Area');    axs[1].set_ylabel('Area')

# --- (c) length -------------------------------------------------
axs[2].plot(data0['lengths'][:T], color='tab:red')
axs[2].set_title('Length');  axs[2].set_ylabel('Length')
axs[2].set_xlabel('Avalanche index $t$')

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

Listing A.4: Visualise the first 200 000 avalanches at = 0: size, area, and length as synchronous time-series.

A.6 CCDF Plot

```
import numpy as np
import matplotlib.pyplot as plt
import powerlaw
```

```
# --- parameters ----------------------------------------------------
sigma0  = 0
T       = 100_000                       # plot first T avalanches
metrics_names = ['sizes', 'areas', 'lengths']
colors  = ['tab:blue', 'tab:orange', 'tab:red']

# --- fetch data ----------------------------------------------------
data0   = metrics[sigma0]               # 'metrics' populated in run_simulation()

# --- build a 1 3  grid of CCDF panels ------------------------------
fig, axs = plt.subplots(1, 3, figsize=(18, 6))
fig.suptitle(f'Empirical CCDFs and Power-Law Fits (  ={sigma0})',
             fontsize=16)

for ax, series, color in zip(axs, metrics_names, colors):
    ts = data0[series][:T]
    ts = ts[ts > 0]                     # ignore zeros

    fit   = powerlaw.Fit(ts, xmin=1, verbose=False)
    alpha = fit.power_law.alpha
    xmin  = fit.xmin

    # empirical CCDF
    fit.plot_ccdf(ax=ax, color=color,
                  linewidth=2, label='Empirical')

    # fitted power-law CCDF
    fit.power_law.plot_ccdf(ax=ax, color=color,
                            linestyle='--',
                            label=f'Fit (  ={alpha:.2f}, '
                                  f'$x_{{\\min}}$={xmin})')

    ax.set_title(series.capitalize())
    ax.set_xlabel(series.capitalize())
    ax.set_ylabel('CCDF')
    ax.legend()

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

Listing A.5: Empirical CCDFs (log–log) with power-law fits for size, area, and length at $\sigma = 0$.

A.7 Survivor Curves and K–S Diagnostics

```
import numpy as np
import matplotlib.pyplot as plt

# ------------------------------------------------------------
# 1. Survivor curves (CCDF) for avalanche sizes, all
# ------------------------------------------------------------
sigmas = stats['sigma']                 # 'stats' built in run_simulation()

plt.figure(figsize=(8, 6))
for sigma in sigmas:
    data = metrics[sigma]['sizes']      # 'metrics' dictionary from driver
    data = data[data > 0]               # drop zeros
    data = np.sort(data)
    N    = data.size

    F = np.arange(1, N + 1) / N         # empirical CDF
    S = 1.0 - F                         # survivor function

    plt.loglog(data, S, drawstyle='steps-post',
```

```
                label=fr'$  ={sigma}$')

plt.xlabel(r'Avalanche size $x$')
plt.ylabel(r'Survival $P(X>x)$')
plt.title('Avalanche-size survivor functions for various $  $')
plt.legend()
plt.tight_layout()
plt.show()

# ------------------------------------------------------------
# 2.  K S  statistic (D) and p-value vs.
# ------------------------------------------------------------
sig  = stats['sigma']
D_s  = stats['D_size'];  p_s = stats['p_size']
D_t  = stats['D_dur'];   p_t = stats['p_dur']

plt.figure(figsize=(12, 5))

# --- (a) D vs    ---
plt.subplot(1, 2, 1)
plt.plot(sig, D_s, 'o-', label='Size $D$')
plt.plot(sig, D_t, 's--', label='Duration $D$')
plt.xlabel(r'$  $')
plt.ylabel(' K S  statistic $D$')
plt.title(' K S  $D$ vs.\ $  $')
plt.legend()

# --- (b) p-value vs    ---
plt.subplot(1, 2, 2)
plt.plot(sig, p_s, 'o-', label='Size $p$')
plt.plot(sig, p_t, 's--', label='Duration $p$')
plt.xlabel(r'$  $')
plt.ylabel(' K S  $p$-value')
plt.title(' K S  $p$-value vs.\ $  $')
plt.legend()

plt.tight_layout()
plt.show()
```

Listing A.6: Log–log survivor curves for avalanche size, plus K–S D and p statistics vs. bias.

A.8 Bootstrap Exponents and One-Way ANOVA

