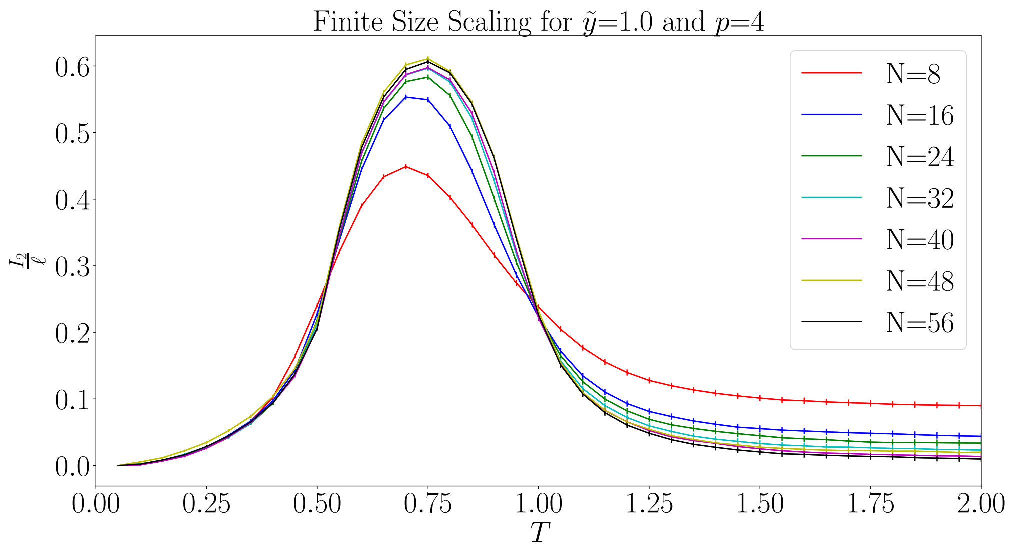


Numerical Methods in Computing Renyi Entropy

Ben Kolligs

May 2, 2018

Contents

Acknowledgements

I'd like to thank Mohamed Anber for the wonderful opportunity to join him in conducting research. I'd also like to thank the NSF, and the Murdock Charitable Trust for funding our research. I'd also like to thank Will Garrick and the Portland Institute of Compututational Sciences for the use of their powerful computer cluster. It was *critical* in getting our simulations done in a reasonable amount of time. I also would like to acknowledge the newly acquired BLT cluster at Lewis & Clark College.

Abstract

This research addresses the question of confinement/deconfinement transition in quantum chromodynamics (QCD). We study this theory using a new approach by compactifying two of the four dimensions over a torus. Further, we use Renyi Mutual Information (RMI) as an order parameter of the transition. RMI is a more dependable order parameter than traditional thermodynamic quantities, such as susceptibility, because it quantifies the information flow in a system irrespective of the details of the microscopic degrees of freedom. We also discuss the parallel computing techniques employed to generate the data using the Coeus computer cluster at Portland State University.

Chapter 1

Introduction

The strong nuclear force makes up most of the mass of the visible matter in the universe. This force is responsible for the structure of protons and neutrons and other particles in the baryon and meson families. Baryons and mesons are particles that are built of different combinations of *quarks*.

The strong force can be described by *quantum chromodynamics* theory (henceforth QCD). QCD clues us in to the mechanism of *confinement*. This is the mechanism by which quarks are held inside nucleons by the strong nuclear force. Quarks are connected by a linear potential called the "string". The string has a mathematical form of $V = \sigma r$, where σ is some constant and r is the distance between the quarks. When one heats up nuclei, this "string" melts, and an interesting phenomenon called *deconfinement* occurs. Deconfinement results in a state of matter called *quark-gluon plasma*.

We are interested in learning more about the deconfinement phase transition, and whether we can qualitatively analyze it using what is called an *order parameter*. An order parameter is a quantity that can be used to determine what phase a system is in.

A classic example is the density of water, shown in figure 1.1.

One might expect the string to function as an order parameter after our earlier discussion, since we have deconfinement when the string is gone. Unfortunately things are not so simple. At zero temperature, if one were to theoretically "pull" apart the quarks, it is energetically more favorable for two more quark pairs to form, as in figure 1.2

QCD without quarks has an order parameter. QCD with quarks, however, doesn't have an order parameter. Our research aimed to find a more general way to define an order parameter; this is Renyi Mutual Information. We found that RMI can indeed be a good order parameter for QCD with quarks.

The central question to our research is: is there an order parameter for quark deconfinement?

