Prifysgol Abertawe
Swansea University

University of Swansea

Masters Dissertation

# BL1: 2D Potts Model with a Twist

*Author:*
Robert James

*Supervisor:*
Professor Biagio Lucini

May 23, 2015

## Abstract

The goals of this project were to perform a Monte Carlo simulation of the Potts Model, to numerically generate the Density of States of a periodic and twisted lattice and investigate the behaviour of the interfaces that occur when a twist is introduced to the system, for a variety of Q states. These goals have been accomplished using a variety of techniques, including using both the Metropolis update algorithm and the Wang-Landau restricted sampling technique in a custom lattice solver written in C++.

# Contents

# Chapter 1

# Introduction and Theory

This project studies the behaviour of interfaces that occur when a twist is added to a discrete state lattice. The Potts Model as originally proposed was to reinterpret the Ising Model as a system of interacting spins. Generalising such a system of interacting spins in 2 dimensions leads to a system of Q equally distributed angles.

$$\theta_n = \frac{2\pi n}{q}, n = 1, 2, ...q \tag{1.1}$$

Such as system has a Z(q) symmetry and the Hamiltonian can be written as.

$$\mathcal{H} = -\sum_{\langle ij \rangle} J(\theta_{ij}) \tag{1.2}$$

$J(\theta)$ is periodic in $2\pi$ and $\theta_{ij}$ is the difference in angles of the two neighbour states. In this project we discard the planar Potts Model and focus on the standard Potts model that can be written in the form.

$$J(\theta_{ij}) = J_c \delta_{Kr}(n_i, n_j) \tag{1.3}$$

Where $n_i$ and $n_j$ are unit vectors for equal of the Q directions. Taking the coupling constant $J_c$ to be equal to 1 we can write the Hamiltonian of the Potts Model used in this project as

$$\mathcal{H} = -\sum_{\langle ij \rangle} \delta_{Kr}(n_i, n_j) \tag{1.4}$$

It is useful to look at the energy per unit volume. The Minimum energy value per unit volume will be

$$E_{Min} = -\text{Number of Links per Lattice Site} \tag{1.5}$$

You would expect the value for the Maximum energy per unit volume to be 0 however this not quite the case due to the number of possible configurations that lead to the same system energy. In other papers the issue of overlapping energy configurations is nullified by a scaling term. This leads to a slightly modified Hamiltonian.

$$\mathcal{H} = \sum_{\langle ij \rangle} \frac{1}{q} - \delta_{Kr}(n_i, n_j) \tag{1.6}$$

However in this project the issue of incorrect scaling is dealt with by ignoring non-physical values so the Hamiltonian remains the same as before.

From the paper by F. Y. Wu the critical point on a square lattice in the anti-ferromagnetic case which the project focuses on can be written as [8]

$$\beta_c = \frac{1}{k_B T} = \ln\left(1 + \sqrt{Q}\right) \tag{1.7}$$

## 1.1 Lattice

It is impossible to perform computer simulations for infinite lattice sizes. By using finite lattice sizes you introduce boundary effects that can, in particularly small lattices produce noise of significance. For small lattices, it is often useful to introduce periodic boundary conditions so that this effect is diminished. On a smaller lattice, there is a higher ratio of lattice sites close to the boundaries, this leads to a higher number of sites being affected by the non standard interactions that occur at these points. In 2D the lattice is a regular euclidean grid where the sites in each direction are labelled between 0 and $L - 1$. In this project the following periodic boundary conditions are set on the lattice.

$$\begin{aligned} s_{L,0} &= s_{0,0} \\ s_{0,L} &= s_{0,0} \end{aligned} \tag{1.8}$$

In 3 dimensions by wrapping the boundaries of this 2 dimensional lattice you generate a torus.

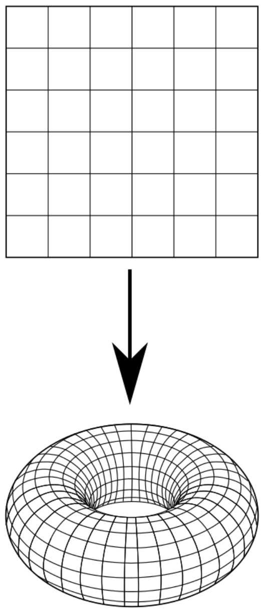


Figure 1.1: 2 dimensional lattice transforming to torus in 3 dimensions under periodic boundary conditions

In this project I add a twist to the lattice in one axis then take separate measurements for this twisted lattice. To add a twist to the lattice in a single axis you need to relax the periodic condition in the axis of choice and define a boundary point. At this point every lattice site on the right has it's value set to be the lattice site on the left plus an arbitrary constant this new value is then put through the modulo operator to ensure that new Q states aren't generated during the process. The new state can be described as

$$s_{x,bp} = (s_{x,bp-1} + 1)\%Q \tag{1.9}$$

Where x is any integer value between 0 and L-1, bp is the boundary point defined earlier.

## 1.2 Observables

In this project, all of the results are given in terms of $\beta$ because it is the variable that is manipulated directly in the code.

$$\beta = \frac{1}{k_B T} \tag{1.10}$$

A simplification that can be made with ease involves setting the Boltzmann constant $k_B$ to be 1.

### 1.2.1 Magnetisation

Spontaneous Magnetisation is also known as the Magnetisation in the absence of an external Magnetic field. At high $\beta$, $\beta > \beta_c$, any of the lattice spin states can influence the states of its neighbours. In this lattice simulation the system has a net magnetisation even without a magnetic field. At the critical point, $\beta_c$, the Magnetisation of the system can no longer remain zero. We can define the Magnetisation as the sum of each individual spin states.

$$M = \sum_i^V \sigma_i \tag{1.11}$$

However when calculating the magnetisation from the $\theta_i$ form the Magnetisation becomes

$$M = \sum_i^V e^{i\theta_i} \tag{1.12}$$

### 1.2.2 Specific Heat

In a general case, the Specific Heat C can be defined as

$$C_V = \left.\frac{\partial E}{\partial T}\right|_V \tag{1.13}$$

It is a measurable physical quantity based upon the ratio between the heat added to an object and the relating temperature change.

The form I use for the program

$$C_V = \beta^2 \left[\left\langle E^2 \right\rangle - \langle E \rangle^2\right] \tag{1.14}$$

### 1.2.3 Magnetic Susceptibility

The Magnetic Susceptibility is a dimensionless value that shows how much an extensive parameter changes when an intensive parameter increases. In this simulation the magnetic susceptibility tells us how much the magnetisation changes by increasing the temperature. In this project I take measurements using this form.

$$\chi = \beta \left[\left\langle M^2 \right\rangle - \langle M \rangle^2\right] \tag{1.15}$$

## 1.3 Partition Function

You can describe the Partition function as

$$Z = \sum_{\{i\}} e^{-\beta E_i} \equiv \sum_{i} g(E_i) e^{-\beta E_i} \tag{1.16}$$

Where $\{i\}$ is a generic configuration and $g(E_i)$ is the Density of States. Traditional approaches to calculate the Partition Functions through simulation have exponential fluctuations in volume, as such I used the algorithm proposed by F. Wang and D. P. Landau [7].

1. Starting with any lattice configuration
2. Choose a random site and change to a random value
3. Accept the change with probability

   $$P = min\left\{\frac{g(E_{old})}{g(E_{new})}, 1\right\} \tag{1.17}$$

   If rejected revert the change.
4. Update g(E$_{new}$)$Set$E$_{old} = E_{new}$ and repeat from 2 until g(E) has converged.

Performing this algorithm until the values converge allows for the reconstruction of the Density of States, which can then be used to calculate the Partition Function. In this project I calculate the Partition Function for both a regular Periodic Lattice and a Twisted Lattice for the various grid sizes chosen to study.

## 1.4 Interfaces

The Free Energy of a thermodynamic system is defined to be

$$F = -k_B T \ln Z \tag{1.18}$$

To find the Free Energy of the Interface you need to take the logarithm of the ratio the partition functions calculated for the Periodic Lattice and the Twisted Lattice. This leads to the equation for the Interface Free Energy

$$F_s = -\log \frac{Z^*}{Z} - \log L \tag{1.19}$$

Z is the partition function of the periodic lattice, $Z^*$ is the partition function of the twisted lattice. Because the Interface can form at L points in the axis with the twist condition this needs to be subtracted to provide an accurate representation [3]. From the Interface Free Energy you can calculate the interface tension

$$\sigma = \lim_{L \to \infty} \frac{F_s}{V_{D-1}} \tag{1.20}$$

# Chapter 2

# Code

## 2.1 Design Choices

In this project, I started designing the code such that it would compile into a single executable. To ensure that I didn't need to recompile this code every time I needed to change parameters, I opted for a simple configuration file that would determine the functions and the flow of operations. To this end, I ended up using the C++11 standard[1] and a library designed to parse configuration files libconfig [4].

libconfig has two different interfaces, a C API (Application Programming Interface) and a C++ API. I picked the API written in C++ because it matched the rest of the project. An example configuration file is given below.

5.
```
#Program Mode
wanglandau = true;
coldstart = true;
dim_grid = 16;
dim_q = 2;
interface = true;
#Metropolis Algorithm Parameters
beta = 1.00;
randomspin = false;
n_samples = 1000;
#Wang Landau Algorithm Parameters
target_e = -2.00;
target_width = 16.0;
n_entropic_samples = 1000;
n_asamples = 100;
a0 = 2.0;
```

In the above configuration file you can see 5 configuration options that determine the type of simulation that is going to be performed and 3/5 other options that are for these specific operations.

The first parameter, wanglandau, determines the type of simulation that is going to be run. If this value is false, the program runs a typical Metropolis Algorithm and reads the beta, randomspin and n_samples parameters further down the file. If the parameter is true, it runs the Wang Landau algorithm. This algorithm requires a few more parameters, like the target energy (target_e), the target width, the number of $a_n$ samples and a starting value for the iterative calculation of $a_n$.

The other parameters are required by both of the algorithms, these determine the size of the lattice (dim_grid) and the number of q states (n_q). The parameter, coldstart, determines the initial state of the system be it ordered or disordered.

## 2.2 Code Operation

In this section, I will describe the operation of the program. As described above, there are two different modes in this project so I will explain each type separately. However in both modes, the first stage is the same. We read the file name of the configuration file from the input arguments and parse it into the variables. Then to calculate the angles for each different q state because they only need to be calculated once at the beginning. Now that all the variables are loaded in from the configuration file, it is closed and the lattice/grid variable is constructed as a 2D array of heap memory. The decision to allocate the memory as a 2D array was to improve the readability of the code. In most applications, it is preferable to construct a 1D array and access elements using Row-Major order to improve the access speed of the elements. In this case, the 2D array was preferable as it significantly improved the overall readability and intuitive understanding of the code. By default the C++ memory allocation operator, new[], doesn't assign any value to the assigned memory which means to initialise the system the program needs to run through every lattice point and assign it a value. In the case of a cold start, the program simply generates a random integer between 1 and q and assigned each lattice point to that value. In the case of a hot start, a new random number is generated for each point. From this point on ward the two operating modes diverge in their assigned functions so I will move on to the Metropolis algorithm

### 2.2.1 Metropolis

A description of the Metropolis algorithm is now required to ensure that the next stages of the program are understood. A Metropolis-Hastings algorithm is a Markov Chain Monte Carlo method to obtain random samples from a probability distribution. In general this technique is usually used to solve more complex multidimensional problems despite this it is highly adaptable and suitable for the problem at hand. The Metropolis-Hastings approach to solving this problem involves modification of a single lattice point at each step. There are two methods for determining which lattice point to use. The first method is the simplest, using a PRNG (Pseudo Random Number Generator) generate the coordinates of the lattice site which is to be modified. The second method involves progressing through each point in the lattice in order. In this project, I elected to use the second method because at the time it seemed to require less computation and as such be faster. However due to the nature of the 2D array used as the lattice it requires the use of two nested for loops which could slow down the process. Once the lattice site has been chosen from whatever method chosen, the program makes a random change to the state of that point and calls it a *Proposal State*. The energy change between the original state and the proposed state is calculated. Again there are two methods to do this, one a complete sweep across the lattice applying the Hamiltonian and using the sum and the other which looks only at the *links* between lattice points that have changed because everything else will cancel out. Again both methods have a function in the final source code but the second method is used as this reduces the total number of operations required. The energy difference, delta, between the two states is then put through a simple algorithm, shown below, to determine if the change is accepted or not.

**Algorithm 1** Accepting or Rejecting a Proposed Change for the Metropolis Algorithm

```
if $delta < 0.0$ then
    ACCEPT
else
    $0 <= random() < 1$
    if $e^{-\beta \text{ delta}} > rand$ then
        ACCEPT
    else
        REJECT
    end if
end if
```

$\beta$ is a variable that is provided in the configuration file. Now that the Metropolis-

Hastings method has been explained in the context of this particular problem the remaining behaviour of the Metropolis portion of the code can be explained. Before any measurements can be taken from this system, it needs to be thermalised so that it reaches equilibrium. The average time taken for the system to reach thermal equilibrium is known as the thermalisation time $\tau_e q$ and it varies upon lattice size. After the system thermalises, the program progresses onto the measurement component of the program. In this mode of operation, there are 3 measured quantities, the Acceptance, the Magnetisation and the Energy. The acceptance is the ratio of accepted changes against the total number of proposed states, the Magnetisation and Energy as described earlier in the report are shown below.

$$\begin{aligned} \mathcal{M} &= \sum_{i}^{V} \sigma_i \\ E &= \sum_{\{i,j\}} \delta_{Kr}(\sigma_{i,j}) \end{aligned} \tag{2.1}$$

Now using the n_samples parameter the program starts a FOR loop which performs an update and a measurement algorithm at each loop. By storing the recorded measurement at each step the program stores a 1D array of measurement values as it goes through the loop. When the loop is finished the program has finished the measurement stage and now starts to perform analysis on the measurements. The next stage will be described in a subsequent section.