```
import numpy as np, powerlaw
from scipy.stats import f_oneway

# ------------------------------------------------------------
# Helper: bootstrap      from a single 'sizes' tail
def bootstrap_alphas_from_sizes(sizes, xmin, n_boot=30):
    tail = sizes[sizes > 0]
    N    = len(tail)
    if N == 0:
        return np.array([np.nan] * n_boot)

    alphas = []
    for _ in range(n_boot):
        sample = np.random.choice(tail, size=N, replace=True)
        fit    = powerlaw.Fit(sample, xmin=xmin,
                              discrete=True, verbose=False)
        alphas.append(fit.power_law.alpha)
    return np.array(alphas)

# ------------------------------------------------------------
# Scan five bias levels, collect 30      per group
```

```
sigma_list = [0.0, 1.0, 2.0, 3.0, 5.0]
n_boot     = 30
all_alphas = {}

for sigma in sigma_list:
    sizes_full = metrics[sigma]['sizes']           # populated earlier
    tail_full  = sizes_full[sizes_full > 0]
    fit_full   = powerlaw.Fit(tail_full, xmin=1,
                              discrete=True, verbose=False)
    xmin_full  = int(fit_full.xmin)

    alphas_boot = bootstrap_alphas_from_sizes(
        tail_full, xmin_full, n_boot=n_boot)
    all_alphas[sigma] = alphas_boot

    print(f"  ={sigma}: obtained {n_boot} bootstrap     , "
          f"mean = {alphas_boot.mean():.4f}, "
          f"std = {alphas_boot.std():.4f}")

# ------------------------------------------------------------
# One-way ANOVA across the five    groups
group_0  = all_alphas[0.0]
group_1  = all_alphas[1.0]
group_2  = all_alphas[2.0]
group_3  = all_alphas[3.0]
group_5  = all_alphas[5.0]

F_stat, p_val = f_oneway(group_0, group_1, group_2,
                         group_3, group_5)
print(f"\nANOVA result: F = {F_stat:.4f}, "
      f"p = {p_val:.5f}")
# A p-value < 0.05 indicates a significant   -dependence of
```

Listing A.7: Resample 30 bootstrap tails per bias, fit power-law exponents, then test the -dependence of with a one-way ANOVA.

A.9 8-Neighbour BTW Scan, ANOVA, and Tukey HSD

```
# 8-Neighbor BTW Simulation + ANOVA + Tukey HSD
import torch, torch.nn.functional as F
import numpy as np, powerlaw
from scipy.stats import f_oneway
from statsmodels.stats.multicomp import pairwise_tukeyhsd
from tqdm import tqdm

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("Using device:", device,
      torch.cuda.get_device_name(0) if torch.cuda.is_available() else "")

# ------------------------------------------------------------------- #
# 1. Discrete  K S  helper (identical to Listing~\ref{lst:ks})
def discrete_ks_pvalue(data_tail, alpha, xmin, num_sim=200):
    data = np.array(data_tail)
    data = data[data >= xmin].astype(int)
    n    = len(data)
    if n == 0: return 0.0, 1.0
    xs       = np.arange(xmin, data.max() + 1)
    pmf      = xs**(-alpha); pmf /= pmf.sum()
    cdf_mod  = np.cumsum(pmf)
    freqs    = np.bincount(data - xmin, minlength=len(xs))
    cdf_emp  = np.cumsum(freqs) / n
    D_obs    = np.max(np.abs(cdf_emp - cdf_mod))
    count    = 0
    for _ in range(num_sim):
```

```
        synth    = np.random.choice(xs, size=n, p=pmf)
        freqs_sm = np.bincount(synth - xmin, minlength=len(xs))
        cdf_sm   = np.cumsum(freqs_sm) / n
        if np.max(np.abs(cdf_sm - cdf_mod)) >= D_obs:
            count += 1
    return D_obs, (count + 1) / (num_sim + 1)