1.1 Compactification and XY Spin Models

To answer this question, we must simplify the QCD model to a point which we can derive non trivial information yet still be able to perform simulations in a reasonable amount of time. The problem is too difficult to solve in four dimensions, where QCD normally lives. If we wanted to simulate the four dimensional lattice, then there are

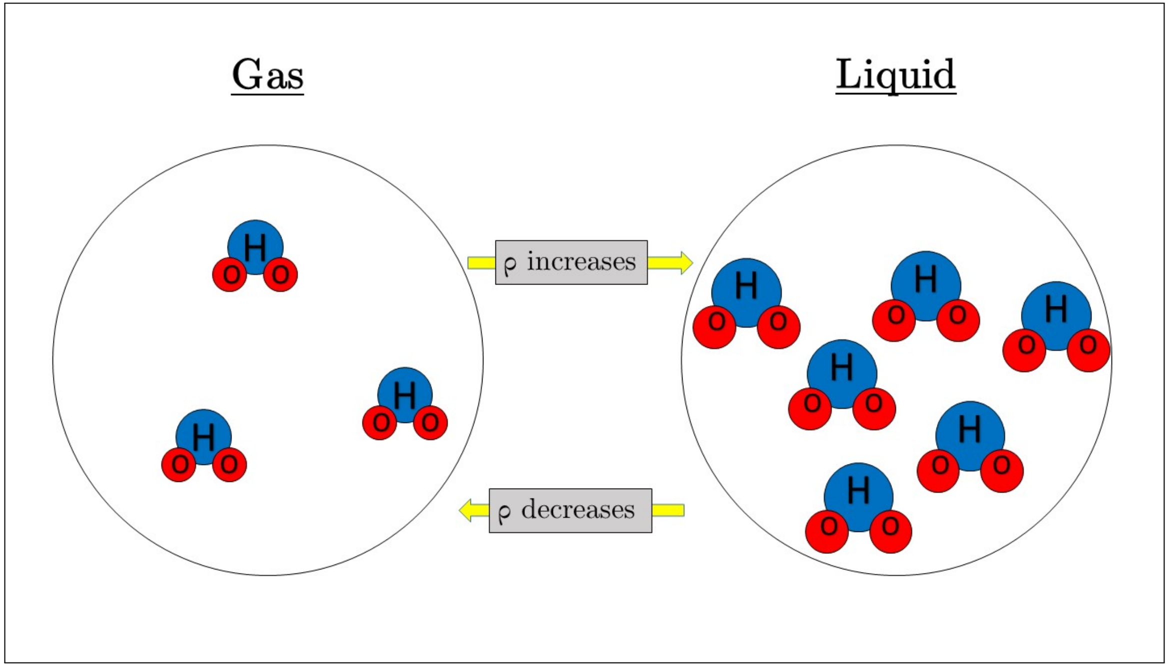


Figure 1.1: The phases of water shown with respect to the density ρ of water. Based on the numerical value of ρ at any given temperature, we can determine what phase the water is in. We hope to do the same with the deconfinement transition.

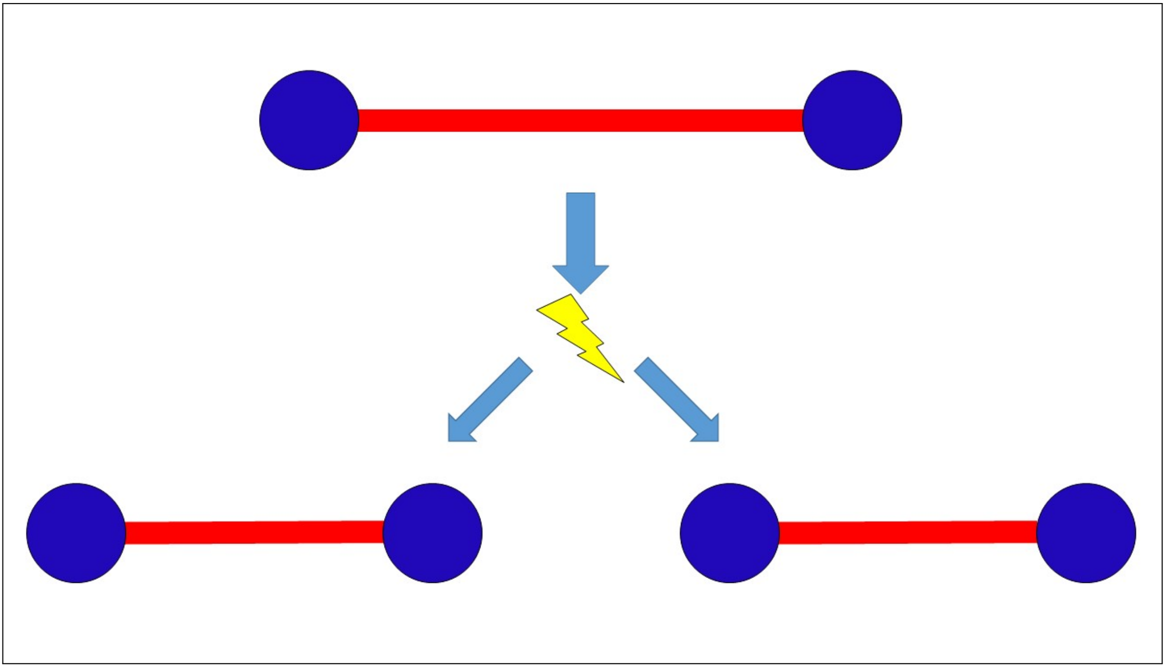


Figure 1.2: Dynamical quarks cannot use the string as an order parameter because breaking the string at low temperatures just creates more quarks.

connecting "struts" that look like the lattice in figure 1.3.

