Critical Properties of Driven Dissipative Systems Through the KPZ equation

by

Thomas Mellor

Supervisors: Dr Marzena Szymańska and Dr Alejandro Zamorae

March 2018

Abstract

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgements and the people to thank go here, don't forget to include your project advisor...

Contents

Chapter 1

Introduction and Background

1.1 Introduction

Condensation–the phenomenon whereby a macroscropic number of particles occupy the ground state a system–has been of theoretical and experimental interest for physicists for almost a century. Bose Einstein Condensation (BEC), perhaps the most familiar example, was predicted to occur for a system of non-interacting Bosons at a sufficiently low temperature. This was observed in ultrocold atoms in the 1990s [**PhysRevLett.75.3969**], but there are many other examples of condensation, such as Bardeen Cooper Srhrieffer superconductivity [3]. The excitations can even be quasi particles, for example in the case of magnons–electronic spin waves [**magnon**].

More recently, another type of quasiparticle, the exciton-polariton, formed from the coupling of electron-hole (exciton) pairs and photons, has been observed to undergo condensation in microcavities [5]. However, to maintain the condensate, the continual pumping of a laser is required, inhibiting the system from attaining equilibrium. Because it cannot, many questions still remain as to the exact nature of the condensate and how the understanding of condensation derived from equilibrium physics must be modified.

The aim of this project is to address some of these issues by confirming behaviour of the condensate expected from a theoretical analysis. In particular, when describing the condensate as a macroscopic wavefunction $\rho(r,t)e^{i\theta(r,t)}$, it is found that the long range behaviour of the condensate can be described strictly by the evolution of the phase θ through the Karder-Parisi-Zhang (KPZ) equation [14, 2]. Originally used to describe height growth[**PhysRevLett.56.889**], in its current context the compactness of the θ, i.e. that it is limited between 0 and 2π, allows for the existence of topoligal defects which are crucial in explaining the physics.

By employing techniques from the renormalistion group, it is found that depending on the parameters of the equation there are effectively two regimes [14, 2]. The first is described by Berezinskii-Kosterlitz-Thouless (BKT) physics, which is that of the XY model. The second, the KPZ phase. During the porject, the evolution of the condensate phase was simulated through the KPZ equation, to confirm that in the different regime the behaviour was as expected.

1.2 Exciton-Polariton Condensates

Exciton-polaritons are formed in a laser illuminated microcavity–an example of a so-called drive dissipative system: the laser contiually *drives* the microcavity and photons inside the cavity escape or *dissipate* continually out of the microcavity. As such, the system is distinctly non equilibrium due to the external influence.

To create these quasiparticles, a thin superconducting surface is sandwitched between two reflective Bragg mirrors on either side. The material is then illuminated with a laser, exciting an electron, which, because of the Coulombic interaction, forms a hydrogen-like bound state with the remaining hole: an exciton. The excitons are also confined to very narrow widths in the material due to the alternating doping that varies the band gap, making the system two dimensional. This is illustrated in Figure. 1.1. Perpendicular to the surface [9], the photon wavevector is restricted due to the finite length L of the z plane, the possible values given by $2\pi N/L$ where N is an integer. The possible frequencies are therefore

$$\omega = \frac{c}{n}\sqrt{k^2 + (2\pi N/L)^2}$$

where n is the material refractive index and k is the magnitude of the 'in-plane' wavevector. If this is small, we may expand this dispersion to obtain

$$\hbar\omega = \hbar\omega_0 + \hbar^2 k^2/2m$$

where $m = \hbar(n/c)(2\pi N/L)$ is the effective mass of the photon and is set by the cavity length. The energy $\omega_0 = (c/n)(2\pi N/L)$ at the bottom of the dispersion should be near to that of the energy rquired to create an exciton. This fixes the photon mass to be $m \sim 10^{-4} m_e$ while the exciton mass is typically $\sim m_e$, meaning we can neglect the exciton dispersion compared to that of the polariton.

Although the mirrors are not perfectly reflective, with the lifetime of the polaritons being limited to $\sim 200ps$ in current experiments [11], if the rate of recombination and electron excitation exceeds that of photon dissipation, the excitons are able to couple with the photons. To describe this [**RevModPhys.85.299**], the Hamiltonian of the photons is

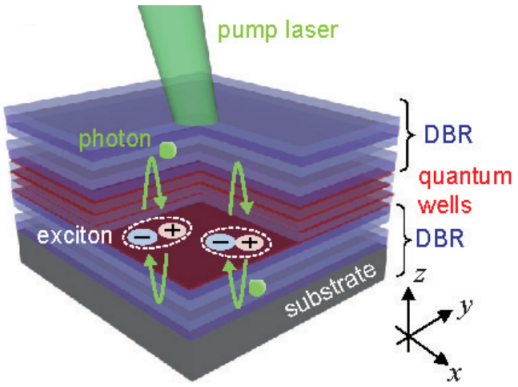

FIGURE 1.1: The exciton-polariton microcavity. Superconducting material of with an alternating band gap is sandwiched between Bragg mirrors. A laser illuminates the semiconducting material which creates excitons that are confined in a 2D plane due to the doping. With sufficient photon lifetime, the photons can couple with the exciton, effectively becoming a quasi particle known as a exciton-polariton [5].

given by

$$\mathcal{H}_{\text{cav}} = \int \frac{\mathrm{d}^2\boldsymbol{k}}{(2\pi)^2} \sum_\sigma \hbar\omega_{\text{cav}}(\boldsymbol{k}) \hat{a}^\dagger_{C,\sigma}(\boldsymbol{k}) \hat{a}_{C,\sigma}(\boldsymbol{k})$$

where $\hat{a}^\dagger_{C,\sigma}(\boldsymbol{k})$ is the Bosonic creation operator for a photon with in-plane wavevector $\boldsymbol{k}$ and polarisation σ. The exciton Hamilotian is given by

$$\mathcal{H}_{\text{exc}} = \int \frac{\mathrm{d}^2\boldsymbol{k}}{(2\pi)^2} \sum_\sigma \hbar\omega_{\text{exc}}(\boldsymbol{k}) \hat{a}^\dagger_{X,\sigma}(\boldsymbol{k}) \hat{a}_{X,\sigma}(\boldsymbol{k})$$

with analogous definitions. The interaction Hamiltonian is then

$$\mathcal{H}_{\text{int}} = \int \frac{\mathrm{d}^2\boldsymbol{k}}{(2\pi)^2} \sum_\sigma \hbar\Omega_R [\hat{a}^\dagger_{X,\sigma}(\boldsymbol{k}) \hat{a}_{C,\sigma}(\boldsymbol{k}) + \hat{a}^\dagger_{C,\sigma}(\boldsymbol{k}) \hat{a}_{X,\sigma}(\boldsymbol{k})]$$

describing the interconversion of photons and excitons, with the strength given by the Rabi frequency Ω_R. Defining

$$\hat{\boldsymbol{a}}_\sigma(\boldsymbol{k}) = \begin{pmatrix} \hat{a}_{C,\sigma}(\boldsymbol{k}) \\ \hat{a}_{X,\sigma}(\boldsymbol{k}) \end{pmatrix}$$

we can write the total Hamiltonian by

$$\mathcal{H} = \int \frac{\mathrm{d}^2\boldsymbol{k}}{(2\pi)^2} \sum_\sigma \hat{\boldsymbol{a}}_\sigma(\boldsymbol{k})^T M(\boldsymbol{k}) \hat{\boldsymbol{a}}_\sigma(\boldsymbol{k})$$

where $M(\boldsymbol{k})$ is given by

$$\begin{pmatrix} \hbar\omega_{\text{cav}}(\boldsymbol{k}) & \hbar\Omega_R \\ \hbar\Omega_R & \hbar\omega_{\text{exc}}(\boldsymbol{k}) \end{pmatrix}.$$

Diagonalising, we see that the frequencies of the eigenfunctions, which are linear combination of the exciton and photon creation operators (exicton-polaritons), is given by

$$\omega_{(\mathrm{UP,LP}),\sigma}(\boldsymbol{k}) = \frac{\omega_{\mathrm{cav},\sigma}(\boldsymbol{k}) + \omega_{\mathrm{exc},\sigma}(\boldsymbol{k})}{2} \pm \left[\Omega_R^2 + \left(\frac{\omega_{\mathrm{cav},\sigma}(\boldsymbol{k}) - \omega_{\mathrm{exc},\sigma}(\boldsymbol{k})}{2}\right)^2\right]^{1/2}$$

where 'UP' and 'LP' stand for upper and lower polariton, respectively. As mentioned, the exciton dispersion is very flat compared to the photon. Thus, the polariton dispersion is almost totally dependent on the photon. At low frequencies the LP dispersion is approximately quadratic but has a point of inflection as the momentum is increased and approaches the exciton dispersion. During laser puming it is the LP modes that are occupied.

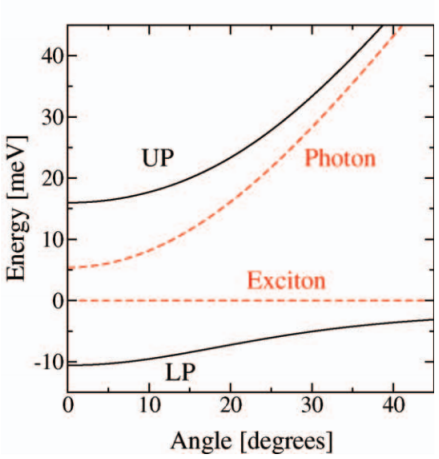

FIGURE 1.2: A dispersion graph for the exciton-polariton system. The angle corresponds to the momentum of the polariton. LP and UP are upper and lower polariton, respectively, and are the two eigenvalues for the Hamiltonian. Due to the photon's low effective mass compared to the exciton, the dispersion for the polariton is almost totally dependent on the photon dispersion. In experiments it is the LP modes that are become occupied. Its dispersion relation looks quadratic at low momentum but has a point inflection, tending to the exciton dispersion, at high momentum. [9]

One can obtain a wide range of behaviour in the microcavity by varying the method of illumination. The momenta of the created polaritons in the microcavity plane depends on the $\sin\theta$ where θ is the angle of the laser with respect to the plane [9]. To obtain a condensate where a low momentum mode is macroscopically occupied, there a few standard methods, two of which we will describe, however in general they depend on the pump strength, and, after a level has been passed, accumulation of the low momentum states increases dramatically.

The first of these is the incoherent pump. Here the laser excites high momentum polaritons. These can decay through phonon scattering at first, but as the dispersion steepens this process becomes more and more inefficient and this creates a bottle-neck

effect near the inflection. However, if the density of the polaritons is high enough then polariton-polariton scattering can take over, scattering the polaritons to high and low momentum states, the high momentum states then repeating this process. By ensuring the polariton is more exciton-like, so that their interactions are strengthened, and that the density is high enough, this process leads to the formation of the condensate [9].

The second method is the coherent pumping scheme which leads to opitical parametric oscillation. This is a 'cleaner' method. Here the pump is tuned at the inflection point and so any subsequent polariton interaction can cause scattering of two low and high momentum states which conserve both momentum and energy, known as the signal and idler modes, respectively. The idler can then decay to the lower momentum state via phonon emission. Beyond a threshold pump power the density of the low and high momentum states increases [9].

These techniques allow experimenters to observe a BEC, for example in [12], and illustrated in Figure. 1.3.

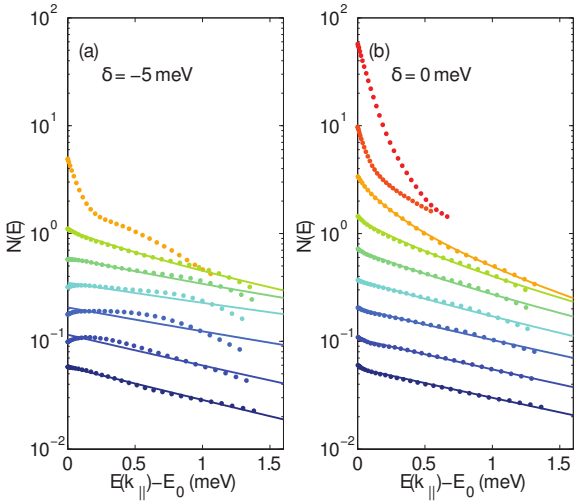

FIGURE 1.3: From [12], the energy distribution of polaritons at two detunings δ (which is the energy difference between the cavity resonance and exciton energy at $k = 0$) and various pump powers, shown here as increasing when going from blue to red. For zero detuning at a sufficient pump power, the distribution unambiguously points to Bose-Einstein statistics.

In general, however, condensation is not necessarily indicative of the system being a BEC. Various regimes are possible where the system behaves more like a laser, a distinctly non-equilibrium system, or like a BEC as above. It is thus most fruitful to consider the system as lying somewhere on a spectrum from the laser of BEC system [5]. When condensation occurs, it is possible consider a macroscopic wavefunction $\psi(\boldsymbol{r}, t)$ of the system, which can also be written as $\sqrt{\rho(\boldsymbol{r}, t)}e^{i\theta(\boldsymbol{r},t)}$ where $\rho(\boldsymbol{r}, t)$ is the (real) amplitude and $\theta(\boldsymbol{r}, t)$ is the phase.

We will look more in detail about the dynamics and phases of this condensate, but first we must discuss a related model.

1.3 XY model and the BKT transition

The two dimensional XY model consists of a finite lattice of unit length spins that can rotate freely rotate the plane of the lattice [1]. The Hamiltonian describing the system is given by

$$\mathcal{H} = -J \sum_{\langle i,j \rangle} \sum_{\{x,y\}} \boldsymbol{s}_{i,j} \cdot \boldsymbol{s}_{i+x,j+y} = -J \sum_{(i,j)} \sum_{\{x,y\}} \cos(\theta_{i,j} - \theta_{i+x,j+y})$$

where i, j signifies point (i, j) and $\{x, y\}$ are the pairs $\{0, 1\}\{0, 1\}$, i.e. nearest neighbours, and J is a constant. Periodic boundary conditions have been employed. We can form the continuum long wavelength (slowly varying) version of this equation by expanding in a Taylor series and keeping only terms to second order (as higher order terms will have Fourier transformed components greater than k^2 and we are ignoring those)

$$-J \sum_{(i,j)} \sum_{\{x,y\}} \left(1 - \frac{a^2}{2} \frac{(\theta_{i,j} - \theta_{i+x,j+y})^2}{a^2}\right).$$

The first term is a constant that can be ignored in what follows. Taking $a \to 0$, we have that $\sum_{(i,j)} a^2 \to \int \mathrm{d}x\, \mathrm{d}y$ and the terms in brackets become partial derivative, i.e. we obtain

$$\frac{J}{2} \int \mathrm{d}x\, \mathrm{d}y\, (\nabla\theta)^2$$

where now θ is a continuous variable. The Hamiltonian under consideration is invariant under global SO(2) transformation, i.e. in the replacement of $\theta \to \theta + \delta\theta$ as the change in cancelled in the cos terms. Because the system is two dimensional and interacting, it is impossible for the symmetry to be spontaneously broken, that is, the system cannot occupy a state in which all the spins point in the same direction [7]. To demonstrate this, we will show that $\left\langle (\theta(\boldsymbol{r})\theta(\boldsymbol{0}))^2 \right\rangle$ diverges as $r \to \infty$, which means the fluctuations between the origin and $\boldsymbol{r}$ diverges as $r \to \infty$, so there can be no long range order [**8.334 lecture notes**]. We note that the probability of a configuration is given by

$$\mathcal{P}[\theta(\boldsymbol{r})] = \exp\left[-\beta J/2 \int \mathrm{d}^2\boldsymbol{r}\, (\nabla\theta)^2\right]$$

If instead we decompose θ into its Fourier decomposition $\theta(\boldsymbol{r}) = \sum_{\boldsymbol{k}} \theta_{\boldsymbol{k}} e^{i\boldsymbol{k}\cdot\boldsymbol{r}}$ where the sum is over the Brillouin zone, we then obtain for the probability of $\theta_{\boldsymbol{k}}$

$$\mathcal{P}[\theta_{\boldsymbol{k}}] = \exp\left[-\beta J L^2/2k^2|\theta_{\boldsymbol{k}}|^2\right]$$

where L^2 denotes the system size and we use $\theta_{\boldsymbol{k}} = \theta^*_{-\boldsymbol{k}}$. Now $|\theta_{\boldsymbol{k}}|^2 = (\operatorname{Re}\theta_{\boldsymbol{k}})^2 + (\operatorname{Im}\theta_{\boldsymbol{k}})^2$ and we can write the probability of a configuration as

$$\begin{aligned}\mathcal{P}[\{\theta_{\boldsymbol{k}}\}] &= \prod_{\boldsymbol{k}} \exp\left[-\beta J L^2/2k^2|\theta_{\boldsymbol{k}}|^2\right] \\ &= \prod_{\boldsymbol{k}, k_x>0} \exp\left[-2\beta J L^2/2k^2((\operatorname{Re}\theta_{\boldsymbol{k}})^2 + (\operatorname{Im}\theta_{\boldsymbol{k}})^2)\right]\end{aligned}$$

where in the second we restrict ourselves to the positive x and thus obtain completely decoupled Gaussian variables with variance $1/\beta J L^2$. Then the variances of $\theta_{\boldsymbol{k}}$ are given by

$$\langle\theta_{\boldsymbol{k}}\theta_{\boldsymbol{k}'}\rangle = \langle\operatorname{Re}\theta_{\boldsymbol{k}}\operatorname{Re}\theta_{\boldsymbol{k}'}\rangle + \langle\operatorname{Im}\theta_{\boldsymbol{k}}\operatorname{Im}\theta_{\boldsymbol{k}'}\rangle = \delta_{\boldsymbol{k},-\boldsymbol{k}'}/\beta J L^2$$

We wish to calculate

$$\left\langle\theta(\boldsymbol{r})\theta(\boldsymbol{r}')\right\rangle = \sum_{\boldsymbol{k},\boldsymbol{k}'} e^{i\boldsymbol{k}\cdot\boldsymbol{r}+i\boldsymbol{k}'\cdot\boldsymbol{r}'}\left\langle\theta_{\boldsymbol{k}}\theta_{\boldsymbol{k}'}\right\rangle = \frac{1}{\beta J L^2}\sum_{\boldsymbol{k}}\frac{e^{i\boldsymbol{k}\cdot(\boldsymbol{r}-\boldsymbol{r}')}}{k^2}$$

which can be replaced by an integral by $1/L^2\sum_{\boldsymbol{k}} \to \int \mathrm{d}^2\boldsymbol{k}\,/(2\pi)^2$ so that we obtain

$$\frac{1}{\beta J}\int\frac{\mathrm{d}^2\boldsymbol{k}}{(2\pi)^2}\frac{e^{i\boldsymbol{k}\cdot(\boldsymbol{r}-\boldsymbol{r}')}}{k^2}.$$

The negative of the integral is the Green's function of ∇^2:

$$-\int\frac{\mathrm{d}^2\boldsymbol{k}}{(2\pi)^2}\nabla^2\frac{e^{i\boldsymbol{k}.(\boldsymbol{r}-\boldsymbol{r}')}}{k^2} = \int\frac{\mathrm{d}^2\boldsymbol{k}}{(2\pi)^2}k^2\frac{e^{i\boldsymbol{k}.(\boldsymbol{r}-\boldsymbol{r}')}}{k^2} = \delta^2(\boldsymbol{r}-\boldsymbol{r}')$$

A solution V depends only on $|\boldsymbol{r}-\boldsymbol{r}'|$ as the integrand has an inner product with $\boldsymbol{k}$ which takes on all values, so we can find V with the divergence thoerem:

$$\int\nabla\cdot\nabla V\,\mathrm{d}^2\boldsymbol{r}\int r\nabla V\,\mathrm{d}\Omega = 2\pi r\nabla V = 1.$$

With $\nabla V = \partial V/\partial r$ this is $V(r) = \ln(r)/2\pi + C$. Now $\left\langle(\theta(\boldsymbol{r})-\theta(\boldsymbol{r}'))^2\right\rangle =$

$$2\left\langle\theta(\boldsymbol{r})^2\right\rangle - 2\left\langle\theta(\boldsymbol{r})\theta(\boldsymbol{r}')\right\rangle.$$

We also have $\langle\theta(\boldsymbol{r})\rangle = 0$ as it is the sum of the Gaussian distributed variables. Since the range of θ is no longer $[0, 2\pi]$ in this analysis, we interpret the mean of θ being 0 and

the variance infinite in that it can occupy any value with equal probability, as there is no preference for a direction. Clearly due to our simplification there will be undefined behaviour, so we set

$$2\left\langle\theta(\boldsymbol{r})^2\right\rangle - 2\left\langle\theta(\boldsymbol{r})\theta(\boldsymbol{r}')\right\rangle = \frac{\log(|\boldsymbol{r}-\boldsymbol{r}'|/a)}{\beta J\pi}$$

where a is of the order of the lattice spacing ensuring the log is unitless. We have ignored the infinite constant. From this result we see that the correlations diverge as $|\boldsymbol{r}-\boldsymbol{r}'|\to\infty$. For free, we can calculate the spin correlations in space $\langle\cos\theta(\boldsymbol{r}-\boldsymbol{r}')\rangle$. We use

$$\left\langle e^A\right\rangle = e^{\left(\langle A\rangle+\frac{1}{2}\langle A^2\rangle\right)}.$$

Thus, the correlations in space are of the form

$$\left\langle\cos\theta(\boldsymbol{r}-\boldsymbol{r}')\right\rangle = \mathrm{Re}\left\langle e^{i\theta(\boldsymbol{r}-\boldsymbol{r}')}\right\rangle \propto \frac{1}{|\boldsymbol{r}-\boldsymbol{r}'|^\eta}$$

for an exponent η. We then see that while there is no spontaneous symmetry breaking, there still exists a phase with 'algebraic' decay of correlations. Since in making the continuum assumption we only retained the leading order term in the Taylor expansion, we are implicitly assuming assuming large βJ that penalises large fluctuations of the spins [1]. This is the low temperature phase. Assuming βJ small, we may expand the exponential of the original Hamiltonian in powers of βJ so that the partition function approximately:

$$\mathcal{Z} = \int\prod_{\boldsymbol{r}}\mathrm{d}\theta(\boldsymbol{r})\prod_{\boldsymbol{r}'}[1+\beta J\cos\big(\theta(\boldsymbol{r})-\theta_(\boldsymbol{r}')\big)+\mathcal{O}(J^2)]$$

In determining the correlations in this regime, we note that any integral of the form

$$\int_0^{2\pi}\mathrm{d}\theta_k\cos(\theta_i-\theta_j)$$

is zero whereas

$$\int_0^{2\pi}\mathrm{d}\theta_k\cos(\theta_i-\theta_j)\cos(\theta_j-\theta_k) = \pi\cos(\theta_i-\theta_k)$$

Thus for the correlation we have

$$\langle\cos(\theta(\boldsymbol{r})-\theta(\boldsymbol{0}))\rangle \propto (\pi J)^{|\boldsymbol{r}|}$$

as there is an order $|\boldsymbol{r}|$ of cosine terms in the product that are required to 'connect' $\boldsymbol{0}$ to $\boldsymbol{r}$. We may rewrite this expression as $\exp[-|\boldsymbol{r}|/\xi]$ where $\xi^{-1} = 1/\pi\beta J$ and is the 'correlation length' of the system. It quantifies the distance that the spins are correlated.