### 2.2.2 Wang Landau

This mode of operation is significantly different to the Metropolis algorithm. After all of the parameters have been loaded and the lattice states set either randomly (hotstart) or all aligned (coldstart) the program needs to drive the system state into the desired energy band. The parameters, target_e and target_width which define the lattice are provided in the configuration file. In general, the target width is taken to be equal to the grid size in an attempt to match the conditions in the Guagnelli paper studying this particular problem for Q=10[2] but it can be set to smaller values to try and restrict the sampling region more. I found the simplest way to drive the system towards the target energy band was for each point on the lattice suggest a random state, if the difference in energy between this new state and the target is smaller than that of the original state accept it. The program does this repeatedly until it reaches it's target energy band this is one of the more time consuming parts of the code because it doesn't try to reduce the length of any interfaces, which would allow a significantly higher level of control and it can get stuck, especially in low $\beta$ conditions. By the time the loop stops, the program will have reached it's target energy band. It is unlikely but it remains possible that the energy of the system is exactly equal to the configuration file parameter due to the discrete nature of the energies that arises from the discrete number of possible states. The next stage of the program is to take the value $a_0$ provided by the configuration file and perform an update on the system. This time however the algorithm is slightly modified from the original.

**Algorithm 2** Accepting or Rejecting a Proposed Change for the Wang Landau Algorithm

```
if outsideenergyband() then
  REJECT
else
  CONTINUE
  if $\delta < 0.0$ then
    ACCEPT
  else
    $0 <= random() <= 1$
    if $e^{-a_n \delta} > random()$ then
      ACCEPT
    else
      REJECT
    end if
  end if
end if
```

After running this update algorithm across the lattice and taking a measurement of $E^\star$. $E^\star$ is the current energy less the target energy $E_0$. The program moves onto the iterative equation for $a_n$ which is given below for $Q = 2$.

$$a_{n+1} = a_n + \frac{12}{L^2}\langle E^\star \rangle \tag{2.2}$$

For simulations that involve higher numbers of states taking the form for $a_n$ to be

$$a_{n+1} = a_n + \frac{12}{4L + L^2}\langle E^\star \rangle \tag{2.3}$$

After updating the value for $a_n$ the program returns to the beginning of a FOR loop, that runs a configurable number of times dependent on the value of n_asamples. The aforementioned FOR loop includes updating the lattice a number of times based upon the configuration parameter, n_entropic_samples taking measurements of the lattice and recalculating $E^\star$ to be used in the calculation of $a_n$ using the method described above.

### 2.2.3 Adding a Twist

Adding a twist to the lattice requires a slight modification to the update algorithm used in the Wang Landau method. In my code, I add a twist to the lattice close to the midpoint in the x direction. It is trivial to calculate what point it occurs using the floor function provided by the C++ math library. The modifications to the code that was used in the Wang Landau method are simple and are switched on and off using the interface boolean parameter in the configuration file. This ensures a twist exists in the lattice and allows us to calculate $Z^*$ that is required to study the interface tension. At the interface point, whose value is determined using $floor(L/2)$, the update algorithm is replaced by setting the value of the lattice on the other side of the interface plus a constant and then put the modulus function to ensure the new state wouldn't be a state that wasn't expected in the program.

## 2.3 Run

To get meaningful data from the code, it needs to be run multiple times with many different parameters. This means that my code is SPMD, Single Program Multiple Data which is a Subcategory of MIMD, Multiple Instruction Multiple Data. Despite design choices in my code causing it to run slower than expected, it was still sensible to explore a method of pregenerating the configurations and scheduling them to run as efficently as possible. To this end, as part of the process I wrote several shell scripts that were designed to improve this stage of the process. When writing these shell scripts I ran into a very useful utility GNU Parallel[6]. This utility allows a simple single threaded program to run with several different configurations on an arbitary number of cores. An example of one of these shell scripts is below.

```
rm -R -- */
rm output.dat
rm Q*.dat
filename='target.list'
filelines=`cat $filename`
gridsize=18
interface=false
sed -i '/interface\s=\s.*/c\interface = '$interface'' param
    .cfg
sed -i '/dim_grid\s=\s.*/c\dim_grid = '$gridsize'' param.
    cfg
```

```
sed -i '/target_width\s=\s.*/c\target_width = '$gridsize
   '.0' param.cfg

for q in 2 3 4 8 10
do
rm -R -- */
for i in $filelines
do
        mkdir energy$i
    cd energy$i
    cp ../param.cfg .
    sed -i '/target_e\s=\s.*/c\target_e = '$i'' param.cfg
    sed -i '/dim_q\s=\s.*/c\dim_q = '$q'' param.cfg
    cd ..
done

parallel -j16 --eta "cd energy{}; ../potts.app param.cfg;
   cd ../" :::: target.list

filelines=`cat $filename`
for i in $filelines
do
        anaverage=`tail -n 30 energy$i/an.dat | awk '{ sum
           +=$1} END {print sum/30}'`
    energy=$i
    echo "$energy $anaverage" >> Q$q.dat
done
done
```

This script, takes a template configuration file and fills out all of the parameters and creates a directory for the code to run in, runs the code on the number of cores determined by the j parameter of the parallel exeuctable and then collects the results averages the $a_n$ and returns the results for each energy.

## 2.4 Output

The program was several modes of operation. In the Metropolis mode, the program creates several files that contain the results of the simulation for the measured parameters, Energy Per Unit Volume, Magnetisation Per Unit Volume, Specific Heat and the Magnetic Susceptibility. The output from a single file in this mode contains 3 columns. The value of $\beta$, the actual value of the measurement and the error in the measurement. For the Wang Landau mode, the code returns a file with all of n measurements of $a_n$. Because the programs themselves are run in parallel with others the single outputs are appended to one another at the end so that, in the Metropolis case you get a list of the results at each $\beta$ which is in an easily plottable format. In the case of the Wang Landau mode, the nature of the output is such that it isn't simply a case of appending each files contents together. As shown in the example above the final results of the output file are averaged and the averages are appended together to give a single output file. For error estimation and calculation, I chose to use a Jack Knife error calculation routine. For each array of measured values the program calculates the average of the results then bins the data. The program calculates the averages of each bin and calculates the difference between the bin average and the total average for each bin and combines the results to calculate the non biased error of the samples.

# Chapter 3

# Results

From the earlier chapter I described the final form of the data output from the program. Below is a sample from the Metropolis data file.

```
  0          0             0.000277831
0.05   0.00000183835   0.000301732
 0.1   0.00000759666   0.000289046
 ...        ...             ...
 2.5     0.000177274     0.000889706
2.55     0.000121696     0.000737807
 2.6     0.000113462     0.000829744
```

The Wang Landau output file looks like this

```
 -1.972222222     117.227
 -1.916666667     114.667
 -1.861111111     112.106
      ...            ...
 -0.1388888889   -28.7273
-0.08333333333   -26.1667
-0.02777777778   -28.7273
```

Using this data we can start constructing the physical quantities and plotting them.

## 3.1 Metropolis

### 3.1.1 Range of beta

For the Metropolis data, the first step is to plot the data to ensure the program produced expected behaviour. In this case expected behaviour for the Energy as a function of $\beta$ is that at low $\beta$, high T, the energy would be nearing it's maximum and that for high $\beta$, low T the energy would be close to its minimum and at there would need to be a transition between these two points.

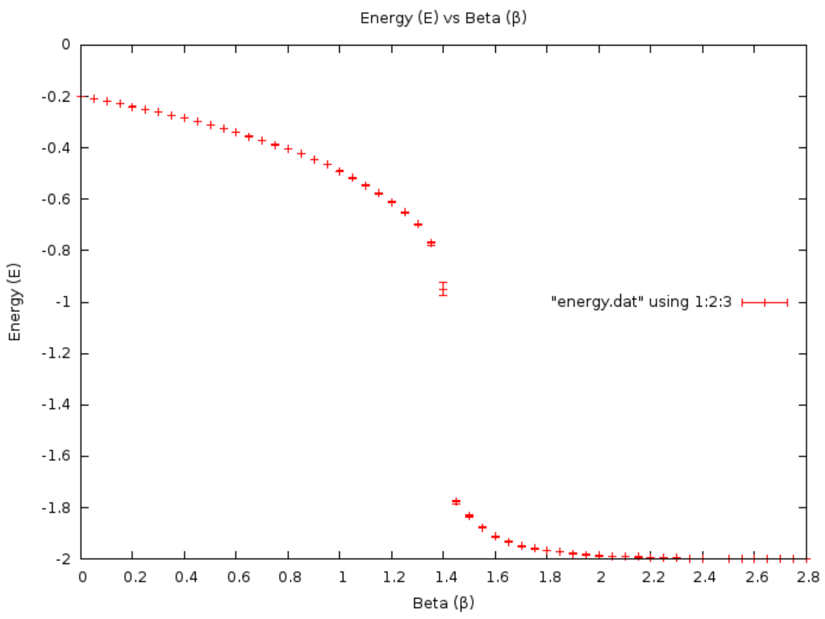


Figure 3.1: Graph showing Energy vs $\beta$

In the graph above, the expected behaviours are clearly visible. The data shows that the energy approaches it's maximum at low $\beta$ and it's minimum at high $\beta$. The above graph is set for $Q = 10$ which has a phase transition at $\beta_c = \ln(1 + \sqrt{Q}) \approx 1.426$ in the thermodynamic limit, while the graph clearly shows a transition it isn't at this $\beta_c$ but this is due to the finite size of the lattice used in this simulation.

For Magnetisation the expected behaviour is that for high $\beta$, low T, that magnetisation per unit volume would be equal to 1. At low $\beta$, high T, this property disappears and should approach a minimum. Again the expected behaviour is seen in the graph below. The phase transition occurs at around the same position as in the Energy case which is another quick check to make.

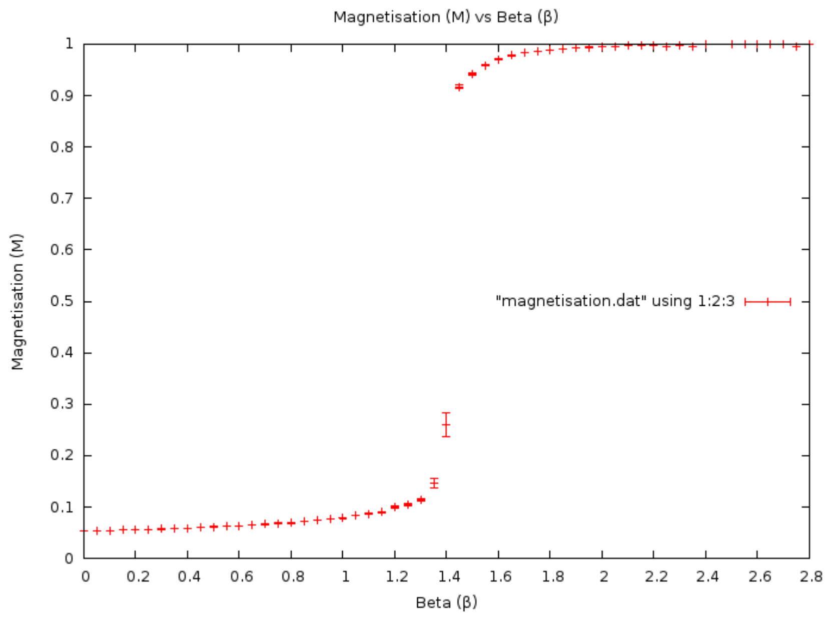
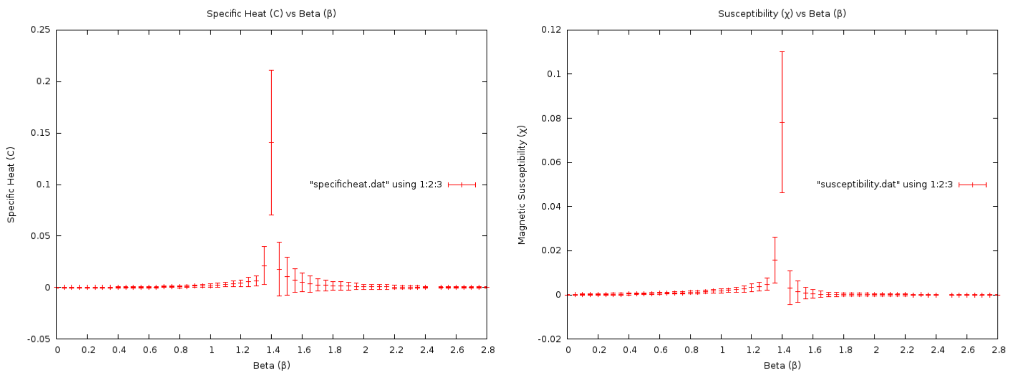


Figure 3.2: Graph showing Magnetisation vs $\beta$

For the other data files that are measured, which include the Specific Heat, C and the Magnetic Susceptibility,($\chi$) the behaviour should be 0 everywhere except at the phase transition as stated above, $\beta_c \approx 1.426$ in the thermodynamic limit.