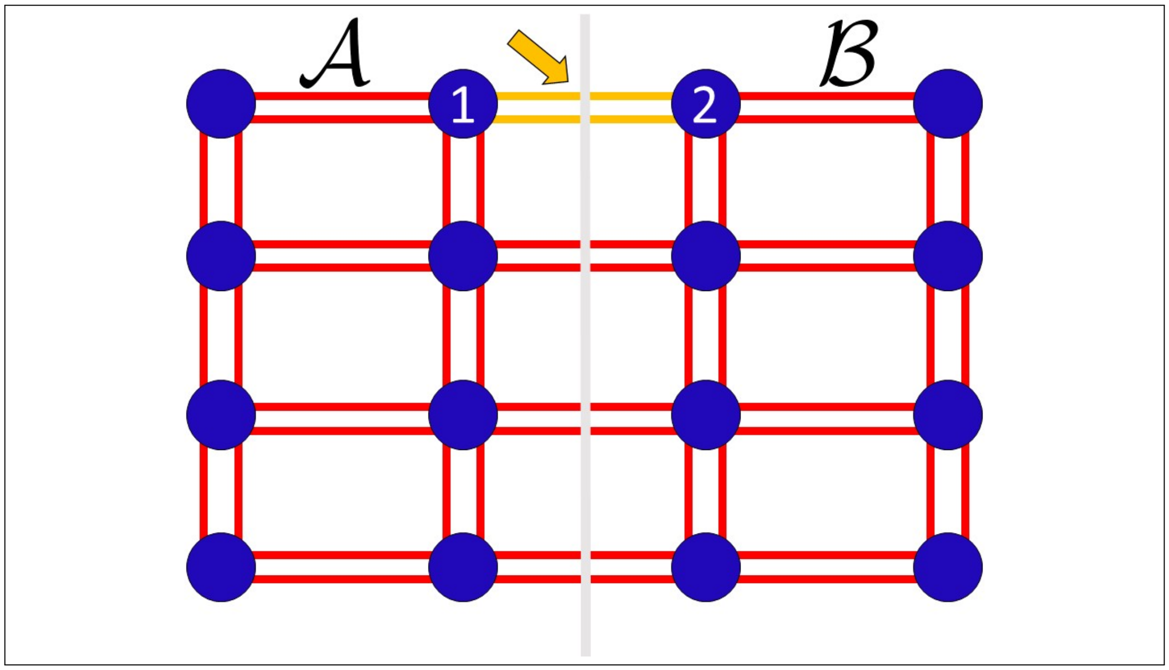


Figure 1.3: Does the orange strut belong to spin 1 or 2? It belongs to both, but there is no way to distinguish this if we split the lattice into two regions, which is necessary to do when calculating Renyi Mutual Information.

This is a novel approach developed specifically for this problem. So we compactify one of the spatial dimensions, which effectively rolls the spatial dimensions into a cylinder. However we also compactify the time dimension, which means the theory effectively lives on a torus.