This intuitive approach has revealed that, in high temperature phase, the correlations decay exponentially. The two types of decay – algebraic and exponential – are hallmarks of two phases in certain 2D systems, and the transition between the two is known as the BKT transition. More rigorous analysis reveals that the correlation length diverges at the critical point of the transition, and tends to zero as one moves away in either direction.

What differentiates the two phases? The main insight is that the system enables the existence of topoligcal defects known as 'vortices'. This means that we cannot continuously deform a state with vortices into one without them, and thus it cannot be obtained through perturbation theory. To see their origin, note that the phase at each point is restricted to $[0, 2\pi]$, i.e. it is compact, and so the circulation, defined by

$$\int \nabla\theta \cdot \mathrm{d}\boldsymbol{l} = 2\pi n$$

must be restricted to $2\pi n$ where n is an integer so that the phase angle is not multiple valued. Now by the gradient theorem we know that if the gradient is everywhere defined inside the loop then this loop integral should be zero, already signaling that there must be a singularity for vortices to exist. Phase transitions are manifested by singularities in various quantities, and it is indeed it is in the absence or presence of vortices that is indicative of the phase the system occupies.

To obtain a better intuition for the subject, let us consider an isotropic solution $\nabla\theta = \frac{n}{r}\hat{\boldsymbol{r}}$. The integer n is known as the charge of the vortex, as then the solution is $\theta(\boldsymbol{r}) = \log(r)$ which describes a Coulombic charge in two dimensions. This provides a revealing picture for the system: in the low temperature phase the 'charges' are bound, but increasing the temperature beyond the transition is marked by an unbinding and proliferation of these vortices. Since the free energy $F = U - TS$ is minimised at a given temperature and volume, the proliferation at the critical point must mean the entropy increase is more favourable than the energy increase.

To estimate both, imagine that we have one vortex in the system. We use our expression for the energy

$$\int \mathrm{d}^2\boldsymbol{r}\, |\nabla\theta|^2 \propto \log(L)$$

where L is the system size. The number of places to put the vortex is proportional to L^2, so the entropy $k \log \Omega$ must also be proportional to $\log L$. As both have this proportionality, it is then clear that, at the critical point, $U - T_c S$ makes vortex proliferation favourable. All these points will come into play in a modified form in the study of polariton condensates.

After considering equilibrium issues, what of dynamics? We describe [10] the evolution of $\theta_{i,j}$ according the Langevin equation

$$\partial_t \theta_{i,j} = -\Gamma \frac{\delta \mathcal{H}}{\delta \theta_{i,j}} + \eta = -\Gamma \frac{\partial \mathcal{H}}{\partial \theta_{i,j}} + \eta$$

where η is a Guassian white noise term $\langle \eta(\boldsymbol{r}, t), \eta(\boldsymbol{r}', t') \rangle = 2\Delta\delta(t - t')\delta(\boldsymbol{r} - \boldsymbol{r}')$. In the discrete version this is

$$\partial_t \theta_{i,j} = +\Gamma J \sum_{\{x,y\}} \sin(\theta_{i,j} - \theta_{i+x,j+y})$$

where $\{x, y\}$ are as before. The continuum version of this equation requires $\delta\mathcal{H}/\delta\theta$ which is

$$\frac{\delta \mathcal{H}}{\delta \theta} = \frac{J}{2} \int \mathrm{d}^2 \boldsymbol{r}\, 2(\nabla\theta)(\nabla\delta\theta).$$

Intgerating by parts we find that the integrand is zero for $J\nabla^2\theta$ so the equation is

$$\partial_t \theta = -\Gamma J \nabla^2 \theta + \eta.$$

We will see that this is the equation of our system describing the long wavelength behaviour of the condensate phase when we are in a certain regime.

1.4 XY model scaling violations

To understand the approach taken in the project, we describe the approach taken in [4] to determine dynamical evolution of the correlation length ξ as the system XY system is quenched from the disordered (high temperature) phase to the ordered (low temperature) phase. Using renormalisation group methods [**Janssen1989**], the dynamical exponent z is predicted to be 2 so that the correlation length grows as $\xi(t) = t^{1/2}$ and t is a substitute of the relaxation time of the system to the evolution of the system. The relaxation time also diverges at criticality, known as 'critical slowing down' [**nishimori2011elements**]. To test this prediction, for various system sizes a plot of the time dependent Binder cumulant defined by

$$g_L(t) = 2 - \frac{\langle \boldsymbol{M}^2 \rangle^2}{\langle \boldsymbol{M}^4 \rangle}$$

was plotted where $\boldsymbol{M}(t)$ is the magnetisation (the average of $(\cos\theta_{i,j}, \sin\theta_{i,j})$ on the lattice. The mean of $\boldsymbol{M}$ is over independent Monte Carlo realisations. In a disordered condition where each spin is random, each component magnetisation is the sum of independent uniformly distributed numbers, and so by the central limit theorem are

approximately Gaussian distributed. Then $\langle \boldsymbol{M}^2 \rangle^2 / \langle \boldsymbol{M}^4 \rangle$ is 2, so g_L is 0. In an ordered state where all spins are aligned, the ratio is one, and g_L is 1. The Binder cumulant is thus a convenient quantity to act as an ordered parameter.

Crucially, it is dimensionless. Therefore, for different system sizes, plotting the Binder cumulant as a function of another dimensionless quantity should result in all the curves 'collapsing' into one. Since we are varying the system size L, the ratio $\xi^2/L^2 = t/L^2$ is appropriate.

This was done in two cases: first in the quench from a completely ordered phase to one below the critical temperature, where $z = 2$ provided a good collapse. In the quench from the random initial condition to the ordered phase, however, a much higher exponent $z = 2.35$ was required to achieve the same quality in collapse. The disparity was explained by the requirement of vortex-antivortex binding necessary when queching from the disordered initial condition, which slows the approach to equilibrium. Indeed, a modification of the functional form of ξ was found to be necessary, and was $\xi(t) = (t/\log(t/t_0))^{1/2}$. The collapse is shown in Figure. 1.4.

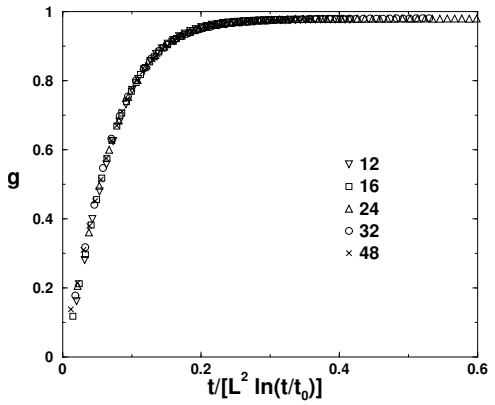

FIGURE 1.4: Scaling violation of the Binder cumulant g when quenching from the disordered phase to the critical temperature. Rather than standard scaling form $\xi \propto t^{1/z}$ (z being 2 when quenching from the ordered phase), it is modified by $\xi \propto (t/\ln(t))^{1/2}$ due to vortex interactions [4].

The physical origin of this was in the friction and attraction experienced by the vortices. For the attraction, we found earlier that the vortices behave as 2D Coulombic charges, and thus the force between them is proportional to $1/R$ where R is the vortex distance. The friction was determined from the evolution equation (without noise, or $T = 0$), i.e. $\partial_\theta = -\delta H/\delta\theta$ (the prefactors being set to 1). If $\theta(x, y, t) = \theta_v(x - vt, y)$ is the vortex motion is now

$$\frac{\mathrm{d}E}{\mathrm{d}t} = \int \mathrm{d}^2\boldsymbol{r}\, \frac{\delta E}{\delta\theta}\frac{\partial\theta}{\partial t} = -\int \mathrm{d}^2\boldsymbol{r} \left(\frac{\partial\theta}{\partial t}\right)^2 = -v^2 \int \mathrm{d}^2\boldsymbol{r} \left(\frac{\partial\theta_v}{\partial x}\right)^2$$

Now for a frictional force $-\gamma\dot{x}$ we see that the last integral must be the friction constant. Using the equilibrium vortex configuration we found earlier, which is isotropic, and hence we replace the derivative by x with the gradient, we see that the zero velocity vortex friction is the equilibrium energy, and thus scales like $\log R$, the vortex distance.

Taking the correlation length as the vortex distance, the velocity of the vortices is $\mathrm{d}\xi/\mathrm{d}t \sim F/\gamma \sim 1/(\xi \log \xi)$ so that $\xi \sim (t/\log t)^{1/2}$.

1.5 Mapping to the KPZ Equation

We now in see what way our polariton condensate relates to the XY model. Previously we discussed two pumping regimes for the microcavity: incoherent and the OPO regimes. The long wavelength dynamics of the phase in both cases is governed by the compact anisotropic KPZ equation [14]

$$\partial_t \theta = D_x \frac{\partial^2 \theta}{\partial x^2} + D_y \frac{\partial^2 \theta}{\partial y^2} + \lambda_x \left(\frac{\partial \theta}{\partial x}\right)^2 + \lambda_y \left(\frac{\partial \theta}{\partial y}\right)^2 + \eta$$

where η is the Gaussian white noise term as before. We see that if the λs are set to zero (the Ds can always be set to one by rescaling the coordinates) we reobtain the equation for the XY model evolution.

However, the physical origin and the interpretation of θ is different in both cases, which will be discussed. First, as we described earlier the polariton system can form a condensate by adjusting the drive parameters appropriately, and hence we can describe the condensate by a macroscopic wave function $\rho(\boldsymbol{r})e^{i\theta(\boldsymbol{r})}$. Since the phase angle is compact, by the same arguments as before, vortices are allowed to exist.

In the incoherent pump regime the dynamics of the condensate can be phenomenologically described by the dissipative Gross-Pitaevskii equation [2]

$$\partial_t \psi(\boldsymbol{r}, t) = -\frac{\delta H_d}{\delta \psi^*} + i\frac{\delta H_c}{\delta \psi^*} + \eta$$

where H_d and H_c are responsible for the coherent and dissipative dynamics and are given by

$$H_l = \int r_l |\psi|^2 + K_l^x |\partial_x \psi|^2 + K_l^y |\partial_y \psi|^2 + \tfrac{1}{2} u_l |\psi|^4$$

where $l = \{c, d\}$. The last term is a complex Gaussian white noise. The equation then reads

$$\partial_t \psi = -r_{\mathrm{eff}}\psi + K_{\mathrm{eff}}^x \partial_x^2 \psi + K_{\mathrm{eff}}^y \partial_y^2 \psi - u_{\mathrm{eff}} |\psi|^2 \psi + \eta$$

where $A_{\text{eff}} = A_d - iA_c$. To simplify this equation, and in particular to determine an equation only dependent on θ, the condensate is expanded about a mean $(M_0 + \chi)e^{i\theta}$ where M_0 is set to be a static and uniform solution when $\eta = 0$ and $\theta = 0$. This gives $r_{\text{eff}}M_0 = -u_{\text{eff}}M_0^3$. The coeffient r_c is arbitrary and can be set by a local and time dependent transformation $\psi' = \psi e^{i\omega t}$, and this freedom is used to set $r_c = -u_c M_0^2$ and $r_d = -u_d M_0^2$. The physical origin of the other terms is as follows: r_d is the particle loss rate less the pump rate. Clearly the loss rate depends on the density of polaritons, and in the incoherent regime a sufficient density is required for polariton creation. The K_c terms are the momentum coefficents in the x and y directions, inversely proportional to m_x and m_y, while K_d are diffusion-like terms. Finally, the u_c term describes a two polariton interaction, and u_d is also a loss term but is non linear. The latter ensures that the polariton density does not grow without bound.

Substituting the mean expansion and linearising the χ fluctuation (also ignoring terms like $\partial_x\chi\partial_x\theta$), gives

$$\partial_t\chi = -2u_dM_0^2\chi - K_c^xM_0\partial_x^2\theta - K_c^yM_0\partial_y^2\theta - K_d^xM_0(\partial_x\theta)^2 - K_d^yM_0(\partial_y\theta)^2 + \operatorname{Re}\eta$$

$$M_0\partial_t\theta = -2u_cM_0^2\chi + K_d^xM_0\partial_x^2\theta + K_d^yM_0\partial_y^2\theta - K_c^xM_0(\partial_x\theta)^2 - K_c^yM_0(\partial_y\theta)^2 + \operatorname{Im}\eta$$

where the real and imaginary parts were equated. In the low frequency limit, the left hand side is negligible compared to the first term on the right as as it has a time derivative, therefore can be ignored in when $\omega \to 0$. The first equation can then be solved for χ and substituted into the second, from which the KPZ equation is obtained. That the long wavelength limit results in only θ dependence is as expected as the U(1) symmetry (clear from the Hamiltonian) means that the excitation about small k is 'gapless' in that the energy of the excitation tends to zero (since a slowly varying rotation in space can be made to require arbitrarily small energy). In contrast, excitations of the other variables are 'gapped' and the dispersion does not tend to zero at low k but some finite value as there is no corresponding symmetry. Thus we need only retain θ dependence.

For the coherent pump regime, purportedly the same dynamics is followed. However, here an *ab initio* rather than phenomlogical model is possible, and the meaning of θ is different: $\theta = \theta_s - \theta_i$ where s and i stand for the signal and idler modes. There again exists a U(1) symmetry as now polariton scattering is described by terms such as $\psi_s\psi_i\psi_p^{*2}$+h.c. which are invariant under a transformation $\psi_s \to \psi_s e^{i\alpha}$, $\psi_i \to \psi_i e^{-i\alpha}$, and ψ_p kept the same. This symmetry results in a gapless excitation and the KPZ equation is obtained with the modified θ.

To obtain the discretised version of this equation [**PhysRevB.94.10452**], the linear terms are transformed as before in the XY model, i.e. by

$$\partial_x^2 \theta_{i,j} = - \sum_{x=\{-1,1\}} \sin(\theta_{i,j} - \theta_{i+x,j})$$

and analogously for the y direction. The nonlinear terms are the second order coefficients in the expansion of cos, so the prescription for these is

$$(\partial_x \theta_{i,j})^2 = - \sum_{x=\{-1,1\}} (\cos(\theta_{i,j} - \theta_{i+x,j}) - 1)$$

and analogously for the y direction. There is no factor of 2 as we are summing two such cos terms in each case.

1.6 Polariton Condensate Regimes

Using the KPZ equation parameters, the regimes of the condensate are parametrised [14] by the scaled nonlinearity

$$g \equiv \lambda_x^2 / D_x^2 \sqrt{D_x D_y}$$

and the scaled anisotropy

$$\Gamma = \lambda_y D_x / \lambda_x D_y.$$

For the KPZ equation to be stable, the Ds must be positive. The sign of the anisotropy therefore depends on the signs of the λs, and only their relative sign matters. The cases $\Gamma > 0$ and $\Gamma < 0$ are called the weak and strong anisotropic phases, respectively.

To determine the long range behaviour in the absense of vortices, rernormalisation group (RG) analysis has been peformed. The essential idea is as follows: over small length scale the system is course grained so is in a sense averaged. A microscropic length scale of the system (such as lattice spacing) is then obtained by shrinking the course grained length scale down to the lattice spacing. This process results in a new description of the system with the parameters adjusted. After repeated application of this procedure, a 'flow' of the parameter values is obtained (essentially a phase portrait) and the terms that are non zero at the fixed points, which correspond to points where the correlation length is zero, it shrinking after each iteration, are the terms that are important.

To leading order, the RG evolution of these parameters is given by

$$\frac{\mathrm{d}g}{\mathrm{d}l} = \frac{g^2}{32\pi}(\Gamma^2 + 4\Gamma - 1)$$

$$\frac{\mathrm{d}\Gamma}{\mathrm{d}l} = \frac{\Gamma g}{32\pi}(1-\Gamma^2)$$

where l is the scale factor during an application of the RG method. The flows are illustrated in XX Two cases are important. For the weak anisotropic regime all flow lines tend to $g = \inf$ and $\Gamma = 1$. In the strongly anisotropic regime there is a fixed point at $g = 0$ and $\Gamma = -1$. This is the XY model regime where $\lambda = 0$ and one expects BKT physics.

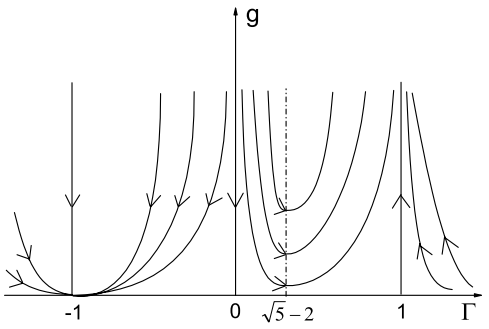

FIGURE 1.5: RG flow lines for the parameters Γ and g defined in the main body. For the strongly anisotropic case $\Gamma < 0$ all flows go to the fixed point $\Gamma = 0$ and $g = -1$. For the weak isotropic case $\Gamma > 0$, flow lines tend to $\Gamma = 1$ and $g = \infty$ [6].

In practice, only the weakly anisotropic regime is accessible in the case of incoherent pumping [2], whereas for the OPO pumping it is experimentally viable to traverse different regimes by the adjustment of the pump strength. We shall now describe the various possible regimes for both pumping types and for both anisotrpic and isotropic parameters.

In the weakly isotropic case, a dual modified electrodynamic theory has been mapped from the KPZ equation which describes the vortices as charges, extending the mapping that was used in the BKT transition. In the present case, the interactions of these vortices have repulsive contributions that grow with λ. Thus it was determined that beyond a length scale $L_v = e^{2\pi/\lambda}$, vortices are screened and thus always proliferate. Beyond this distance they are expected to show exponential correlations, while beyond the KPZ distance L_{KPZ} stretched exponential correlations are expected, i.e. $e^{-r^{2\chi}}$ is the form of the correlation function, with $\chi \sim 0.39$ determined from numerical investigations.

In the case of incoherent pumping in the isotropic regime it is found [2] that although in the thermodynamic limit the algebraic order will always be destroyed, the size L_{KPZ} to which a finite system can show algebraic correlation depends on a tuning parameter $x = \gamma_p/\gamma_l - 1$ where the p and l are pump power and loss, respectively. Algebraic correlations may be destroyed in one of two ways: for a sufficiently large tuning parameter and system, decreasing the tuning parameter increases the nonequilibrium fluctuations so that the system enters the KPZ phase. Alternatively, a BKT transition may take place

first. This is illustrated in Figure. 1.6. It is noted that the KPZ length scale L^* is much larger than realistic system sizes, and therefore may not be observed for incoherently pumped systems.

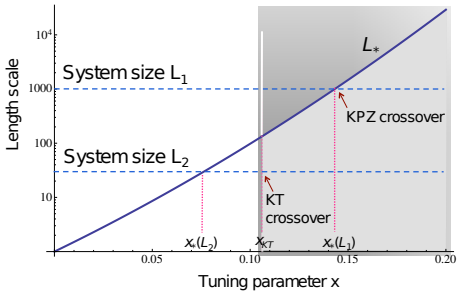

FIGURE 1.6: The different regimes the system is in in the case of incoherent pumping depends on the tuning parameter x and system size. At a combination with a sufficiently small system size and sufficiently large x, the system displays algebraic correlations. Depending on the system size, decreasing x can lead to stretched exponential correlations either through entering the KPZ phase or via a BKT transition [2].

For coherent pumping, the length scales as a function of the pump strength relative to the maximum pump strength that allows the OPO regime is shown in Figure. 1.7. Although the length scale L_v is less than that of L_{KPZ} where the KPZ phase would be observable, thus seemingly precluding the KPZ phase, it has been pointed out that in numerical studies of up to 1000 μm that the vortex rich phase appears absent XX, which could suggest a prevention of the vortex proliferation, or an underestimation of L_v.

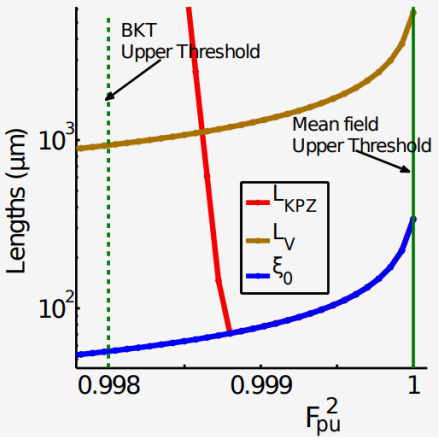

FIGURE 1.7: The case of coherent pumping in the weak anisotropic regime. This shows the various length scales as a function of the pump power normalised to the maximum pump that allows the OPO regime. For most values of pump power the L_v is less than that of L_{KPZ} where the KPZ phase would be observable [8].

In systems with strong anisotropy there only exists a phase transition between the BKT phases [2, 8]. An incoherent pump phase diagram that illustrates this is shown in Figure. [**fig:incoherentanisotropic**]. In particular, the system exhibits reentrance which means that by increasing the pump power it is possible to enter into the algebraic regime,

and then subsequently exit it on further increase. Coherent pumping displays similar phases as the incoherent pump, specifically in reentrance, however in experimental realisations the various regimes can be explored much more easily with the available system size by varying the detuning (the energy difference between the cavity resonance and exciton energy at $k = 0$), pump power, and pump wavevector.

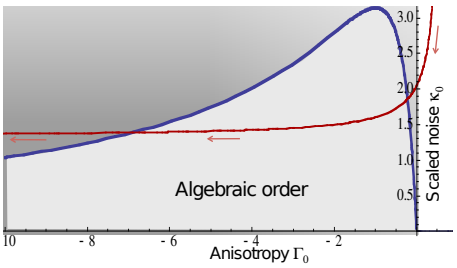

FIGURE 1.8: A phase diagram for the anistropic KPZ state for the incoherent pump for the parameters Γ_0 (as defined in the text) and the parameter $\kappa_0 = \Lambda/\sqrt{D_x D_y}$ where Λ is the noise strength. The curve shows a potential route that would be traversed in experiment as pump power is increased: the key feature of reentrance where the algebraic phase is entered and left as this path is crossed [2].

1.7 Project Aim

The aim of the project will be to simulate the discretised KPZ equation on a finite lattice of phase angles in for various values of λ both in the anisotropic regime and isotropic regime. The Binder cumulant will be plotted as a function of $(t/\log t)^{1/2}$, first to obtain the same collapse as in the study of the XY model, then to test for the expected collapse in the anisotropic regime. The behaviour of the Binder cumulant will also be checked for in the isotropic regime.