(a) Q10 Graph showing Specific Heat ($C$) vs $\beta$ (b) Q10 Graph showing Magnetic Susceptibility ($\chi$) vs $\beta$

The peaks in the two graphs above are obvious and fit with the expected results for the respective quantities.

While these quick checks are useful while debugging the code to ensure that the simulated behaviours are correct, on their own they don't provide enough information to verify the code is working correctly this will be dealt with in the next step.

## 3.2 Wang Landau

### 3.2.1 Plotting the Output

The first step of dealing with the Wang Landau output is, like the Metropolis, is to plot it. This is to visually verify that the behaviour is as expected, in this case the iterative process for calculating $a_n$ leads to data that starts at an initial value $a_0$ and that will converge towards and oscillate around in a band of acceptable values.

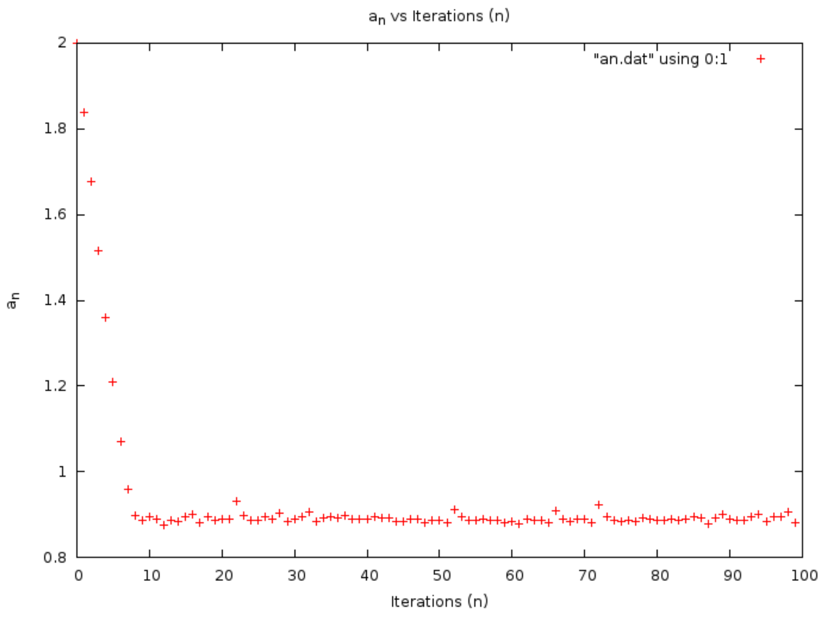


Figure 3.3: Graph showing the $a_n$ convergence vs n

It is clear that in the above graph the behaviour is as expected. The value of $a_n$ converges extremely quickly towards it's target value and oscillates slightly around its value.

### 3.2.2 Are the results compatible with the Metropolis?

In an earlier section, I stated that the information provided by the metropolis wasn't enough information to verify the code was working correctly. The Metropolis part of the program was a stepping stone to the implementation of the Wang

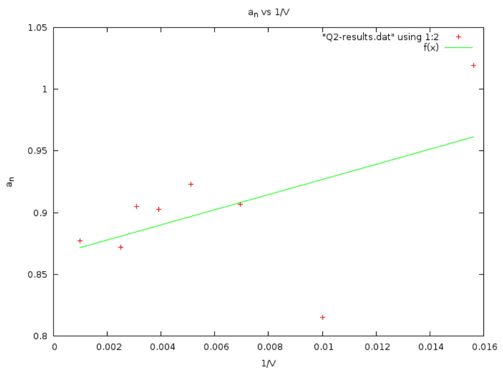
Landau algorithm that was the primary portion of this project. That being said, you can verify the results returned by the Wang Landau mode of this project by using the energy values returned by the Metropolis code at the $\beta_c$ at the thermodynamic limit. $a_n$ should be roughly equivalent to the $\beta$ required to keep the system at the target energy using this information and the value of $\beta_c$ I calculated the value of $a_n$ at the critical point using the energy value returned from the Metropolis data. This leads to a data file looking similar to the one below.

```
 8   -1.74296   0.00591688
10   -1.72614   0.00648999
12   -1.71512    0.0058484
14   -1.69895    0.0067316
...     ...         ...
28    -1.7049   0.00379986
30    -1.7069   0.00395975
32   -1.72059   0.00357536
```

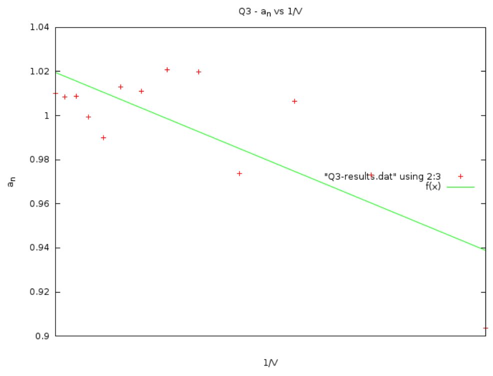
The first column is the lattice size, the $2^{nd}$ column is the Energy at $\beta_c$ and the third column is the error in the $2^{nd}$ column. Using this data I wrote several shell scripts that are designed to run the program in the Wang Landau mode using the target energy provided in the $2^{nd}$ column. Taking the results from the Wang Landau mode of the program using the shell scripts described above to collect the values of $a_n$ for the various grid sizes you can plot them against the inverse volume. At the thermodynamic limit, the inverse volume should be 0. If you fit the results with a linear function $y = ax + b$, it's intercept represents the value at the thermodynamic limit. The intercepts for this particular test are shown in the table below.

| Q | Theoretical | Result | Error |
|---|---|---|---|
| 2 | 0.881373587 | 0.865616 | 0.03173 |
| 3 | 1.005052539 | 0.996294 | 0.01123 |
| 4 | 1.098612289 | 1.11538 | 0.002026 |
| 8 | 1.342454046 | 1.34759 | 0.008933 |
| 10 | 1.426062439 | 1.43679 | 0.0152 |

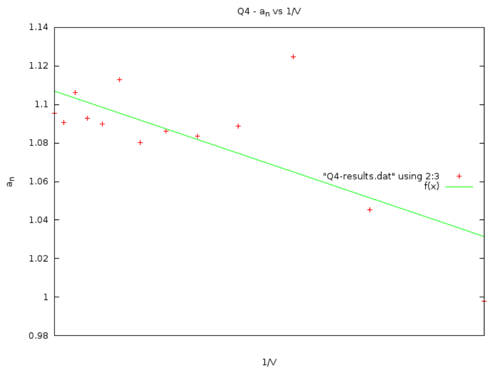
A plot of the data as it appeared is now provided below.



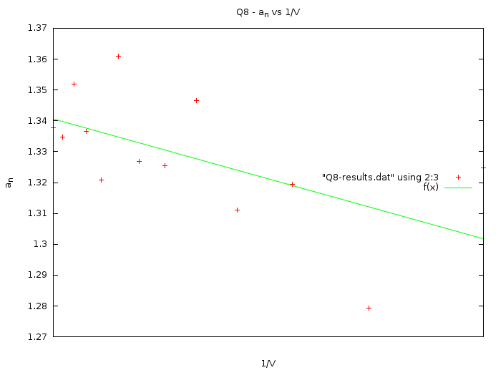
(a) Q2 $a_n = 0.881373 \pm 0.03173$



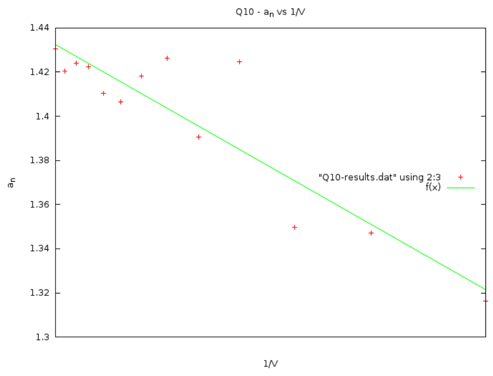
(b) Q3 $a_n = 0.996294 \pm 0.01123$



(c) Q4 $a_n = 1.11538 \pm 0.02026$



(d) Q8 $a_n = 1.34759 \pm 0.008933$



(e) Q10 $a_n = 1.43679 \pm 0.0152$

The above graphs show that in the thermodynamic limit, the calculated values of $a_n$ agree with the $\beta_c$ which is a form of verification to the results from the Wang Landau.

### 3.2.3 Calculating the Partition Function

The next stage of the project, which is required to calculate the Partition Function, $Z$, is to generate values of $a_n$ at regular intervals between the minimum and maximum energies. The regular intervals are determined by the energy band width. That means that the midpoints of these intervals for a small grid size for $Q = 2$ would be as shown below.

| Minimum | Maximum | Midpoint |
|---|---|---|
| -2 | -1.9375 | -1.96875 |
| -1.9375 | -1.875 | -1.90625 |
| -1.875 | -1.8125 | -1.84375 |
| ... | ... | ... |
| -0.1875 | -0.125 | -0.15625 |
| -0.125 | -0.0625 | -0.09375 |
| -0.0625 | 0.000 | -0.03125 |

Running the Wang Landau mode of the program with the midpoint as the target energy and the target width as the grid size.

The output from the program for each midpoint when plotted against the energy leads to the plot below.

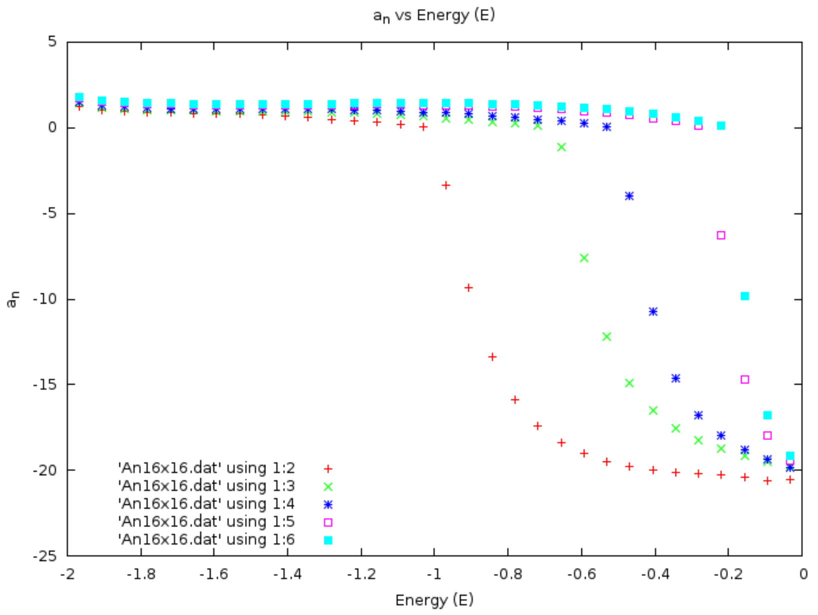


Figure 3.4: Graph showing the plot of $a_n$ across the energy bands for various Q

This graph has issues with scaling due to the fact that the maximum energy is dependant on Q. The maximum energy is dependant on Q because the number of configurations that can represent each energy increases with Q. A rescaled graph shows a clearer picture for the different Q.

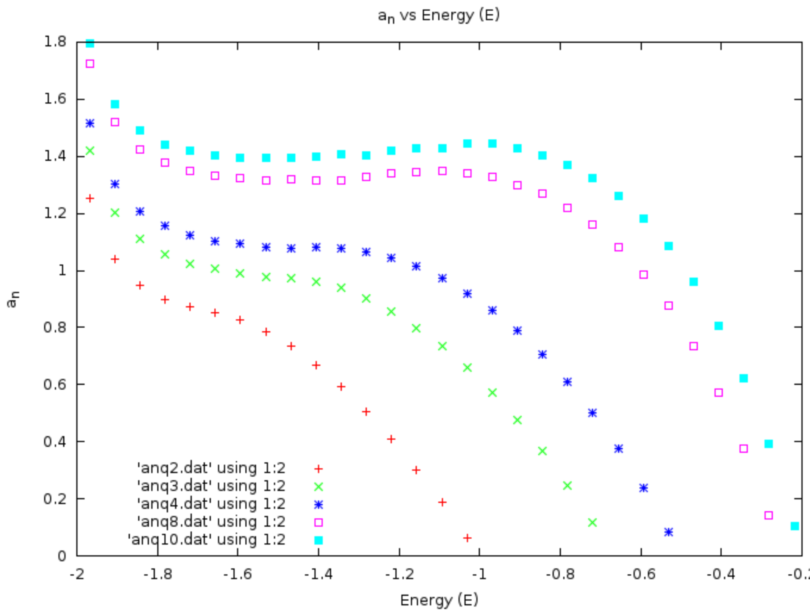
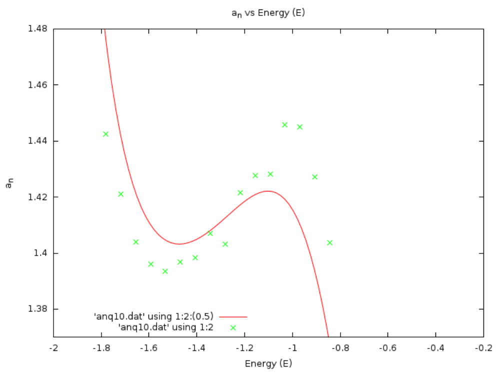
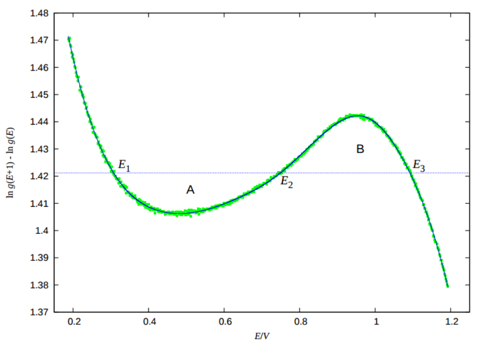


Figure 3.5: Graph showing the rescaled plot of $a_n$ across the energy bands for various Q

A comparing my results with a similar result from the paper by Guagnelli for Q=10 is shown below.





