BWS Numerical Implementation Notes

These notes describe implementation details for the implementation of the Branching Wiener Sausage simulation. Simulation code is written in C with supporting code written in Python for analysis.

I. INTRODUCTION

Branching Wiener Sausage is a Reaction-Diffusion processes that consists of a branching, diffusion and extinction. A particle either diffuses with probability h, branches with probability σ or becomes extinct with probability ϵ. We focus only on the critical branching rate which is at $\sigma = \epsilon$. In the current implementation parameters are chosen such that $h + \sigma + \epsilon = 1$ and h is chosen to be high e.g. 0.95. Each of these subprocesses is a *Poisson* process. As the system evolves, an exponential waiting time for an event is simulated as $-\ln(1-U)/N$ where $U \sim U(0,1)$ and N is the size of the population. Time is stored as a `long double`. The process is *bosonic*. When branching occurs, offspring are added *locally* i.e. to the same site as the parent. We are primarily interested in the distinct sites visisted by the process.

II. LATTICES

We consider regular lattices in d dimensions in addition to a fractal (Sierpinski) lattice and small-world network.

A. Regular Lattice

For regular lattices we typically choose system sizes of $2^n - 1$. We choose an upper bound that accounts for memory usage in higher dimensions. For example, two dimensions may exceed $L = 2^{10}$ while three and five dimensions may struggle depending on hardware. We typically consider $L \leq 63$ for 5D simulations. The d-dimensional lattice is mapped to a one dimensional data structure using a mapping $f : \mathcal{R}^d \to \mathcal{R}$,

$$f(x_0, x_1, ..., x_d) = \sum_{d=0}^{D-1} x_d L^d \tag{1}$$

With this mapping, a particle that hops 1 site in any direction in $\mathcal{R}^D$ jumps $\pm L^d$ in R^1 for a given component axis[1]. Note also the inverse mapping $f^{-1}(i, d)$ requires iteratively extracting components by dividing i by decreasing powers of L to the remainder of the result on each iteration.

The mapping maps each site to a *partition hierarchy* of the space and plays a role in implementing boundary conditions. A bitmask is used to close boundaries on given axes. For example setting bits $0, 1$ closes the first component axis at both ends to create a toroidal axis (as opposed to closed/reflecting boundaries). It is illegal for a particle to hop between partitions[2]. When a jump is computed along an axis such that $i' = i + L^d$, we can check the condition $h(i, d) \stackrel{?}{=} h(i', d)$, where h gives the partitions for i and i'.

$$h(i) = \left\lfloor \frac{i}{L^d} \right\rfloor,\ d = 1, ..., D \tag{2}$$

. Considering the mapping $\mathcal{R}^d \to \mathcal{R}$, we allow particles to move in one direction in any plane but we are careful to check when a particle moving in $\mathcal{R}$ jumps a partition illegally.

B. Sierpinski Fractal Lattice

We implement a Sierpinski fractal in d dimensions, $\mathcal{S}^{(d)}_{b,m}$. In 2 dimensions it is called a carpet. It is parametrized by b and m and like fractals in general, it is generated recursively. In this case the rule is to partition a lattice *element* into p^2 partitions and erase $p^2 - m$ partitions. An example of $\mathcal{S}_{3,8}$. In our simulations we will restrict b, m to certain convenient values and we will choose lattices with sides that are powers of b rather than considering all possible combinations of these parameters. We are consistent with choices of b, m as the fractal dimension is given in terms of these parameters. Hausdorff is

$$\mathcal{H}_d = \log m / \log b \tag{3}$$

$\mathcal{S}_{3,8}$ as an fractal lattice is a convenient extension to our existing experiments on square lattices because the overall structure does not change and we simply add holes or exclusions in the lattice. As such, a walker can simply check a lattice site to see if is *accessible*. We keep the existing boundary conditions for the extremities of the lattices but holes are *strictly inaccessible* i.e. a walker does not fall into holes but rather is reflected from them regardless of what external boundary conditions we choose. [3]. For any site i we require some accessibility rule

$$g(i) \to [0, 1] \tag{4}$$

---

[1] Conventionally we assign even numbers in $2D$ to positive(forward) moves and odd numbers to negative (back) moves

[2] TODO: Performance hint to predetermine boundary sites with lattice flags to avoid checking partitions each time.

[3] It is perhaps not quite accurate to say the walker is *reflected* as the site in the hole is not part of the adjacency matrix at all. In

In an example, choosing $b = 3, m = 8$, we always remove the middle tile out of 9 tiles. Given the recursive nature of the fractal, we can consider partitions at various zoom levels scaling in powers of 3. A rule of the form given by (4) is defined below. It determines if a particular lattice site falls in a hole of the fractal (in this example, the central tile) for *any* zoom level. We take zoom levels and corresponding partitions.

Over all zoom levels, there is a k-hierarchy of lattices with $p = b^k$ *tiles* at each level. Now let us use the notation $p = b^k$, $p' = b^{k-1}$. We can check that the following equality holds for *any* p

$$f_p(i) := \left\lfloor \frac{x_d(i) \ mod \ p}{p'} \right\rfloor \stackrel{?}{=} \lfloor b/2 \rfloor, \forall d \tag{5}$$

This defines if a site on $\mathcal{S}_{3,8}$ is inaccessible where the boolean result is understood to map to $[0, 1]$ and a site indexed i is mapped onto d components in D dimensions via $x_d(i)$. So, for a Sierpinksi fractal on a hypercubic lattice one must check that this condition holds *for all* c but *for any* p.

When initialising the process we start the walker a random direct offset from the boundary of the middle square e.g. from the centre we jump b to the k in a random d. Note the largest central tile in our case will cover L elements, L a square. So we can initialise by moving $\lceil (L/3)/2 \rceil$ in one random direction on the lattice.

III. WAITING TIMES AND RANDOM VARIABLES

We use the SMID-orientated Fast Mersenne Twister random number generator [ref]. We simulate Poisson waiting times but write output data deterministically according to a write time

$$W(i) = t_0 \Big(\frac{t_\infty}{t_0} \Big)^{\frac{i}{n_b - 1}} \tag{6}$$

where t_0, t_∞ are parameters for minimum and maximum timescales set to $10^{-1}, 10^9$. The number of bins, n_b is set to 1000.

When drawing random numbers from the generator and projecting to a range n, we make sure not to introduce bias due to rounding via modulus errors. Given a range maximum of the generator rng_∞, we draw again in the rare case where

$$x \geq rng_\infty - (rng_\infty \ mod \ n) \tag{7}$$

IV. DATA OUTPUT AND PARAMETER SPACE

We write tab delimited data for each experiment defined by the parameter space. An experiment $E(L, B, D, h)$ is parametrised by the system size L, the boundary conditions B, the dimensionality D and the hopping rate h. Each experiment will write metadata such as the random number generator seed. Each experiment produces N samples in chunks of size C. Chunks are simply used to aggregate statistics and flush data at regular intervals. The tabular data consists of columns that combined parameter space and values $M0, ...M8$ which correspond to the sample size in that chunk i.e. and the observables to higher orders $< s^n >$. Other data can be optionally written including the trace histogram.