Chapter 2

Method

2.1 Data Generation

The evolution of the condensate phase was simulated using code written in C++. This defined the codensate as a square lattice consisting of N^2 points, where N is the linear size of the lattice, and each point restricted to be between $[0, 2\pi]$. Initially, the angle of each point was random, the system therefore being in the disordered phase. The simulation updated each lattice point according to the compactified and discretised equation:

$$\begin{aligned}\theta_{i,j}(t+\mathrm{d}t) = \theta_{i,j}(t) + \mathrm{d}t\,[&- D_x(\cos(\theta_{i,j}-\theta_{i+1,j}) + \cos(\theta_{i,j}-\theta_{i-1,j}) - 2) \\ &- D_y(\cos(\theta_{i,j}-\theta_{i,j-1}) + \cos(\theta_{i,j}-\theta_{i,j+1}) - 2) \\ &- \frac{\lambda_x}{2}(\sin(\theta_{i,j}-\theta_{i+1,j}) + \sin(\theta_{i,j}-\theta_{i-1,j})) \\ &- \frac{\lambda_y}{2}(\sin(\theta_{i,j}-\theta_{i,j-1}) + \sin(\theta_{i,j}-\theta_{i,j+1}))] \\ &+ 2\pi c_L \times \sqrt{\mathrm{d}t} \times \xi\end{aligned}$$

where $\theta_{i,j}(t)$ is the value of the condensate at points i, j of the lattice and $\mathrm{d}t$ is the timestep used. Periodic boundary conditions were used. The final term is the stochastic term where ξ is a uniformaly distributed random number (restricted to $[-0.5, 0.5]$) that was also added at each timestep. Finally, for all simulations D_x and D_y were set to 1 as the coordinates can always be rescaled to ensure this.

The timestep was initially chosen to be $\mathrm{d}t = 0.01$ and this was the value used to generate the significant data. However, other values were considered with: namely, $\mathrm{d}t = 0.02$ and $\mathrm{d}t = 0.001$ with system size 32. Figure. 2.1 shows the Binder cumulant for this system size for the three values of $\mathrm{d}t$. The evolution for $\mathrm{d}t = 0.02$ did not follow that of

$dt = 0.01$, implying that $dt = 0.02$ was potentially too large of a timestep, whereas the behaviour of $dt = 0.01$ reasonably matched that of $dt = 0.001$, demonstrating that $dt = 0.01$ was a suitable choice of timestep for the simulation, and a smaller one was computationally unnecessary.

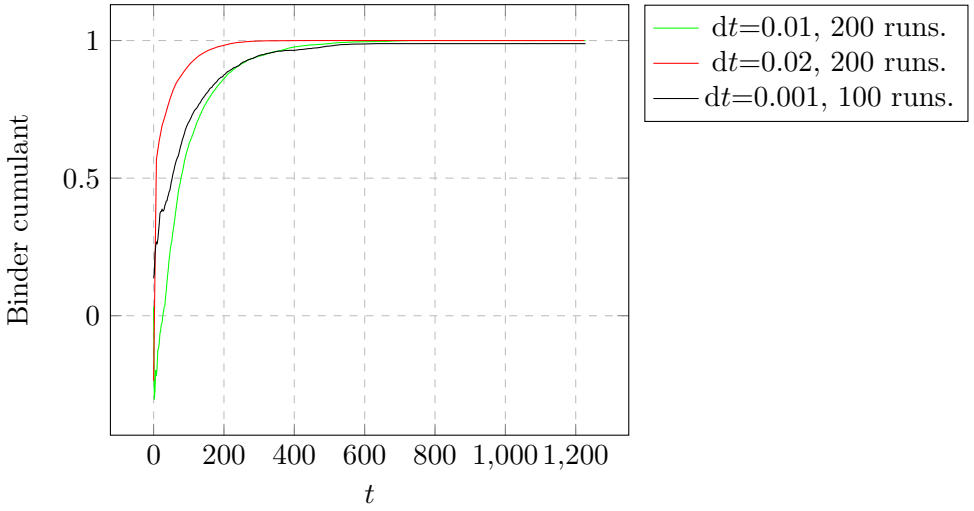

FIGURE 2.1: The Binder cumulant convergence for $L = 32$ at different values of dt. The convergence of $dt = 0.001$ compared to $dt = 0.01$ suggests that a timestep of $dt = 0.001$ was unnecessary, while the behaviour of $dt = 0.02$ although qualitatively correct, deviated from that $dt = 0.02$ significantly to be considered trustworthy.

The value of c_L at which the phase transition occurs was determined. Figure. 2.2 shows the number of vortices at $t = 400$ for a system size of 64 at various values of c_L. The value $c_L = 0.2$ was then chosen for the remainder of project. For the value of $c_L = 0.1$ the Binder cumulant did not approach one at all in the case of $L = 128$, as shown in Figure. 2.3. This is likely a numerical artefact and can be explained by the existence of vortices stuck to lattice sites and not able to anihilate due to the low noise. The number of vortices is higher in Figure. 2.2 for $c_L = 0.1$ than higher values below the transition. This issue was also mentioned in [**PhysRevE.47.1525**].

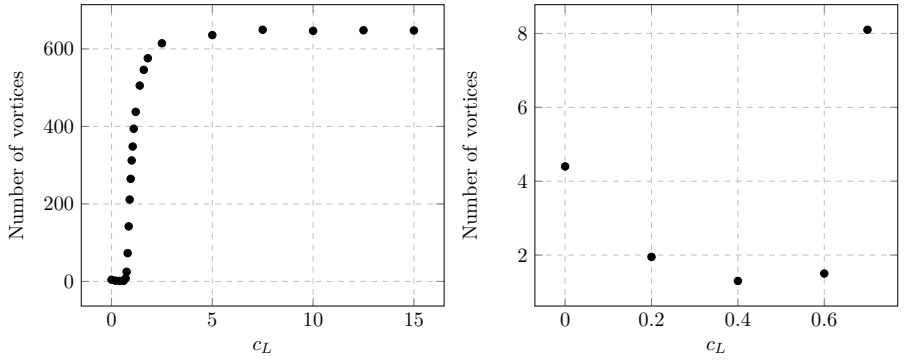

FIGURE 2.2: The number of vortices at the end of a simulation ($t = 400$) as a function of c_L for $L = 64$ with 20 runs.

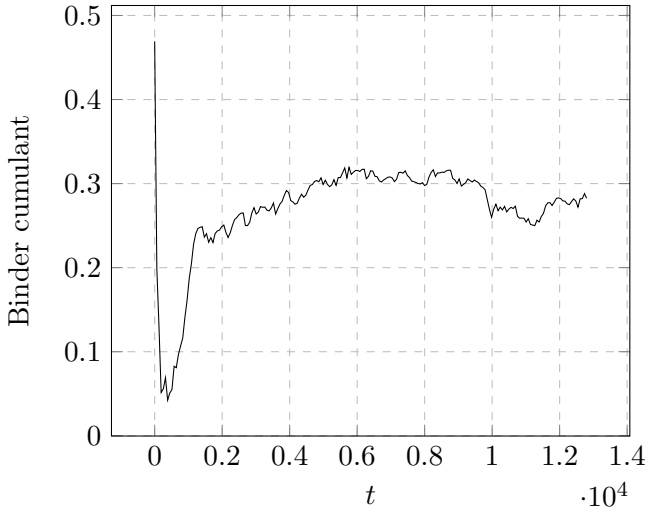

FIGURE 2.3: The Binder cumulant for $L = 128$ and $c_L = 0.1$. Despite being in the ordered phase, the Binder cumulant does not converge to near one due to the vortices.

2.2 Data Extraction

Using the generated data, the Binder cumulant was calculated as follows: for each timestep of a simulation, the magnetisation $\boldsymbol{M}$, defined by

$$\boldsymbol{M} = \frac{1}{N^2}\sum_{i,j}(\cos(\theta_{i,j}), \sin(\theta_{i,j})),$$

where the sum is over all lattice points, was calculated. The averages (over all the realisations) $\left\langle \boldsymbol{M}^2 \right\rangle$ and $\left\langle \left(\boldsymbol{M}^2\right)^2 \right\rangle$ were then used in the Binder cumulant given by

$$g = 2 - \frac{\left\langle (\boldsymbol{M}^2)^2 \right\rangle}{\left\langle \boldsymbol{M}^2 \right\rangle^2}$$

To estimate the error, the Binder cumulant was considered to be a function of the variables $\boldsymbol{M}^2 >$, i.e.

$$g = 2 - N\frac{\sum_i (\boldsymbol{M}_i^2)^2}{\left(\sum_i \boldsymbol{M}_i^2\right)^2}$$

where the sum is over every realisation and N is the number of realisations. These variables are indentical and independent, so the error of each is the same and is approximated by

$$\sigma^2_{\boldsymbol{M}^2} = \frac{1}{N-1}\sum_i (\boldsymbol{M}_i^2 - \left\langle \boldsymbol{M}^2 \right\rangle)^2 = \frac{N}{N-1}(\left\langle (\boldsymbol{M}^2)^2 \right\rangle - \left\langle \boldsymbol{M}^2 \right\rangle^2).$$

Using error propagation, the error on the Binder cumulant is

$$\begin{aligned}
\sigma_g^2 &= 4N^2 \sum_k \left(-\frac{\boldsymbol{M}_k^2}{\left(\sum_i \boldsymbol{M}_i^2\right)^2} + \frac{\sum_i (\boldsymbol{M}_i^2)^2}{\left(\sum_i \boldsymbol{M}_i^2\right)^3} \right)^2 \sigma_{\boldsymbol{M}^2} \\
&= \frac{4N^3}{N-1} \left(\left\langle (\boldsymbol{M}^2)^2 \right\rangle - \left\langle \boldsymbol{M}^2 \right\rangle^2 \right) \sum_k \left(\frac{(\boldsymbol{M}_k^2)^2}{\left(\sum_i \boldsymbol{M}_i^2\right)^4} + -2\frac{\boldsymbol{M}_k^2 \sum_i (\boldsymbol{M}_i^2)^2}{\left(\sum_i \boldsymbol{M}_i^2\right)^5} + \frac{\left(\sum_i (\boldsymbol{M}_i^2)^2\right)^2}{\left(\sum_i \boldsymbol{M}_i^2\right)^6} \right) \\
&= \frac{4N^3}{N-1} \left(\left\langle (\boldsymbol{M}^2)^2 \right\rangle - \left\langle \boldsymbol{M}^2 \right\rangle^2 \right) \left(\frac{N \left\langle (\boldsymbol{M}^2)^2 \right\rangle}{N^4 \left\langle \boldsymbol{M}^2 \right\rangle^4} - 2\frac{N^2 \left\langle \boldsymbol{M}^2 \right\rangle \left\langle (\boldsymbol{M}^2)^2 \right\rangle}{N^5 \left\langle \boldsymbol{M}^2 \right\rangle^5} + \frac{N^3 \left\langle (\boldsymbol{M}^2)^2 \right\rangle^2}{N^6 \left\langle \boldsymbol{M}^2 \right\rangle^6} \right) \\
&= \frac{4}{N-1} \left(\left\langle (\boldsymbol{M}^2)^2 \right\rangle - \left\langle \boldsymbol{M}^2 \right\rangle^2 \right)^2 \frac{\left\langle (\boldsymbol{M}^2)^2 \right\rangle}{\left\langle \boldsymbol{M}^2 \right\rangle^6}
\end{aligned}$$

which is proporiotnal to $1/N$, as expected.

To calculate the number of vortices or anti-vortices, we use a discretised version of their defining equation. The 'loop', starting at a point $\theta_{i,j}$, is the path

$$\theta_{i,j} \to \theta_{i+1,j} \to \theta_{i+1,j+1} \to \theta_{i,j+1} \to \theta_{i,j}$$

and the value of one angle minus the value of the previous angle is calculated and all such differences are summed. If the total is greater than or equal to 2π, there is a vortex, whereas if the total is less than or equal to -2π, there is an antivortex.

The uncertainty on the number of vortices was unfortunately not calculated as the necessary data for larger system sizes had to be periodically deleted and the need for the vortex uncertainty was not considered (in fact it was fortuotous that the uncertainty on the Binder cumulant was calculable without requiring extra data). That being said, there are indications that such an error would would be small based on the variation of the average number of vortices vs the number of realisations, as in Figure. 2.4. Here it is plotted for the $L = 128$ linear case at snapshot 10. This was chosen as the largest system size and the most realisations for that size. There is clearly a downward trend, but the relative variation $\sim 0.15/4.725 = 0.03$ or 3% which is not extreme. A lack of an uncertainty is not acceptable for exact values but should be acceptable to observe general trends and also in the comparison to the expectation in the linear case. To calculate the uncertainties, we used Python's `scimpy` package.

For calculating the correlation functions, the average of $\cos(\theta_{i,j} - \theta_{i,0})$ was calculated from $i = 0$ to $i = L - 1$ and over all realistations at a specific time step. The correlation distance is given by j and $j = 1$ to $j = L/2$ was used.

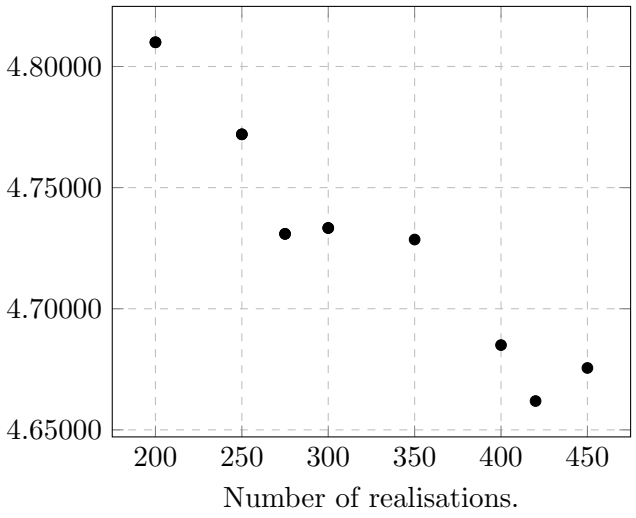

FIGURE 2.4: The number of vortices as a function of the number of realisatations for $L = 128$ and $\lambda_x = \lambda_y = 0$ at snapshot 10.

Chapter 3

Results

Simulations were peformed on system sizes $L = 40, 48, 64, 80, 104, 128$ with the λs taken from zero to one in steps of 0.2 and also 1.5 where λ_x and λ_y had the same magnitude and the same or opposite signs.

3.1 Linear Case

Figure. 3.1 overlays the Binder cumulant evolution for different sizes as a function of $t/L^2 \log t$ in the linear ($\lambda_x = \lambda_y = 0$) case which corresponds to the XY model. As is clear, this reobtains the result from [4] while using Stochastic evolution rather than Monte Carlo and provides confidence in the results for the non-linear cases. The collapse is not perfect: Figures 3.2 and 3.3, which display the Binder cumulant as a function of the number of realisations for all the system sizes, show how far apart the curves are. A value of $t/L^2 \log t$ was calculated for $L = 40$ for two points in the simulation, namely, midway through and three quarters of the way thourgh, and the corresponding t was chosen for all other system sizes that was closest to this value. In this report we call these the midway and three quarters graph for brevity. For the first case, the uncertainties of the points do not all cross, although none are isolated. In the second case, the collapse is weaker and $L = 48$ is consistently isolated, as can be seen from the graph of the collapse. This also occurs for $L = 64$ earlier on in the simulation. For $L = 48$ there could be an underestimation of the error, and there is a slight downward decrease in its value as the number of realisations is increased. Ideally one would perform further simulations to ensure this, but this was not possible in the time constraints of the project.

With those caveats in mind, we may be assured that the result has been obtained, and this is further corroborated by the plots of $\log n_v$, where n_v is the number of vortices,

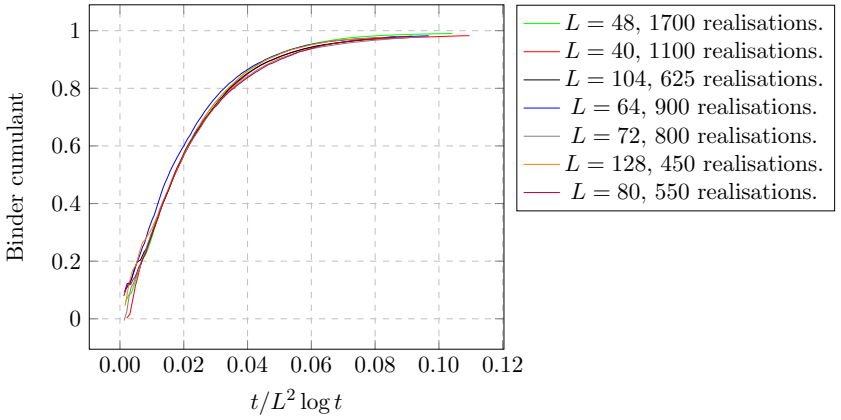

FIGURE 3.1: The Binder cumulant as a function of $t/L^2 \log t$ for different sizes with $\lambda_x = \lambda_y = 0$.

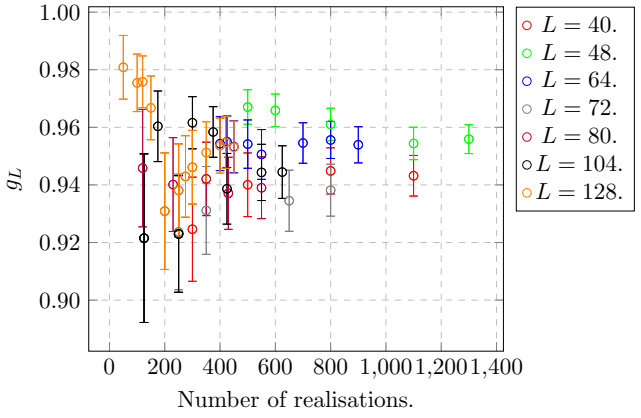

FIGURE 3.2: The uncertainty in the Binder cumulant as a function of the number of realisations at a point closest to $t/L^2 \log t$ for $t = 6500$ (the mid-point of the simulation) for $L = 40.\lambda_x = \lambda_y = 0$.

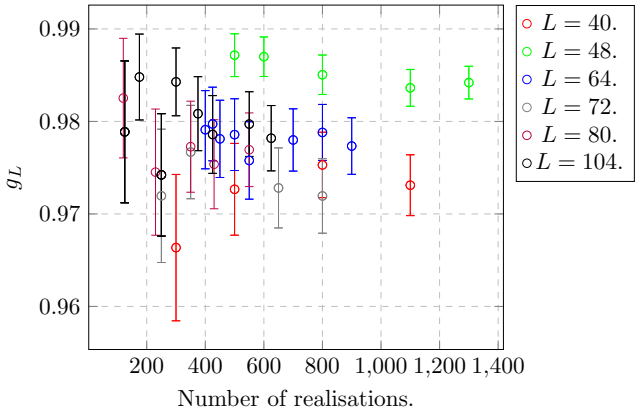

FIGURE 3.3: The uncertainty in the Binder cumulant as a function of the number of realisations at a point closest to $t/L^2 \log t$ for $t = 937.5$ (three quarters through the simualtion) for $L = 40$. $\lambda_x = \lambda_y 0$. There is no value for 128.

as a function of $\log(t/\log t)$, shown in Figures 3.37 to 3.3 for each system size. For the linear case, it is expected that $\log n_v = -\log(t/\log t)$. This can be explained as follows: using the result in [4] that the distance between vortices R is approximately proportional to $(t/\log t)^{1/2}$, we have that the vortex density is then $1/R^2$, and the number of vortices $n_v \sim L^2/R^2$, and therefore $\log n_v \sim -\log R^2 = -\log(t/\log t)$ up to some additive constants. For brevity, we shall call this the 'vortex gradient.' As the figures show, the gradient was obtained to good accuracy for all system sizes: taking an average for all sizes, we obtain the value -1.001 ± 0.003. Due to the finite size of the system, the approximation breaks down at large times once the majority of the vortices have annihilated, and therefore a time scale must be chosen with which to calculate the gradient. In practice, the time scale maximised but chosen to be prior to the finite size behaviour of the system. Therefore it is sensible not to read the values too literally, but take them as indicative of the behaviour of the linear case.

3.2 Anistropic Case

For non-zero λs, when $\lambda_x = -\lambda_y$, we expect to obtain the same result as the linear case. The following pages show the Binder cumulant plotted as a function of t and also $t/L^2 \log t$ for several values of increasing λ_x. Although the collapse is not as tight as for the linear case, it clearly still occurs to a large extent. For the case $\lambda_x = 0.2$, the curves do not cross each other as closely as in the linear case at the mid-point of the simulation, however looking at the uncertainties shows that only the $L = 80$ size deviates from the others. All other points are within each other when also considering the errors on the points. These conclusions also apply after three quarters of the simulation has completed. This is corroborated by the vortex vortex gradient which is averaged to be -1.001 ± 0.003, supporting linear behaviour.

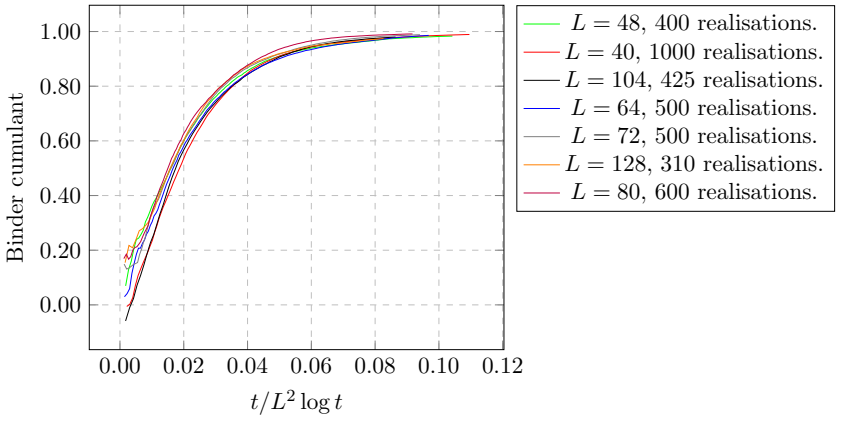

FIGURE 3.4: The Binder cumulant as a function of $t/L^2 \log t$ for different sizes with $\lambda_x = -\lambda_y = 0.2$.

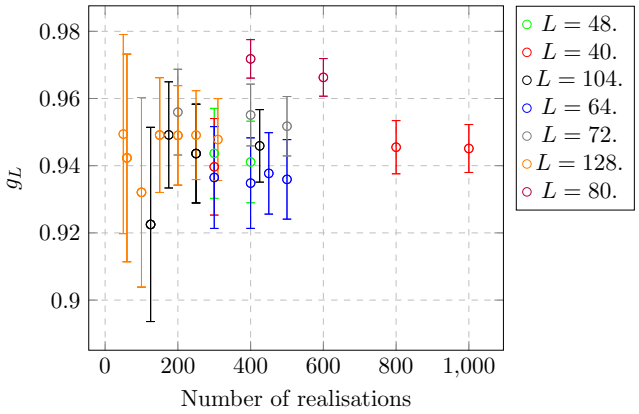

FIGURE 3.5: The uncertainty in the Binder cumulant as a function of the number of realisations at a point closest to $t/L^2 \log t$ for $t = 6500$ (the mid-point of the simulation) for $L = 40$. $\lambda_x = -\lambda_y = 0.2$.