(a) Q10 Graph showing the rescaled plot of $a_n$ across the energy bands from the program

(b) Q10 Graph showing the rescaled plot of $a_n$ across the energy bands from the Guagnelli paper[5]

The primary discrepancies between the two results is likely to occur due to the difference in lattice sizes used to generate the results and some arbitrary scaling factors used in the simulation. To calculate the partition function, $Z$, from this data I turned to the symbolic computational mathematics tool kit Mathematica to calculate the constants that keep the Piecewise function of $\log \rho(E)$ continuous. The table 5.3.2 contains these constants for each of the 5 Q tested, the Null values occur due to the aforementioned scaling issues.

By plotting a Piecewise function in side Mathematica where each component is of the form.

$$f(E) = C_i + a_i(E - E_i) \tag{3.1}$$

$E_i$ is the Midpoint of the interval, $C_i$ is the constant value for that interval to ensure continuity and $a_i$ is the measured value of $a_n$ at the midpoint. A sample of the piecewise function that Mathematica generates for Q2 is provided in the table at 5.3.3. Plotting the these Piecewise functions is a good way to ascertain the continuity of the newly generated functions. However due to the way Mathematica renders these functions there appears to be gaps in the rendered output. Included below is the plot for the Q2 piecewise function of $\log g\,(E)$.

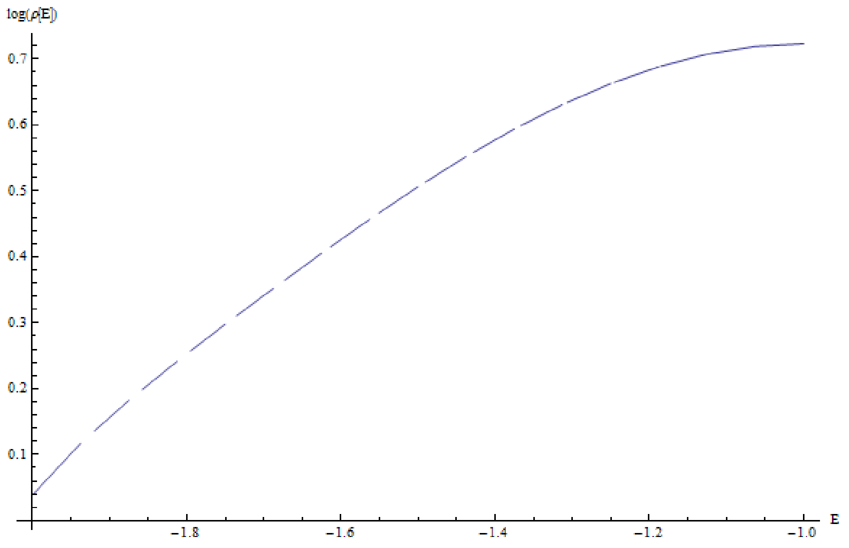


Figure 3.6: Graph showing the $\log g\,(E)$ vs E

The plots for Q3, Q4, Q8, Q10 can be found in the appendix at 5.3.4. All the plots show similar behaviour for all the various values of Q. Now the only piece of information that remains to provide the complete Density of States is the constant, $c_0$ that is added to all of the piecewise functions. As explained earlier this value is calculated using the following equation

$$C_0 = \frac{2L^2}{\int g(E)dE} \tag{3.2}$$

In this case, $g(E)$ is the incomplete Density of States. The integral can be computed easily using the tools provided by Mathematica however it could also be computed using a simple Simpson's rule integrator due to the linear nature of the approximations used to calculate the Density of States. After adding this constant to the incomplete Density of States, you get the complete Density of States for that simulation. The next stage of the processing for the Mathematica program is to calculate the Partition Function. The partition function can be written in the form.

$$\mathcal{Z} = \int g(E)e^{-\beta E}dE \tag{3.3}$$

Using Mathematica to perform the integration numerically across the sampled energy range you can extract the Partition Function as a function of $\beta$. With this done the analysis of this part of the project is complete.

## 3.3 Wang Landau with a Twist

After modifying the configuration file to add the twist into the simulation and running the simulation across various lattice sizes. After extracting the data from the programs output files and importing it into Mathematica the code is nearly indistinguishable from that described earlier for the Wang Landau data so to ensure that I am not repeating any material I will describe the next stages of analysis here. The plots of the incomplete Density of States are available in the appendix at 5.3.5.

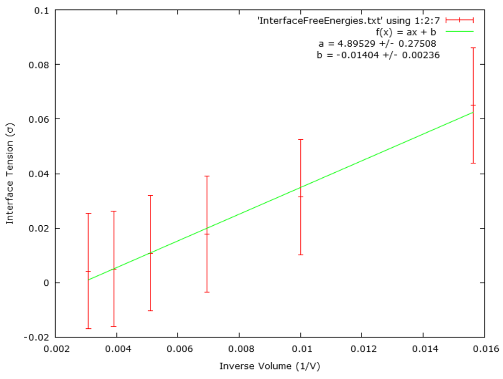
Having now calculated the Partition Functions for the untwisted $Z$ and twisted $Z^*$ lattice, the next stage is to calculate the interface free energy. The Interface Free Energy as described earlier can be written as

$$F_I\left(L\right) = -\log\frac{Z^*}{Z} - \log L \tag{3.4}$$

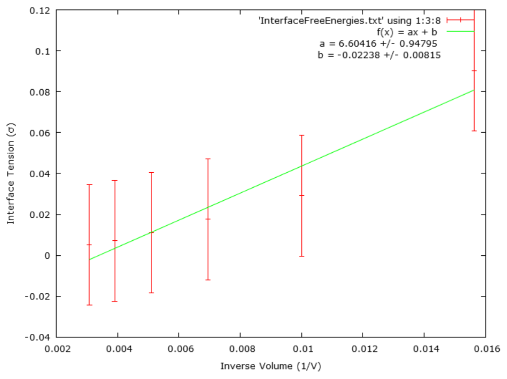
The Mathematica notebook provided in the appendix 5.3.6 is a copy of the working code used to perform all of the analysis for the Wang Landau model. Taking the results from the Mathematica notebook for the Interface Free Energy at $\beta_c$ and using the form for the interface tension [3].

$$\sigma = \lim_{L\to\infty}\frac{F_I}{V_{D-1}} \tag{3.5}$$

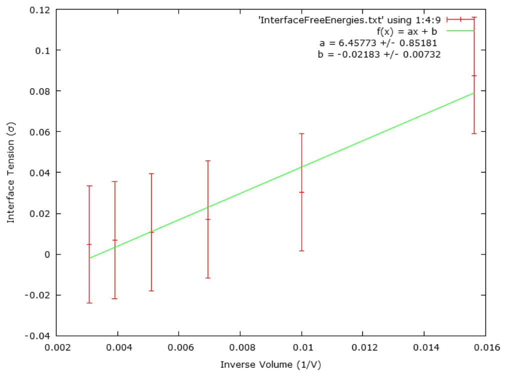
Taking the results from the Mathematica notebook for the various grid sizes and taking a fit of the plot to the thermodynamic limit leads to the results below for the interface tension $\sigma$ at $\beta_c$.



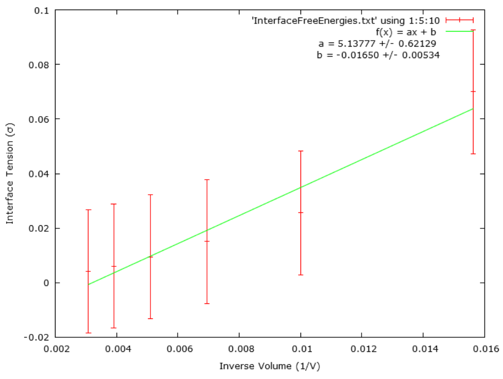
(a) Q2 $\sigma = -0.01404 \pm 0.00236$



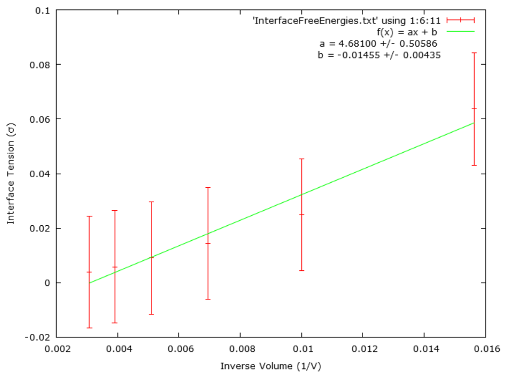
(b) Q3 $\sigma = -0.02238 \pm 0.00815$



(c) Q4 $\sigma = -0.02183 \pm 0.00732$

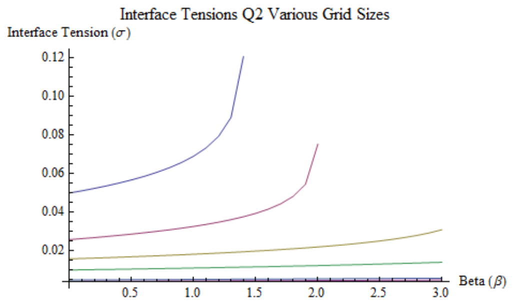


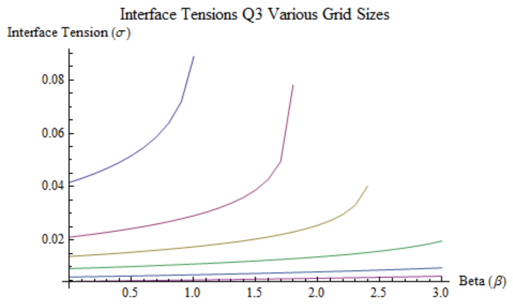
(d) Q8 $\sigma = -0.01650 \pm 0.00534$

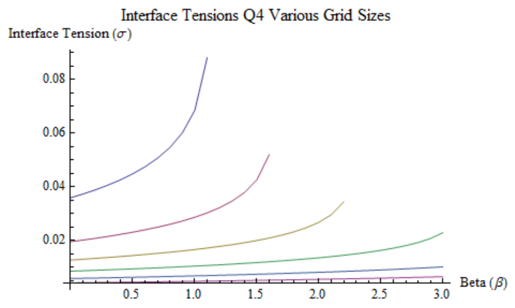


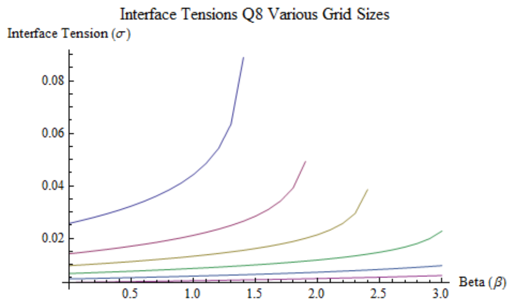
(e) Q10 $\sigma == 0.01455 \pm 0.00435$

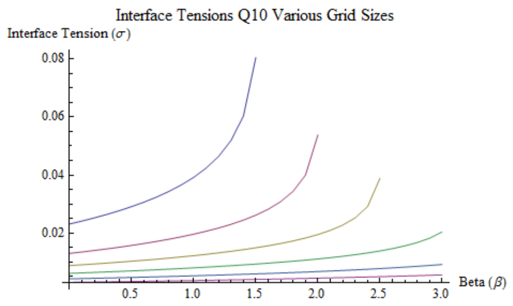
This information isn't useful for determining the behaviour of the Interface tension reformatting the output from the Mathematica notebook to extract the interface tension at a range of $\beta$ and for all of the lattice sizes provides further insight into the behaviour of the interface tension.











In the graphs above you can clearly see that the interface tension on the smaller grids rapidly increases with $\beta$. This is primarily due to the fact that on a smaller lattice, the affect of the twist is significantly larger than on a bigger lattice. On the larger lattices which almost appear to be linear in this scale you would expect to have the same or similar behaviour over a larger $\beta$ range. For a large $\beta$, low T, the twist artificially creates an interface that leads to an increase in the energy. The overall effect of the interface to the systems energy is directly proportional to the lattice size which would explain why for the larger lattice sizes the tension doesn't grow rapidly in the range of $\beta$ sampled.

# Chapter 4

# Conclusion and Reflection

To conclude this project I will discuss what work has been done, how it related to the original goals of the project and where further work and progress could have been made. The first goal, to perform a Monte Carlo simulation of the Potts Model. Using the a Metropolis update algorithm and later a slightly modified version for the Wang Landau sampling method, I developed a computer simulation program that was met the first goal. While I achieved this I feel that the large amount of time spent at this phase reduced the effectiveness of the latter stages and that focus here wasn't the best use of my time. The second goal which was to access the traditionally harder to calculate Partition Functions which required the reconstruction of the Density of States. I believe this goal has been achieved but I feel that sampling a larger range of Lattice Sizes would be essential to reduce the errors and provide a more complete picture of the results. Further to this point I feel that it would be essential to sample as many lattice sizes as possible to ensure that the behaviour shown still occurs as the size increases. The final goal of observing the behaviour of the interface tensions has been partially completed. While I did manage to collect some information about the Interface Tensions at $\beta_c$ and for a range of $\beta$ values the small number of different lattice sizes could have severely restricted any more complex behaviours that may have arisen. I would like to have computed comparable results to that of the prevailing literature referenced throughout my paper and I feel that the end results of my project for this goal are missing a final stage of analysis. With regards to the programming component of this project, I feel that some of the design choices I made restricted the performance of the code too much and as such contributed to the reduced lattice sizes that were sampled. Modification to the lattice memory allocation from 2 dimensional array to a 1 dimensional is the most glaring potential optimisation.