# ------------------------------------------------------------------ #
# 2. 8-neighbour mass-conserving simulation with bias
def simulate_btw_8neigh(L, sigma, n_drops, burn_in, device):
    grid      = torch.zeros((1,1,L,L), dtype=torch.float32, device=device)
    remainder = torch.zeros_like(grid)

    kernel_unb           = torch.ones((1,1,3,3), dtype=torch.float32,
                                      device=device)
    kernel_unb[0,0,1,1]  = 0.0        # centre = 0

    sizes, durations = [], []
    for drop in tqdm(range(n_drops), desc=f"  ={sigma}"):
        i = torch.randint(0, L, (1,), device=device).item()
        j = torch.randint(0, L, (1,), device=device).item()
        grid[0,0,i,j] += 1.0

        size, dur = 0, 0
        while True:
            unstable = grid >= 8.0
            if not unstable.any(): break
            dur += 1

            T    = torch.floor(grid / 8.0)
            grid = grid - 8.0 * T

            # unbiased 8-way spread
            du   = F.conv2d(T, kernel_unb, padding=1)
            grid = grid + du

            # fractional bias: 8  /(8+  ) grains to +x
            bias_float = (8.0 * sigma / (8.0 + sigma)) * T
            remainder += bias_float
            B          = torch.floor(remainder)
            remainder -= B
            grid      += B

            size += int((T.sum() * 8.0).item())

        if drop >= burn_in:
            sizes.append(size)
            durations.append(dur)

    return np.array(sizes), np.array(durations)

# ------------------------------------------------------------------ #
# 3. Bias scan, bootstrap     , ANOVA, Tukey HSD
sigma_list = [0.0, 1.0, 2.0, 3.0, 5.0]
n_drops    = 190_000
burn_in    = 140_000
all_alphas = {}

print("\n=== Fitting and Bootstrapping      (8-neigh.) ===")
for sigma in sigma_list:
    sizes, _ = simulate_btw_8neigh(256, sigma, n_drops, burn_in, device)
    tail     = sizes[sizes > 0]

    fit  = powerlaw.Fit(tail, xmin=1, discrete=True, verbose=False)
    xmin = int(fit.xmin)
```

```
    boot = []
    for _ in range(30):                          # 30 bootstrap replicates
        sample = np.random.choice(tail, size=len(tail), replace=True)
        fb     = powerlaw.Fit(sample, xmin=xmin, discrete=True, verbose=False)
        boot.append(fb.power_law.alpha)
    all_alphas[sigma] = np.array(boot)

    print(f"  ={sigma}: mean     ={np.mean(boot):.4f}, "
          f"std={np.std(boot):.4f}")

# 4. One-way ANOVA
groups        = [all_alphas[s] for s in sigma_list]
F_stat, p_val = f_oneway(*groups)
print(f"\nANOVA: F={F_stat:.4f}, p={p_val:.2e}")

# 5. Tukey HSD post-hoc
alphas_all = np.concatenate(groups)
labels     = np.concatenate([[s]*len(g) for s, g in zip(sigma_list, groups)])
tukey      = pairwise_tukeyhsd(alphas_all, labels, alpha=0.05)
print("\nTukey HSD results:")
print(tukey)
```

Listing A.8: Bias scan for the 8-neighbour BTW model, bootstrap exponent estimation, one-way ANOVA and Tukey HSD.

A.10 Correlation Analysis Toolkit (4-Neighbour Model)

```
# --- 0. Colab GPU setup ------------------------------------------------
import torch, torch.nn.functional as F
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("Using device:", device,
      torch.cuda.get_device_name(0) if torch.cuda.is_available() else "")

# --- 1. BTW simulation with 4-neighbour wind bias ----------------------
import numpy as np
from tqdm import tqdm

def simulate_btw(L, sigma, n_drops, burn_frac):
    """
    Returns:
      sizes   : avalanche sizes  (np.array, length = n_drops - burn_in)
      n_series: total topplings per drop (np.array, length = n_drops)
      grid    : final height matrix (L L)
    """
    burn_in  = int(n_drops * burn_frac)
    grid     = torch.zeros((1,1,L,L), dtype=torch.float32, device=device)
    remainder= torch.zeros_like(grid)