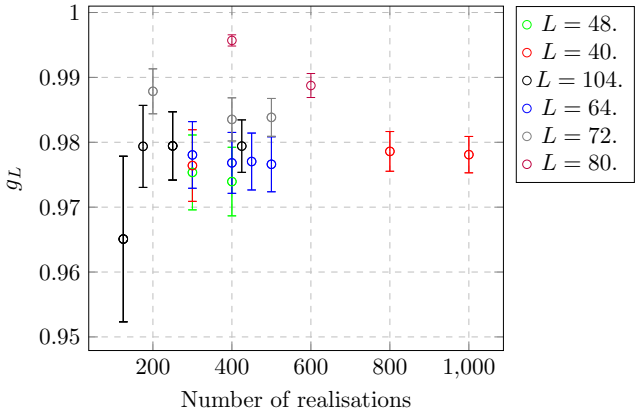

FIGURE 3.6: The uncertainty in the Binder cumulant as a function of the number of realisations at a point closest to $t/L^2 \log t$ for $t = 937.5$ (three quarters through the simualtion) for $L = 40$. $\lambda_x = -\lambda_y = 0.2$. There is no value for 128.

For $\lambda_x = 0.4$ there are similar conclusions to be made, shown in Figure. 3.7. $L = 80$ also deviates, while the mid point uncertainties in the Binder cumulant are within each other for other system sizes. Due to these errors on the Binder cumulant (shown in Figures 3.8 and 3.9, it would appear that it is undetermined whether more realisations are necessary to tighten the collapse further, or if the dynamical exponent growth rate $(t/\log t)^{1/2}$ is simply not as good an approximation as in the linear case. On the other hand, the vortex gradient does show deviation in its calculated average of -1.013 ± 0.003. As these decrease for higher values of λ monotonically it is clear that the effects of higher values of λ are beginning to be felt. In particular, the convergence of the Binder cumulant becomes faster as λ increases, indicating a faster transition to the ordered phase.

To test how this effects the collapse of the Binder cumulant, we can reverse the process which determined the gradient from the correlation length. Therefore the Binder cumulant should be plotted as a function of $(t/logt)^{\alpha/2}$ where α is the gradient of the vortex plot. Similarly to how [4] plotted the collapse with an effective exponent $z = 2.35$, the

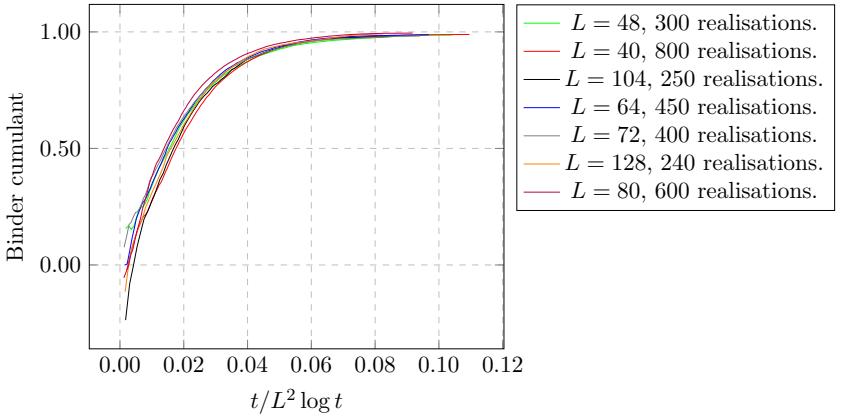

FIGURE 3.7: The Binder cumulant as a function of $t/L^2 \log t$ for different sizes with $\lambda_x = -\lambda_y = 0.4$.

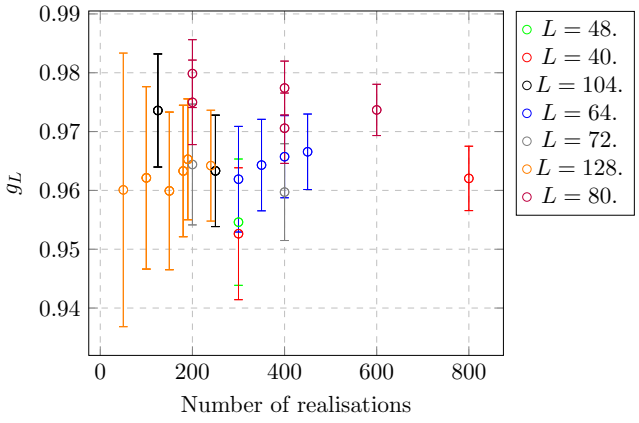

FIGURE 3.8: The uncertainty in the Binder cumulant as a function of the number of realisations at a point closest to $t/L^2 \log t$ for $t = 6500$ (the mid-point of the simulation) for $L = 40$. $\lambda_x = -\lambda_y = 0.4$.

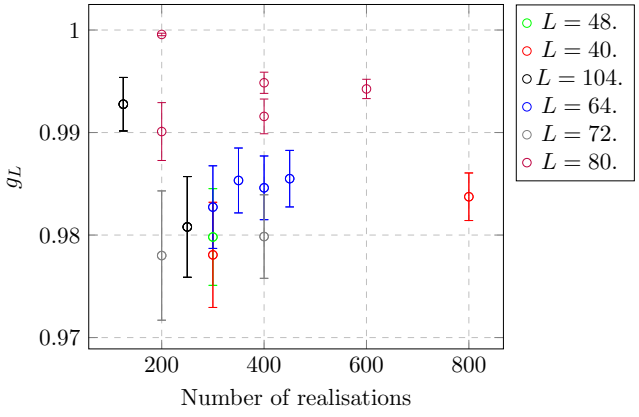

FIGURE 3.9: The uncertainty in the Binder cumulant as a function of the number of realisations at a point closest to $t/L^2 \log t$ for $t = 937.5$ (three quarters through the simualtion) for $L = 40$. $\lambda_x = \lambda_y - 0.4$. There is no value for 128.

purpose of this plot does not have a physical origin but to demonstrate that the $t/\log t$ must be modified. It should be again noted dual electrodynamic theory in [13] only looked at the weakly isotropic case in which the second order (in λ) term of the force between vortices was repulsive at large distances. Due to the decrease in the vortex gradient and a faster convergence, in the following they appear to be attractive.

For $\lambda_x = 0.4$, the plot of the Binder cumulant as a function of $(t/\log t)^1.013/L^2$ is shown in Figure. 3.10. The collapse is only slightly tighter. The gradient, however, has not significantly deviated from the linear case, so this is not unexpected. In both cases the collapse is still weaker than the linear case. Noting the previous comments on the uncertainties, further simulations are indeed necessary, but there are also are differences to the linear case.

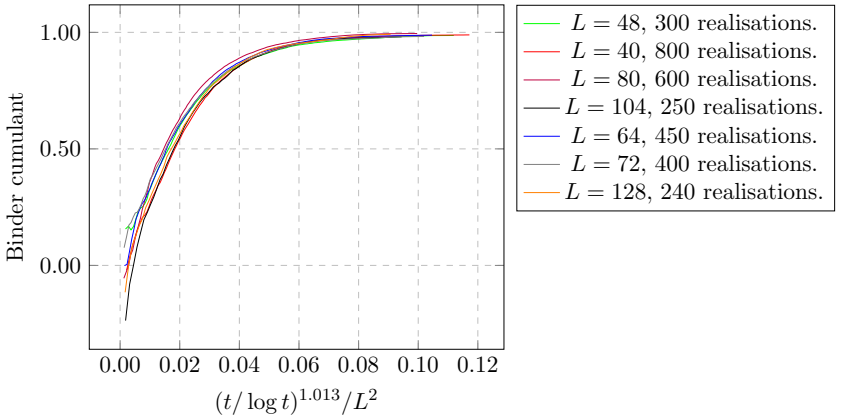

FIGURE 3.10: The Binder cumulant as a function of $(t/\log t)^{1.013}/L^2$ for different sizes with $\lambda_x = -\lambda_y = 0.4$.

For the remaining values of λ, we compare both the collapse with an exponent $\alpha = 1$ and one calculated from the gradient. As one increases λ the collapse becomes weaker with a linear exponent but does not appear to degrade in any linear fashion with the modified exponent. In fact, a subjective glance seems to indicate it actually improves, with $\lambda = 1.5$ showing the strongest collapse besides the linear case.

To check if this is perhaps a finite size effect, although there is no immediate indication that this should be the case, with the calculations in [14] showing that spatial correlations are algebraic as in the XY model below the BKT transitions, the gradient was checked for system sizes of $L = 512$, shown in Figure. 3.19. This was for $\mathrm{d}t = 0.05$ and a much lower number of realisations, so not directly comparable, but at least the behaviour of decreasing gradient with increasing λ also follows. The $\lambda = 0$ gradient under estimated (in magnitude), though the low number of realisations could be the explanation. Further study into these areas is clearly necessary.

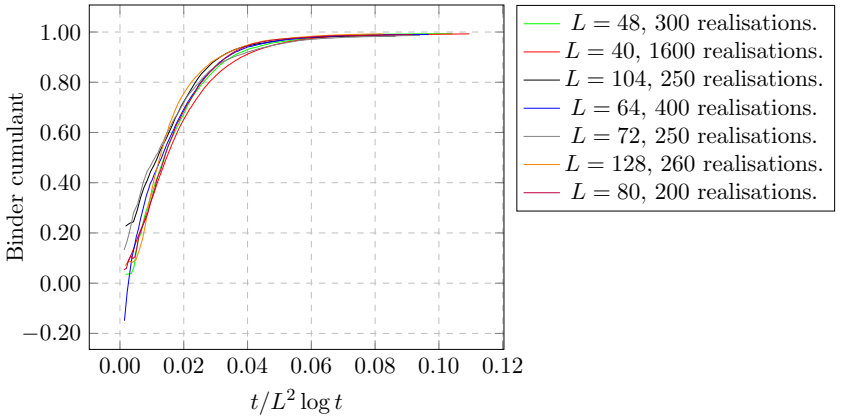

FIGURE 3.11: The Binder cumulant as a function of $t/L^2 \log t$ for different sizes with $\lambda_x = -\lambda_y = 0.6$.

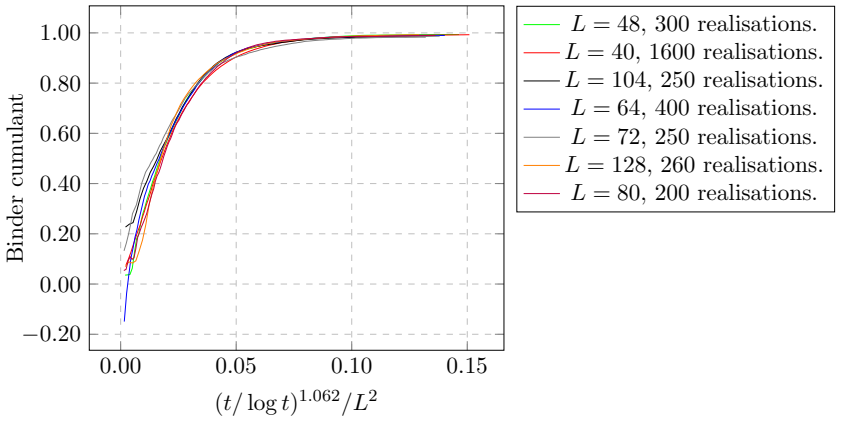

FIGURE 3.12: The Binder cumulant as a function of $(t/\log t)^{1.062}/L^2$ for different sizes with $\lambda_x = -\lambda_y = 0.6$.

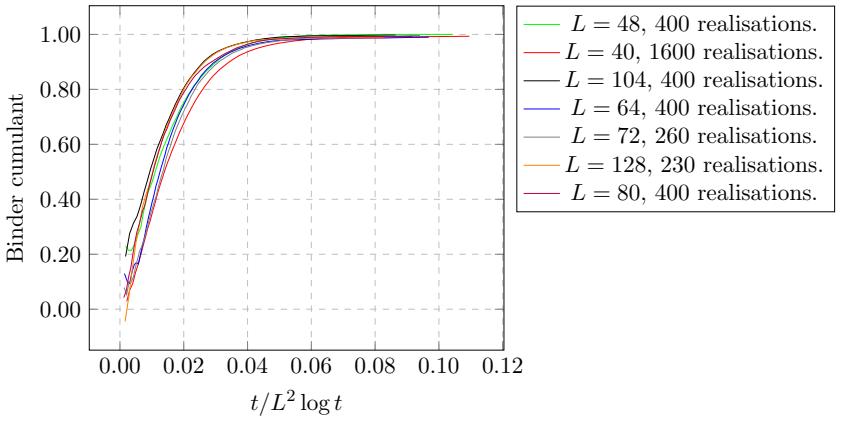

FIGURE 3.13: The Binder cumulant as a function of $t/L^2 \log t$ for different sizes with $\lambda_x = -\lambda_y = 0.8$.

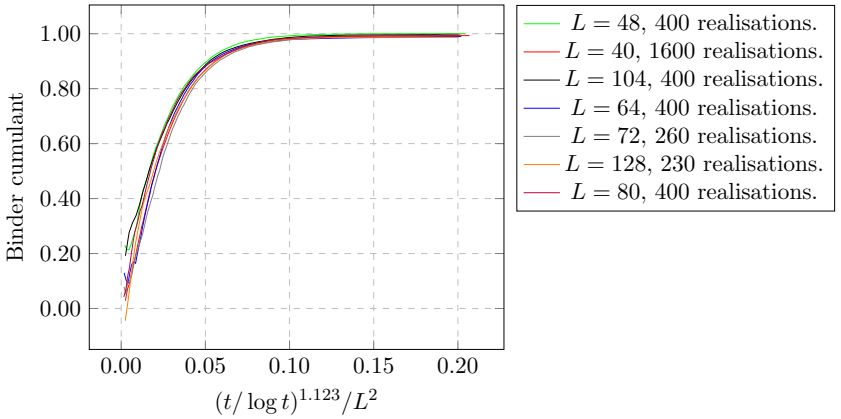

FIGURE 3.14: The Binder cumulant as a function of $(t/\log t)^{1.123}/L^2$ for different sizes with $\lambda_x = -\lambda_y = 0.8$.

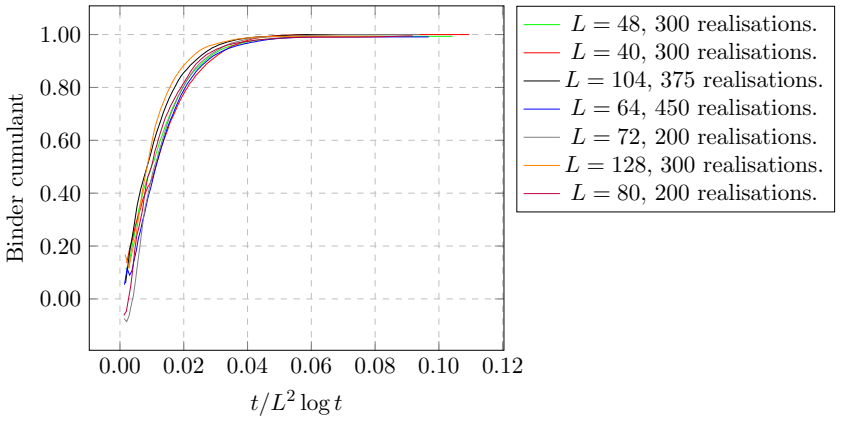

FIGURE 3.15: The Binder cumulant as a function of $t/L^2 \log t$ for different sizes with $\lambda_x = -\lambda_y = 1$.

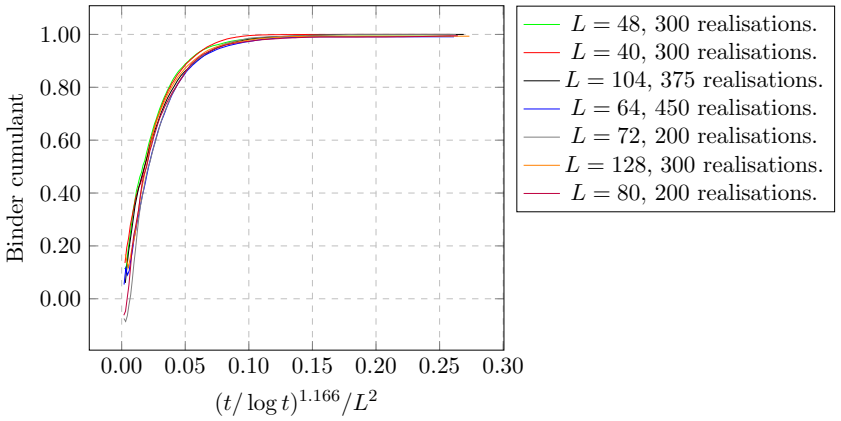

FIGURE 3.16: The Binder cumulant as a function of $(t/\log t)^{1.166}/L^2$ for different sizes with $\lambda_x = -\lambda_y = 1$.

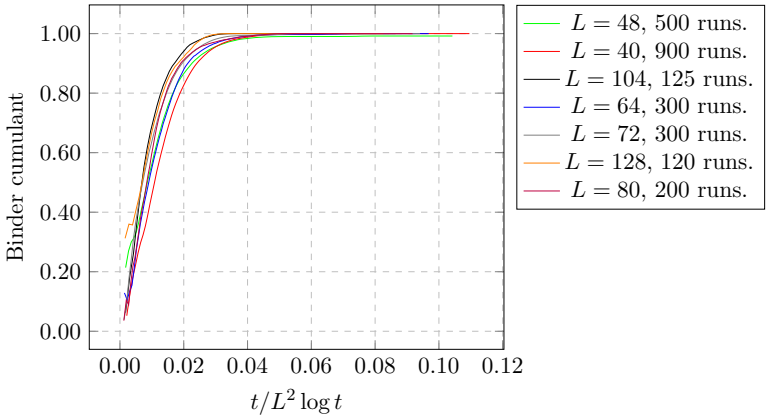

FIGURE 3.17: The Binder cumulant as a function of $t/L^2 \log t$ for different sizes with $\lambda_x = -\lambda_y = 1.5$.

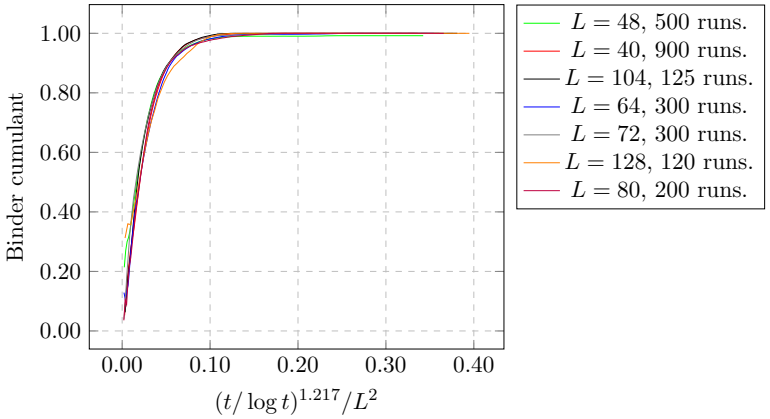

FIGURE 3.18: The Binder cumulant as a function of $(t/\log t)^{1.217}/L^2$ for different sizes with $\lambda_x = -\lambda_y = 1.5$.

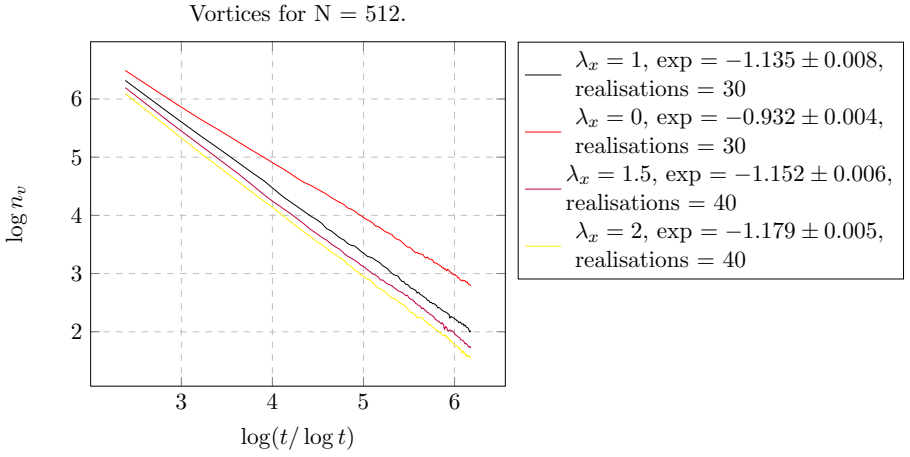

FIGURE 3.19: Log of the number of vortices as a function of $\log(t/\log t)$ for $L = 512$ in the anisotropic case.

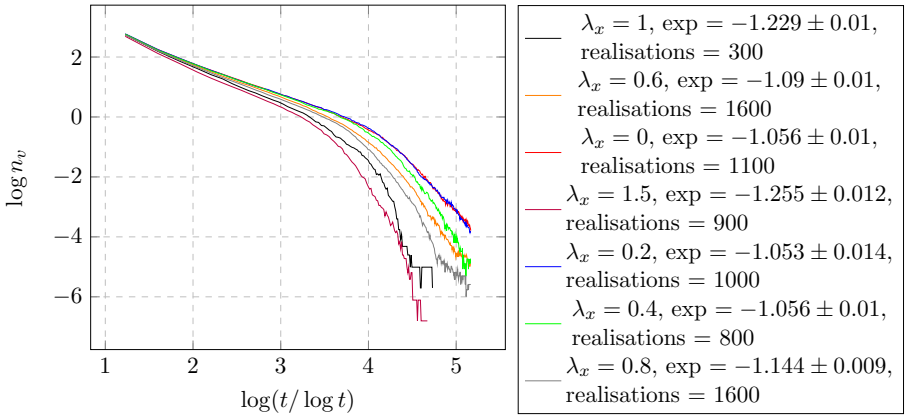

FIGURE 3.20: Log of the number of vortices as a function of $\log(t/\log t)$ for $L = 40$ in the anisotropic case.

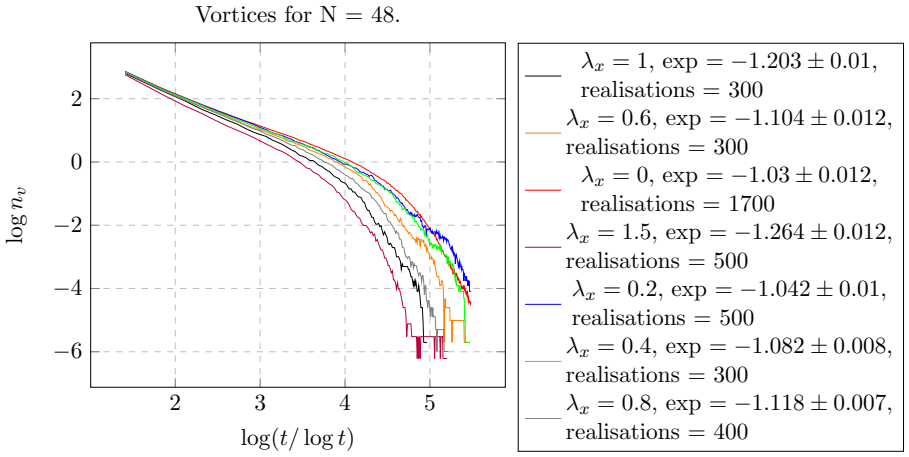

FIGURE 3.21: Log of the number of vortices as a function of $\log(t/\log t)$ for $L = 48$ in the anisotropic case.