I would say that the use of the Wang Landau algorithm provides a method of calculating the Partition Functions of arbitrary lattice configurations. The implementation of the algorithm required several minor modifications to a standard

Metropolis update algorithm and that should be the first step of all future implementations of similar work. The inclusion of a Metropolis update algorithm allows for a self verification process to occur between the two operation modes and this should be seen as a positive. I would have liked to have tied my simulation results down to real world experimental data as part of the verification process but due to simplifications that I had put in my code i.e.$/k_B = J = 1$ I found finding comparable results extremely difficult and I would suggest that while the model behaviour appears correct some experimental verification can go a long way. I found the Energy scaling issues to be rather time consuming to debug because the source wasn't initially obvious. I would therefore suggest another test in the case of bizarre energy measurements from similar problems be plotting the maximum energy returned against the number of states. It helped narrow down the potential sources of the issue. While I found the benefits of using C++ over C to be numerous, there are several potential improvements that would feature if I were involved in any future work. Aside from the modifications to the array allocation the Object orientation helped add a layer of abstraction but added several layers of potential performance hits. This being said, further investigation could focus on the implementation of parallelism into the programs operation. While the Wang Landau algorithm isn't embarrassingly parallel the PRNG could be placed in an alternate thread to increase the speed of updates.

This project has developed my programming skills, pushed my understanding of computational and statistical physics to their limits and has given me an understanding of the strengths and weaknesses of several sampling techniques. While I feel the information gleaned about the behaviour of the Interfaces is limited due to a number of factors I believe that given the opportunity the limitations could be resolved.

# Bibliography

[1] Working draft, standard for programming language c++, 2011.

[2] M. Guagnelli. Sampling the density of states, 2012.

[3] A. Hietanen and B. Lucini. Interface tension of the 3d 4-state potts model using the wang-landau algorithm, 2011.

[4] Mark A. Lindner. libconfig - a library for processing structured configuration files, 2012.

[5] Elliott W. Montroll, Renfrey B. Potts, and John C. Ward. Correlations and spontaneous magnetization of the two dimensional ising model. *Journal of Mathematical Physics*, 4(2):308–322, 1963.

[6] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[7] Fugao Wang and D. P. Landau. Efficient, multiple-range random walk algorithm to calculate the density of states. *Phys. Rev. Lett.*, 86:2050–2053, Mar 2001.

[8] F. Y. Wu. The potts model. *Rev. Mod. Phys.*, 54:235–268, Jan 1982.

# Chapter 5

# Appendices

## 5.1 Program Source Code

### 5.1.1 main.cpp

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
# define M_PIl
   3.141592653589793238462643383279502884L /* pi */
#include <cstdlib>
#include <string>
#include <random>
#include <libconfig.h++>

#include "potts.h"
#include "utilityfunctions.h"

int main(int argc, char **argv) {
        std::string filename;
        if(argc != 2){
                std::cout << "No Configuration File
                   provided" << std::endl;
                exit(1);
        } else {
                filename = argv[1];
        }
        POTTS_MODEL potts;
        read_input(filename,&potts);

        // Pre Generate the angles for the cos(theta) so
           you can just refer
        // back to them
        potts.angles = new double[potts.n_q + 1];
        potts.angles[0] = 0; // So ID's match to angles
           this array is 1 bigger
        // than it needs

        for(unsigned int i = 1; i <= potts.n_q; i++){
                potts.angles[i] = (2.0 * M_PIl * (i-1)) / (
                   double)potts.n_q;
                //std::cout<< potts.angles[i] << std::endl;
        }

        // Main trigger to switch between the different
           algorithms. I should
        // probably stop typing with such a flourish
        if(potts.wanglandau == false){
                // Run the Metropolis Algorithm
                potts.metropolis();
        } else {
                // Run the Wang Landau Algorithm Method
                potts.wang_landau();
        }

        delete [] potts.angles;


        return(0);
}
```

### 5.1.2 utilityfunctions.cpp

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <cstdlib>
#include <stdio.h>
```

```
#include <string>

#include <libconfig.h++>

#include "potts.h"
#include "utilityfunctions.h"

using namespace libconfig;

int read_input(std::string file, POTTS_MODEL *potts){
        Config cfg;
        try{
                        cfg.readFile(file.c_str());
                } catch(const FileIOException &fioex){
                        std::cerr << "I/O error while 
                           reading file." << std::endl;
                        return(1);
                } catch(const ParseException &pex){
                        std::cerr << "Parse error at " <<
                           pex.getFile() << ":" << pex.
                           getLine()
                                << " - " << pex.getError()
                                   << std::endl;
                        return(1);
                }

        // Type of Simualation to Run. Either WangLandau (
           entropic sampling) or Metropolis
        try{
                potts->wanglandau = cfg.lookup("wanglandau"
                   );
        }
        catch(const SettingNotFoundException &nfex){
                std::cerr << "No 'wanglandau' parameter 
                   found" << std::endl;
                std::cerr << "Unrecoverable Error. Add a 
                   wanglandau to the " << file << std::endl
                   ;
                return(1);
        }
```

```
try{
        potts->interface = cfg.lookup("interface");
}
catch(const SettingNotFoundException &nfex){
        std::cerr << "No 'interface' paramter found
           " << std::endl;
        std::cerr << "Unrecoverable Error. Add a
           interface to the " << file << std::endl;
        return(1);
}


// Collect type of simulation invarient parameters
   here
try{
        potts->size = cfg.lookup("dim_grid");
}
catch(const SettingNotFoundException &nfex){
        std::cerr << "No 'dim_grid' parameter found
           " << std::endl;
        std::cerr << "Unrecoverable Error. Add a
           dim_grid to the " << file << std::endl;
        return(1);
}

try{
        potts->n_q = cfg.lookup("dim_q");
}
catch(const SettingNotFoundException &nfex){
        std::cerr << "No 'dim_q' parameter found"
           << std::endl;
        std::cerr << "Unrecoverable Error. Add a
           dim_q to the " << file << std::endl;
        return(1);
}

try{
        potts->coldstart = cfg.lookup("coldstart");
}
```

```
catch(const SettingNotFoundException &nfex){
        std::cerr << "No␣'coldstart'␣parameter␣
           found" << std::endl;
        std::cerr << "Unrecoverable␣Error.␣Add␣a␣
           coldstart␣to␣the␣" << file << std::endl;
        return(1);
}


//Now you've got the invarient parameters sorted
   now do the rest of them
if(potts->wanglandau == true){
        // Collect specifics for that
        try{
                potts->a0 = cfg.lookup("a0");
                potts->target_e = cfg.lookup("
                   target_e");
                potts->target_e *= (potts->size *
                   potts->size);
                potts->target_width = cfg.lookup("
                   target_width");
                potts->n_entropic_samples = cfg.
                   lookup("n_entropic_samples");

                potts->n_entropic_samples *= potts
                   ->size * potts->size;
                potts->n_asamples = cfg.lookup("
                   n_asamples");
        }
        catch(const SettingNotFoundException &nfex)
           {
                return(1);
        }
} else {
        // Collect specifics for Metropolis
           algorithm beta etc.
        try{
                potts->beta = cfg.lookup("beta");
                potts->randomspin = cfg.lookup("
                   randomspin");
```

```
                        potts->n_samples = cfg.lookup("
                           n_samples");
                }
                catch(const SettingNotFoundException &nfex)
                   {
                        return(1);
                }
        }

        return(0);
}
```

### 5.1.3 potts.cpp

```
#define _USE_MATH_DEFINES
#include <cstdio>
#include <iostream>
#include <fstream>
#include <chrono>
#include <stdlib.h>
#include <cstdlib>
#include <string>
#include <random>
#include <cmath>
#include <complex>
#include <cstdlib>
#include "potts.h"


// Class Initialisation, goes though and assigns values
POTTS_MODEL::POTTS_MODEL(){
        generator.seed(std::chrono::system_clock::now().
           time_since_epoch().count());
}

POTTS_MODEL::~POTTS_MODEL(){
        if(wanglandau == true){
                // Wang Landau Cleanup
                for(unsigned int i = 0; i < size; i++){
                        delete [] grid[i];
                }
                delete [] grid;
                delete [] estar;
                delete [] aguess;
        } else {
                // Metropolis Cleanup
                for(unsigned int i = 0; i < size; i++){
                        delete [] grid[i];
                }
                delete [] grid;
                // Delete Measurements
                delete [] energy;
                delete [] magnetisation;
        }

}

double POTTS_MODEL::energycalc(){
        double energy = 0.0;

        for(unsigned int j = 0; j < size; j++){
                for(unsigned int i = 0; i < size; i++){
                        if(grid[i][j] == (grid[(i+1)%size][
                           j])){
                                energy++;
                        }
                        // For neighbour below
                        if(grid[i][j] == grid[i][(j+1)%size
                           ]){
                                energy++;
                        }
                }
        }
        //energy *= -1 * ((n_q - 1) / (n_q));
        energy *= -1;
        return(energy);
}

double POTTS_MODEL::energychange(unsigned int i, unsigned
    int j){
        double energy = 0.0;
        if(grid[i][j] == grid[(i+1)%size][j]){
                energy++;
        }

        if(grid[i][j] == grid[i][(j+1)%size]){
                energy++;
        }
        if(grid[i][j] == grid[i][(j-1)%size]){
                energy++;
        }

        if(grid[i][j] == grid[(i-1)%size][j]){
                energy++;
        }

        energy *= -1;
        return(energy);
}

double POTTS_MODEL::magnetisationcalc(){
        double magnetisation = 0.0;
        double real = 0.0;
        double imag = 0.0;
        for(unsigned int j = 0; j < size; j++){
                for(unsigned int i = 0; i < size; i++){
                        real += cos(angles[grid[i][j]]);
                        imag += sin(angles[grid[i][j]]);
                }
        }
        magnetisation = sqrt( (real * real) + (imag * imag)
            );
        return(magnetisation);
}
```

