On Interacting Particles in 1D and 2D

Joshua DM Hellier

Doctor of Philosophy
The University of Edinburgh
February 2019

Abstract

Interface growth, and in particular the prediction of its rate, has long been a tough problem in statistical physics. In this thesis, I will outline my personal take on the matter, and will showcase a possible approach to it consisting of constructing a microscopic model on a lattice and using this to parametrise a large-scale model of the phenomenon. I will then discuss how to do this with multiple interacting particle species in play.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Parts of this work have been published in .

(*Joshua DM Hellier, February 2019*)

Acknowledgements

Insert people you want to thank here.

Contents

Abstract i

Declaration ii

Acknowledgements iii

Contents iv

List of Figures vi

List of Tables vii

1 Monte-Carlo Simulations of the SPM 1

1.1 Numerical Simulations of Continuous-Time Markov Processes 1

1.1.1 Purpose of Monte-Carlo Methods 2

1.1.2 Evenly-Spaced Timesteps 2

1.1.3 The N-Fold Way, or Gillespie Algorithm 4

1.2 Implementation of Monte Carlo Methods 6

1.2.1 Our Implementation of a Metropolis-Hastings Algorithm with Evenly-Space Timesteps 6

1.2.2 `KMCLib` 6

1.2.3 Managing `KMCLib` Calculations in Parallel 10

1.3 1D Calculation Results 12
1.3.1 Flow Patterns 12
1.3.2 Scans Through λ with Constant Boundary Densities 16
1.3.3 Varying λ and Boundary Density Difference Together 16
1.3.4 Varying the Boundary Densities with Constant λ 16
1.3.5 Diffusion Coefficient 16
1.4 2D Calculation Results 16
1.4.1 Calculational Choices 16
1.4.2 Results 16
1.5 Conclusions 16

A Code Listings 18

A.1 1d Ising Correlation Functions 18
A.2 n-Dimensional Continuum-Limit MFT 20

Bibliography 21

List of Figures

(1.1) Illustration of the method for choosing successor states in the n-fold way. 6

(1.2) The flow pattern of sticky particles in 1D 14

(1.3) The flow pattern of sticky particles in 1D 17

List of Tables

Chapter 1

Monte-Carlo Simulations of the SPM

We now have numerical results for SPM systems using TRM analysis; however, this only allows us to study relatively small systems. In order to study larger ones, we have used Monte-Carlo methods. In this chapter, we will discuss the methods we used, the results they yielded and their meaning, with particular emphasis on what they tell us about the suspected transition between low and high-λ behaviours.

1.1 Numerical Simulations of Continuous-Time Markov Processes

Here we will discuss the theory behind the Monte-Carlo methods used to simulated continuous-time Markov processes. We will assume throughout that we have the computational means to produce pseudorandom floating-point numbers in a way which which closely approximates the uniform real distribution over $(0, 1)$.

1.1.1 Purpose of Monte-Carlo Methods

We should first really describe what we mean by a Monte-Carlo method. In essence, Monte-Carlo methods refer to numerical routines in which we attempt to characterise an unknown distribution, generated via known rules, by using pseudorandom numbers in order to produce sample data which is hopefully faithful to the original distribution, at least in terms of the statistics we are trying to calculate. A good example of a commonly-used Monte-Carlo method in Physics is the Metropolis-Hastings algorithm, which in its original for is used to calculate statistics for equilibrium statistical mechanics systems.

In our situation, we wish to be able to mimic a continuous-time discrete-state Markov process. As we saw in Chap. **??**, the state space for a TRM system of size L scales as $\mathcal{O}(2^L)$; thus we quickly run out of size if we try to consider exactly probability distributions, which correspond to vectors in $\mathbb{R}^{L^2}$. We can, however, store individual configurations, which only occupy $\mathcal{O}(L)$ space. Therefore, we need to find a way to produce trajectories through the discrete state space which sample the actual space of system trajectories well enough to allow us to access the statistics we want. Of course, there isn't a unique "best" way to do this. We have considered two contrasting methods, which differ primarily in the way in which they convert the original continuous time into discrete steps which we can use in an algorithm.

1.1.2 Evenly-Spaced Timesteps

If we wished to numerically approximate an ODE system, one might use the Euler forward or Runge-Kutte methods. These both involve discretising time simply by dividing it into evenly-sized pieces, and then converting the ODE into a discrete form by using finite differences to approximate derivatives. We need to be careful to choose a small enough timestep for the approximation to the derivative to remain good, but otherwise it is a very simple and effective approach.

We can do a very similar technique with continuous time Markov processes. In our SPM system, if we ignore the boundaries, there are two rates, 1 and λ, and our system is homogeneous. Let us represent the system with a binary array of length $L+2$, with L sites for the bulk and a site each representing the boundaries. Therefore, in order to simulate the action of the SPM as defined in ¡reference to

appropriate section in introduction¿, we can use the following recipe:

1. **START**. Advance time by Δt. Pick a site, which we will call Site, (of which there are $L+2$) at random. If the site chosen is one of the boundaries with density ρ, reset the site to be occupied with probability ρ and unoccupied with probability $1-\rho$.

2. If Site is occupied, pick one of the two adjacent sites, which we will call Target, at random with equal probability. This will be the site we attempt to move into. If it is not, go back to **START**.