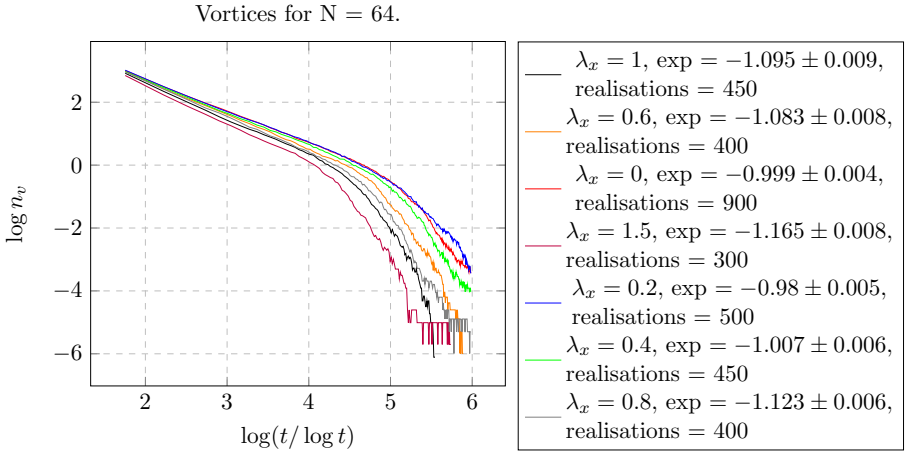

FIGURE 3.22: Log of the number of vortices as a function of $\log(t/\log t)$ for $L = 64$ in the anisotropic case.

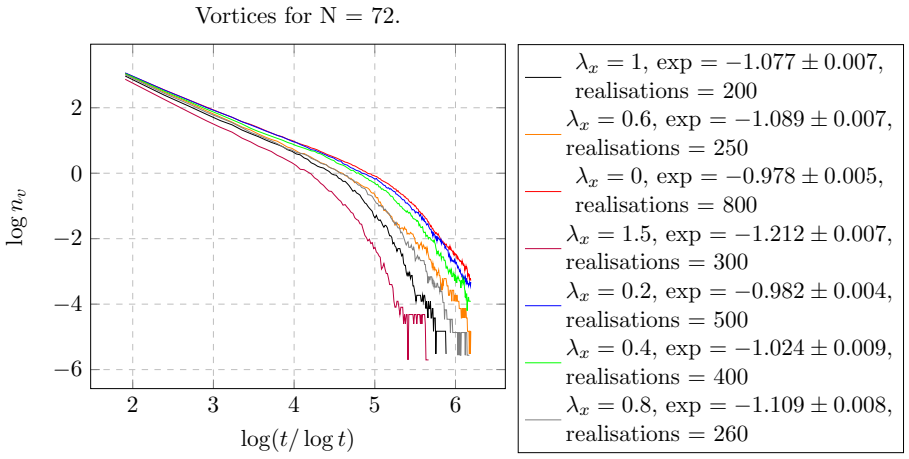

FIGURE 3.23: Log of the number of vortices as a function of $\log(t/\log t)$ for $L = 72$ in the anisotropic case.

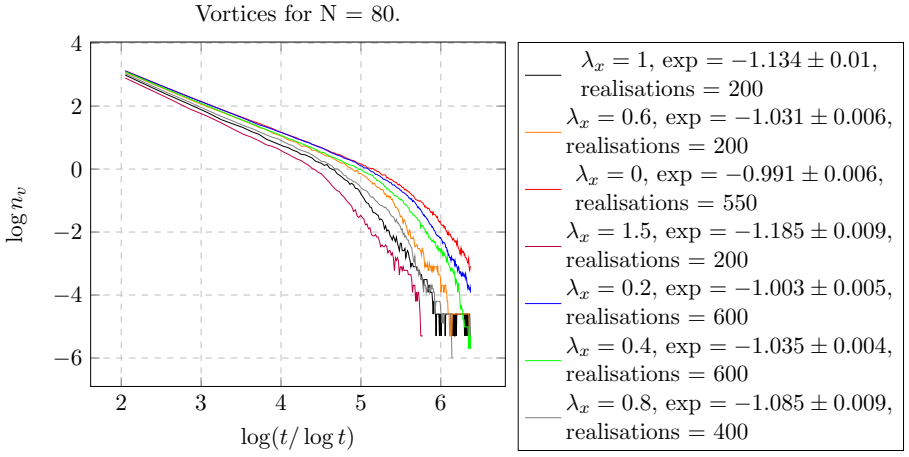

FIGURE 3.24: Log of the number of vortices as a function of $\log(t/\log t)$ for $L = 80$ in the anisotropic case.

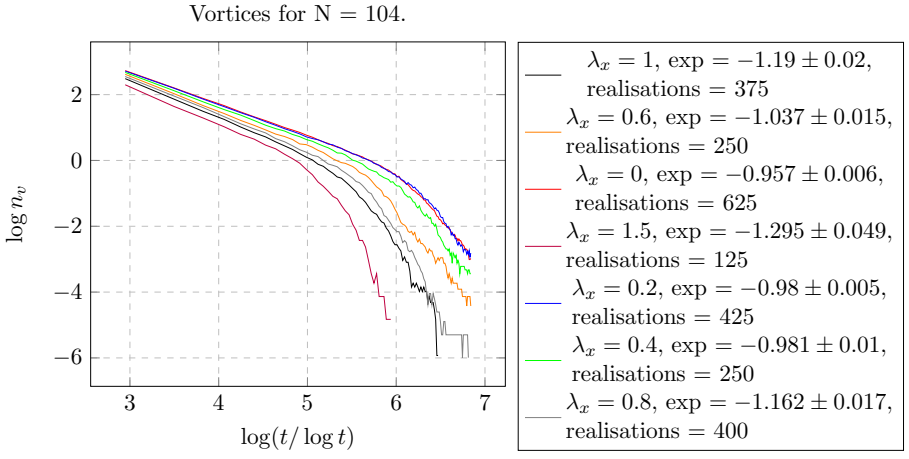

FIGURE 3.25: Log of the number of vortices as a function of $\log(t/\log t)$ for $L = 104$ in the anisotropic case.

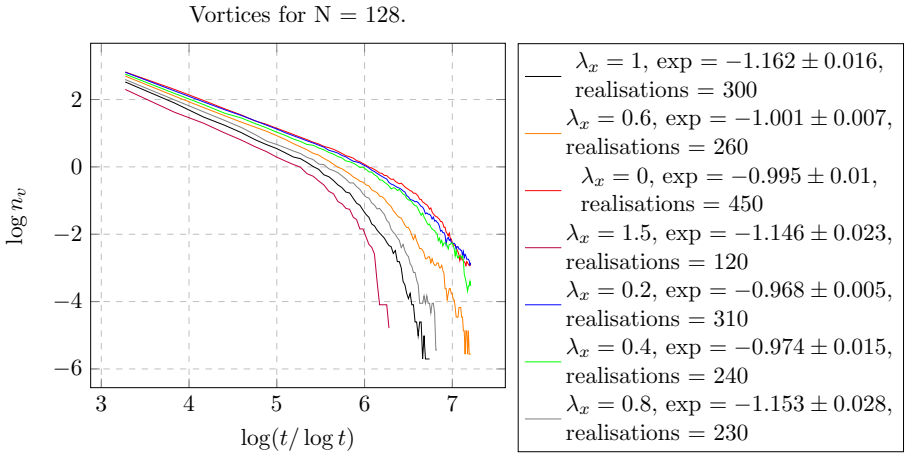

FIGURE 3.26: Log of the number of vortices as a function of $\log(t/\log t)$ for $L = 128$ in the anisotropic case.

3.3 Isotrpic case

In the isotropic case, due to the non-linearity, we expect a repulsive contribution of the vortex force at distances large calculated in [13] with a second order result, the first order being a force perpendicular to the line joining the vortices, of

$$\boldsymbol{f}(\boldsymbol{R}) = \frac{1}{8}\left(\frac{\lambda}{2D}\right)^2 + \frac{1}{\epsilon}\frac{\boldsymbol{R}}{R^2}(8\log(R/a)^2 + 4\log(R/a) - 1)$$

where R is the vortex separation, ϵ the dielectric constant, and a the lattice spacing. Thus, beyond a certain length scale vortex unbinding should occur–the vortex dominated phase–although this is not the same as the entropic unbinding in the XY case, estimated from equlibrium thermodynamics. Also, in the paper the estimate for the distance at which this correction term dominates is given by

$$L_v = ae^{\frac{2D}{\lambda}}.$$

These conclusions are matched by the results. Figure. 3.27 shows the collapse for $\lambda_x = 0.2$ and the quality is comparable to the anisotropic case. This is as anticipated as we do not expect drastic change in the qualitative behaviour away from a critical point, and provides further confidence in the obtained results. Although the points in the uncertainty graph (Figures 3.28 and 3.29) appear to all cross each other when including the errors, the uncertainties are much larger. The average vortex gradient is -0.953 ± 0.005, so has already altered unlike in the anisotropic case. This is in the direction we expect, where the the vortex recombination occurs at a slower rate due to their reduced attractions. In brief, the behaviour is slighty but not significantly different from the linear case.

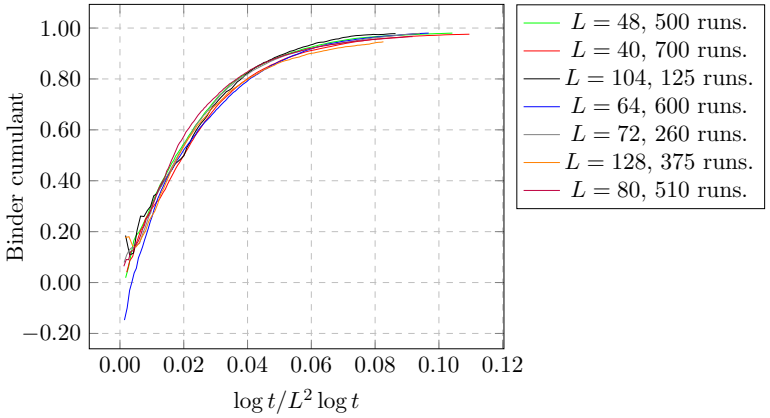

FIGURE 3.27: The binder cumulant as a function of $t/L^2 \log t$ for different system sizes and $\lambda_x = \lambda_y = 0.2$.

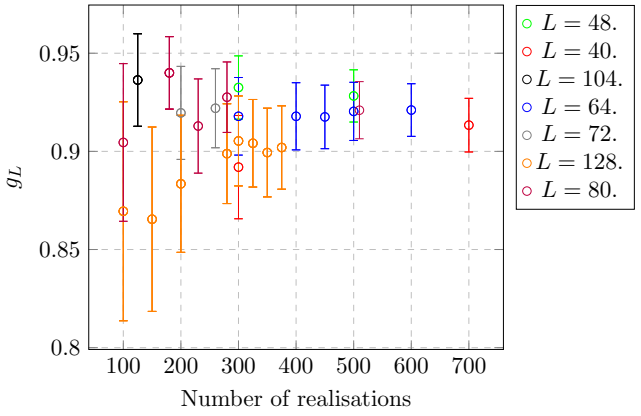

FIGURE 3.28: The uncertainty in the Binder cumulant as a function of the number of realisations at a point closest to $t/L^2 \log t$ for $t = 6500$ (the mid-point of the simulation) for $L = 40.\lambda_x = \lambda_y = 0.2$.

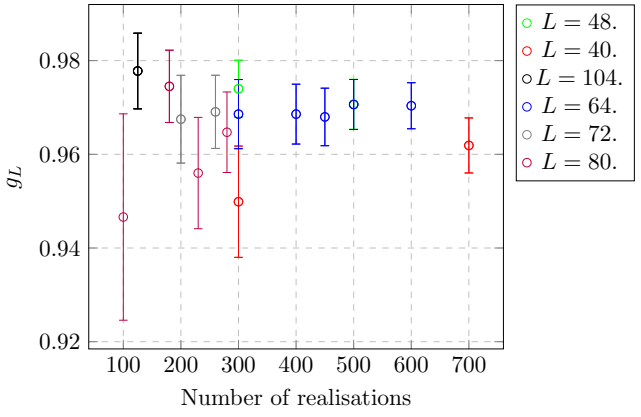

FIGURE 3.29: The uncertainty in the Binder cumulant as a function of the number of realisations at a point closest to $t/L^2 \log t$ for $t = 9370$ (the mid-point of the simulation) for $L = 40.\lambda_x = \lambda_y = 0.2$.

The case $\lambda_x = 0.4$ is more interesting, shown in Figure. 3.30, where the effects of the non-linearity are much more pronounced. This provides a clear demonstration of the expectation in that the 'destruction' of the convergence of the Binder cumulant occurs first for the largest system and gradually lessens for the smaller system as can be seen from the order of the curves in the plot. There is also a weak pattern in the increase of the vortex gradient as the system size is increased, ranging from -0.928 ± 0.015 for $L = 40$ to -0.807 ± 0.009 for $L = 128$, whereas no such pattern occurs in the linear case.

As for the uncertainties, shown in Figures 3.31 and 3.32, they are all larger and in general the points are not overlapping, although this can be seen from the graph of the Binder cumulant. The uncertainty in the sizes $L = 104$ and $L = 128$ are much larger, and for these sizes the Binder cumulant is fluctuating.

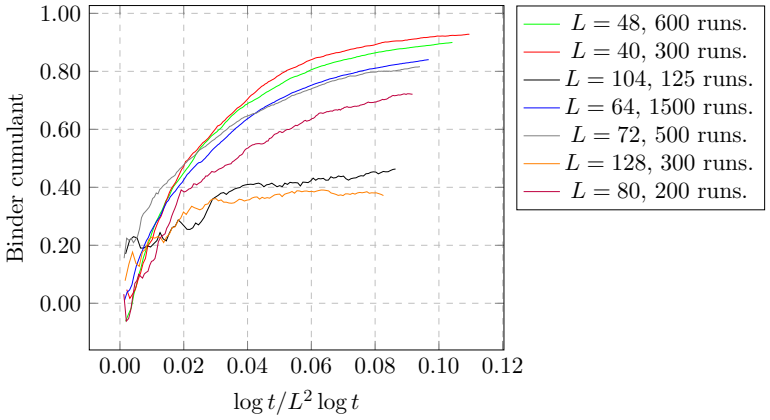

FIGURE 3.30: The Binder cumulant as a function of $t/L^2 \log t$ for different sizes with $\lambda_x = \lambda_y = 0.4$. The range has been restricted to be $(-1, 1)$.

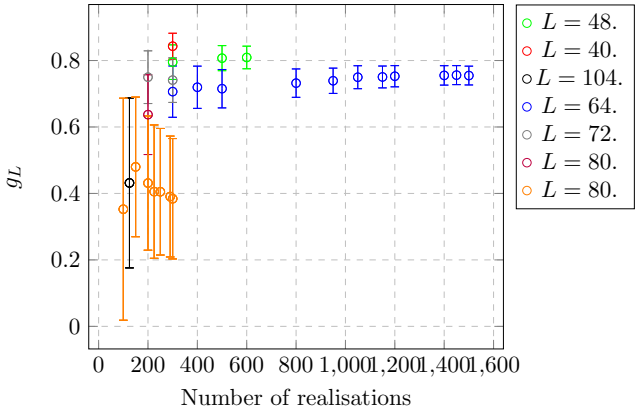

FIGURE 3.31: The uncertainty in the Binder cumulant as a function of the number of realisations at a point closest to $t/L^2 \log t$ for $t = 650$ (three quarters through the simualtion) for $L = 40$. $\lambda_x = \lambda_y = 0.4$.

By observing the behaviour of the Binder cumulant as λ is increased further, shown in Figures 3.33 through 3.33, a clear pattern emerges. The largest systems ($L = 128$ and $L = 104$) first fluctuate around 0 with a negative bias before fluctuating aronud 0

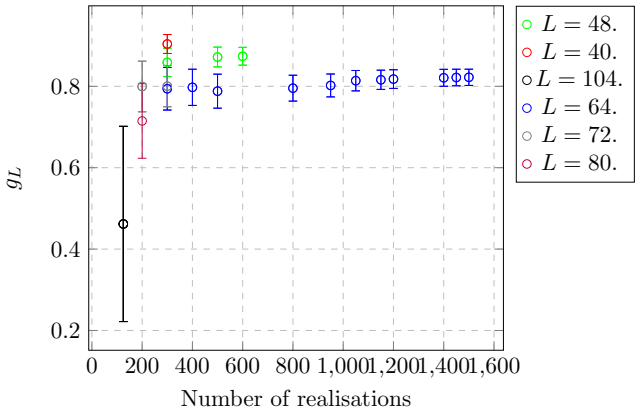

FIGURE 3.32: The uncertainty in the Binder cumulant as a function of the number of realisations at a point closest to $t/L^2 \log t$ for $t = 937.5$ (three quarters through the simualtion) for $L = 40$. $\lambda_x = \lambda_y = 0.4$. There is no value for 128.

with an amplitude of around 0.5. Smaller systems, in order of size, first begin decreasing significantly and also fluctuate quite severely, which explains the very large uncertainties (not shown) obtained, before finally also fluctuating around 0 with the same amplitude. For the larger systems the former occurs at $\lambda_x = 0.6$ to $\lambda_x = 0.8$, while $L = 72$ and $L = 80$ are finally fluctuating around 0 by $\lambda_x = 1$.

The plots do not show the very large negative values obtained by the systems, which increase as the system size is decreased. This can be explained by the fact thet Binder cumulant contains a ratio of two quantities. In the disordered state, these are small, so the ratio may be very large. We do not show the plots for $\langle \boldsymbol{M}^2 \rangle$: it suffices to point out that the value is close to zero. For a completely random phase angle at each point, the Binder cumulant should be zero. The same may not be the case in this regime. However, this transition appears to stabilise eventually. Prior to that it is interesting to note that the decrease of the Binder cumulant only occurs after a certain time and occurs quite rapidly, signaling a transition. An example in shown in Figure. 3.36 for $L = 64$. The values of $\langle \boldsymbol{M}^2 \rangle$ and $\langle \boldsymbol{M}^2 \rangle^2$ also both increased, although they still remained quite small, around 0.04 and 0.02, respectively.

The larger decrease during the transition for small sizes could be due to the structure of the vortex interactions, or also due to the lower number of phase points used in calculating the magnetisation. In this way, the behaviour may be a finite size effect: both the rate (in terms of the value of λ) of the transition to the final behaviour and also the how was reduced for larger system sizes.

For the gradient of the vortices, shown in Figures Figure. 3.37 through Figure. 3.3, they behave as expected for smaller values of λ in that increasing λ results in the gradient increasing. The repulsive force due to the non linearity should either prevent the vortices from recombining, or slow their rate of recombination. The upshot is that for

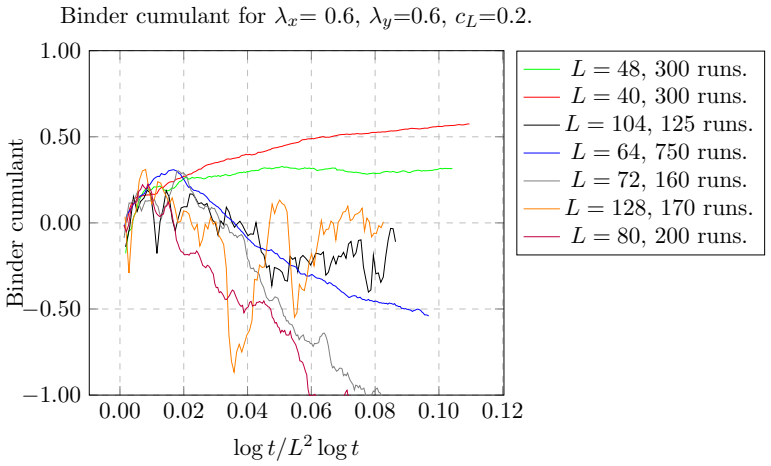

FIGURE 3.33: The Binder cumulant as a function of $t/L^2 \log t$ for different sizes with $\lambda_x = \lambda_y = 0.6$. The range has been restricted to be $(-1, 1)$.

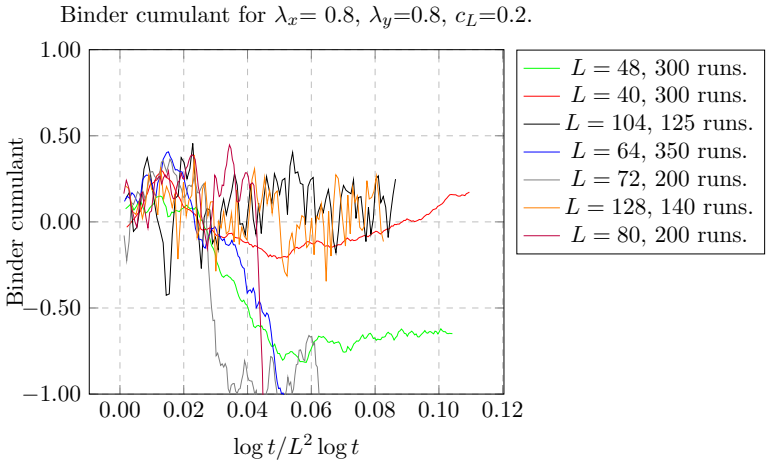

FIGURE 3.34: The Binder cumulant as a function of $t/L^2 \log t$ for different sizes with $\lambda_x = \lambda_y = 0.8$. The range has been restricted to be $(-1, 1)$.

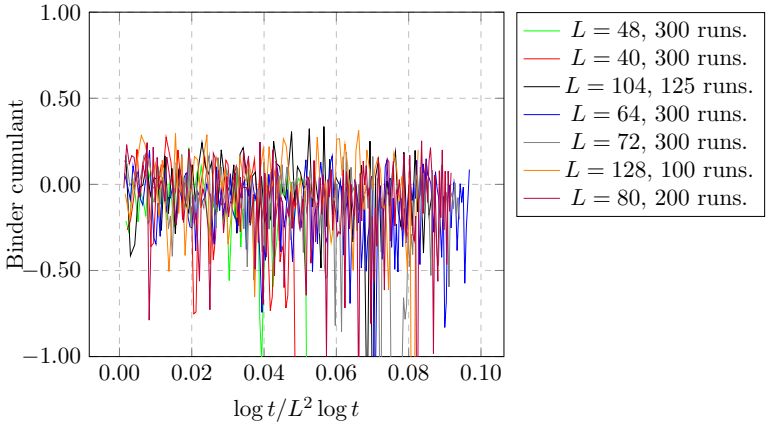

FIGURE 3.35: The Binder cumulant as a function of $t/L^2 \log t$ for different sizes with $\lambda_x = \lambda_y = 1.5$. The range has been restricted to be $(-1, 1)$.