### 5.1.4 metropolis.cpp

```
#define _USE_MATH_DEFINES
#include <cstdio>
#include <iostream>
#include <fstream>
```

```
#include <chrono>
#include <stdlib.h>
#include <cstdlib>
#include <string>
#include <random>
#include <cmath>
#include <complex>
#include <cstdlib>
#include "potts.h"

void POTTS_MODEL::metropolis(){

    grid = new unsigned int *[size]; // 2D array for ease
        of use
    for(unsigned int i = 0; i < size; i++){
        grid[i] = new unsigned int [size];
    }

    // To ensure the number of samples is the average of
        samples
    // that each point on the lattice recieves multiply by
        volume
    n_samples *= (size * size);

    //Setup Arrays for Measurements
    energy = new double[n_samples];
    magnetisation = new double[n_samples];

    // Use a Mersenne Prime Twister Random Number Generator
    std::uniform_int_distribution<int> distribution(1,n_q);

    if(coldstart == true){
        // Set everything to a random q value
        unsigned int rand_q = distribution(generator);
        for(unsigned int j = 0; j < size; j++){
            for(unsigned int i = 0; i < size; i++){
                grid[i][j] = rand_q;
            }
        }
    } else {
        // Set every point randomly :)
        for(unsigned int j = 0; j < size; j++){
            for(unsigned int i = 0; i < size; i++){
                grid[i][j] = distribution(generator);
            }
        }
    }
    //std::cout << "Program Gets To JUST BEFORE
       THERMALISATION" << std::endl;
    // A Metropolis Algorithm needs Thermalising.



    acceptance = 0;
    n_therm *= (size * size);
    if(randomspin == true){
        for(unsigned int i = 0; i < n_therm; i++){
            metropolis_update();
        }
    } else {
        for(unsigned int n = 1; n < n_therm; n++){
            unsigned int y = n % size;
            unsigned int x = (n % (size*size)) / size;
            smooth_metropolis_update(x,y);
        }
    }
    //std::cout << "Program COMPLETES THERMALISATION" <<
       std::endl;
    // Reset the acceptance
    acceptance = 0;
    if(randomspin == true){
        for(unsigned int i = 0; i < n_samples; i++){
            metropolis_update();
            metropolis_measurement(i);
        }
    } else {
        for(unsigned int n = 0; n < n_samples; n++){
            unsigned int y = n % size;
            unsigned int x = (n % (size*size)) / size;
            smooth_metropolis_update(x,y);
        metropolis_measurement(n);
    }
}

// Errors and Thermodynamic Derived Quantities
double *specificheat = new double[n_samples];
double *susceptibility = new double[n_samples];

for(unsigned int i = 0; i < n_samples ; i++){
    specificheat[i] = energy[i] * energy[i];
    susceptibility[i] = magnetisation[i] *
       magnetisation[i];
}

double energy_avg = metropolis_average(energy);
double magnetisation_avg = metropolis_average(
   magnetisation);
double specificheat_avg = metropolis_average(
   specificheat);
double susceptibility_avg = metropolis_average(
   susceptibility);

double energy_err = metropolis_error(energy,energy_avg)
   ;
double magnetisation_err = metropolis_error(
   magnetisation,magnetisation_avg);
double specificheat_err = metropolis_error(specificheat
   ,specificheat_avg);
double susceptibility_err = metropolis_error(
   susceptibility,susceptibility_avg);

delete [] specificheat;
delete [] susceptibility;

specificheat_avg -= (energy_avg * energy_avg);
specificheat_avg *= (beta * beta);

susceptibility_avg -= (magnetisation_avg *
   magnetisation_avg);
susceptibility_avg *= beta;
```

```
    specificheat_err = sqrt((specificheat_err *
       specificheat_err) + (energy_err * energy_err));
    susceptibility_err = sqrt((susceptibility_err *
       susceptibility_err) + (magnetisation_err *
       magnetisation_err));

    // Write the Data to File
    std::ofstream file;
        file.open("specificheat.dat");
        file << beta << "␣" << specificheat_avg << "␣" <<
           specificheat_err << std::endl;
        file.close();

        file.open("susceptibility.dat");
        file << beta << "␣" << susceptibility_avg << "␣" <<
            susceptibility_err << std::endl;
        file.close();

        file.open("energy.dat");
        file << beta << "␣" << energy_avg << "␣" <<
           energy_err << std::endl;
        file.close();

        file.open("magnetisation.dat");
        file << beta << "␣" << magnetisation_avg << "␣" <<
           magnetisation_err << std::endl;
        file.close();

    file.open("acceptance.dat");
    file << beta << "␣" << acceptance << std::endl;
    file.close();

}

void POTTS_MODEL::metropolis_measurement(unsigned int k){
    energy[k] = energycalc(); //Total on Lattice
    magnetisation[k] = abs(magnetisationcalc()); // Total
       on Lattice
    //magnetisation[k] = abs(magnetisation[k]); // Absolute
```

```
        Value

    unsigned int volume = size * size;

    energy[k] /= volume; // Per Lattice Site
    magnetisation[k] /= volume; // Per Lattice Site
}

void POTTS_MODEL::smooth_metropolis_update(unsigned int x,
   unsigned int y){
    std::uniform_int_distribution<unsigned int>
       distribution(1,n_q);
    double energy_pre = energychange(x,y);
    unsigned int old_q = grid[x][y];

    unsigned int new_q = distribution(generator);
    grid[x][y] = new_q;
    double energy_post = energychange(x,y);

    std::uniform_real_distribution<double> pdistribution
       (0,1);
    double delta = energy_post - energy_pre;
    double rand = pdistribution(generator);

    if( delta < 0.0 ){
        grid[x][y] = new_q;
        acceptance++;
    } else {
        if(exp(-1 * beta * delta) > rand){
            grid[x][y] = new_q;
            acceptance++;
        } else {
            grid[x][y] = old_q;
        }
    }
}

void POTTS_MODEL::metropolis_update(){
    std::uniform_int_distribution<unsigned int>
       distribution(1,n_q);
    std::uniform_int_distribution<unsigned int> coordinates
        (0,size-1);

    //std::cout << "Program Gets INSDIE metropolis_update()
        " << std::endl;

    unsigned int x = coordinates(generator);
    unsigned int y = coordinates(generator);
    double energy_pre = energychange(x,y);
    unsigned int old_q = grid[x][y];

    unsigned int new_q = distribution(generator);
    grid[x][y] = new_q;
    double energy_post = energychange(x,y);

    std::uniform_real_distribution<double> pdistribution
        (0,1);
    double delta = energy_post - energy_pre;
    double rand = pdistribution(generator);

    if( delta < 0.0 ){
        grid[x][y] = new_q;
        acceptance++;
    } else {
        if(exp(-1 * beta * delta) > rand){
            grid[x][y] = new_q;
            acceptance++;
        } else {
            grid[x][y] = old_q;
        }
    }
}



double POTTS_MODEL::metropolis_average(double *array){
    double average = 0.0;
    for(unsigned int i = 0; i < n_samples; i++){
        average += array[i];
    }
    average /= n_samples;
    return(average);
}

double POTTS_MODEL::metropolis_error(double *array, double
   average){
        double *bin, *jackbins;
        unsigned int numbins = 100;
        bin = new double[numbins];
        jackbins = new double[numbins];

        unsigned int slice = n_samples / numbins;
        double sumbins = 0.0;
        for(unsigned int l = 0; l < numbins; l++){
            bin[l] = 0.0;
            for(unsigned int k = 0; k < slice; k++){
                bin[l] += array[(l * slice)+k];
            }
            bin[l] /= slice;
            sumbins += bin[l];
        }

        // Forming Bins
        for(unsigned int l = 0; l < numbins; l++){
            jackbins[l] = (sumbins - bin[l]) / (numbins -
               1.0);
        }

        double error = 0.0;
        for(unsigned int l = 0; l < numbins; l++){
            error += (average - jackbins[l]) * (average -
               jackbins[l]);
        }
        error *= (numbins - 1.0) / (double)numbins;
        error = sqrt(error);
        return(error);
}
```

### 5.1.5 wanglandau.cpp

```
#define _USE_MATH_DEFINES
```

```
#include <cstdio>
#include <iostream>
#include <fstream>
#include <chrono>
#include <stdlib.h>
#include <cstdlib>
#include <string>
#include <random>
#include <cmath>
#include <complex>
#include <cstdlib>
#include "potts.h"


void POTTS_MODEL::wang_landau(){
        grid = new unsigned int *[size]; // 2D array for
           ease of use
        for(unsigned int i = 0; i < size; i++){
                grid[i] = new unsigned int [size];
        }

        // Use a Mersenne Prime Twister Random Number
           Generator
        std::uniform_int_distribution<int> distribution(1,
           n_q);
        k = distribution(generator);
        k = 1;

        interfacepoint = floor(size/2);

        if(coldstart == true){
                // Set everything to a random q value
                unsigned int rand_q = distribution(
                   generator);
                for(unsigned int j = 0; j < size; j++){
                        for(unsigned int i = 0; i < size; i
                           ++){
                                unsigned int y = i % size;
                                unsigned int x = (i % (size
                                   *size))/size;
```

```
                    if(x == interfacepoint &&
                        interface == true){
                                // This is idiotic
                                    of me. State 0
                                    isn't a state it
                                    's empty.
                                // So need to catch
                                    that.
                                if( (grid[x-1][y]+k
                                    )%n_q == 0){
                                        grid[x][y]
                                            = 1;
                                } else {
                                        grid[x][y]
                                            = (grid[
                                            x-1][y]+
                                            k)%n_q;
                                }
                    } else {
                                grid[i][j] = rand_q
                                    ;
                    }
            }
    }
} else {
    // Set every point randomly :)
    for(unsigned int j = 0; j < size; j++){
            for(unsigned int i = 0; i < size; i
                ++){
                    if(i == interfacepoint &&
                        interface == true){
                    // This is idiotic of me.
                        State 0 isn't a state it
                        's empty.
                    // So need to catch that.
                                if( (grid[i-1][j]+k
                                    )%n_q == 0){
                                        grid[i][j]
                                            = 1;
                                } else {
                                        grid[i][j]
                                           = (grid[
                                           i-1][j]+
                                           k)%n_q;
                                }

                        } else {
                                grid[i][j] =
                                   distribution(
                                   generator);
                        }
                }
        }
}

// Drive Energy into Target

while(outsideenergyband()){
        for(unsigned int j = 0; j < size; j++){
                for(unsigned int i = 0; i < size; i
                   ++){
                        drivetotarget(i,j);
                }
        }
}
// Do Measurements

estar = new double[n_entropic_samples];
for(unsigned int i = 0; i < n_entropic_samples; i
   ++){
        unsigned int y = i%size;
        unsigned int x = (i % (size*size))/size;
        if(x == interfacepoint && interface == true
           ){
        // This is idiotic of me. State 0 isn't a
           state it's empty.
                                // So need to catch
                                   that.
                                if( (grid[x-1][y]+k
                                   )%n_q == 0){
                                        grid[x][y]
                                           = 1;
                              } else {
                                        grid[x][y]
                                           = (grid[
                                           x-1][y]+
                                           k)%n_q;
                              }
            } else {
                      smooth_wanglandau_update(x,y);
            }
            wanglandau_measurement(i);
}
double estar_avg = wanglandau_average(estar);

// Set a_0 in the aguess array to stop segfaults
cur_a = a0;

aguess = new double[n_asamples];

aguess[0] = cur_a;

//std::cout << "Gets to n_asamples nested for loop"
    << std::endl;

// Loop around until n_asamples is reached
for(unsigned int i = 1; i < n_asamples; i++){
            for(unsigned int n = 0; n <
                n_entropic_samples; n++){
                      unsigned int y = n % size;
                      unsigned int x = (n % (size*size))
                          / size;
                      if(x == interfacepoint && interface
                          == true){
                      // This is idiotic of me. State 0
                          isn't a state it's empty.
                              // So need to catch
                                  that.
                              if( (grid[x-1][y]+k
                                  )%n_q == 0){
```

```
                                                grid[x][y]
                                                   = 1;
                                        } else {
                                                grid[x][y]
                                                   = (grid[
                                                   x-1][y]+
                                                   k)%n_q;
                                        }

                        } else {
                                smooth_wanglandau_update(x,
                                   y);
                        }
                        wanglandau_measurement(n);
                }

                estar_avg = wanglandau_average(estar);
                //std::cout << estar_avg << std::endl;
                if(n_q == 2){
                        aguess[i] = aguess[i-1] + (12 / (
                           target_width * target_width)) *
                           estar_avg;
                } else {
                        aguess[i] = aguess[i-1] + (12 / ((4
                            * target_width) + (target_width
                            * target_width))) * estar_avg;
                }
                //std::cout << (i/(double)n_asamples)*100
                   << "%" << std::endl;
                cur_a = aguess[i];
        }

        std::ofstream file;
        file.open("an.dat");
        for(unsigned int i = 0; i < n_asamples; i++){
                file << aguess[i] << std::endl;
        }
        file.close();
}
```

```
void POTTS_MODEL::smooth_wanglandau_update(unsigned int x,
   unsigned int y){
        std::uniform_int_distribution<unsigned int>
           distribution(1,n_q);
        double H_old = energychange(x,y);
        unsigned int old_q = grid[x][y];

        unsigned int new_q = distribution(generator);
        grid[x][y] = new_q;

        double H_new = energychange(x,y);
        double delta = H_new - H_old;

        if( outsideenergyband() == 1){
                grid[x][y] = old_q;
                //std::cout << "Outside Band: Ignoring" <<
                   std::endl;
        } else {
                //std::cout << "Inside Band: Running
                   Metropolis" << std::endl;
                std::uniform_real_distribution<double>
                   pdistribution(0,1);
                double rand = pdistribution(generator);
                if( delta < 0.0 ){
                        grid[x][y] = new_q;
                        acceptance++;
                } else {
                        if(exp(-1 * cur_a * delta) > rand){
                                grid[x][y] = new_q;
                                acceptance++;
                        } else {
                                grid[x][y] = old_q;
                        }
                }
        }
}

void POTTS_MODEL::wanglandau_measurement(unsigned int k){
        estar[k] = energycalc()- target_e; //Total on
           Lattice less target
        //std::cout << energycalc() << std::endl;
}

void POTTS_MODEL::wanglandau_update(){
        std::uniform_int_distribution<unsigned int>
           distribution(0,size-1);
        unsigned int x = distribution(generator);
        unsigned int y = distribution(generator);
        double H_old = energychange(x,y);
        H_old = energycalc();
        unsigned int old_q = grid[x][y];

        //double H_old = energycalc();
        std::uniform_int_distribution<unsigned int>
           qdistribution(1,n_q);
        unsigned int new_q = qdistribution(generator);
        grid[x][y] = new_q;
        double H_new = energychange(x,y);
        H_new = energycalc();
        double delta = H_new - H_old; // This looks weird
           should be
        //double delta = H_new - target_e;

        if( outsideenergyband() == 1 ){
                grid[x][y] = old_q;
        } else {
                std::uniform_real_distribution<double>
                   pdistribution(0,1);
                double rand = pdistribution(generator);
                if (delta < 0.0){
                        grid[x][y] = new_q;
                        acceptance++;
                } else {
                        if(exp(-1 * cur_a * delta) > rand){
                                grid[x][y] = new_q;
                                acceptance++;
                        } else{
                                grid[x][y] = old_q;
                        }
                }
        }
}

double POTTS_MODEL::wanglandau_average(double *array){
        double average = 0.0;
        for(unsigned int i = 0; i < n_entropic_samples; i
           ++){
                average += array[i];
        }
        average /= n_entropic_samples;
        //std::cout << average << std::endl;
        return(average);
}

double POTTS_MODEL::wanglandau_error(double *array, double
   average){
        double *bin, *jackbins;
        unsigned int numbins = 100;
        bin = new double[numbins];
        jackbins = new double[numbins];
        unsigned int slice = n_asamples / numbins;
        double sumbins = 0.0;
        for(unsigned int l = 0; l < numbins; l++){
                bin[l] = 0.0;
                for(unsigned int k = 0; k < slice; k++){
                        bin[l] += array[(l * slice)+k];
                }
                bin[l] /= slice;
                sumbins += bin[l];
        }
        // Forming Bins
        for(unsigned int l = 0; l < numbins; l++){
                jackbins[l] = (sumbins - bin[l]) / (numbins
                    - 1.0);
        }
        double error = 0.0;
        for(unsigned int l = 0; l < numbins; l++){
                error += (average - jackbins[l]) * (average
                    - jackbins[l]);
        }
        error *= (numbins - 1.0) / (double)numbins;
        error = sqrt(error);
        return(error);
}


int POTTS_MODEL::outsideenergyband(){
        double target_lb, target_ub;
        target_lb = target_e - (target_width/2);
        target_ub = target_e + (target_width/2);
        double energy = energycalc();
        if( energy < target_ub && energy > target_lb){
                return(0);
        } else {
                return(1);
        }
}

void POTTS_MODEL::drivetotarget(unsigned int i, unsigned
   int j){
        unsigned int q_before = grid[i][j];
        double energy_before = energycalc();

        unsigned int rand_q;
        std::uniform_int_distribution<unsigned int>
           qdistribution(1,n_q);
        rand_q = qdistribution(generator);
        grid[i][j] = rand_q;
        double energy_after = energycalc();

        if( abs(energy_before - target_e) < abs(
           energy_after - target_e) ){
                grid[i][j] = q_before;
                //std::cout << "Stuck with Original Q" <<
                   std::endl;
        } else {
                grid[i][j] = rand_q;
                //std::cout << "Gone with Random Q of " <<
                   rand_q << std::endl;
        }
        //std::cout << energycalc() << std::endl;
}
```