3. *If Site is not on the boundary*: If Target is occupied, go to **START**. Otherwise, consider the other adjacent site, which we will refer to as Rear. If Rear is empty, move the particle in Site into Target randomly with probability $\frac{1}{1+\lambda}$; otherwise, move the particle with probability $\frac{\lambda}{1+\lambda}$. Return to **START**.
   If Site is on the boundary: If Target is outside the system, go to **START**. If Target is occupied, go to **START**. Assign an occupation value for Rear randomly, occupied with probability ρ, unoccupied with probability $1-\rho$, where ρ is the density of the relevant boundary. Now, if Rear is empty, move the particle in Site into Target randomly with probability $\frac{1}{1+\lambda}$; otherwise, move the particle with probability $\frac{\lambda}{1+\lambda}$. Return to **START**.

We define Δt via

$$\Delta t = \frac{\tau_0}{L(1+\lambda)}. \tag{1.1}$$

In terms of the algorithm's correctness at producing reasonable trajectories, we simply need note that the rates at which particular transitions should occur are in the correct proportions, and that the boundaries result in the correct densities in equilibrium; then, we just need to verify that the rate at which free particles move is the correct one in absolute terms, which it is, and we're done.

For Monte-Carlo methods, we generally rate their performance by the amount of computational power required to explore a given amount of the probability space. In methods in which we are exploring this space by advancing though time (and invoking ergodicity) we desire methods which move us quickly through time whilst maintaining good sampling and performing little computation.

The advantage of this method is that it is very simple; thus, there aren't too many opportunities for error when writing the code, it uses very little memory

(all calculations can be performed in-place), and each iteration should be very fast as there are very few overheads. It should also produce trajectories which are good samples of the original probability distribution we are trying to replicate.

If λ is close to 1, the probability of rejection (i.e. a step which results in no overall change to the system) is $\sim \frac{1}{2}$, and this is the situation in which the algorithm really shines; similarly it also performs well for large or small *lambda* if the system density is very high or very low respectively. For extreme λ in general however, performance drops off considerably, as we are often performing lots of calculations and advancing time very little, and thus not seeing much of the distribution simply because we aren't moving much.

We could have made this code marginally more efficient by making the more likely moves certain, and correspondingly adjusting the timestep size Δt to account for this; however, this only improves efficiency by a factor of around 2 , whilst making the code more complicated, so as we only used this method to verify the results of our main code we didn't bother. It is possible for us to get around this issue by advancing time in a variable fashion, although this comes at the cost of a little more computation per iteration.

1.1.3 The N-Fold Way, or Gillespie Algorithm

A popular way to produce trajectories for a continuous-time Markov process is the N-Fold Way, also known as BKL, or Gillespie Algorithm [1, 5, 6]. It evolves us through time as follows:

1. **START**: Make a list of all states which can be transitioned to in a single move from the current state, and the associated rates at which this occurs.
2. Weight each successor state by the transition rate into it, and then select a successor state by random selection from a uniform distribution over the weighted possible successors. Change the system state to the chosen successor.
3. Now advance time by an increment chosen from an exponential distribution whose decay rate is the sum of all of the rates of the possible transitions to a successor. Go back to **START**.

Now we just need to supply the rates ¡from introduction¿ that define the

SPM, along with some additional rates describing processes at the boundary. Specifically, we use the method described in **??** to do this, whereby we have a double layer of "blinking" boundary sites and sites in the internal layer undergo the same transitions as in the bulk. However, unlike in our TRM calculations, we should not set the incoming and outgoing rates to be extremely high, as then these rather trivial processes come to dominate the calculation and cause the timesteps to be on average extremely small, wasting our computing time. Instead, we set them to be proportional to the geometric mean of 1 and λ, and thus in the language of **??** this corresponds to setting $B_0 = \sqrt{\lambda}$. This way the boundaries refresh often, but not too often, and should still act as suitable reservoirs.

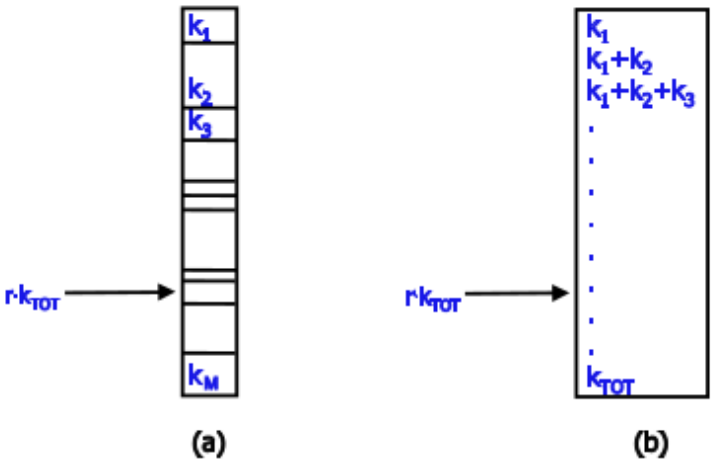
We will get into the fine details about how the software we use implements KMC in Sec. 1.2.2. Let us instead discuss how we obtain the required probability distributions using the uniform distribution on $(0, 1)$, $U(0, 1)$:

- We can randomly choose the successor state required in step 2 by creating a list of weighted partial sums. If the transition rate from the current state the i^{th} potential successor state is k_i, then let us define $k_{\text{Tot.}} = \sum_i^n$, where n is the number of potential successors. Create the list of partial sums via $s_i = \sum_j^i k_j$, then generate the random number $u = rk_{\text{tot}}$ where r is drawn from $U(0, 1)$. We can then use a binary search to find $i : s_{i-1} \leq u \leq s_i$, and then this i indicates the successor state which has been chosen. This process is illustrated more visually in Fig. 1.1.