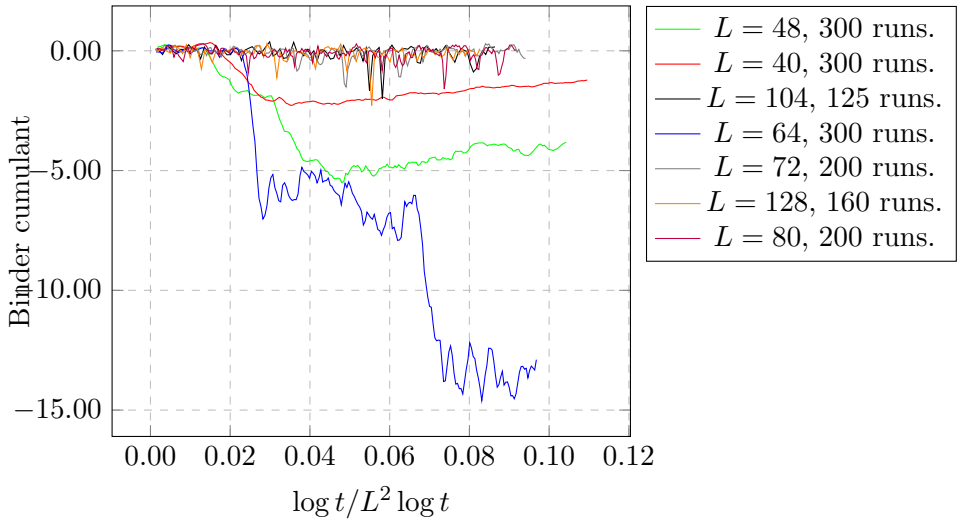

FIGURE 3.36: The Binder cumulant as a function of $t/L^2 \log t$ for different sizes with $\lambda_x = \lambda_y = 1$.

high enough values of λ there should be a saturation on the number of vortices, at least on large enough systems. The caveat of a large system is because initially the vortex density is very higher which means that vortices closest to each other can recombine and only after the vortex density is low enough does the saturation take place. In previous simulations that have been performed, eventually the number of vortices saturates for system sizes of $L = 512$.

The slightly concerning nature of these graphs is that at a sufficiently large λ the gradient actually *decreases* again, and this occurs for all system sizes with the rate being faster the larger the system. To ensure this is not perhaps a finite size effect or a numerical artefact, a simulation with size $L = 512$ was performed as shown in Figure. 3.44. Here a saturation is obtained, matching the previous result. However, the decrease in the gradient also occurs around the same values of λ as in the other system sizes, so further investigation is required.

3.4 Correlation Functions

As an aside to the main purpose of the project, a small subset of the data was utilised to calculate the form of the correlations functions and compare them to those obtained from the theory. In particular, we test the correlation function $\langle \cos(\theta_(\boldsymbol{r}) - \theta(\mathbf{0})) \rangle = g(r)$ as a function of r. We are interested in qualitative behaviour, not the in values of any exponents, and therefore this has not been done very rigorously and there are no uncertainties. As not all the data was able to be saved, specifically for higher lattice sizes, $L = 64$ was the size chosen for two different regimes: the linear regime and the

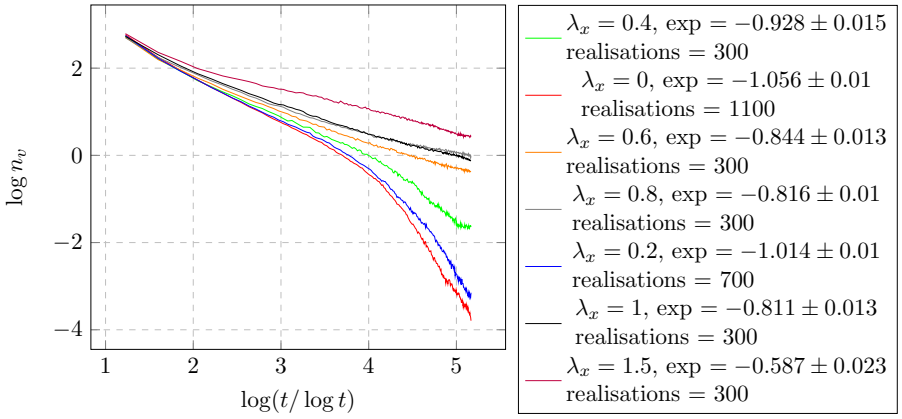

FIGURE 3.37: Log the number of vortices as a function of $\log(t/\log t)$ for $L = 40$ in the isotropic case.

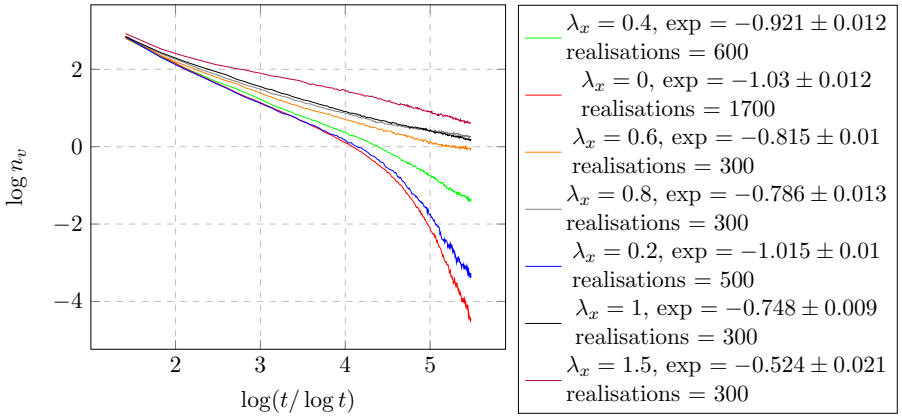

FIGURE 3.38: Log the number of vortices as a function of $\log(t/\log t)$ for $L = 48$ in the isotropic case.

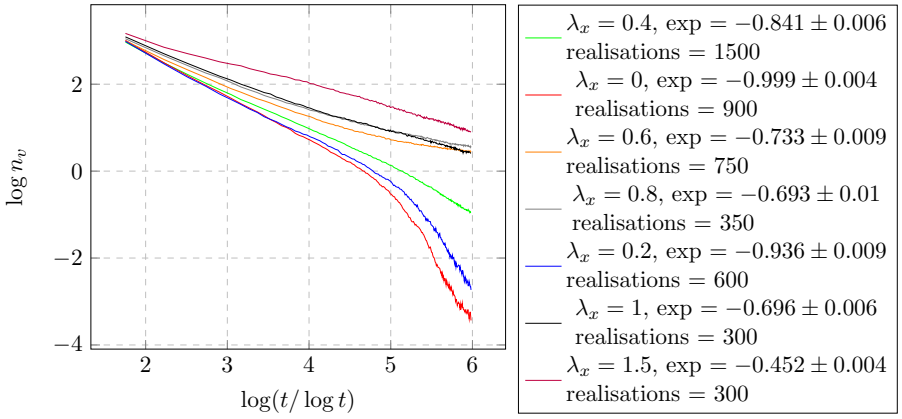

FIGURE 3.39: Log the number of vortices as a function of $\log(t/\log t)$ for $L = 64$ in the isotropic case.

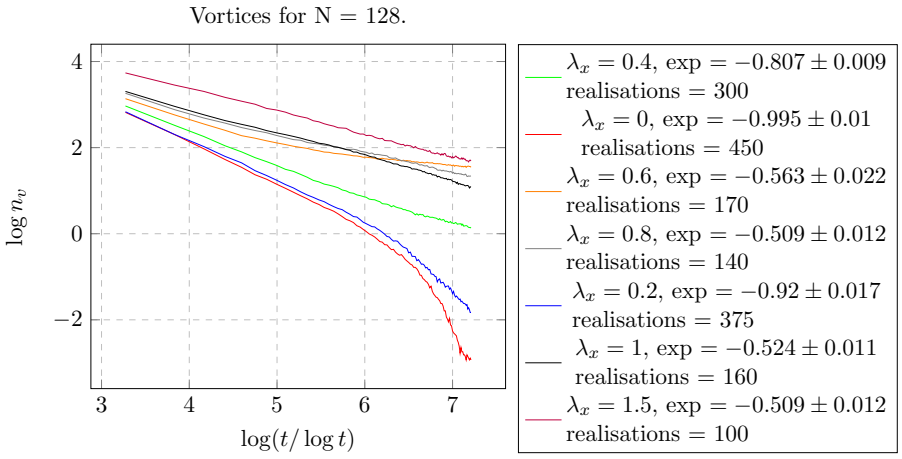

FIGURE 3.40: Log the number of vortices as a function of $\log(t/\log t)$ for $L = 128$ in the isotropic case.

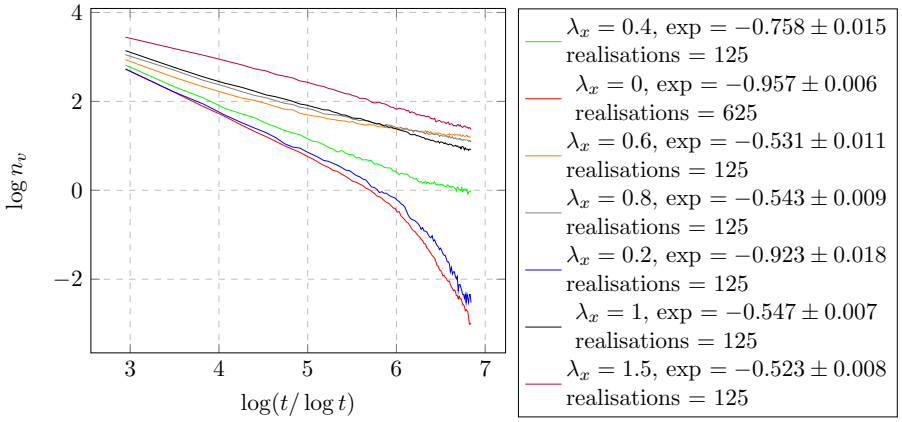

FIGURE 3.41: Log the number of vortices as a function of $\log(t/\log t)$ for $L = 104$ in the isotropic case.

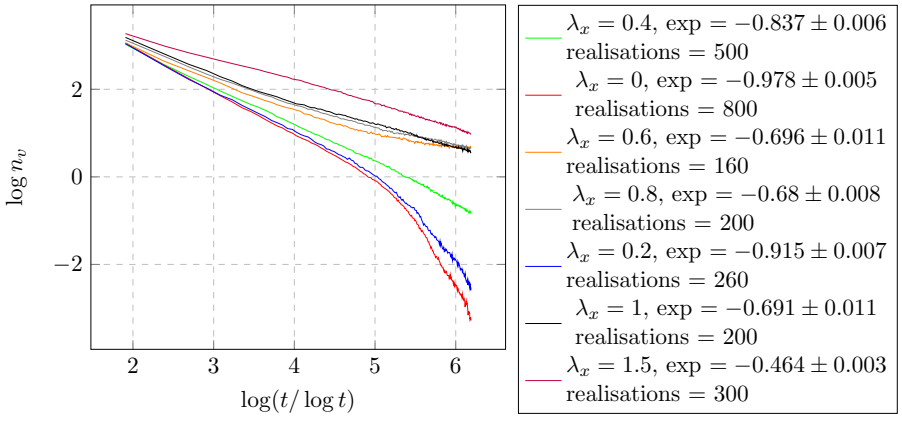

FIGURE 3.42: Log the number of vortices as a function of $\log(t/\log t)$ for $L = 72$ in the isotropic case.

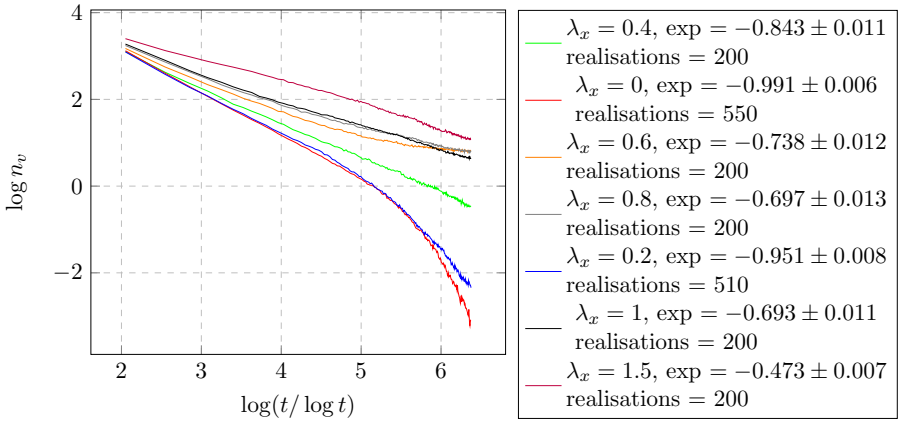

FIGURE 3.43: Log the number of vortices as a function of $\log(t/\log t)$ for $L = 80$ in the isotropic case.

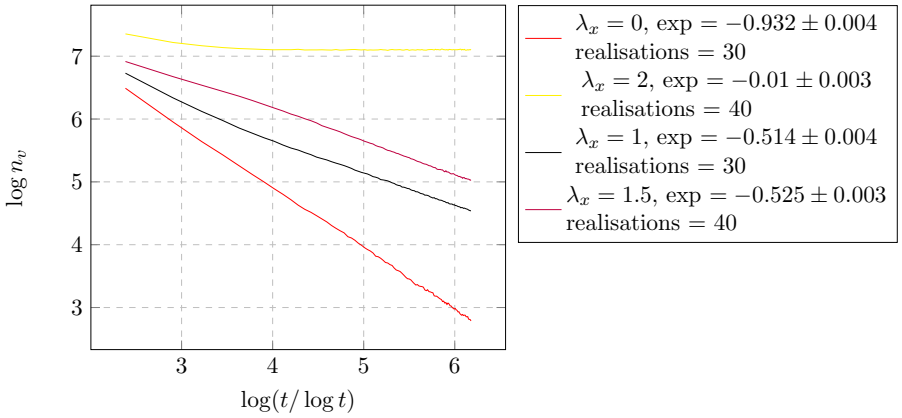

FIGURE 3.44: Log the number of vortices as a function of $\log(t/\log t)$ for $L = 512$ in the isotropic case.

highly ($\lambda = 1.5$) non-linear isotropic regime. For the isotropic case either exponential or stretched exponential correlations are expected. To differentiate these would require exact values (a gradient of 1 in a $\log(-\log g(r))$ vs $\log r$ plot or a value other than one), therefore our approach is not suitable.

In the linear case, Figure. 3.45 shows the correlation at the beginning of the siulation. Since the spins were completely random, we expect the correlation to be 0 for all distances. Although this is primarily to ensure the simulation and correlations behave as expected as we manually insert these conditions, it corresponds to the infinite temperature case with a correlation length of zero and therefore a correlation $exp(-r/\xi)$ of zero at all distances. Small scale fluctuations about 0 are observed, which we take as a confirmation of the desired behaviour.

At the end of the simulation in the linear regime, the system is in the ordered phase where the correlations are algebraic. We plot $\log g(r)$ as a function of $\log r$ which is expected to be linear (with a negative gradient). As shown in Figure. 3.46, we obtain

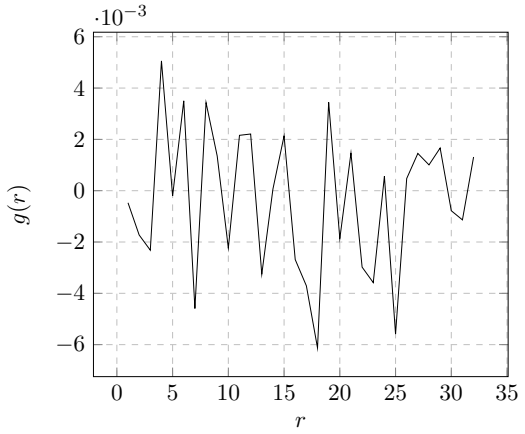

FIGURE 3.45: The spatial correlation function as a function of distance for $L = 64$ linear case with an inifinite noise (temperature).

a reasonable plot. The curve is linear to good approximation. A change in gradient appears as the end is approached. This distance is half the lattice size. The periodic boundary conditions may be interfering with the expected behaviour–in other words, a finite size effect. As usual, perforing this on a large system would have been prefereble. Nonetheless the results support, or at the very least do not conflict with, the expectation.

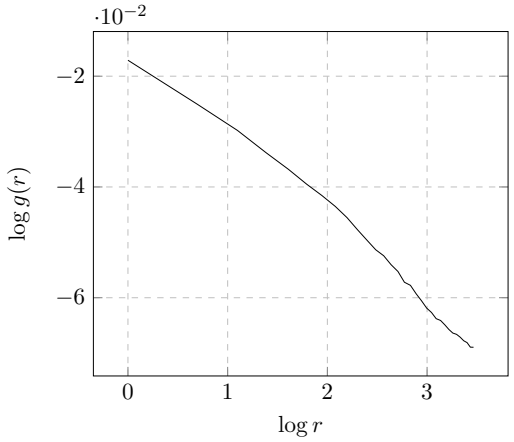

FIGURE 3.46: The spatial correlation function as a function of distance for the $L = 64$ linear case in the ordered phase.

The anisotropic case is displayed in Figure. 3.47. Like the linear case, the approximation of a linear curve is also quite good. The magnitudes of the correlation are also of a similar magnitude up to ~ 2, but then their values deviate from each other. This perhaps is further support that finite size effects are responsible for the non linear behaviour beyond this point. This data sheds light on previous results: the expectation that the anisotropic regime would be that of the XY model was obtained through a renormalisation group analysis. For long time behaviour (once the system has reached steady state), the correlation functions support this prediction. However the approach to this phase is not determined and is why the behaviour of the linear and anisotropic regimes is different.

This can be see in the plot of the correlation functions roughly midway through the simulation for both anisotropic and linear cases, shown in Figure. 3.48. Clearly, despite the correlations being the same at the end by the simulation, the linear case takes longer to reach steady state.

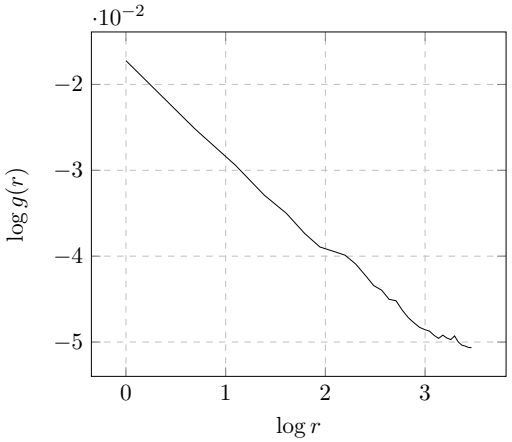

FIGURE 3.47: The spatial correlation function as a function of distance for the $L = 64$ and $\lambda = 1.5$ anisotropic case.

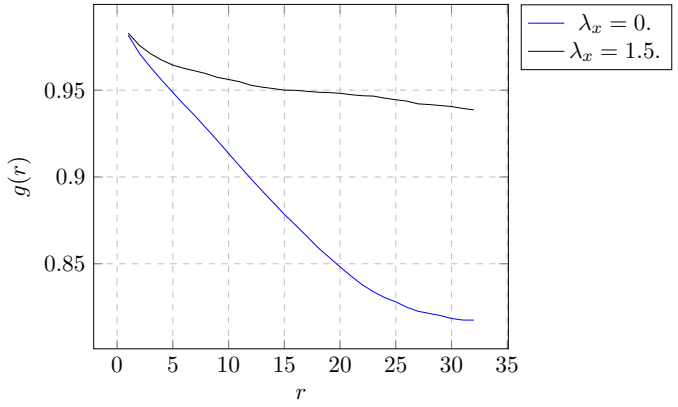

FIGURE 3.48: The spatial correlation function as a function of distance for the $L = 64$ and $\lambda = 1.5$ anisotropic case and the linear case. The linear case is at $t = 1424$ and the anisotropic is at $t = 1200$.

Bibliography

[1] Alexander Altland. *Condensed Matter Field Theory*. Leiden: Cambridge University Press, 2010. ISBN: 978-0521769754.

[2] E. Altman et al. "Two-Dimensional Superfluidity of Exciton Polaritons Requires Strong Anisotropy". In: *Physical Review X* 5.1, 011017 (Jan. 2015), p. 011017. DOI: 10.1103/PhysRevX.5.011017.

[3] J. Bardeen, L. N. Cooper, and J. R. Schrieffer. "Theory of Superconductivity". In: *Phys. Rev.* 108 (5 Dec. 1957), pp. 1175–1204. DOI: 10.1103/PhysRev.108.1175. URL: https://link.aps.org/doi/10.1103/PhysRev.108.1175.

[4] A. J. Bray, A. J. Briant, and D. K. Jervis. "Breakdown of Scaling in the Nonequilibrium Critical Dynamics of the Two-Dimensional XY Model". In: *Phys. Rev. Lett.* 84 (7 Feb. 2000), pp. 1503–1506. DOI: 10.1103/PhysRevLett.84.1503. URL: https://link.aps.org/doi/10.1103/PhysRevLett.84.1503.

[5] Tim Byrnes, Na Young Kim, and Yoshihisa Yamamoto. "Exciton-polariton condensates". In: *Nat Phys* 10.11 (Nov. 2014). Review, pp. 803–813. ISSN: 1745-2473. URL: http://dx.doi.org/10.1038/nphys3143.

[6] Leiming Chen and John Toner. "Universality for Moving Stripes: A Hydrodynamic Theory of Polar Active Smectics". In: *Phys. Rev. Lett.* 111 (8 Aug. 2013), p. 088701. DOI: 10.1103/PhysRevLett.111.088701. URL: https://link.aps.org/doi/10.1103/PhysRevLett.111.088701.

[7] Sidney Coleman. "There are no Goldstone bosons in two dimensions". In: *Communications in Mathematical Physics* 31.4 (Dec. 1973), pp. 259–264. URL: https://doi.org/10.1007/BF01646487.

[8] G. Dagvadorj et al. "Nonequilibrium Phase Transition in a Two-Dimensional Driven Open Quantum System". In: *Phys. Rev. X* 5 (4 Nov. 2015), p. 041028. DOI: 10.1103/PhysRevX.5.041028. URL: https://link.aps.org/doi/10.1103/PhysRevX.5.041028.

[9] Jonathan Keeling and Natalia G. Berloff. "Exciton-polariton condensation". In: *Contemporary Physics* 52.2 (2011), pp. 131–151. DOI: 10.1080/00107514.2010.550120. eprint: http://dx.doi.org/10.1080/00107514.2010.550120. URL: http://dx.doi.org/10.1080/00107514.2010.550120.

[10] L. M. Sieberer et al. "Lattice duality for the compact Kardar-Parisi-Zhang equation". In: *Physical Review B* 94.10, 104521 (Sept. 2016), p. 104521. DOI: 10.1103/PhysRevB.94.104521. arXiv: 1604.01043 [cond-mat.stat-mech].

[11] M. Steger et al. "Slow reflection and two-photon generation of microcavity exciton-polaritons". In: *ArXiv e-prints* (Aug. 2014). arXiv: 1408.1680 [physics.optics].

[12] Yongbao Sun et al. "Bose-Einstein Condensation of Long-Lifetime Polaritons in Thermal Equilibrium". In: *Phys. Rev. Lett.* 118 (1 Jan. 2017), p. 016602. DOI: 10.1103/PhysRevLett.118.016602. URL: https://link.aps.org/doi/10.1103/PhysRevLett.118.016602.