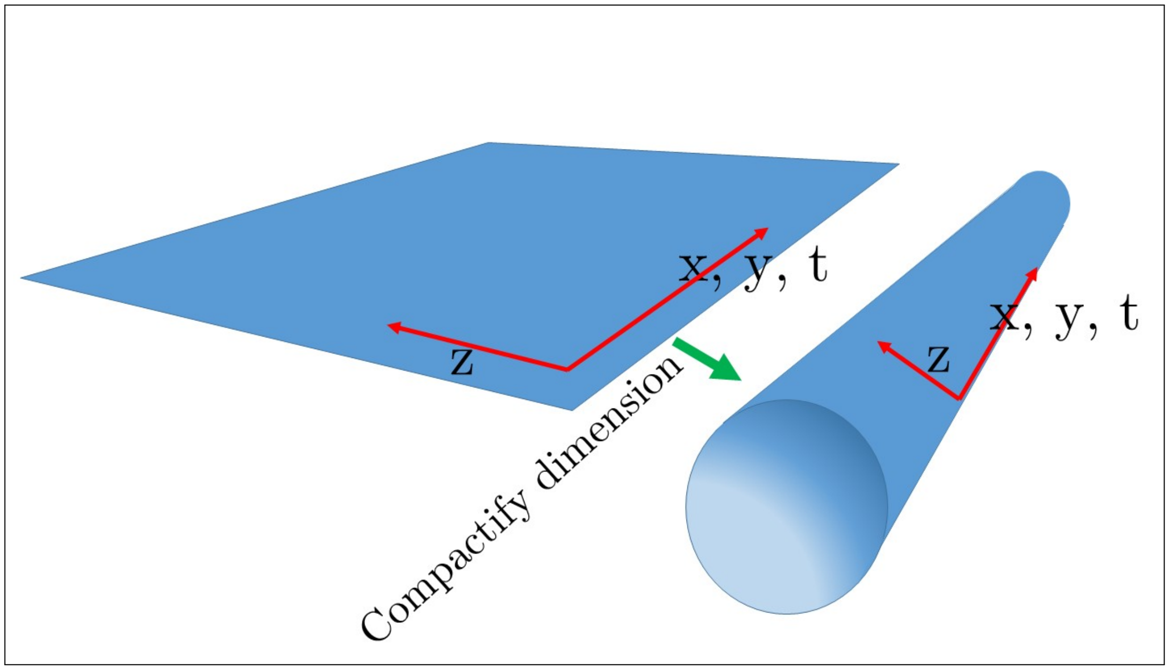


Figure 1.4: Rolling the QCD model into a cylinder.

This simplified model of QCD now exists in 2 dimensions and is exactly equivalent to a specific version of the XY spin model. This is a lattice of spins that can each rotate from 0 to 2π as pictured in figure 1.5. This spin model can be simulated at different temperatures using Monte Carlo methods. We can calculate different quantities (specifically candidates for an order parameter) from these simulations, which will indirectly tell us about the deconfinement transition in four dimensions.

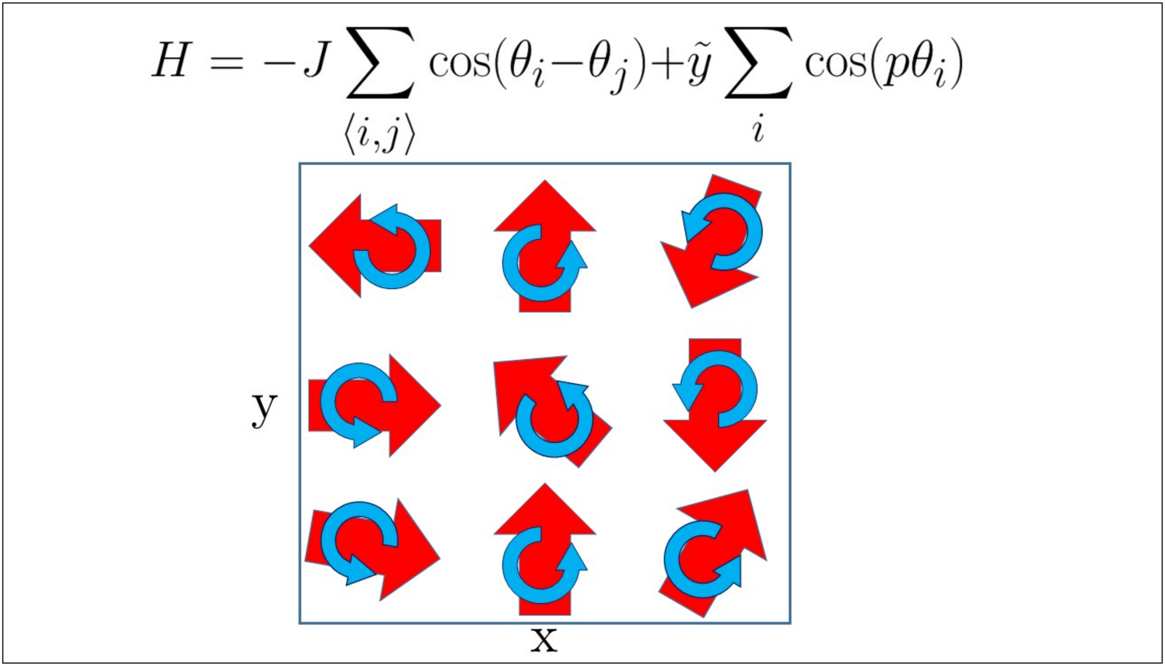

$H = -J\sum_{\langle i,j\rangle}\cos(\theta_i - \theta_j) + \tilde{y}\sum_i \cos(p\theta_i)$
y
x

Figure 1.5: The XY model. A lattice of spins magnetically interacting with each other according to the Hamiltonian displayed.

In particular, the equivalent XY model has the Hamiltonian,

$$H = -J\sum_{\langle i,j\rangle}\cos(\theta_i - \theta_j) + \tilde{y}\sum \cos(p\theta_i). \tag{1.1}$$

Where J is the temperature constant, the sum $\langle i, j\rangle$ is a nearest neighbor sum (closest spins to s_i) for spins s_i, s_j with rotation angles θ_i, θ_j, and symmetry constant p. We simulated different situations depending on the value of p.