- In step 3, we need to generate random numbers in an exponential distribution with decay rate k_{tot}. We can do this by generating r from $U(0, 1)$, and then $w = -\frac{1}{k_{\text{tot}}} \log r$ follows the desired distribution.

I will defer to Voter (see in particular Sec. 5 of [6] for the "proof of correctness" of the method. The primary advantage of this method is that we are certain to advance time every step, so we are not potentially "wasting" steps as when we use even timestepping; this comes at the cost of having to compute which transitions are possible from the current state. For our SPM, a given state has $\mathcal{O}(L)$ possible transitions, thus the time complexity of a single timestep is $\mathcal{O}(L)$; note that our method with evenly-spaced timesteps has constant time complexity, but the size of each timestep scales as $\mathcal{O}(L^{-1})$; thus we're not actually losing as much as it appears by using variable timesteps. Furthermore, there is the possibility that the process by which we calculate which transitions are possible could be performed

Figure 1.1 *An illustration of the suggested method for choosing a successor state in the n-fold way. Reproduced from [6].*



in parallel, and so the walltime cost of a single timestep in the n-fold way can end up comparatively cheap. The process of choosing a successor state once the options are found involves performing the equivalent of a search, and therefore takes $\mathcal{O}(\log L)$ time and so should be insignificant.

1.2 Implementation of Monte Carlo Methods

1.2.1 Our Implementation of a Metropolis-Hastings Algorithm with Evenly-Space Timesteps

We have written a Fortran code which implements the algorithm in Sec 1.1.2. This is stored in ¡location of code¿. This programme initialises the system to have a particle density equal to the average of the two desired boundary densities, and then proceeds in a manner extremely faithful to the simple accept/reject algorithm.

1.2.2 `KMCLib`

The vast majority of our Monte-Carlo calculations have been performed using the n-fold way, described in Sec. 1.1.3. This is implemented for continuous-time

Markov processes on crystalline lattices (of which the SPM is an example) in a software package called `KMCLib`, documented at [3] developed by Dr Mikael Leetmaa.

`KMCLib` is a Python-wrapped C++ package. This means that the frontend, where one specifies the system to be simulated, the data to be recorded, and how the simulation is run, is written in a Python script; then, when this script is run, it executes C++ code in order to represent the system and actually carry out the desired operations. Furthermore, `KMCLib` can perform calculations in parallel if so desired. Whilst there exist examples [2, 4] of kinetic Monte-Carlo codes other than `KMCLib`, we chose to use that one due to our familiarity with all of the languages involved, and preference for a Python frontend.

Of course, setting up different calculations which vary different parameters or measure different things require different scripts. Going through every script we wrote individually would cause this thesis to be around twice as long and three times more dull; therefore we have instead chosen to focus upon a single set of codes designed to perform a particular calculation, which we have annotated and included here ¡location¿; the intention is that a reader wishing to reproduce any of our results could do so by performing a few simple modifications to the code listed there. A more comprehensive codebase is stored at ¡location¿, but this is sparsely annotated working code, and so might not be very helpful.

The exemplary code which actually interfaces directly with `KMCLib` is contained within `concFlow.py`. This script takes in several command line inputs. These provide the parameters for a simulation of the SPM, with the desired value of λ, system size and boundary conditions. It then sets up the representation of the system configuration and the means to enumerate possible transitions and their associated rates, as is necessary to implement the n-fold way. The initial configuration is generated by randomly inserting particles into the system until its density is equal to $\frac{1}{2}(\rho_0 + \rho_L)$; we then perform N_{eq} KMC steps in order to equilibrate the system (in case the initial configuration we chose was highly deviant from the norm for the prescribed parameters). The actual measurements are performed by time-averaging values for system quantities (e.g. the number of particles entering the system at one end) over N_{meas} steps, relaxing the system (in other words, performing steps but taking no measurements) for N_{req} steps, and then repeating this process N_{pass} times. This way, we can generate N_{pass} time-separated observations of, say, the total current through the system, and because we are relaxing the system between measurement runs we should not

have to worry too much about the results being unduly correlated with each other, (assuming we set N_{req} high enough). Thus, we supply `concFlow.py` with the following parameters as command line inputs:

1. The particle reservoir concentration at one end of the domain, ρ_0.
2. The particle reservoir concentration at the other end of the domain, ρ_L.
3. The value of λ to use in the simulation.
4. The system size, L.
5. The interval between measurements performed by the analysis routines, N_{anal}. This should be set to 1 in order to measure the current.
6. The number of equilibration steps, N_{eq}.
7. The number of analysis steps per pass, N_{meas}.
8. The number of reequilibration steps per pass, N_{req}.
9. The total number of passes, N_{pass}, which give separate sets of observations, performed during this calculation.
10. A timescale, δt, which indicates how often to evaluate, and to what accuracy to record times, when measuring the number of particles in the system. This should probably be small compared to the expected KMC timestep size.