[13] G. Wachtel et al. "Electrodynamic duality and vortex unbinding in driven-dissipative condensates". In: *Phys. Rev. B* 94 (10 Sept. 2016), p. 104520. DOI: 10.1103/PhysRevB.94.104520. URL: https://link.aps.org/doi/10.1103/PhysRevB.94.104520.

[14] A. Zamora et al. "Tuning across Universalities with a Driven Open Condensate". In: *Phys. Rev. X* 7 (4 Oct. 2017), p. 041006. DOI: 10.1103/PhysRevX.7.041006. URL: https://link.aps.org/doi/10.1103/PhysRevX.7.041006.

Appendix A

Source Code

```
#ifndef LATTICE_H
#define LATTICE_H

#include <vector>
#include <iostream>
#include <math.h>

/* Header code for the lattice templates and
 * implementations
 */



/*
 * Basic lattice template
 */
template<typename T> class lattice{
        private:
                const unsigned int N; //size
                std::vector<T> lattice_points = std::vector<T>(pow(N, 2)); //array of points
                static int point_1D(int, int, int);
                static int mv_inside(int, int);
        protected:
                template <typename R, typename S> static  T mod(R, S) ;
        public:
                lattice(unsigned int);
                lattice(const lattice&);
                lattice& operator=(const lattice&);
                ~lattice() {};

                T point(int, int) const;
                void set(int, int, T);
                unsigned int size() const;
};

template <typename T>
unsigned int lattice<T>::size() const {
        return N;
```

```
}

template <typename T>
int lattice<T>::point_1D(int i, int j, int N) {
        return N*mv_inside(i, N) + mv_inside(j, N);
};

template <typename T>
int lattice<T>::mv_inside(int i, int j) {
        if(i >= 0 && i < j){
                return i;
        } else if(i < 0) {
                return j - 1;
        } else {
        return 0;
        };
};

template <typename T>
T lattice<T>::point(int x, int y) const {
        return (*this).lattice_points[point_1D(x, y, N)]; //mod for periodic boundaries
};

template <typename T>
void lattice<T>::set(int x, int y, T val) {
        this->lattice_points[point_1D(x, y, N)] = val; //mod for periodic boundaries
};

/*
 * Constructors for lattice templates
 */

template<typename T>
lattice<T>::lattice(unsigned int s) : N(s) {};

template<typename T>
lattice<T>::lattice(const lattice& lat) : N(lat.size()) {
            this-> lattice_points = lat.lattice_points;
};

template <typename T>
lattice<T>& lattice<T>::operator=(const lattice& lat) {
        if( this -> size() != lat.size()) {
                throw std::invalid_argument("lattices not the same size");
        };
        this -> lattice_points = lat.lattice_points;
        return *this;
};

/*
 * Prints lattice to standard output
*/
template <typename T>
void print(lattice<T> const &lat) {
        for(int j = 0; j < lat.size(); j++) {
```

```
                for(int i = 0; i < lat.size(); i++) {
                        std::cout << lat.point(i, j) << " ";
                };
                std::cout << std::endl;
        };
};

/*
 *returns the positive a mod b
 */
template <typename T>
template <typename R, typename S>
T lattice<T>::mod(R a, S b) {
        T ret = fmod(a, b);
        if(ret < 0) {
                ret += b;
        };
        return ret;
};

/*
 * Lattice of angles
 */
class angle_lattice : public lattice<double> {
        public:
                virtual void set(int, int, double); // overloaded to be mod 2pi;
                angle_lattice(int N);
};

/*
 * Vortex lattice class which includes vortex number member
 * variables
 */
class vortex_lattice : public lattice<int> {
        private:
                int n_vor = 0;
                int n_anti_vor = 0;
        public:
                virtual void set(int, int, int); //overloaded so each point is an integer;
                vortex_lattice(int N);
                int num_vor() const {return n_vor;};
                int num_anti_vor() const {return n_anti_vor;}
                /*
                 * Sets the vortex lattice from the angle lattice lat
                 */
        vortex_lattice& set_vortex_lattice(angle_lattice const& lat);
};

template <typename T> int sign(T val) {
        if(val > 0) {return 1;};
        if(val < 0) {return -1;};
        return 0;
};

#endif
```

```
#include <vector>
#include <iostream>
#include <math.h>
#include "system.h"
#include <stdexcept>
#include "vortex_calculation.h"


void angle_lattice::set(int x, int y, double ang) {
        lattice<double>::set(x, y, mod(ang, 2*M_PI));
};

/*
 * Constructor
 */
angle_lattice::angle_lattice(int N) : lattice<double>(N) {};

/*
 * Sets (x,y) to charge 'charge' and updates the number of charges accordingly
 */
void vortex_lattice::set(int x, int y, int charge) {
        int pre_charge = this -> point(x, y);
        int added_charge = sign(charge);
        lattice<int>::set(x, y, added_charge);
        if(pre_charge == 0 && charge == 0) {
                return;
        }
        else if(pre_charge != 0 && charge != 0){
                n_vor += added_charge;
                n_anti_vor += pre_charge;
        }
        else if (pre_charge == 0) {
                if(charge > 0) {
                        n_vor++;
                } else {
                        n_anti_vor++;
                };
        } else {
                if(pre_charge > 0) {
                        n_vor--;
                } else {
                        n_anti_vor--;
                };
        };
};

/*
 * Constructor
 */
vortex_lattice::vortex_lattice(int N) : lattice<int>(N) {};

/*
 *Returns a vortex lattice set from the angle lattice lat
 */
```

```
vortex_lattice& vortex_lattice::set_vortex_lattice(angle_lattice const& lat) {
        if((*this).size() != lat.size()){
                std::cerr << "Vortex lattice and angle lattice not the same size. Error. " << st
                exit(1);
        };
        for(int i = 0; i < lat.size(); i++) {
                for(int j = 0; j < lat.size(); j++) {
                        (*this).set(i, j, circulation(lat, i, j));
                };
        };
        return *this;
};
```

```
#ifndef SIMULATION_H
#define SIMULATION_H

#include "updater.h"
#include "fstream"
#include <string>
#include "math.h"
#include <random>

/*
 * Simulation parameters object. These related to the simulation type.
 */
class sim_parameters {
        friend void simulate(int, parameters&, sim_parameters&, std::string, int);
        friend void multiple_simulate(int, parameters&, sim_parameters&, std::string);
        friend void update_progress_bar(int, sim_parameters&, int&);
        private:
                const double dt;
                const unsigned int num_iter;
                const unsigned int num_sim;
                const unsigned int num_prev_sim;
                const unsigned int num_per_save;
                const std::string init_cond;
        public:
                sim_parameters(double timestep, int iter, int prev, int sim, int save, std::stri
                        dt(timestep), num_iter(iter), num_sim(sim), num_prev_sim(prev), num_per_
                        init_cond(initial) {}
};



void simulate(int, parameters&, sim_parameters&, std::string, int);
void multiple_simulate(int, parameters&, sim_parameters&, std::string);
void update_progress_bar(int, sim_parameters&, int&);


#endif
```

```
#include "simulation.h"
#include "system.h"
#include "updater.h"
#include <iostream>
```

```
#include <fstream>
#include <string>
#include "lattice_io.h"
#include "initial_condition.h"

void update_progress_bar(int, sim_parameters&, int&);


void simulate(int lattice_size, parameters& par, sim_parameters& sim_par, std::string dir, int s
        int counter = 1;
        int progress_counter = 1;
        angle_lattice lat(lattice_size);
        std::cout << sim_par.init_cond;
        if(sim_par.init_cond == "disordered") {
                lat = disordered(lattice_size);
        };
        angle_lattice new_lat(lattice_size);
        std::string path = dir + "/simulation:dt," + std::to_string(sim_par.dt)
                + ",iter_num," + std::to_string(0)
                + ",sim_num," + std::to_string(sim_number) + ".txt";

        lattice_write(lat, path);

        for(int iteration_num = 0; iteration_num < sim_par.num_iter; iteration_num++) {
                update_progress_bar(iteration_num, sim_par, progress_counter);
                new_lat = update_lattice(lat, par, sim_par.dt);

                if(counter == sim_par.num_per_save) {
                        counter = 0;
                        std::string path = dir + "/simulation:dt,"+ std::to_string(sim_par.dt) +
                                + ",iter_num," + std::to_string(iteration_num + 1)
                                + ",sim_num," + std::to_string(sim_number) + ".txt";
                        lattice_write(new_lat, path);
                };
                counter++;
                lat = new_lat;
        };
};

void multiple_simulate(int lattice_size, parameters& par, sim_parameters& sim_par, std::string d
        for(int i = sim_par.num_prev_sim; i < sim_par.num_sim + sim_par.num_prev_sim; i++) {
                std::cout << "Simulation " << i+1 << ":" << std::endl;
                simulate(lattice_size, par, sim_par, dir, i+1);
                std::cout << std::endl;
        };
};

void update_progress_bar(int iteration_num, sim_parameters& sim_par, int& progress_counter) {
        if(iteration_num == 0){
                std::cout << "[";
        };
        if(( (double) iteration_num)/sim_par.num_iter*10 > progress_counter) {
                std::cout << "#" << std::flush;
                progress_counter++;
        };
```

```
        if(iteration_num == (sim_par.num_iter - 1)){
                std::cout << "]" << std::flush;
        };
};
```

```
#ifndef SIMULATION_UTILITY_H
#define SIMULATION_UTILITY_H
#include <iostream>
#include <sstream>
#include <vector>
#include <string>
#include "lattice_io.h"
/*
template <typename T> bool value_check(std::string line, T&val) {
        std::istringstream is {line};
        if((is >> val) && !(is >> line)){
                return true;
        } else {
                return false;
        };
};

template <typename T> void value_save(std::string name, T& val) {
        std::cout << name << ": ";

        std::string line;
        while(std::getline(std::cin, line)){
                std::istringstream is {line};
                if(value_check(line, val)){
                        break;
                };
                std::cerr << "Invalid input. Please try again: ";
        };
};

template <typename T> bool check_positive_sign(T& val) {
        if(val > 0) {
                return true;
        } else {
                return false;
        };
};
*/

void check_dir(std::string&);

bool check_files_exist(std::vector<std::string>);



#endif
```

```
#include "simulation_utility.h"

void check_dir(std::string& dir) {
```

```
        while(!check_dir_exists(dir)) {
                std::cout << "Directory does not exist. Please try again:" << std::endl;
                getline(std::cin, dir);
        };
};

bool check_files_exist(std::vector<std::string> files) {
        if(files.size() == 0){
                return false;
        } else {
                return true;
        };
};
```

```
#ifndef MAGNETISATION_H
#define MAGNETISATION_H
#include "system.h"
#include <vector>
#include <map>

std::vector<double> angle_components(double const&); // (cos val, sin val) from val

std::vector<double> calculate_components(angle_lattice const&); // av angle_components in the la

/*
 *  Stores av angle_components in lattice
 */
class magnetisation {
        private:
                std::vector<double> comp;
        public:
                std::vector<double> components() const {return comp;};
                magnetisation(angle_lattice const& lat) : comp(calculate_components(lat)) {};
};

/*
 * Stores the average of the magnitude of the magnetisation to a power
 */
class av_magnetisation {
        private:
                double total;
                unsigned int averaging_num;
                const int power;
        public:
                        int get_power() const {return power;}
            av_magnetisation(int pow) : power(pow) {};
            av_magnetisation& add(magnetisation const&); // reaverages with the addition on obje
            double get_average() const;
            av_magnetisation() : power(0), total(0), averaging_num(0) {};
                        double get_total() const { return total;};
                        double get_averaging_num() const {return averaging_num;};
                        av_magnetisation(int pow, int av_num, double tot) : power(pow), averagin
};

double magnitude_pow(magnetisation const&, int);
```

```
double binder_cumulant(av_magnetisation const&, av_magnetisation const&);

std::map<double, double> make_binder_cumulant_map(std::map<double, av_magnetisation>,
                                                  std::map<double, av_magnetisatio

void print(magnetisation const&);

#endif
```

```
#include "magnetisation.h"
#include <iostream>
#include <math.h>
#include <map>

av_magnetisation& av_magnetisation::add(magnetisation const& mag) {
        (*this).total += magnitude_pow(mag, (*this).power);
        (*this).averaging_num++;
        return *this;
};

double av_magnetisation::get_average() const {
        if ((*this).averaging_num == 0) {
                return 0;
        } else {
                return (*this).total/(*this).averaging_num;
        };
};

std::vector<double> angle_components(double const& val) {
        std::vector<double> comps = {cos(val), sin(val)};
        return comps;
};

std::vector<double> calculate_components(angle_lattice const& lat) {
        std::vector<double> comp = {0, 0};
                int N = lat.size();
                for(int i = 0; i < N; i++) {
                for(int j = 0; j < N; j++) {
                        std::vector<double> added_comps = angle_components(lat.point(i, j));
                        for(int k = 0; k < comp.size(); k++) {
                                                comp[k] += added_comps[k];
                        };
                };
        };
        for(int i = 0; i < comp.size(); i++) {
                double val = comp[i]/(N*N);
                comp[i] = val;
        };
        return comp;
};

double magnitude_pow(magnetisation const& mag, int n) {
        double val = 0;
        for(int i = 0; i < mag.components().size(); i++) {
```

```
                val += pow(mag.components()[i], 2);
        };
                double exp = ((double) n)/2;
        return pow(val, exp);
};

double binder_cumulant(av_magnetisation const& av_mag_2, av_magnetisation const& av_mag_4) {
        if(av_mag_2.get_power() !=2 || av_mag_4.get_power() !=4) {
                std::cerr << "Error. The powers in the magnetisations are not correct" << std::e
                exit(1);
        };
        return 2 - av_mag_4.get_average()/(pow(av_mag_2.get_average(), 2));
};

std::map<double, double> make_binder_cumulant_map(std::map<double, av_magnetisation> mag2_map, s
        if(mag2_map.at(0).get_power() != 2 || mag4_map.at(0).get_power() !=4 ) {
                std::cerr << "Cannot make binder cumulant map as incorrect powers of average mag
                exit(1);
        };
        std::map<double, double> binder_map;
        for(std::map<double, av_magnetisation>::const_iterator iter = mag2_map.cbegin(); iter !=
                double time = iter -> first;
                std::cout << time << " " << mag2_map.at(time).get_average() << " " << mag4_map.a
                binder_map[time] = binder_cumulant(mag2_map.at(time), mag4_map.at(time));
                std::cout << binder_map[time] << std::endl;
        };
        return binder_map;
};

void print(magnetisation const& mag) {
        std::cout << mag.components()[0] << " " << mag.components()[1] << std::endl;
};
```

```
#ifndef VORTEX_CALCULATION_H
#define VORTEX_CALCULATION_H

#include "system.h"


int circulation(angle_lattice const&, int, int);
double angle_difference(angle_lattice const&, int, int, int, int);

/*
 * Stores the number of vortices and anti vortices in an angle lattice
 */
class vortex_number {
        private:
                const unsigned int num_vor;
                const unsigned int num_anti_vor;
        public:
                vortex_number(int n_v, int n_av) : num_vor(n_v), num_anti_vor(n_av) {};
                int get_num_vor() const {return num_vor;};
                int get_num_anti_vor() const {return num_anti_vor;};
};
```

```
/*
 * Stores average number of vortices and anti vortices as you
 * 'add' vortex_number objects to it
 */
class av_vortex_number {
        private:
                double n_vor;
                double n_anti_vor;
                unsigned int averaging_num;
        public:
                double get_av_vor() const;
                double get_av_anti_vor() const;
                av_vortex_number() : n_vor(0), n_anti_vor(0), averaging_num(0) {};
                                av_vortex_number& add(vortex_number const&); // reaverages vorte
                                av_vortex_number(int av_num, double n_v, double n_av) : averagin
                                n_anti_vor(n_av) {};
};

vortex_number make_vortex_number(angle_lattice const&);

#endif
```

```
#include "vortex_calculation.h"
#include <math.h>
/*
vortex_lattice make_vor_lattice(angle_lattice const& lat) {
        vortex_lattice vor_lat(lat.size());
        for(int i = 0; i < lat.size(); i++) {
                for(int j = 0; j < lat.size(); j++) {
                        vor_lat.set(i, j, circulation(lat, i, j));
                };
        };
        return vor_lat;
};
*/


int circulation(angle_lattice const& lat, int i, int j) {
        double total_angle = 0;
        total_angle += angle_difference(lat, i+1, j, i, j);
        total_angle += angle_difference(lat, i+1, j+1, i+1, j);
        total_angle += angle_difference(lat, i, j+1, i+1, j+1);
        total_angle += angle_difference(lat, i, j, i, j+1);

        double residual = 1.0e-12;
        if(total_angle <= -2*M_PI + residual) {
                return -1;
        } else if(total_angle >= 2*M_PI - residual) {
                return 1;
        } else {
                return 0;
        };
};

double angle_difference(angle_lattice const& lat, int i1, int j1, int i2, int j2){
```

```
        double dif = lat.point(i1, j1) - lat.point(i2, j2);
        if (dif < -M_PI) {
                dif += 2*M_PI;
        } else if(dif > M_PI) {
                dif -= 2*M_PI;
        };
        return dif;
};

vortex_number make_vortex_number(angle_lattice const& lat) {
        int num_vor = 0;
        int num_anti_vor = 0;
        for(int i = 0; i < lat.size(); i++) {
                for(int j = 0; j < lat.size(); j++) {
                        int circ = circulation(lat, i, j);
                        if(circ > 0) {
                                num_vor++;
                        } else if(circ < 0) {
                                num_anti_vor++;
                        };
                };
        };
    return vortex_number(num_vor, num_anti_vor);
};

av_vortex_number& av_vortex_number::add(vortex_number const& vor) {
        (*this).n_vor += vor.get_num_vor();
        (*this).n_anti_vor += vor.get_num_anti_vor();
        (*this).averaging_num++;
        return *this;
};

double av_vortex_number::get_av_vor() const {
        if ((*this).averaging_num == 0) {
                return 0;
        } else {
                return (*this).n_vor/(*this).averaging_num;
        };
};
double av_vortex_number::get_av_anti_vor() const {
        if ((*this).averaging_num == 0) {
                return 0;
        } else {
                return (*this).n_anti_vor/(*this).averaging_num;
        };
};
```

```
#include "system.h"
#include "vortex_calculation.h"
#include "lattice_io.h"
#include "sstream"
#include "simulation_utility.h"

int main() {
```

```
std::cout << "Welcome to the vortex location extraction. Please type the directory where the fil
                  << "that has the simulation results:" << std::endl;
        std::string read_directory;
        getline(std::cin, read_directory);
        check_dir(read_directory);

        std::cout << "Please type the directory in which to save the files:" << std::endl;
        std::string write_directory;
        getline(std::cin, write_directory);
        check_dir(write_directory);
        std::vector<std::string> file_vec = list_files(read_directory);
        if(!check_files_exist(file_vec)){
                std::cerr << "No files!" << std::endl;
                exit(1);
        };

                std::cout << "Please type in the simulation number:" << std::endl;
                std::string sim_num_str;
                getline(std::cin, sim_num_str);
                std::stringstream sim_num_ss(sim_num_str);
                int sim_num = 0;
                sim_num_ss >> sim_num;
                std::cout << "We will perform vortex location analysis from directory " << read_
                                  << " and save the results to " << write_directory <<  std::end
                vortex_lattice vor_lat = empty_vortex_lattice_from_path(file_vec[0]);
                angle_lattice  ang_lat = empty_angle_lattice_from_path(file_vec[0]);
                std::string dt = std::to_string(deltat(file_vec[0]));
                for(std::string file : file_vec) {
                        if(simulation_number(file) != sim_num) {
                                continue;
                        };
                        lattice_read(ang_lat, file);
                        vor_lat.set_vortex_lattice(ang_lat);
                        std::stringstream stream(file);
                        std::string file_name = write_directory +  "/vortex_loc:dt," + dt + ",it
                        lattice_write(vor_lat, file_name);
                };
};
```

```
#include <iostream>
#include "simulation_utility.h"
#include "lattice_io.h"
#include "magnetisation.h"

int main() {
    std::cout << "Welcome to the magnetisation data extraction. Please type the directory where
                          << "that has the simulation results:" << std::endl;
    std::string read_directory;
    getline(std::cin, read_directory);
    check_dir(read_directory);

    std::cout << "Please type the directory in which to save the file:" << std::endl;
    std::string write_directory;
    getline(std::cin, write_directory);
    check_dir(write_directory);
```

```
    std::vector<std::string> file_vec = list_files(read_directory);
    if(!check_files_exist(file_vec)){
            std::cerr << "No files!" << std::endl;
            exit(1);
    };
        std::cout << file_vec.size();
    std::cout << "Please type in the file you want to name this:" << std::endl;
    std::string file_name;
    getline(std::cin, file_name);

        std::cout << "This is a magnetisation data extraction from directory " << read_directory
                     << " and saved in " << write_directory << "/" << file_name << std::endl;

        std::vector<std::map<double, av_magnetisation>> mag_map_list = make_magnetisation_map(fi
    std::map<double, double> binder_cumulant_map = make_binder_cumulant_map(mag_map_list[0], mag
        write_binder_cumulant(binder_cumulant_map, write_directory, file_name);
        write_magnetisation(mag_map_list[0], write_directory, "mag_2" + file_name);
        write_magnetisation(mag_map_list[1], write_directory, "mag_4" + file_name);
};
```

```
#include <iostream>
#include "simulation_utility.h"
#include "lattice_io.h"
#include "vortex_calculation.h"

int main() {
        std::cout << "Welcome to the vortex data extraction. Please type the directory where the
                  << "that has the simulation results:" << std::endl;
        std::string read_directory;
        getline(std::cin, read_directory);
        check_dir(read_directory);

        std::cout << "Please type the directory in which to save the file:" << std::endl;
        std::string write_directory;
        getline(std::cin, write_directory);
        check_dir(write_directory);
        std::vector<std::string> file_vec = list_files(read_directory);
        if(!check_files_exist(file_vec)){
                std::cerr << "No files!" << std::endl;
                exit(1);
        };

        std::cout << "Please type in the file you want to name this:" << std::endl;
        std::string file_name;
        getline(std::cin, file_name);
                std::cout << "We will perform vortex analysis from directory " << read_directory
                                  << " and save the results to " << write_directory << "/" << fi
                std::map<double, av_vortex_number> vortex_map = make_vortex_map(file_vec);
                write_vortex_number(vortex_map, write_directory, file_name);
};
```

```
#include "simulation.h"
#include <iostream>
#include "lattice_io.h"
#include "simulation_utility.h"
```

```
#include <sstream>
#include <sys/stat.h>

template <typename T> void value_save_config_string(std::ifstream& file, T& val) {
        std::string line;
        if(getline(file, line)){
                std::istringstream is{line};
                getline(is, val, '=');
                getline(is, val);
                return;
        };
};

template <typename T> void value_save_config(std::ifstream& file, T& val) {
        std::string line;
        if(getline(file, line)){
                std::istringstream is{line};
                std::string param;

                getline(is, param, '=');