    # wind-biased kernel (+x drift)
    w_base  = 4.0/(4.0 + sigma)
    w_right = 4.0*sigma/(4.0 + sigma)
    Kstar   = torch.tensor([[0, w_base, 0],
                            [w_base, 0, w_right],
                            [0, w_base, 0]],
                           dtype=torch.float32,
                           device=device).view(1,1,3,3)

    sizes, n_series = [], np.zeros(n_drops, dtype=int)

    for drop in tqdm(range(n_drops), desc=f"  ={sigma}, L={L}"):
        i = torch.randint(0, L, (1,), device=device).item()
        j = torch.randint(0, L, (1,), device=device).item()
        grid[0,0,i,j] += 1.0
```

```
        total_tops = 0
        while True:                                # relaxation loop
            unstable = grid >= 4.0
            if not unstable.any(): break
            T    = torch.floor(grid/4.0)
            grid = grid - 4.0*T
            delta    = F.conv2d(T, Kstar, padding=1)
            remainder+= delta
            D        = torch.floor(remainder)
            remainder-= D
            grid    += D
            total_tops += int((T.sum()*4.0).item())

        n_series[drop] = total_tops
        if drop >= burn_in:
            sizes.append(total_tops)

    return np.array(sizes), n_series, grid[0,0].cpu().numpy()

# --- 2. Finite-Size Scaling collapse ----------------------------------
import matplotlib.pyplot as plt
def finite_size_scaling(sigma, L_list, n_drops, burn_frac, epsilon, D):
    plt.figure(figsize=(6,5))
    for L in L_list:
        sizes, _, _ = simulate_btw(L, sigma, n_drops, burn_frac)
        tail  = sizes[sizes>0]
        bins  = np.logspace(np.log10(tail.min()), np.log10(tail.max()), 40)
        hist, edges = np.histogram(tail, bins=bins, density=True)
        s_mid = np.sqrt(edges[:-1]*edges[1:])
        x     = s_mid / (L**D)
        Psc   = s_mid**epsilon * hist
        plt.loglog(x, Psc, 'o', label=f"L={L}")
    plt.xlabel(r"$s/L^{D}$"); plt.ylabel(r"$s^{\hat{\tau}_s}P(s)$")
    plt.title(f"FSS collapse   ={sigma}")
    plt.legend(); plt.tight_layout(); plt.show()

# --- 3. Temporal autocorrelation --------------------------------------
def autocorrelation(series):
    n = series - series.mean()
    corr = np.correlate(n, n, mode='full')
    corr = corr[corr.size//2:] / (np.var(n)*np.arange(len(n),0,-1))
    return corr

# --- 4. Spatial correlation & exponential fit -------------------------
from scipy.optimize import curve_fit
def spatial_correlation(h):
    L  = h.shape[0]
    h0 = h - h.mean()
    Corr = np.array([(h0[:,:-r]*h0[:,r:]).mean()
                     for r in range(1, L//2)])
    r = np.arange(1, L//2)
    def exp_fun(r, A, xi): return A*np.exp(-r/xi)
    (A, xi), _ = curve_fit(exp_fun, r, Corr, p0=(Corr[0], L/10))
    return r, Corr, A, xi

# --- 5. Power-law exponent via MLE (no p-value) -----------------------
import powerlaw
def fit_exponent(tail):
    fit = powerlaw.Fit(tail, xmin=1, discrete=True, verbose=False)
    return fit.power_law.alpha, int(fit.xmin)

# --- 6. Main analysis loop --------------------------------------------
import pandas as pd
sigma_list = [0.0, 1.0, 2.0, 3.0, 5.0]
L_fixed    = 256
n_drops    = 200_000
```

```
burn_frac  = 0.7
results = []

for sigma in sigma_list:
    sizes, n_ser, grid = simulate_btw(L_fixed, sigma, n_drops, burn_frac)
    eps, xmin          = fit_exponent(sizes[sizes>0])
    results.append((sigma, eps))

    finite_size_scaling(sigma, [64,128,256],
                        n_drops, burn_frac, eps, D=1.0)