In terms of the output of the code, it produces a short file summarising the input parameters, some trajectory dump information (usually redirected to `/dev/null` in order to save hard memory, which is often in short supply), as well as data taken by measurement routines. We nominate, from a suite of possible routines, which measurements we would like it to take during analysis phases. Note that in our calculations, we do not consider any quantity's value on during particular KMC steps; rather, we always average our quantities over some amount of time. This is partly because some of the quantities we are interested in do not really have any value during a single timestep (e.g. the flux of particles through one of the boundaries), and also because the amount of time spent in particular configurations could potentially vary wildly between configurations. The amount of time spent in a particular configuration in the n-fold way is drawn from an exponential distribution with decay rate k_{tot}, as we saw in Sec. 1.1.3; thus, one

could easily imagine a situation in which the transition time varies wildly. For example, say we have a system with very low λ. If this system was quite full, there would be few transitions possible, and those possible transitions would likely occur with low rates, therefore the kmc timesteps would tend to be very long. However, later during the same simulation we could find ourselves in a situation where the system is less full, and so more transitions can occur, and generally with much higher rates, leading to much shorter timesteps. Thus, we shouldn't really treat particular quantities derived from these configurations with an equal footing, as the amounts of time the system spends in each are so very different.

The precise nature of our time-averaging depends a little on the measurement in question. The types of measurements we usually perform are the following, where T is the total time elapsed during our N_{anal} step measurement run:

- **Current** We count the total number of particles which enter or leave a given boundary over the course of the measurement run. Let the number of particles entering and exiting at the 0 boundary be u_0 and w_0 respectively, and likewise for the L boundary with u_L and w_L. Then

$$J = \frac{u_0 + w_L - u_L - w_0}{2T} \tag{1.2}$$

  should be a good estimate of the total current through the system during that time period.

- **Block Size Distribution** In one dimension, we can look at a configuration and count how many contiguous runs ("blocks") of particles there are of different sizes (e.g. size 1 means a single particle sandwiched between adjacent vacancies). We can find the distribution of block sizes, weight it by the length of the associated kmc timestep, and then add this to a running total. If we do this over our N_{meas} analysis steps and then normalise, we can build a histogram of the block sizes during that time period.

- **Particle Density** Similarly, we can count the total number of particles in the system, weight it by the length of the kmc timestep, and then use this to build another histogram of the system particle density. By keeping track of particles entering and leaving the system, it would be possible to code this very efficiently to take $\mathcal{O}(1)$ time; however, as our routine to detect block sizes scans through the system and counts as it goes along, we have just opted for a simple $\mathcal{O}(L)$ scan of the whole system for our density

measurement as well.

Using these analysis routines, we can generate time-averaged values for particle density histograms, the block size distribution and the current. By calculating N_{pass} separate instances of these observables, we get N_{pass} samples from the relevant distributions, and from there we can probe the statistics of these variables.

1.2.3 Managing `KMCLib` Calculations in Parallel

Of course, it is one thing to have a code which can run on a laptop to produce the output of a particular simulation over the course of a day. It is quite another undertaking to run thousands of separate calculations in order to map out parameter spaces and compute derived quantities such as the diffusion coefficient.

We have been running our calculations on Edinburgh University's `Eddie3` computing cluster. This machine does not boast the high level of processor interconnection density of `ARCHER` or the extremely high working memory of `DiRAC`; however, for the purposes of our calculations it turns out that we need neither. The KMC algorithm only stores a single state of the system under simulation at any given time, therefore its space complexity only scales as $\mathcal{O}(L)$. Furthermore, whilst `KMCLib` can be run in parallel mode in order to take advantage of a multithreaded environment, this isn't actually an advantage when we wish to run very large numbers of separately-parametrised calculations, as the total amount of CPU time required remains the same, whilst incurring additional overheads associated with parallelism. Therefore, we have used a single-threaded environment for all of the calculations featured in this thesis.

In order to set up a batch of calculations, we use the following procedure, implemented by the codes stored in `kmc/1d/` within ¡location¿:

1. Created a batch of input files, in the subdirectory `jobInputs/`. In our setup, we require that files titled `testInput.`i are generated, with $i \in [1, n]$ where n is the total number of calculations to be performed. These input files are typically generated by a code such as `lambdaFlucCreator.py`, and typically the parameters which determine the overall structure of the system (e.g. L, the system size) will be held constant across calculations, whilst parameters such as the stickiness or the boundary densities will be varied

between them. These input files contain a single line of code, which will be appended to *python* and called in the command line.

2. In order to actually perform the calculations, we submit them as `gridengine` batch jobs. We then run the script `kmcSubmit.sh`, which submits the nominated tasks whose input files are in `jobInputs/`, using the scripts `kmcJobArray.sh` and `initKmc.sh` for intermediate steps. `kmcSubmit` is also the place where we specify calculational parameters, such as maximum memory usage, maximum runtime, etc; these will be taken into consideration by `gridengine` when it comes to scheduling these calculations, so it is important that the maxima be relatively tight upper bounds, otherwise the calculation priority will be extremely low, assuming that the cluster in question is being simultaneously for many other calculations.

3. The jobs will then be executed, in their own time. The results will be placed in the location nominated by the input files.