                getline(is, param);
                std::istringstream iss{param};
                if((iss >> val) && !(iss >> param)){
                        return;
                };
                std::cerr << param << " is invalid input. Please edit file." << std::endl;
                exit(1);
        };
};

template <typename T> void check_sign_config(T& val) {
    if(val <= 0) {
        std::cerr << std::to_string(val) << " has the wrong sign. Please edit file" << std::endl
            exit(1);
    };
    return;
};

template <typename T> void check_non_neg_config(T& val) {
        if(val < 0) {
                std::cerr << std::to_string(val) << " has the wrong sign. Please edit file" << s
                exit(1);
        };
        return;
};

int main() {
        std::cout << "Welcome to the KPZ simulation. Please type in the configuration file name:
        std::string config_file;
        getline(std::cin, config_file);
        std::ifstream file(config_file);
        if(!file) {
                std::cerr << "File does not exist. Error." << std::endl;
                exit(1);
        };
```

```
double Dx;
value_save_config(file, Dx);
double Dy;
value_save_config(file, Dy);
double Lx;
value_save_config(file, Lx);
double Ly;
value_save_config(file, Ly);
double cL;
value_save_config(file ,cL);
double dt;
value_save_config(file, dt);
check_sign_config(dt);
int lat_size;
value_save_config(file, lat_size);
check_sign_config(lat_size);
int num_iter;
value_save_config(file, num_iter);
        check_non_neg_config(num_iter);
int num_sim;
value_save_config(file, num_sim);
check_sign_config(num_sim);
        int num_prev;
        value_save_config(file, num_prev);
        check_non_neg_config(num_prev);
        int num_save;
value_save_config(file, num_save);
check_sign_config(num_save);

std::string cond_input;
std::string init_cond = {"ordered"};


value_save_config_string(file, cond_input);
if(cond_input == "y") {
        init_cond = "disordered";
};

std::cout << "Please type the directory in which to save the simulation:" << std::endl;
std::string directory;
getline(std::cin, directory);
check_dir(directory);

parameters param(Dx, Dy, Lx, Ly, cL);
sim_parameters sim_param(dt, num_iter, num_prev, num_sim, num_save, init_cond);

std::cout << "This is a simulation with parameters: Dx = " << Dx << ", Dy = " << Dy
        << ", Lx = " << Lx << ", Ly = " << Ly << ", cL = " << cL << ", lattice size = "
                        << lat_size << std::endl;
std::cout << "The simulation parameters are: timestep is " << dt << ", number of iterati
        << num_iter << ", the number of simulations is " << num_sim
                        << ", the previous number of simulations is " << num_prev
        << ", and the number of iterations per save is " << num_save << std::endl;
std::cout << "The initial condition is " << init_cond << " and the directory the files w
<< directory << std::endl;
```

```
        multiple_simulate(lat_size, param, sim_param, directory);
};
```

```
#include "correlation.h"
#include <math.h>

double av_correlation::get_average() const {
        if(averaging_num == 0) {
                return 0;
        } else {
                return total/averaging_num;
        };
};

av_correlation& av_correlation::add(double val) {
        averaging_num++;
        total += val;
        return *this;
};
```

```
#ifndef UPDATER_H
#define UPDATER_H
#include "system.h"
#include <random>
#include "pcg_random.hpp"

/*
 * Physical variables object
 */
class parameters {
        private:
                const static int rn_div = pow(2,20);
                const double Dx;
                const double Dy;
                const double Lx;
                const double Ly;
                const double Cl;
                std::random_device rd;
                pcg32 rng;
        std::uniform_int_distribution<int> urd;

        public:
                parameters(double dx, double dy, double lx, double ly, double cl) : Dx(dx), Dy(d
                , rng(rd()), urd(std::uniform_int_distribution<int>(0, rn_div)) {}
                double random_real();
                double get_cL() const;
                double get_Dx() const;
                double get_Dy() const;
                double get_Lx() const;
                double get_Ly() const;
};

angle_lattice update_lattice(const angle_lattice&, const parameters&, double);
angle_lattice update_even(const angle_lattice&, const parameters&, double);
```

```
angle_lattice update_odd(const angle_lattice&, const parameters&, double);

double sin_points(const angle_lattice&, int, int, int, int); //sin of difference of two points
double cos_points(const angle_lattice&, int, int, int, int); //cos of difference of two points

#endif
```

```
#include "system.h"
#include "updater.h"
#include <math.h>
#include <random>
#include "pcg_random.hpp"

double sin_points(const angle_lattice&, int, int, int, int);
double cos_points(const angle_lattice&, int, int, int, int);

angle_lattice update_lattice(const angle_lattice& lat, const parameters& par, double dt) {
        if((lat.size() % 2) == 0) {
                angle_lattice new_lat = update_even(lat, par, dt);
                return new_lat;
        } else {
                angle_lattice new_lat = update_odd(lat, par, dt);
                return new_lat;
        };
};

double parameters::random_real() {
        return ((double) urd(rng)/rn_div) - 0.5;
};

double parameters::get_cL() const {
        return Cl;
};

double parameters::get_Dx() const {
        return Dx;
};

double parameters::get_Dy() const {
        return Dy;
};

double parameters::get_Lx() const {
        return Lx;
};

double parameters::get_Ly() const {
        return Ly;
};

double sin_points(const angle_lattice& lat, int i1, int j1, int i2, int j2) {
        return sin(lat.point(i1, j1) - lat.point(i2, j2));
};

double cos_points(const angle_lattice& lat, int i1, int j1, int i2, int j2) {
```

```
        return cos(lat.point(i1, j1) - lat.point(i2, j2));
};

angle_lattice update_even(const angle_lattice& lat, const parameters& par, double dt) {

        //std::random_device rd;
        //std::mt19937 rng(rd());

        pcg_extras::seed_seq_from<std::random_device> seed_source;
        pcg32_fast rng(seed_source);

        int rn_div = pow(2,20);
        std::uniform_int_distribution<int> urd(0, rn_div);
        int N = lat.size();
        angle_lattice new_lat(N);
        double sqrtdt = sqrt(dt);

        for(int j = 0; j < N; j++) {
                int i = 0;
                if ( (j % 2) != 0) {
                        i = 1;
                };

                for(i; i < N; i+=2) {

                        double ang_mid = lat.point(i, j);
                        double ang_up = lat.point(i, j + 1);
                        double ang_down = lat.point(i, j - 1);
                        double ang_left = lat.point(i - 1, j);
                        double ang_right = lat.point(i + 1, j);

                        double sin_left = sin(ang_mid - ang_left);
                        double sin_right = sin(ang_mid - ang_right);
                        double sin_up = sin(ang_mid - ang_up);
                        double sin_down = sin(ang_mid - ang_down);

                        double cos_left = cos(ang_mid - ang_left);
                        double cos_right = cos(ang_mid - ang_right);
                        double cos_up = cos(ang_mid - ang_up);
                        double cos_down = cos(ang_mid - ang_down);

                        double mid_val =
                                dt*(-par.get_Dx()*(sin_right + sin_left)
                                -par.get_Dy()*(sin_up + sin_down)
                                -par.get_Lx()/2*(cos_left + cos_right - 2)
                                -par.get_Ly()/2*(cos_up + cos_down - 2))
                        +sqrtdt*2*M_PI*par.get_cL()*(( (double) urd(rng))/rn_div - 0.5);
                        //if(i == 5 && j == 5) {
                        //      std::cout << "update val "<< mid_val << std::endl;
                        //      std::cout << "           " << lat.point(5,4) << std::endl;
                        //      std::cout <<  lat.point(4,5) <<  " " << lat.point(5,5) << " " <<
                        //      std::cout << "           " << lat.point(5,6) << std::endl;
                        //};
                        new_lat.set(i, j, ang_mid + mid_val);
```

```
                        double left_val = ang_left + new_lat.point(i - 1, j) +
                +dt*(-par.get_Dx()*(-sin_left) -par.get_Lx()/2*cos_left + par.get_Lx()+ par.get_
                                +sqrtdt*2*M_PI*par.get_cL()*( ((double) urd(rng))/rn_div - 0.5);
                        new_lat.set(i - 1, j, left_val);

                        double right_val = new_lat.point(i + 1, j) +
                                dt*(-par.get_Dx()*(-sin_right) - par.get_Lx()/2*cos_right);
                        new_lat.set(i + 1, j, right_val);

                        double up_val = new_lat.point(i, j + 1) +
                                dt*(-par.get_Dy()*(-sin_up) -par.get_Ly()/2*cos_up);
                        new_lat.set(i, j + 1, up_val);

                        double down_val = new_lat.point(i, j - 1) +
                                dt*(-par.get_Dy()*(-sin_down) - par.get_Ly()/2*cos_down);
                        new_lat.set(i, j - 1, down_val);
                };
        };
        return new_lat;
};

angle_lattice update_odd(const angle_lattice& lat, const parameters& par, double dt) {

        std::random_device rd;
        std::mt19937 rng(rd());
        int rn_div = pow(2,20);
        //pcg_extras::seed_seq_from<std::random_device> seed_source;
        //pcg32_fast rng(seed_source);
        std::uniform_int_distribution<int> urd(0, rn_div);
        double sqrtdt = sqrt(dt);
        int N = lat.size();
        angle_lattice new_lat(N);

        for(int i = 0; i < N; ++i) {
                for(int j = 0; j < N; ++j) {

                        double ang_mid = lat.point(i, j);
                        double ang_up = lat.point(i, j + 1);
                        double ang_down = lat.point(i, j - 1);
                        double ang_left = lat.point(i - 1, j);
                        double ang_right = lat.point(i + 1, j);

                        double sin_left = sin(ang_mid - ang_left);
                        double sin_right = sin(ang_mid - ang_right);
                        double sin_up = sin(ang_mid - ang_up);
                        double sin_down = sin(ang_mid - ang_down);

                        double cos_left = cos(ang_mid - ang_left);
                        double cos_right = cos(ang_mid - ang_right);
                        double cos_up = cos(ang_mid - ang_up);
                        double cos_down = cos(ang_mid - ang_down);

                        double mid_val =
                                dt*(-par.get_Dx()*(sin_right + sin_left)
                                -par.get_Dy()*(sin_up + sin_down)
```

```
                                -par.get_Lx()/2*(cos_left + cos_right - 2)
                                -par.get_Ly()/2*(cos_up + cos_down - 2))
                        +sqrtdt*2*M_PI*par.get_cL()*(((double) urd(rng))/rn_div- 0.5);
                                new_lat.set(i, j, ang_mid + mid_val);
                        };
                };
                return new_lat;
};
```

```
#include "initial_condition.h"
#include <math.h>
#include <random>
#include <iostream>

angle_lattice disordered(int N) {
        std::random_device rd;
    std::mt19937 gen(rd());
        std::uniform_real_distribution<> dis(0,2*M_PI);

        angle_lattice lat(N);
        for(int i = 0; i < lat.size(); i++) {
                for(int j = 0; j < lat.size(); j++) {
                        double val = dis(gen);
                        lat.set(i, j, val);
                };
        };
        return lat;
};
```

```
#ifndef INITIAL_CONDITION_H
#define INITIAL_CONDITION_H

#include "system.h"
angle_lattice disordered(int);
#endif
```

```
#ifndef LATTICE_IO_H
#define LATTICE_IO_H
#include <fstream>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include "system.h"
#include <experimental/filesystem>
#include <sys/stat.h>
#include <map>
#include "vortex_calculation.h"
#include "magnetisation.h"
#include <initializer_list>
#include "correlation.h"

/*
 * Functions related to I/O
 */
```

```
namespace fs = std::experimental::filesystem::v1;

unsigned int num_words_in_string(std::string);

std::vector<std::string> list_files(std::string); // returns list of files in a directory

double timestep(std::string path); // returns simulation time from file name
double simulation_number(std::string path); // return simulation number from file name

double deltat(std::string path);
int iteration(std::string path);

void file_error_message(std::string path);
bool check_dir_exists(std::string);

/*
 * Writes lat to path
 */
template <typename T> void lattice_write(lattice<T> const& lat, std::string  path) {
        std::ofstream file(path);
        for(int j = 0; j < lat.size(); j++) {
                for(int k = 0; k < lat.size(); k++) {
                        file << lat.point(j, k) << " ";
                };
                file << std::endl;
        };
};

/*
 * Reads path to lat
 */
template <typename T> void lattice_read(lattice<T>& lat, std::string const& path) {
        std::ifstream file(path);
        if(!file) {
                std::cerr << "Error opening file" << std::endl;
                exit(1);
        };
        std::string line;
        int line_num = 0;
        getline(file, line);
        if(num_words_in_string(line) != lat.size()) {
                std::cerr << "Cannot read data onto lattice as incorrect size. " << std::endl;
                exit(1);
        };
        write_line(lat, line, line_num);

        while(getline(file, line)) {
                line_num++;
                write_line(lat, line, line_num);
        };
};

angle_lattice angle_lattice_from_path(std::string const& path);
```

```
angle_lattice empty_angle_lattice_from_path(std::string const& path); // empty lattice of size i

vortex_lattice empty_vortex_lattice_from_path(std::string const& path); // empty lattice of size

std::map<double, av_vortex_number> make_vortex_map(std::vector<std::string> const &);

std::map<int, av_correlation> make_correlation_map(std::vector<std::string> const&, double);

std::vector<std::map<double, av_magnetisation>> make_magnetisation_map(std::vector<std::string>

std::map<double, av_magnetisation> mag_averaged(std::string const& file_1, std::string const& fi

std::map<double, av_vortex_number> vortex_averaged(std::string const& file_1, std::string const&

/*
 * Writes line to line_num of lat
 */
template <typename T> void write_line(lattice<T> & lat, std::string const& line, int line_num) {
        int pos_in_line = 0;
        std::istringstream iss(line);
        T val;
        while(iss >> val) {
                lat.set(line_num, pos_in_line, val);
                pos_in_line++;
        };
};


/*
 * Write binder map to file
 */
void write_binder_cumulant(std::map<double, double> const& binder_map, std::string const&, std::

/*
 * Write magnetisation squared map to file
 */
void write_magnetisation(std::map<double, av_magnetisation> const& mag_map, std::string const& d

/*
 * Write vortex map to file
 */
void write_vortex_number(std::map<double, av_vortex_number> const&, std::string const&, std::str

void write_correlation(std::map<int, av_correlation> const&, std::string const&, std::string con

#endif
```

```
#include "lattice_io.h"
#include <vector>
#include <sstream>
#include <stdlib.h>
#include <utility>
#include <iomanip>

bool check_dir_exists(std::string path){
```

```
        struct stat sb;
        if(stat(path.c_str(), &sb) == 0 && S_ISDIR(sb.st_mode)){
                return true;
        } else {
                return false;
        };
};

void file_error_message(std::string file) {
        std::cout << "Error opening file " + file << std::endl;
};

unsigned int num_words_in_string(std::string path) {
        std::stringstream stream(path);
        std::string word;
        int counter = 0;
        while(stream >> word) {
                counter++;
        };
        return counter;
};

std::vector<std::string> list_files(std::string path) {
        std::vector<std::string> files;
        for( auto &p : fs::directory_iterator(path)) {
                std::string file(p.path());
                files.push_back(file);
        };
        return files;
};

double timestep(std::string path) {
    double dt = deltat(path);
        int iter = iteration(path);
        return dt*iter;
};

double deltat(std::string path) {
    std::stringstream stream(path);
    std::string parameter;
    getline(stream, parameter, ',');
    getline(stream, parameter, ',');
    double dt = atof(parameter.c_str());
        return dt;
};

int iteration(std::string path) {
        std::string parameter;
        std::stringstream stream(path);
        getline(stream, parameter, ',');
    getline(stream, parameter, ',');
        getline(stream, parameter, ',');
    getline(stream, parameter, ',');
        int iter = atof(parameter.c_str());
        return iter;
```

```
};

double simulation_number(std::string path) {
        std::stringstream stream(path);
        std::string parameter;
        getline(stream, parameter, ',');
        getline(stream, parameter, ',');
        getline(stream, parameter, ',');
        getline(stream, parameter, ',');
        getline(stream, parameter, ',');
        getline(stream, parameter, ',');
        int sim_num = atof(parameter.c_str());
        return sim_num;
};

angle_lattice angle_lattice_from_path(std::string const& path) {
        std::ifstream file(path);
        if(!file) {
                        file_error_message(path);
            exit(1);
        };
        std::string line;
        int line_num = 0;
        getline(file, line);
        angle_lattice lat(num_words_in_string(line));
        write_line(lat, line, line_num);
        while(getline(file, line)) {
                line_num++;
                write_line(lat, line, line_num);
        };
        return lat;
};

angle_lattice empty_angle_lattice_from_path(std::string const& path) {
        std::ifstream file(path);
        if(!file) {
                        file_error_message(path);
            exit(1);
        };
        std::string line;
        getline(file, line);
        angle_lattice lat(num_words_in_string(line));
        return lat;
};

vortex_lattice empty_vortex_lattice_from_path(std::string const& path) {
        std::ifstream file(path);
        if(!file) {
                file_error_message(path);
                exit(1);
        };
        std::string line;
        getline(file, line);
        vortex_lattice lat(num_words_in_string(line));
        return lat;
```

```
};


std::map<double, av_vortex_number> make_vortex_map(std::vector<std::string> const& files) {
        angle_lattice lat = empty_angle_lattice_from_path(files[0]);
        std::map<double, av_vortex_number> vor_map;
        for(std::string const& st : files) {
                double time = timestep(st);
                lattice_read(lat, st);
                vortex_number vor_num = make_vortex_number(lat);
                vor_map[time].add(vor_num);
        };
        return vor_map;
};

std::map<int, av_correlation> make_correlation_map(std::vector<std::string> const& files, double
        angle_lattice lat = empty_angle_lattice_from_path(files[0]);
        std::map<int, av_correlation> av_cor_map;
        bool not_in = true;
        for(std::string const& st : files) {
                if(time != timestep(st)) {
                        continue;
                        not_in = false;
                };
                lattice_read(lat, st);
                double angle_0 = lat.point(0,0);
                for(int i = 1; i < lat.size()/2; i++) {
                        av_cor_map[i].add(cos(angle_0 - lat.point(i,0)));
                };
        };
        if(not_in == true) {
                std::cerr << "Timestep not part of results!" << std::endl;
                exit(1);
        };
        return av_cor_map;
};


std::vector<std::map<double, av_magnetisation>> make_magnetisation_map(std::vector<std::string>
    angle_lattice lat = empty_angle_lattice_from_path(files[0]);
        std::vector<std::map<double, av_magnetisation>> mag_map_list(pow_args.size());
    for(std::string const& st : files) {
        double time = timestep(st);
        lattice_read(lat, st);
        magnetisation mag(lat);
                for(size_t i = 0; i < pow_args.size(); i++) {
                        if(mag_map_list[i].count(time) == 0) {
                av_magnetisation av_mag(pow_args.begin()[i]);
                mag_map_list[i].insert(std::make_pair(time, av_mag));
                mag_map_list[i].at(time).add(mag); // [] does not work as no default contructor
                        } else {
                mag_map_list[i].at(time).add(mag);
                        };
                };
        };
```

```
    return mag_map_list;
};

void write_binder_cumulant(std::map<double, double> const& binder_map, std::string const& dir, s
        std::string path = dir + "/" + name;
        std::ofstream file(path);
        for(std::map<double, double>::const_iterator iter = binder_map.cbegin(); iter != binder_
                double time = (*iter).first;
                double mag = (*iter).second;
                file << time << " " << mag << std::endl;
        };
};

void write_magnetisation(std::map<double, av_magnetisation> const& mag_map, std::string const& d
        std::string path = dir + "/" + name;
        std::ofstream file(path);
        for(std::map<double, av_magnetisation>::const_iterator iter = mag_map.cbegin(); iter !=
                double time = (*iter).first;
                double mag = (*iter).second.get_average();
                file << time << " " << mag << std::endl;
        };
};

void write_correlation(std::map<int, av_correlation> const& cor_map, std::string const& dir, std
        std::string path = dir + "/" + name;
        std::ofstream file(path);
        for(std::map<int, av_correlation>::const_iterator iter = cor_map.cbegin(); iter != cor_m
                int distance = (*iter).first;
                double cor = (*iter).second.get_average();
                file << distance << " " << cor << std::endl;
        };
};


std::map<double, av_magnetisation> mag_averaged(std::string const& path_1, std::string const& pa
        std::ifstream file_1(path_1);
        std::ifstream file_2(path_2);
        if(!file_1) {
                file_error_message(path_1);
        exit(1);
    };
        if(!file_2) {
                file_error_message(path_2);
        exit(1);
    };
        std::string line_1;
        std::string line_2;

        std::map<double, av_magnetisation> mag_map;
        while(getline(file_1, line_1) && getline(file_2, line_2)) {
                std::stringstream line_1_s(line_1);
                std::stringstream line_2_s(line_2);
                double time_1, time_2;
                line_1_s >> time_1;
                line_2_s >> time_2;
```

```
                if(time_1 != time_2) {
                        std::cerr << "Error. Times in the files are different" << std::endl;
                        exit(1);
                };
                double av_1, av_2;
                line_1_s >> av_1;
                line_2_s >> av_2;
                av_magnetisation av_mag(power, runs_1 + runs_2, av_1*runs_1 + av_2*runs_2);
                std::cout << av_mag.get_averaging_num() << std::endl;
                mag_map.insert(std::make_pair(time_1, av_mag));
        };
        return mag_map;
};

std::map<double, av_vortex_number> vortex_averaged(std::string const& path_1, std::string const&
        std::ifstream file_1(path_1);
        std::ifstream file_2(path_2);
        if(!file_1) {
                file_error_message(path_1);
        exit(1);
    };
        if(!file_2) {
                file_error_message(path_2);
        exit(1);
    };
        std::string line_1;
        std::string line_2;

        std::map<double, av_vortex_number> vor_map;
        while(getline(file_1, line_1) && getline(file_2, line_2)) {
                std::stringstream line_1_s(line_1);
                std::stringstream line_2_s(line_2);
                double time_1, time_2;
                line_1_s >> time_1;
                line_2_s >> time_2;
                if(time_1 != time_2) {
                        std::cerr << "Error. Times in the files are different" << std::endl;
                        exit(1);
                };
                std::cout << time_1 << " " << time_2 << std::endl;
                double nv_1, nv_2, nav_1, nav_2;
                line_1_s >> nv_1;
                line_2_s >> nv_2;
                line_1_s >> nav_1;
                line_2_s >> nav_2;
                av_vortex_number av_vor(runs_1 + runs_2, nv_1*runs_1 + nv_2*runs_2, nav_1*runs_1
                vor_map.insert(std::make_pair(time_1, av_vor));
        };
        return vor_map;
};


void write_vortex_number(std::map<double, av_vortex_number> const& vortex_map, std::string const
        std::string path = dir + "/" + name;
        std::ofstream file(path);
```

```
        for(std::map<double, av_vortex_number>::const_iterator iter = vortex_map.cbegin(); iter
                double time = (*iter).first;
                file << time << " " << std::fixed << std::setprecision(8)  << (*iter).second.get
        };
};
```