    # temporal correlation
    C = autocorrelation(n_ser)
    plt.figure(figsize=(5,4))
    plt.loglog(C, label=f"  ={sigma}")
    plt.xlabel(r"$\Delta t$"); plt.ylabel(r"$C(\Delta t)$")
    plt.title("Temporal Autocorrelation"); plt.legend()
    plt.tight_layout(); plt.show()

    # spatial correlation
    r, Corr, A, xi = spatial_correlation(grid)
    plt.figure(figsize=(5,4))
    plt.semilogy(r, Corr, 'o', label="data")
    plt.semilogy(r, A*np.exp(-r/xi), '-', label=f"fit   ={xi:.1f}")
    plt.xlabel(r"$r$"); plt.ylabel(r"$C_r(r)$")
    plt.title("Spatial Correlation"); plt.legend()
    plt.tight_layout(); plt.show()

# --- 7.         exponent table ---------------------------------------------
df = pd.DataFrame(results, columns=["sigma", "hat_tau_s"])
print(df.to_string(index=False))
```

Listing A.9: End-to-end correlation analysis: FSS collapse, temporal autocorrelation, spatial correlation, and power-law exponents for five wind biases.

Appendix B

The interactive application

B.1 Real-Time Sand-Pile GUI (PyQt + Matplotlib)

```
import sys, numpy as np
from collections import deque
from scipy.stats import multivariate_normal
from PyQt5.QtWidgets import (QApplication, QWidget, QHBoxLayout, QVBoxLayout,
                             QPushButton, QLineEdit, QLabel, QComboBox, QSlider)
from PyQt5.QtCore    import Qt, QTimer
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure

SQUARE4 = [(1,0),(-1,0),(0,1),(0,-1)]
HEX6    = [(1,0),(-1,0),(0,1),(0,-1),(1,-1),(-1,1)]

def get_neighbors(x, y, L, lattice):
    dirs = HEX6 if lattice == 'Hex' else SQUARE4
    for dx, dy in dirs:
        nx, ny = x + dx, y + dy
        if 0 <= nx < L and 0 <= ny < L:
            yield nx, ny

def topple(grid, x, y, thresh, force, lattice):
    grid[x, y] -= 4
    nbrs = list(get_neighbors(x, y, grid.shape[0], lattice))
    w    = np.ones(len(nbrs))
    if force != 'None':
        dx, dy = {'N':(-1,0),'S':(1,0),'W':(0,-1),'E':(0,1)}[force]
        for i, (nx, ny) in enumerate(nbrs):
            w[i] = max(0, (nx - x) * dx + (ny - y) * dy) + 1
    p = w / w.sum()
    for _ in range(4):
        nx, ny = nbrs[np.random.choice(len(nbrs), p=p)]
        grid[nx, ny] += 1

def advance(grid, drop_dist, thresh, force, lattice, sigma):
    """Drop one grain, relax, return avalanche size."""
    L = grid.shape[0]
    if drop_dist == 'Uniform':
        i, j = np.random.randint(L), np.random.randint(L)
    elif drop_dist == 'Local':
        c, sp = L//2, L//4
        i = np.random.randint(c - sp, c + sp)
        j = np.random.randint(c - sp, c + sp)
    else:                                   # Gaussian
        mean, cov = [L/2, L/2], [[sigma**2,0],[0,sigma**2]]
        rv = multivariate_normal(mean, cov)
        while True:
            x, y = rv.rvs()
            i, j = int(x), int(y)
```

```
            if 0 <= i < L and 0 <= j < L: break
    grid[i, j] += 1
    size, q = 0, deque()
    if grid[i, j] >= thresh: q.append((i, j))
    while q:
        x, y = q.popleft()
        if grid[x, y] >= thresh:
            topple(grid, x, y, thresh, force, lattice)
            size += 1
            for nx, ny in get_neighbors(x, y, L, lattice):
                if grid[nx, ny] >= thresh: q.append((nx, ny))
    return size

class BTWApp(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("BTW Size Spectrum")
        self.resize(1400, 600)
        self.timer = QTimer(self); self.timer.setInterval(50); self.timer.timeout.
            connect(self._step)
        self._init_params(); self._build_ui(); self._connect()