This allowed us to truly check if RMI can function as an order parameter, and if the behavior is was we expect. We ran simulations for different p values according to these different situations:

1. $p = 1$: quarks are present in the model. Our model has no order parameter in the classic thermodynamic sense. We will check if RMI detects this.

2. $p = 2$: there are no quarks. This model has an order parameter in the thermodynamic sense. We can measure a phase transition using susceptibility in this case, and compare it to the RMI measurement to test RMI's effectiveness. Our procedure here was to use susceptibility as an experimental control, and RMI as our test variable.

3. $p = 4$: there are benign quarks (quarks that don't cause string breaking when we pull the string apart as described earlier)

Now that we can simulate the strong nuclear force, we can test our different models (the different p values) to see whether RMI might function as an order parameter for the deconfinement transition.

1.2 Renyi Entropy and Mutual Information

First we must build on some concepts from information theory to understand this quantity. Let a region $\mathcal{S}$ be bi-partitioned such that $\mathcal{S} = \mathcal{A} \cup \mathcal{B}$. Say we have two sets of random variables $\{x_i\} \in X$ and $\{y_i\} \in Y$ with support on $\mathcal{A}$ and $\mathcal{B}$ respectively. Let us define the quantity of *mutual information*, which is defined as:

$$I(X;Y) = \sum_{x \in X, y \in Y} p(x,y) \log \left(\frac{p(x,y)}{p(x)p(y)} \right). \tag{1.2}$$

Where $p(x), p(y)$ are the probability distributions of X and Y respectively, and $p(x,y)$ is the joint probability distribution between X and Y. The mutual information measures the amount of information shared between $\mathcal{A}$ and $\mathcal{B}$.

The uncertainty of a physical quantity is quantified by entropy. In information theory this uncertainty is given by Shannon's entropy:

$$S(\mathcal{A} \cup \mathcal{B}) \equiv - \sum_{x \in X, y \in Y} p(x,y) \log(p(x,y)) \tag{1.3}$$

The reduced entropy or von-Neumann entropy, $S(\mathcal{A})$, is obtained by tracing (an operation on a density matrix) out the degrees of freedom of $\mathcal{B}$:

$$S(\mathcal{A}) = - \sum_{x \in X} p(x) \log(p(x)) \tag{1.4}$$

and a similar expression for $S(\mathcal{B})$. Then, one can show that [75]

$$I(X;Y) = S(\mathcal{A}) + S(\mathcal{B}) - S(\mathcal{A} \cup \mathcal{B}). \tag{1.5}$$

The *generalized Renyi entropy* is defined as:

$$S_n(\mathcal{A} \cup \mathcal{B}) = \frac{1}{1-n} \log \left(\sum_{x \in X, y \in Y} p^n(x,y) \right), \tag{1.6}$$

and

$$S_n(\mathcal{A}) = \frac{1}{1-n} \log \left(\sum_{x \in X} p^n(x) \right), \tag{1.7}$$

such that Shannon's entropy is reproduced in the limit $S = lim_{n \to 1} S_n$. There is no direct way to simulate Shannon entropy with simulation techniques, and so this is why the generalized Renyi entropy is useful. Similarly, Renyi Mutual Information is given by the expression:

$$I_n(X;Y) = S_n(\mathcal{A}) + S_n(\mathcal{B}) - S_n(\mathcal{A} \cup \mathcal{B}). \tag{1.8}$$

The reduced Shannon's or the von-Neumann entropy $S(\mathcal{A})$ or $S(\mathcal{B})$ are examples of entanglement entropy. Unlike thermodynamic entropy, which scales with the system size, entanglement entropy scales with area. This area scaling is attributed to the fact that there is a finite correlation length ζ between two disjoint systems $\mathcal{A}$ and $\mathcal{B}$ such that $\mathcal{A}$ is the complement of $\mathcal{B}$ and ℓ is the boundary length between them. Then, the entanglement entropy takes the general form

$$S(\mathcal{A}) = S(\mathcal{B}) = \mathcal{C}\ell + \mathcal{D}\log\ell + \gamma \tag{1.9}$$

The constant $\mathcal{C}$ depends on the correlation length ζ, while the subleading logarithm is typical in quantum critical systems. The constant γ is known as the topological entanglement entropy, which is a quantity that will be addressed in future work. The area law results from the fact that regions that are separated by more than ζ will not contribute to the entanglement entropy. Since the the mutual information $I(X;Y)$ is the sum of entanglement entropies, it will also follow the area law. However, unlike entropy, which measures the uncertainty about the system, mutual information will quantify the amount of information shared between them, and hence, it is a more useful tool to detect phase transitions or other subtle properties of the systems. The remainder of this thesis will outline the methods used to compute and analyze the data that we generated in order to understand *deconfinement*.

1.3 The Strengths of Renyi Mutual Information

A more classic example of an order parameter for statistical systems is susceptibility. However, we are looking for a more general order parameter than susceptibility, because susceptibility focuses on the microscopic elements of a system. Our quark system is harder to define an order parameter for because of the nature of quarks. For example, susceptibility takes into account how magnetic the system is, but quarks do not exhibit magnetism in the Maxwellian sense.