## 5.2 Metropolis

### 5.2.1 Q10EnergyBeta16x16RangeofBeta.dat

```
0 −0.200109 0.00025786
0.05 −0.209624 0.00027772
0.1 −0.21899 0.000264747
0.15 −0.22827 0.000328895
0.2 −0.239697 0.00034315
0.25 −0.249341 0.000322799
0.3 −0.260898 0.000375688
0.35 −0.272723 0.000393963
0.4 −0.284388 0.000390524
0.45 −0.297082 0.000406967
0.5 −0.311139 0.000454669
0.55 −0.324349 0.000481373
0.6 −0.339617 0.000556149
0.65 −0.355262 0.000571083
0.7 −0.370582 0.000588718
0.75 −0.388206 0.000693746
0.8 −0.404432 0.000805053
0.85 −0.423668 0.000777047
0.9 −0.44452 0.000851305
0.95 −0.465189 0.000954719
1 −0.49025 0.00117225
1.05 −0.516876 0.00126606
1.1 −0.546703 0.00145358
1.15 −0.57747 0.0017864
1.2 −0.611437 0.00197478
1.25 −0.650466 0.00275758
1.3 −0.696883 0.00279037
1.35 −0.772292 0.00896837
1.4 −0.948917 0.0261211
1.45 −1.77709 0.00725456
1.5 −1.83122 0.00498143
1.55 −1.87696 0.00297309
1.6 −1.91116 0.00235711
```

```
1.65 −1.93321 0.00199471
1.7 −1.94931 0.00146225
1.75 −1.95923 0.00132297
1.8 −1.96639 0.00101562
1.85 −1.97237 0.001026
1.9 −1.97907 0.000982634
1.95 −1.98302 0.000793976
2 −1.98726 0.000609572
2.05 −1.98861 0.000599667
2.1 −1.99062 0.000576544
2.15 −1.99192 0.000539044
2.2 −1.99415 0.000428889
2.25 −1.9955 0.00038934
2.3 −1.99605 0.000289569
2.35 −1.99703 0.000253472
2.4 −1.99816 0.000224678
2.5 −1.99819 0.000216858
2.55 −1.99873 0.000179629
2.6 −1.99891 0.000202063
2.65 −1.99915 0.000155375
2.7 −1.99914 0.000163213
2.75 −1.99921 0.000150579
2.8 −1.99939 0.000147808
```

### 5.2.2 Q10MagnetisationBeta16x16RangeofBeta.dat

```
0 0.0536258 0.000282599
0.05 0.0540781 0.000279901
0.1 0.0542501 0.000287509
0.15 0.0554078 0.000295225
0.2 0.0564436 0.000321571
0.25 0.0566199 0.000357871
0.3 0.0576936 0.000371024
0.35 0.0579753 0.000341963
0.4 0.0585935 0.000385968
0.45 0.0603006 0.000411977
0.5 0.0615635 0.000414219
0.55 0.0630296 0.000409879
0.6 0.0633744 0.000417007
0.65 0.0654273 0.000408695
0.7 0.0667299 0.000475517
```

```
0.75 0.06832 0.000532082
0.8 0.0693393 0.000636549
0.85 0.0719688 0.00063416
0.9 0.0744893 0.000661863
0.95 0.0769229 0.000845882
1 0.0786101 0.000861817
1.05 0.083604 0.000845359
1.1 0.0866739 0.00114479
1.15 0.0903927 0.00144828
1.2 0.0996223 0.00169752
1.25 0.105431 0.00180421
1.3 0.114108 0.00265604
1.35 0.14669 0.00880269
1.4 0.260539 0.0227491
1.45 0.916998 0.00384802
1.5 0.940809 0.00234376
1.55 0.958311 0.00120807
1.6 0.970224 0.000934043
1.65 0.977959 0.000659634
1.7 0.983036 0.000501082
1.75 0.986376 0.000447583
1.8 0.988708 0.000321665
1.85 0.990527 0.000337317
1.9 0.992693 0.00032042
1.95 0.993967 0.000271542
2 0.995432 0.000211862
2.05 0.99583 0.000202744
2.1 0.996618 0.000202872
2.15 0.997036 0.000191388
2.2 0.997793 0.000166809
2.25 0.995348 7.58233e−05
2.3 0.998512 0.000107969
2.35 0.995623 4.63501e−05pn
2.4 0.9993 8.78052e−05
2.5 0.999318 7.76653e−05
2.55 0.999527 6.85556e−05
2.6 0.999585 7.80653e−05
2.65 0.999671 6.15873e−05
2.7 0.999652 6.92973e−05
2.75 0.995975 2.34157e−05
```

```
2.8 0.999773 5.52442e-05
```

### 5.2.3 Q10SpecificHeatBeta16x16RangeofBeta.dat

```
0 0 0.000277831
0.05 1.83835e-06 0.000301732
0.1 7.59666e-06 0.000289046
0.15 1.8079e-05 0.000362044
0.2 3.32178e-05 0.000381555
0.25 5.3092e-05 0.000361097
0.3 8.05329e-05 0.000424603
0.35 0.000116444 0.00045009
0.4 0.000153867 0.000448959
0.45 0.000200627 0.000474747
0.5 0.000266681 0.000538302
0.55 0.00033652 0.000575057
0.6 0.000424915 0.000675175
0.65 0.000509799 0.000702659
0.7 0.000625063 0.000735869
0.75 0.000778356 0.000885296
0.8 0.000890101 0.00104076
0.85 0.00114436 0.00102571
0.9 0.00135769 0.00114159
0.95 0.00157159 0.00131239
1 0.00188835 0.00166014
1.05 0.0023751 0.00183388
1.1 0.00278538 0.00217225
1.15 0.00348612 0.00274892
1.2 0.00416383 0.00314387
1.25 0.00556986 0.0046199
1.3 0.00643498 0.00486409
1.35 0.0214081 0.0186271
1.4 0.140861 0.0700809
1.45 0.0178388 0.0262297
1.5 0.010962 0.0184243
1.55 0.00702258 0.0114387
1.6 0.00515308 0.00921808
1.65 0.00358491 0.00791604
1.7 0.00256486 0.00585533
1.75 0.00220429 0.00532126
1.8 0.00170033 0.00410681
```

```
1.85 0.00164258 0.00415803
1.9 0.0013511 0.00399262
1.95 0.00107063 0.00323329
2 0.000802398 0.00249073
2.05 0.000762133 0.00244968
2.1 0.000673591 0.00235798
2.15 0.000638553 0.00220467
2.2 0.000472682 0.00175664
2.25 0.000351523 0.00159687
2.3 0.000319642 0.00118773
2.35 0.00025569 0.00103987
2.4 0.000164989 0.000922352
2.5 0.000177274 0.000889706
2.55 0.000121696 0.000737807
2.6 0.000113462 0.000829744
2.65 0.000100027 0.000637751
2.7 9.9788e-05 0.000670199
2.75 9.32963e-05 0.000618449
2.8 7.96827e-05 0.000606782
```

### 5.2.4 Q10SusceptibilityBeta16x16RangeofBeta.dat

```
0 0 0.00028509
0.05 4.29466e-05 0.000282763
0.1 8.58881e-05 0.000290136
0.15 0.000135661 0.000298115
0.2 0.000184004 0.000325097
0.25 0.000230868 0.00036167
0.3 0.000297625 0.00037515
0.35 0.000341974 0.000345617
0.4 0.000402675 0.000390005
0.45 0.000479985 0.000416988
0.5 0.000555462 0.000419142
0.55 0.000639779 0.000415187
0.6 0.000702171 0.000422092
0.65 0.000797168 0.000414669
0.7 0.000880077 0.000481797
0.75 0.000993099 0.000539691
0.8 0.00109877 0.000646136
0.85 0.00125121 0.000643803
0.9 0.00138957 0.000673981
```

```
0.95 0.00166783 0.000861908
1 0.00178388 0.000879005
1.05 0.00208251 0.000864356
1.1 0.0023205 0.00116932
1.15 0.00272497 0.00148544
1.2 0.00330586 0.00174601
1.25 0.00370356 0.00185931
1.3 0.00485822 0.0027615
1.35 0.0158726 0.010339
1.4 0.0781662 0.0319328
1.45 0.00324901 0.00774562
1.5 0.00153613 0.00483858
1.55 0.000770371 0.00257786
1.6 0.000535052 0.00201244
1.65 0.000292019 0.00144091
1.7 0.000207642 0.00110016
1.75 0.000169414 0.000984836
1.8 0.00012585 0.000710757
1.85 0.000119812 0.000746123
1.9 9.35882e-05 0.000710222
1.95 7.74758e-05 0.000602487
2 5.65397e-05 0.000471058
2.05 5.37267e-05 0.000450788
2.1 4.51293e-05 0.000451287
2.15 4.36806e-05 0.00042575
2.2 3.4398e-05 0.000371114
2.25 9.52731e-06 0.000168444
2.3 2.20644e-05 0.000240564
2.35 5.39795e-06 0.000103016
2.4 1.10412e-05 0.000195722
2.5 1.11168e-05 0.00017311
2.55 7.38777e-06 0.000152878
2.6 7.03322e-06 0.000174046
2.65 6.69305e-06 0.000137209
2.7 6.75523e-06 0.000154427
2.75 1.38085e-06 5.20999e-05
2.8 4.30191e-06 0.000123173
```

## 5.3 Wang Landau

### 5.3.1 An16x16Convergence.dat

```
2
1.83742
1.67596
1.51659
1.36017
1.21047
1.07115
0.958293
0.898123
0.886636
0.894645
0.890632
0.876113
0.886485
0.882993
0.893742
0.900179
0.882147
0.896148
0.885572
0.890101
0.889612
0.930086
0.897667
0.886961
0.885495
0.895436
0.889006
0.904736
0.884794
0.88835
0.894414
0.906143
0.885411
0.893166
0.895981
0.892837
```

```
0.896723
0.888887
0.888455
0.889703
0.896108
0.891426
0.891259
0.884917
0.882937
0.888925
0.889294
0.879971
0.887493
0.885633
0.881775
0.912052
0.894377
0.885789
0.886023
0.890293
0.887923
0.886067
0.881623
0.883337
0.879752
0.888423
0.885911
0.8877
0.881466
0.908074
0.889962
0.88437
0.890146
0.890344
0.881311
0.923801
0.895426
0.887581
0.883116
0.887668
```

```
0.883806
0.893626
0.890971
0.886484
0.885689
0.888526
0.885702
0.888504
0.895164
0.892044
0.879771
0.892959
0.900478
0.888948
0.887261
0.888088
0.894206
0.901281
0.885127
0.896582
0.896501
0.90774
0.880374
```

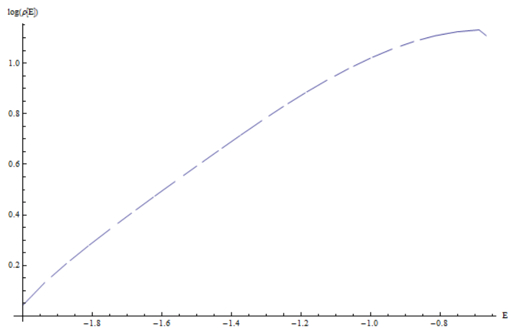
## 5.3.2 Continuity Constants

| $Q2$ | $Q3$ | $Q4$ | $Q8$ | $Q10$ |
|---|---|---|---|---|
| 0.0783575 | 0.0887444 | 0.0948331 | 0.107823 | 0.112169 |
| 0.149986 | 0.170668 | 0.182925 | 0.209201 | 0.217729 |
| 0.212082 | 0.242924 | 0.261318 | 0.301153 | 0.313728 |
| 0.269831 | 0.310671 | 0.335173 | 0.38864 | 0.405326 |
| 0.325173 | 0.375721 | 0.40646 | 0.473809 | 0.494806 |
| 0.378996 | 0.439152 | 0.47602 | 0.557657 | 0.583087 |
| 0.431435 | 0.501496 | 0.544567 | 0.640646 | 0.670586 |
| 0.481899 | 0.562968 | 0.61251 | 0.723112 | 0.75776 |
| 0.529428 | 0.623959 | 0.679999 | 0.805459 | 0.844962 |
| 0.573241 | 0.684446 | 0.74747 | 0.887795 | 0.932316 |
| 0.612643 | 0.743848 | 0.814892 | 0.970099 | 1.01998 |
| 0.647018 | 0.801437 | 0.881851 | 1.05277 | 1.10781 |
| 0.675715 | 0.856414 | 0.947792 | 1.1361 | 1.19608 |
| 0.69803 | 0.908095 | 1.0121 | 1.21999 | 1.28512 |
| 0.713377 | 0.955984 | 1.0742 | 1.30418 | 1.37436 |
| 0.721189 | 0.999568 | 1.13333 | 1.3882 | 1.46418 |
| Null | 1.03814 | 1.189 | 1.4716 | 1.55451 |
| Null | 1.07098 | 1.24065 | 1.55375 | 1.64428 |
| Null | 1.09738 | 1.28747 | 1.63399 | 1.73275 |
| Null | 1.11655 | 1.32871 | 1.71175 | 1.81937 |
| Null | 1.12791 | 1.36349 | 1.78614 | 1.9035 |
| Null | Null | 1.39094 | 1.85622 | 1.98427 |
| Null | Null | 1.41012 | 1.92089 | 2.06066 |
| Null | Null | 1.42016 | 1.97909 | 2.13154 |
| Null | Null | Null | 2.02948 | 2.19551 |
| Null | Null | Null | 2.0704 | 2.25085 |
| Null | Null | Null | 2.10006 | 2.29553 |
| Null | Null | Null | 2.11625 | 2.32719 |
| Null | Null | Null | Null | 2.34277 |
| Null | Null | Null | Null | Null |
| Null | Null | Null | Null | Null |
| Null | Null | Null | Null | Null |