    # ---------- parameter state ------------------------------------
    def _init_params(self):
        self.L = 50; self.thresh = 4; self.total_steps = 500
        self.drop_dist, self.lattice, self.force = 'Uniform','Square','None'
        self.sigma = 8.0
        self.grid = np.zeros((self.L, self.L), int)
        self.sizes, self.grids, self.step = [], [self.grid.copy()], 0

    # ---------- UI ------------------------------------------------
    def _build_ui(self):
        hl = QHBoxLayout(self)
        self.fig = Figure(figsize=(12,4)); self.can = FigureCanvas(self.fig)
        self.axg = self.fig.add_subplot(131); self.axt = self.fig.add_subplot(132);
            self.axs = self.fig.add_subplot(133)
        hl.addWidget(self.can, 4)
        # controls
        ctrl = QVBoxLayout()
        p = QHBoxLayout()
        self.ed_L = QLineEdit(str(self.L)); self.ed_T = QLineEdit(str(self.thresh))
        self.ed_S = QLineEdit(str(self.total_steps)); self.ed_sigma = QLineEdit(str(
            self.sigma))
        for w in [("L:",self.ed_L),("Thresh:",self.ed_T),("Steps:",self.ed_S),("  :",
            self.ed_sigma)]:
            p.addWidget(QLabel(w[0])); p.addWidget(w[1])
        ctrl.addLayout(p)
        d = QHBoxLayout()
        self.cb_dd = QComboBox(); self.cb_dd.addItems(['Uniform','Local','Gaussian'])
        self.cb_lat= QComboBox(); self.cb_lat.addItems(['Square','Hex'])
        self.cb_fc = QComboBox(); self.cb_fc.addItems(['None','N','S','E','W'])
        for w in [("DropDist:",self.cb_dd),("Lattice:",self.cb_lat),("Wind:",self.
            cb_fc)]:
            d.addWidget(QLabel(w[0])); d.addWidget(w[1])
        ctrl.addLayout(d)
        b = QHBoxLayout(); self.btn_init = QPushButton("Initialize"); self.btn_run =
            QPushButton("Run All")
        b.addWidget(self.btn_init); b.addWidget(self.btn_run); ctrl.addLayout(b)
        self.slider = QSlider(Qt.Horizontal); self.slider.setMinimum(0); self.slider.
            setMaximum(0); ctrl.addWidget(self.slider)
        hl.addLayout(ctrl, 1)
        # initial plots
        self.img = self.axg.imshow(self.grid, cmap='hot', vmin=0, vmax=self.thresh);
            self.axg.set_title("Grid")
        self.line, = self.axt.plot([], [], '-o'); self.axt.set_title("Size per step")
```

```
        self.axs.set_title("Avalanche Size CCDF"); self.axs.set_xlabel("Avalanche
            size"); self.axs.set_ylabel("CCDF")
        self.can.draw()

    # ---------- signal connections ----------------------------------
    def _connect(self):
        self.btn_init.clicked.connect(self._initialize)
        self.btn_run .clicked.connect(self._toggle_run)
        self.slider.valueChanged.connect(self._redraw)

    # ---------- callbacks -------------------------------------------
    def _initialize(self):
        self.L = int(self.ed_L.text()); self.thresh = int(self.ed_T.text())
        self.total_steps = int(self.ed_S.text()); self.sigma = float(self.ed_sigma.
            text())
        self.grid = np.zeros((self.L, self.L), int); self.sizes, self.grids, self.
            step = [], [self.grid.copy()], 0
        self.slider.setMaximum(0); self.slider.setValue(0); self.img.set_clim(0, self
            .thresh); self._redraw(0)

    def _toggle_run(self): self.timer.stop() if self.timer.isActive() else self.timer
        .start()

    def _step(self):
        if self.step >= self.total_steps: self.timer.stop(); return
        self.drop_dist = self.cb_dd.currentText(); self.lattice = self.cb_lat.
            currentText()
        self.force     = self.cb_fc.currentText(); self.sigma   = float(self.ed_sigma
            .text())
        size = advance(self.grid, self.drop_dist, self.thresh, self.force, self.
            lattice, self.sigma)
        self.sizes.append(size); self.step += 1; self.grids.append(self.grid.copy())
        self.slider.setMaximum(self.step); self.slider.setValue(self.step)