4. Once the run is complete, and the data has been saved, we are then ready to process it into a more useful format, in our case a data file which can be interpreted by Mathematica, the programme we used for most of our analysis and graphing. This is done using a script such a `lambdaPostProc`. Note that such a script needs to be able to handle the fact that data may not be produced for some of the calculations (around 5% in our experience). The most likely source of the problem seems to be an issue with type conversion between Python and C++, which only seems to become a significant issue in larger calculations.

Note that throughout our calculations, we have stored data in a human-readable format, instead of in a more compressed binary format. This is because we believe that the benefit of having a human-readable format, and therefore a much greater ability to look through data and check the output, outweighs the associated memory cost (around a factor of 10 or so), especially given that any memory reduction attained due to such a format change wouldn't change what was feasible in terms of what we can afford to store.

1.3 1D Calculation Results

Most of the calculations we have performed are for the 1-dimensional version of the SPM. As we have already performed calculations relating to the behaviour in Chaps. **??** and **??**, we already have results which can be compared directly to our Monte Carlo calculations; thus, we will be plotting them together wherever we feel it is appropriate.

In terms of what to calculate, we can use our previous calculations to motivate our future ones. Using Monte Carlo, we can calculate the quantities we already investigated, such as current and particle density. In addition, we can also look at the time-evolution of particular configurations, to gain insight into the mechanisms by which particles are transmitted during flow. This is something our MFT says essentially nothing about, and our TRM calculations are too small to see anything meaningful in this regard.

1.3.1 Flow Patterns

First, let's talk about these flow patterns. Of the two methods available, we have only implemented the visualisation of flow using the KMC calculations. This is because the n-fold way produces a more "realistic" trajectory for the SPM, in the sense that the trajectory is an exact reproduction of the behaviour of the continuous-time Markov system; our other method, the simpler accept-reject algorithm, should reproduce correct behaviour when long-term time averages are taken, but might behave badly over small times. For example, in this method, there is a minimum timescale over which ANY particle can move, which is not the case for KMC, with it's random timestepping.

Whilst this random timestepping makes for a more formally correct trajectory, it does make it a little more difficult to produce visualisations of particle trajectories. Our method for overcoming this is as follows:

1. Calculate a trajectory, with whatever choice of system parameters we like. The most condensed way to do this in terms of hard memory during calculation runs is to keep track of which slots' occupancies change state at each timestep, and retaining the timestamp of each timestep.

2. We from this occupancy data, we can use linear interpolation in order to

assign a continuous occupancy variable to all sites at all times. Between timesteps, all sites not directly involved with a transition would have an occupancy of 1 or 0, and those involved in the transition would smoothly switch from occupancies of 0 to 1 and vice-versa.

3. We can now make a new, evenly-stepped grid of space and time where we use even timesteps. We can integrate the interpolated KMC data over the time spacings of the new grid in order to find average occupations over the specified timeframe.

Thus, we can produce a spacetime diagram which shows the motion of particle density through the system over time. Clearly this method depends upon supplying a timescale δt over which we perform our averaging; a large δt will ignore most of the specifics of the motion, whereas a very small value will reveal a system in which nothing happens most of the time, except when this calm is punctuated by transitions.

Flow Visualization for Sticky Particles

We have performed calculations which illustrate the behaviour of an SPM system in 1 dimension, displayed in Fig. 1.3. These plots were generated by simulating SPM systems with $L = 512$ and boundary conditions $(\rho_0, \rho_L) = (0.75, 0.25)$, with $\lambda \in \{0.05, 0.15, 0.35\}$. We simulated over differing numbers of kmc steps, N_{steps}, in the end performing $N_{\text{steps}} \in \{8192, 262144, 2097152, 8388608\}$ steps respectively. Once the data was collected, we could lookup the total elapsed time in each simulation, T, and then divide that time by 512 in order to obtain a discretization timescale δt, as required by our method for visualising flow patterns as described above in 1.3.1. In this way, we can visualise what the flow looks like over different timescales for different values of λ. Note that the average size of a KMC step does depend implicitly on the value of λ; thus, the timescales portrayed in Fig. 1.3 are not consistent between the plots, as the timesteps are of lengths $\sim \{6, 2, 1\} \times 10^{-2}$s for $\lambda \in \{0.05, 0.15, 0.35\}$ respectively. In all of these plots, dark tones represent low time-averaged particle occupation, whilst light tones represent high time-averaged particle occupation. Each of the systems here show the behaviour of particles with low-λ, so we are in the sticky regime; however, all of our previous results indicate that there should be rather large differences in behaviour as we switch from relatively weak stickiness (here embodied by $\lambda = 0.35$) to strong stickiness (portrayed by the $\lambda = 0.05$ situation). The images used to create

Figure 1.2 *Spacetime plots of the particle flow in 1 spatial dimension, as described in 1.3.1. In each case, the x-axis represents time and the y-axis space. The higher-density boundary is the one at the top of each image.*