In contrast, Renyi Mutual Information is a universal quantity. Information is not concerned with how magnetic the system is, and thus Renyi Mutual Information can actually be calculated and measured in systems *without* order parameters. One may ask how this is possible, since the whole point is that we are trying to use RMI as an order parameter. We can think of RMI as having a dual purpose. RMI can function as an order parameter by detecting phase transitions, and detecting how the phase transition occurs. In addition, RMI can also detect if a system *has* an order parameter in the first place. This is due to the universal nature of RMI, that it can act both as an order parameter and an order parameter detector.

I will refer to RMI as a universal order parameter, and susceptibility and other thermodynamic quantities as thermodynamic order parameters.

Chapter 2

Monte Carlo Methods

Monte Carlo algorithms try to approximate real life physical processes by generating random numbers. It is particularly useful when dealing with statistical mechanics, because random processes are fundamental in this area of physics. Since we are interested in measuring the Energy of a lattice, whose energy states E_i have a probability

$$P(E_i) = \frac{e^{-\beta E_i}}{Z}, \qquad \text{with} \qquad Z = \sum_i e^{-\beta E_i} \tag{2.1}$$

with $\beta = 1/k_B T$, k_B is Boltzmann's constant. Then for some quantity X that has a corresponding value X_i in the ith state is

$$\langle X \rangle = \sum_i X_i P(E_i). \tag{2.2}$$

In most cases we can't calculate this quantity (2.2) analytically because there are far too many terms in this sum. For example, a mole of gas has 10^{23} atoms in it, and if each atom had S states, then there are $S^{10^{23}}$ total states. This is far greater than the number of protons in the universe, and so even the most powerful of computers can't calculate this sum. So we turn to Monte Carlo methods. To evaluate this sum, we essentially choose N states at random and then calculate

$$\langle X \rangle \approx \frac{1}{N} \sum_{k=1}^{N} X_k. \tag{2.3}$$

However, this equation only works if we choose our states nonuniformly. The Boltzmann probability is exponentially small for states with energy $E_i >> k_B T$, which is the majority of all possible states. So if we choose from the possible states with uniform random numbers, then every energy state has equal probability of being chosen. Then since the majority of possible states are inconsequential, our calculation will be inaccurate. Thus, we need to choose random numbers according to the Boltzmann distribution, so that we actually choose meaningful states. We do this through a mechanism called a Markov Chain.

2.1 Markov Chains

Markov chains allow us to generate a string of states one after another according to our Boltzmann distribution. We start with a system state μ, and the Markov process will generate a new system state ν. The probability of generating the state ν given μ is the transition probability $T_{\mu\nu}$. We can choose this transition probability so that the probability of visiting any particular state on any step of the Markov chain is indeed the Boltzmann probability $P(E_i)$ in (2.1). We choose the $T_{\mu\nu}$ so that

$$\frac{T_{\mu\nu}}{T_{\nu\mu}} = \frac{P(E_\nu)}{P(E_\mu)} = \frac{e^{-\beta E_\nu}/Z}{e^{-\beta E_\nu}/Z} = e^{-\beta(E_\nu - E_\mu)}, \tag{2.4}$$

which is the ratio of probability of going from state μ to ν and the probability of going back from ν to μ.

$$\sum_\nu T_{\mu\nu} = 1 \tag{2.5}$$

because we must reach *some* state at each Markov chain step. Now we must choose a value for the transition probabilities. The most successful choice is one that was first made by Nicholas Metropolis and Keith Hastings in 1953, a choice which led to the famous *Metropolis Algorithm*.

2.2 The Metropolis Algorithm

The Metropolis Algorithm assumes that we can visit the same state more than once in the Markov chain, even on two consecutive steps. Suppose we start in state μ and we try to get to a new state ν by changing μ somehow. For our purposes, changing a state from μ to ν will always involve changing the orientation of one of the spins on the lattice. So when we select a random spin on the lattice, the "move" we are considering is changing the spin to a new orientation. We can either *accept* or *reject* the new state based on this *acceptance probability* R:

$$R = \begin{cases} 1 & \text{if } E_\nu \le E_\mu \\ e^{-\beta(E_\nu - E_\mu)} & \text{if } E_\nu > E_\mu \end{cases} \tag{2.6}$$

So if a move is rejected, the system remains in the old state for one more step. If a move is accepted, then the system changes to the new state, and the spin is rotated an appropriate amount. In practice, the spin move itself is also randomly generated, and then the change in energy ΔE is calculated based on this spin move. Then the ΔE is plugged into the acceptance ratio, where the program evaluates whether to execute the spin change or not. Equation (2.6) is saying that if a new state will decrease the energy of the system or keep it the same, then we always accept it. If the new state will increase the energy of the system, then we may accept it with the probability $e^{-\beta(E_\nu - E_\mu)}$. This probability satisfies equation (2.4). The total probability $T_{\mu\nu}$ is the probability that we choose to move out of all M possibilities times the probability of accepting a move:

$$T_{\mu\nu} = \frac{1}{M} e^{-\beta(E_\nu - E_\mu)} \qquad T_{\nu\mu} = \frac{1}{M}. \tag{2.7}$$