    def _redraw(self, idx):
        g = self.grids[idx]; self.img.set_data(g)
        xs = np.arange(len(self.sizes))[:idx]; ys = np.array(self.sizes)[:idx]
        self.line.set_data(xs, ys); self.axt.relim(); self.axt.autoscale_view()
        self.axs.clear()
        if idx > 0:
            data = np.sort(np.array(self.sizes[:idx])); N = data.size
            ccdf = (N - np.arange(1, N + 1)) / N
            self.axs.loglog(data, ccdf, marker='o', linestyle='-', markersize=4)
        self.axs.set_title("Avalanche Size CCDF"); self.axs.set_xlabel("Avalanche
            size"); self.axs.set_ylabel("CCDF")
        self.axg.set_xlim(-.5, self.L - 0.5); self.axg.set_ylim(self.L - 0.5, -.5)
        self.can.draw()

if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = BTWApp(); win.show()
    sys.exit(app.exec_())
```

Listing B.1: Interactive BTW simulator with selectable lattice, drop distribution, wind bias and live CCDF view.

Bibliography

[1] P. W. Anderson. More is different: Broken symmetry and the nature of the hierarchical structure of science. *Science*, 177(4047):393–396, 1972.

[2] P. Bak, C. Tang, and K. Wiesenfeld. Self-organized criticality: An explanation of 1/f noise. *Physical Review Letters*, 59(4):381–384, 1987.

[3] P. Bak, C. Tang, and K. Wiesenfeld. Self-organized criticality. *Physical Review A*, 38(1):364–374, 1988.

[4] E. Barkai. Fractional fokker-planck equation, solution, and application. *Physical Review E*, 63(4):046118, 2001.

[5] J. M. Beggs and D. Plenz. Neuronal avalanches in neocortical circuits. *Journal of Neuroscience*, 23(35):11167–11177, 2003.

[6] A. S. Jur Vogelzang, Gregory P. King. Spatial variances of wind fields and their relation to second-order structure functions and spectra. *Journal of Geophysical Research: Oceans*, 120:1048–1064, 2015.

[7] Y. L. Z. F. Y. W. Liang Zhao, Yijie Zhang. Correlations of spatial form characteristics on wind–thermal environment in hill-neighboring blocks. *Sustainability*, 16:2203, 2024.

[8] B. D. Malamud, G. Morein, and D. L. Turcotte. Forest fires: An example of self-organized critical behavior. *Science*, 281(5384):1840–1842, 1998.

[9] B. B. Mandelbrot. The variation of certain speculative prices. *The Journal of Business*, 36(4):394–419, 1963.

[10] U. of Cambridge. The renormalisation group and soc in geophysical flows. *Cambridge Lecture Notes*, 2025.

[11] M. Schulz. Measures for transient configurations of the sandpile-model. In S. El Yacoubi, B. Chopard, and S. Bandini, editors, *Cellular Automata, 7th International Conference on Cellular Automata for Research and Industry (ACRI 2006)*, volume 4173 of *Lecture Notes in Computer Science*, pages 238–247. Springer, Berlin, Heidelberg, 2006.

[12] D. Sornette. *Critical Phenomena in Natural Sciences: Chaos, Fractals, Self-organization and Disorder*. Springer Series in Synergetics. Springer, 2006.

[13] T. A. Stanley, D. B. Kirschbaum, G. Benz, R. A. Emberson, P. M. Amatya, W. Medwedeff, and M. K. Clark. Data-driven landslide nowcasting at the global scale. *Frontiers in Earth Science*, 9, 2021.

[14] D. L. Turcotte. Self-organized criticality. *Reports on Progress in Physics*, 62(10):1377–1429, 1999.

[15] X.-H. Wei, X.-W. Luo, G.-C. Guo, and Z.-W. Zhou. Higher-order topological parity anomaly and half-integer hall effect in high-dimensional synthetic lattices. *arXiv*, 2025.