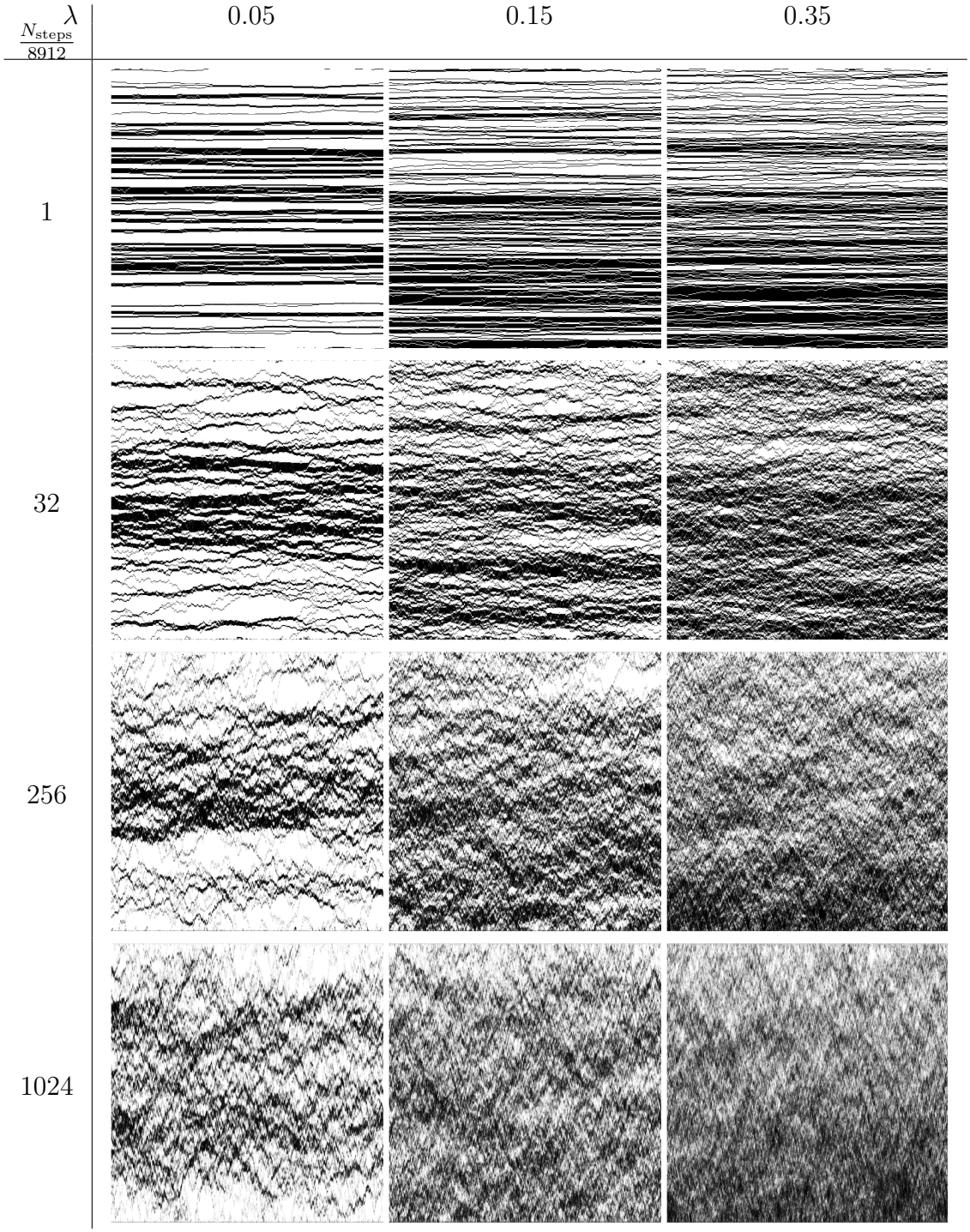


Fig 1.3 are relatively high-definition, which should readers using the digital copy to zoom in order to see the fine details. Our principle observations are as follows:

- At the shortest timescale, one can quite clearly see the motions of individual particles. As one might expect, they are less likely to be seen unbound from neighbours in the lower-λ systems than the higher-λ ones. In the low-λ regime, we are much more like to see big blocks of particles, possibly containing a low concentration of mobile vacancies.

- Focussing now on the $32\times$ longer intermediate timeframe, we can see that in the extremely low-λ situation we have blocks of particles separated by voids of vacancies. These voids contain a dilute gas of particles. Over these longer timescales, we see that the blocks of particles do in fact slowly migrate around the system, occasionally breaking apart or reforming during their travels. Also notice that the voids are more likely to be found towards the centre of the system than adjacent to the boundaries; this is presumably a response to the way that we have implemented the boundaries. The chemical potential (Fig **??**) for small-λ is minimised for high density, thus a boundary held at any density should be expected to in practise generate a high local density regardless of the density it is set to emulate.

- Meanwhile for higher-λ, we see a "tissue paper" pattern over these intermediate timescales; the system is similar to a gas of randomly-walking particles, but there is a little bit more short-range correlation than that, hence the observed texture in the image.

- Now looking over longer timescales, we see that for the lowest-λ it is in fact the case that the voids towards the centre of the system do in fact appear and disappear over time. Given that we know that there are still (small) flows occurring in this regime (see Sec 1.3.2), it is likely that when these voids are created and destroyed, there are small overall biases in terms of which void boundaries more particles are extracted from or shed into. We suspect that this is the primary mechanism by which transport across the system is achieved in this regime. Meanwhile, the higher-λ systems are becoming something closer to a continuous grey gradient from the top boundary to the bottom, suggesting that the overall transport is more diffusive in nature.