Plugging these into equation (2.4) gives the desired result,

$$\frac{T_{\mu\nu}}{T_{\nu\mu}} = \frac{e^{-\beta(E_\nu - E_\mu)}/M}{1/M} = e^{-\beta(E_\nu - E_\mu)}. \tag{2.8}$$

So now if we run the simulation for many steps of the Markov chain, we can accurately measure our quantity X in equation (2.3) according to random states generated according to a Boltzmann distribution! This acceptance ratio we used for the XY Model is expressible in code by the following line:

```
R = exp(dE / T)
if R > 1 or random() < R:
[accept new state]
```

The QCD Model will have the following acceptance ratio, due to its inverted thermodynamics:

```
R = exp(dE * T)
if R > 1 or random() < R:
[accept new state]
```

Now writing the code involves calculating the change in energy `dE` for each proposed state change , and running it through this ”if” statement every Markov chain step. An illustration of what happens each MC step at a given temperature T is shown in figure 2.1.

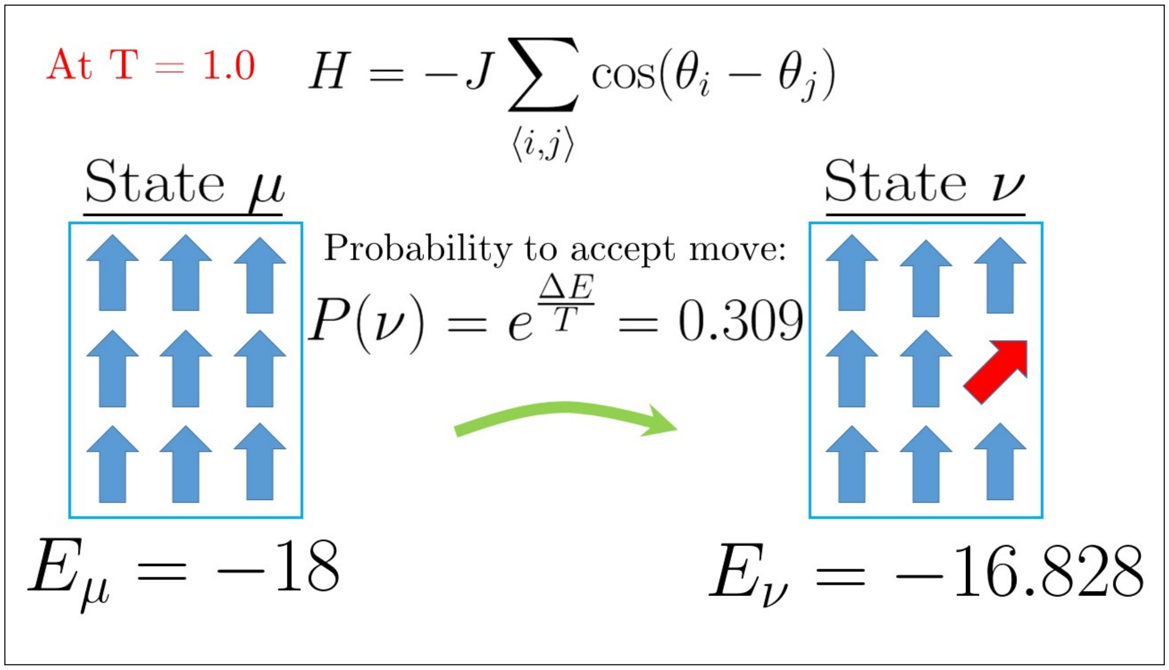

At T = 1.0
$H = -J \sum_{\langle i,j \rangle} \cos(\theta_i - \theta_j)$
State μ
State ν
Probability to accept move:
$P(\nu) = e^{\frac{\Delta E}{T}} = 0.309$
$E_\mu = -18$
$E_\nu = -16.828$

Figure 2.1: Given a system in state μ, the probability the system will switch to state ν. In code, a random number z would be compared to this $P(\nu)$ as in the code snippets above, and if $z < P(\nu)$ then the system would move from μ to ν. This lattice is an example of the XY Model with the Hamiltonian shown on the top of the figure.

To summarize, one step of the Metropolis Algorithm works as follows. This is the procedure used in all of our simulation code:

1. Choose a random spin s_i on the lattice.
2. Generate a random number between 0 and 2π. This is the spin orientation we are *considering* flipping s_i to.