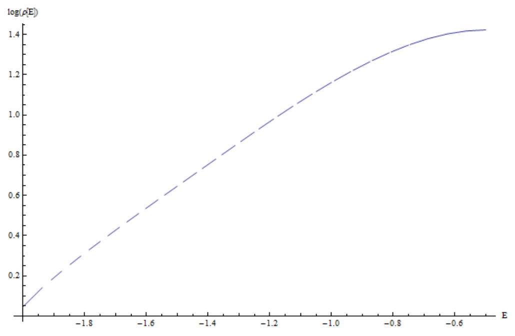
### 5.3.3 Q2 Piecewise Function

$$\begin{array}{cc}
1.25372(x+1.96875)+0.0783575 & -2 \leq x \leq -1.9375 \\
1.03838(x+1.90625)+0.149986 & -1.9375 \leq x \leq -1.875 \\
0.948715(x+1.84375)+0.212082 & -1.875 \leq x \leq -1.8125 \\
0.899257(x+1.78125)+0.269831 & -1.8125 \leq x \leq -1.75 \\
0.87168(x+1.71875)+0.325173 & -1.75 \leq x \leq -1.6875 \\
0.85064(x+1.65625)+0.378996 & -1.6875 \leq x \leq -1.625 \\
0.827427(x+1.59375)+0.431435 & -1.625 \leq x \leq -1.5625 \\
0.787394(x+1.53125)+0.481899 & -1.5625 \leq x \leq -1.5 \\
0.733548(x+1.46875)+0.529428 & -1.5 \leq x \leq -1.4375 \\
0.668473(x+1.40625)+0.573241 & -1.4375 \leq x \leq -1.375 \\
0.592389(x+1.34375)+0.612643 & -1.375 \leq x \leq -1.3125 \\
0.507607(x+1.28125)+0.647018 & -1.3125 \leq x \leq -1.25 \\
0.410693(x+1.21875)+0.675715 & -1.25 \leq x \leq -1.1875 \\
0.303397(x+1.15625)+0.69803 & -1.1875 \leq x \leq -1.125 \\
0.187714(x+1.09375)+0.713377 & -1.125 \leq x \leq -1.0625 \\
0.0622752(x+1.03125)+0.721189 & -1.0625 \leq x \leq -1
\end{array}$$

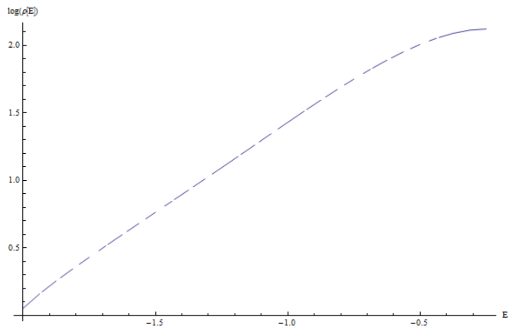
### 5.3.4 Q3 Q4 Q8 Q10 $Log(g(E))$



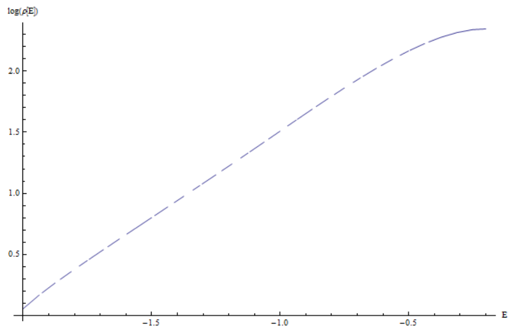
(a) Graph showing the $\log g\,(E)$ vs E for Q3



(b) Graph showing the $\log g\,(E)$ vs E for Q4

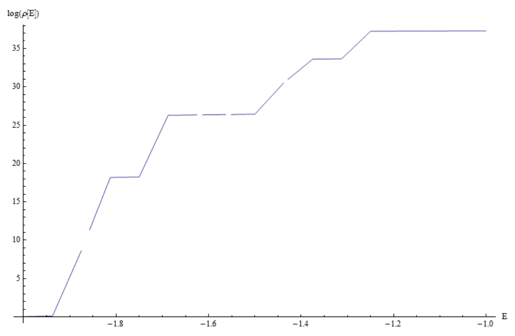


(c) Graph showing the $\log g\,(E)$ vs E for Q8

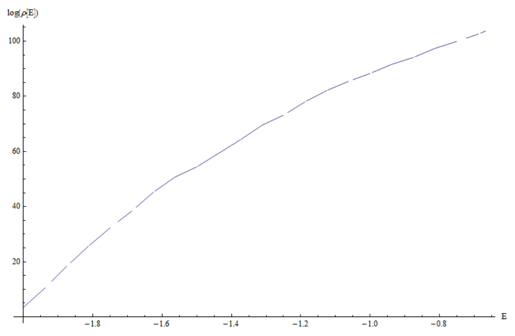


(d) Graph showing the $\log g\,(E)$ vs E for Q10

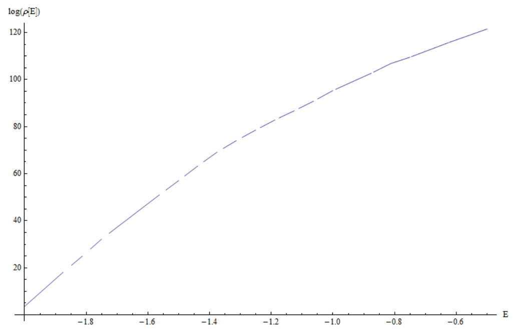
### 5.3.5 Q3 Q4 Q8 Q10 $Log(g(E))$ Twisted



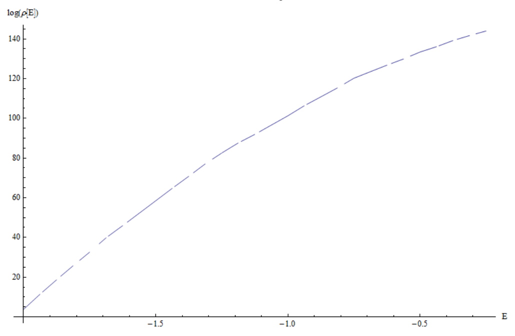
(e) Graph showing the $\log g\left(E\right)$ vs E for Q2



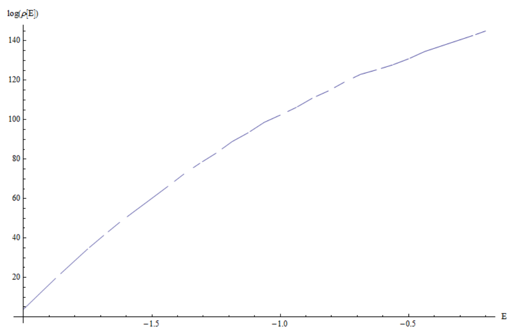
(f) Graph showing the $\log g\left(E\right)$ vs E for Q3



(g) Graph showing the $\log g\left(E\right)$ vs E for Q4



(h) Graph showing the $\log g\left(E\right)$ vs E for Q8



(i) Graph showing the $\log g\left(E\right)$ vs E for Q10

### 5.3.6 Mathematica Data Processing Notebook

## Process simulated data for all Q values simulataneously

Clear all previously calculated variable values

```
ClearAll["Global`*"]
```

Set the grid size for the input data and calculate data

```
GridSize = L;

delta = (GridSize)/(GridSize^2);
```

Now Import the Lattice Data from the spreadsheet

```
DataGrid = ImportString["", "TSV"];
```

### Process the Perioidic Lattice data

Define a function that converts a list {1,2,3,4,5} to access the correct element in the imported data

```
CorrectRegColumn[q_]:=If[q==2, 4, If[q==3, 5, If[q==4, 6, If[q==8, 7, If[q==10, 8, 4]]]]]
```

Define the function that calculates the continuity constants as a function of midpoint number and q.

```
ContinuityConstantsReg[k_, q_]:=(DataGrid[[1]][[CorrectRegColumn[q]]]/2 + Sum[DataGrid[[i]][[CorrectRegColumn[q]]],
{i, 1, k − 1}] + DataGrid[[k]][[CorrectRegColumn[q]]]/2) ∗ delta
```

Define a function of E done as x for simplicity and q using the Piecewise function generator natively embedded in Mathematica

```
S0[x_, q_]:=Piecewise[Table[{ContinuityConstantsReg[i, q] + DataGrid[[i]][[CorrectRegColumn[q]]](x − DataGrid[[i]][[3]]),
DataGrid[[i]][[1]]<=x<=DataGrid[[i]][[2]]}, {i, 1, Length[DataGrid]}]]
```

Plot a graph across the Energy Range {-2,-2/q} for all of the Q values being studied to ensure that continuity occurs as expected.

```
Table[Plot[S0[x, q], {x, −2, −2/q}, ImageSize->Large, AxesLabel->{E, Log[ρ[E]]}], {q, {2, 3, 4, 8, 10}}]
```

Calculate C0 for the Periodic Lattice

```
C0regular[q_]:=(2 ∗ (GridSize^2))/(NIntegrate[S0[x, q], {x, −2, −2/q}])
```

Define the DoS for the Periodic Lattice

```
DoSregular[x_, q_]:=Exp[C0regular[q] + S0[x, q]]
```

Define the Partition Function for the Periodic Lattice

```
Zreg[beta_, q_]:=NIntegrate[Log[DoSregular[x, q]] + beta ∗ x, {x, −2, −2/q}]
```

### Now to process the twisted Data

Define a function that converts a list {1, 2, 3, 4, 5} to access the correct element in the imported data

1

```
CorrectIntColumn[q_]:=If[q==2, 9, If[q==3, 10, If[q==4, 11, If[q==8, 12, If[q==10, 13, 9]]]]]
```

Define the function that calculates the continuity constants as a function of midpoint number and q.

```
ContinuityConstantsInt[k_, q_]:=(DataGrid[[1]][[CorrectIntColumn[q]]]/2 + Sum[DataGrid[[i]][[CorrectIntColumn[q]]],
{i, 1, k − 1}] + DataGrid[[k]][[CorrectIntColumn[q]]]/2) ∗ delta
```

Define a function of E done as x for simplicity and q using the Piecewise function generator natively embedded in Mathematica

```
S0int[x_, q_]:=Piecewise[Table[{ContinuityConstantsInt[i, q] + DataGrid[[i]][[CorrectIntColumn[q]]](x − DataGrid[[i]][[3]]),
DataGrid[[i]][[1]]<=x<=DataGrid[[i]][[2]]}, {i, 1, Length[DataGrid]}]]
```

Plot a graph across the Energy Range {-2,-2/q} for all of the Q values being studied to ensure that continuity occurs as expected.

```
Table[Plot[S0int[x, q], {x, −2, −2/q}, ImageSize->Large, AxesLabel->{E, Log[ρ[E]]}], {q, {2, 3, 4, 8, 10}}]
```

Calculate C0 for the Twisted Lattice

```
C0interface[q_]:=(2 ∗ (GridSize^2))/(NIntegrate[S0int[x, q], {x, −2, −2/q}])
```

Define the DoS for the Twisted Lattice

```
DoSinterface[x_, q_]:=C0interface[q] + Exp[S0[x, q]]
```

Define the Partition Function for the Periodic Lattice

```
Zint[beta_, q_]:=NIntegrate[Log[DoSinterface[x, q]] + beta ∗ x, {x, −2, −2/q}]
```

**NowtocalculatetheFreeEnergyoftheInterfaceattheCriticalPoint**

atcalculateCriticalEnergyFreeInterfaceNowofPointthe$^3$to

### Calculate the Interface Free Energy from the Twisted and Periodic Partiton Functions

```
IntFreeEnergy[beta_, q_]:= − Log[Zint[beta, q]/Zreg[beta, q]] − Log[GridSize]

Table[IntFreeEnergy[Log[1 + Sqrt[q]], q], {q, {2, 3, 4, 8, 10}}]
```

**Acknowledgements**

I would like to thank, My friends and family for keeping me going throughout this challenging project. My project supervisor, without him, I wouldn't have made it as far as I did. My peers inside the Masters physics group whose collective ethos and support has provided invaluable insights over the last few years.