Repulsive Particles

Of course, we can do similar calculations with repulsive particles, for which $\lambda > 1$. Of the most interest is the extreme case in which $\lambda >> 1$, when we should expect that particles have an almost explosive tendency to separate if brought together. We have performed such a calculation

1.3.2 Scans Through λ with Constant Boundary Densities

1.3.3 Varying λ and Boundary Density Difference Together

1.3.4 Varying the Boundary Densities with Constant λ

1.3.5 Diffusion Coefficient

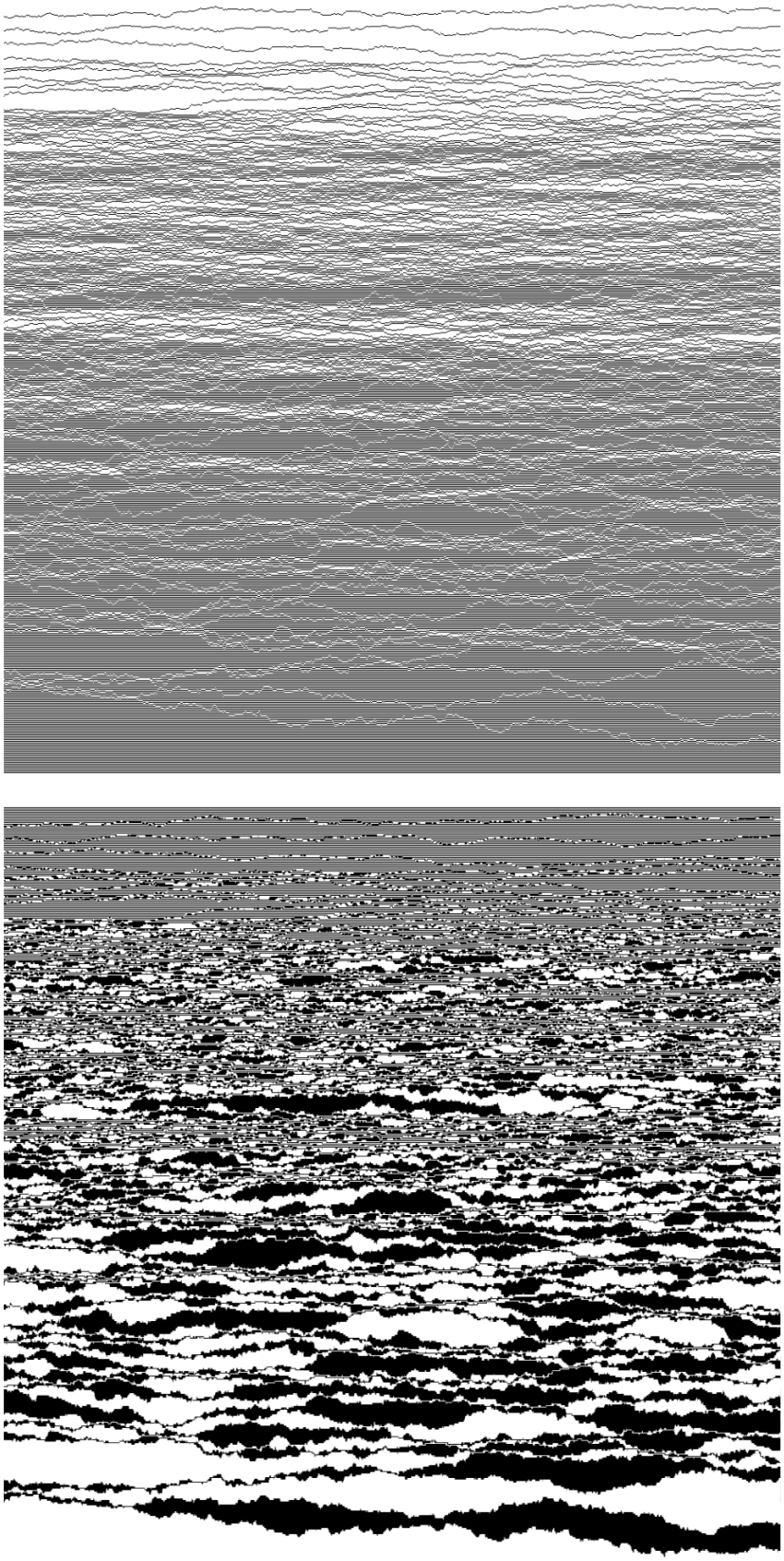
1.4 2D Calculation Results

1.4.1 Calculational Choices

1.4.2 Results

1.5 Conclusions

Figure 1.3 *pff*



Appendix A

Code Listings

A.1 1d Ising Correlation Functions

This `Python` script computes the probability of a site being occupied l lattice spacings away from an occupied site. It requires the system size L and the number of particles N as inputs. The output is saved in a file called `corrFnResults.m`, which is formatted so that it may be used by `Mathematica`.

```
import copy
import sys

def configMake(L, N, prevList, totList):
    if L==1:
        endList = [copy.deepcopy(prevList), N]
        totList.append(unfold(endList))
        return [N]
    if N==0:
       return configMake(L-1, 0, [copy.deepcopy(prevList), 0], totList)
    if L==N:
        return configMake(L-1, N-1, [copy.deepcopy(prevList), 1], totList)
    return [configMake(L-1, N, [copy.deepcopy(prevList), 0], totList),
    configMake(L-1, N-1, [copy.deepcopy(prevList), 1], totList)]

def adjSum(candList):
    listLen = len(candList)
    total = 0
    for index in range(0, listLen):
        total += candList[index-1]*candList[index]
    return total

def unfold(candList):
    if isinstance(candList, list):
        if len(candList)==2:
            return unfold(candList[0])+unfold(candList[1])
```

```
        if len(candList)==1:
            return candList
        if len(candList)==0:
            return []
    return [candList]

def listCollate(candList):
    maxItem = 0
    for index in candList:
        if index > maxItem:
            maxItem = index
    outPut = []
    for size in range(0, maxItem+1):
        numCounts = 0
        for index in candList:
            if index == size:
                numCounts += 1
        outPut.append((size, numCounts))
    return outPut

def genCorrFn(L, N):
    totList = []
    allStates = configMake(L, N, [], totList)
    restStates = []
    weightList = []
    maxAdj = 0
    for state in totList:
        if state[0]==1:
            restStates.append((state, adjSum(state)))
            if restStates[-1][1]>maxAdj:
                maxAdj = restStates[-1][1]
            weightList.append(restStates[-1][1])
    partFnList = listCollate(weightList)
    print(partFnList)
    partitionFn = "("
    for pair in partFnList:
        partitionFn += str(pair[1])+"␣Exp["+str(pair[0]-maxAdj)+"b]␣+␣"
    partitionFn += "0)"
    print(partitionFn)
    finalOut = "{"
    for shift in range(0, L-L/2):
        tempList = []
        for config in restStates:
            if config[0][shift] == 1:
                tempList.append(config[1])
        stateDist = listCollate(tempList)
        outSum = "{"+str(shift)+",␣("
        for pair in stateDist:
            outSum += str(pair[1])+"␣Exp["+str(pair[0]-maxAdj)+"b]␣+␣"
        outSum += "0)/"+partitionFn+"}"
        finalOut += outSum
        if shift != L-L/2-1:
            finalOut += ",␣"
    finalOut+="}"
    return finalOut

L = int(sys.argv[1])
```

```
with open("corrFnResults.m", 'w') as f:
    f.write("{")
    for n in range(2, L-2):
        f.write("{"+str(n)+"/"+str(L)+",␣"+genCorrFn(L, n)+"},␣")
    f.write(genCorrFn(L, L-2) + "}")
```

A.2 n-Dimensional Continuum-Limit MFT

This Mathematica script computes the current which flows between two adjacent sites (offset in the e_1 direction) in the MFT of the n-dimensional SPM; due to symmetry, this tells us what happens in an arbitrary direction. In this case n is set to 3, but it still works if changed to any positive number.

```
n = 3;
i = 1;
zero = 0*UnitVector[n, 1];
e[i_] := UnitVector[n, i];
Hess = Table[
   Piecewise[{{d2p[j, i], j > i}}, d2p[i, j]], {i, 1, n}, {j, 1, n}];
Jacob = Table[dp[i], {i, 1, n}];
p[x_] := p0 + Jacob.x + 1/2 x.(Hess.x);
rightJ = 1/
    t0 (1 - p[1/2 a e[i]]) p[-(1/2) a e[i]] (1 -
     z p[-(3/2) a e[i]]) Product[
    Piecewise[{{(1 - z p[-a e[j] - 1/2 a e[i]]) (1 -
          z p[a e[j] - 1/2 a e[i]]), j != i}}, 1], {j, 1, n}];
leftJ = 1/t0 (1 - p[-(1/2) a e[i]]) p[
    1/2 a e[i]] (1 - z p[3/2 a e[i]]) Product[
    Piecewise[{{(1 - z p[-a e[j] + 1/2 a e[i]]) (1 -
          z p[a e[j] + 1/2 a e[i]]), j != i}}, 1], {j, 1, n}];
fullJ = rightJ - leftJ + O[a]^3;
FullSimplify[fullJ]
```

Bibliography

[1] Bortz, A., M. Kalos, and J. Lebowitz. "A new algorithm for Monte Carlo simulation of Ising spin systems." *Journal of Computational Physics* 17, 1: (1975) 10 – 18. http://www.sciencedirect.com/science/article/pii/0021999175900601.

[2] Hoffmann, M. J., S. Matera, and K. Reuter. "kmos: A lattice kinetic Monte Carlo framework." *Computer Physics Communications* 185, 7: (2014) 2138–2150.

[3] Leetmaa, M., and N. V. Skorodumova. "KMCLib: A general framework for lattice kinetic Monte Carlo (KMC) simulations." *Computer Physics Communications* 185: (2014) 2340–2349.

[4] Plimpton, S., C. Battaile, M. Chandross, L. Holm, A. Thompson, V. Tikare, G. Wagner, E. Webb, X. Zhou, C. G. Cardona, et al. "Crossing the mesoscale no-mans land via parallel kinetic Monte Carlo." *Sandia Report SAND2009-6226* .

[5] Prados, A., J. J. Brey, and B. Sánchez-Rey. "A dynamical monte carlo algorithm for master equations with time-dependent transition rates." *Journal of Statistical Physics* 89, 3: (1997) 709–734. https://doi.org/10.1007/BF02765541.

[6] Voter, A. *Radiation Effects in Solids edited by KE Sickafus and EA Kotomin*. Springer, NATO Publishing Unit, 2005.