3. Calculate ΔE by summing the nearest neighbor energy around s_i according to the Hamiltonian of the system. If we are simulating the XY Model, then it is $H = -J \sum_{i,j} cos(\theta_i - \theta_j)$.

4. Plug the ΔE from step 3 into the acceptance ratio. If the move is rejected, start over from step 1.

5. If the move is accepted, flip the spin and repeat from step 1.

2.2.1 Equilibration Time

One thing to note is that although our Markov chain always converges to the Boltzmann distribution, it takes time (i.e. a certain number of Markov chain steps) to do it. This length of time is called the *equilibration time*. It is very important to begin taking measurements *after* the system equilibrates, otherwise the system states are not necessarily in agreement with the Boltzmann distribution.

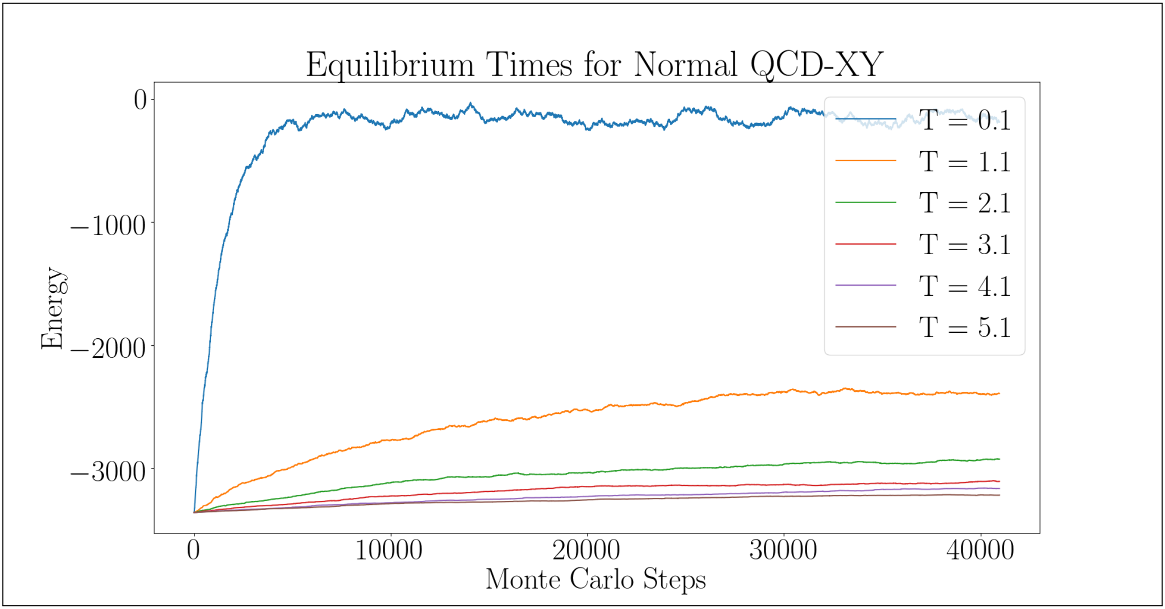


Figure 2.2: An energy vs MC step graph used to estimate equilibration time for *one* of the Monte Carlo Simulations we use to calculate RMI. In this case, this is for the un-replicated lattice energy for the QCD model.

In all of our simulations, equilibrium time was determined graphically. The Monte Carlo functions were modified to output their energy reading each step, to produce a graph of energy vs. Monte Carlo (Markov Chain) steps, like figure 2.2. The step number where the system starts to stabilize around an energy value is the equilibration time. In figure 2.2, for $T = 0.1$ (The blue line), it looks like the system equilibrates around the 7500th Monte Carlo step, around an energy of $E \approx -250$.

The equilibrium time increases as the lattice size increases. It will also increase when the exponential in the acceptance ratio is smallest. This means more steps need to be run before an energy move is accepted, since the probability of acceptance is so much smaller.

2.2.2 Correlation Time

Now that the system is equilibrated, we can start taking measurements. However, in order to make an accurate overall estimate of energy (or any other measurable quantity), we need to take an average over a large number of measurements. The quality of these measurements is also important. We want to take measurements that are statistically independent, and that do not have any correlation. By nature, our Markov process produces a chain of states that are generated from each other, thus there is inherently some correlation between states. So we must determine something called the *correlation time*, which tells us how long between measurements we must wait in order to get independent measurements. For our purposes, we used a correlation time $\tau = 2\tau_{eq}$ where τ_{eq} is the equilibrium time we determined by inspection in the previous section. In the appendices, most of the code displayed will have a chain of "if-statements" that will assign the proper correlation and equilibration time depending on the lattice size. This code looks like this:

Listing 2.1: The equilibration time for each lattice size of the QCD Model.

```
if N_global == 8:
    tau_global = 9000
if N_global == 16:
    tau_global = 14000
if N_global == 24:
    tau_global = 40000
```

Where `N_global` is the lattice size we want to run simulations for, and `tau_global` is the equilibration time for that lattice size. When the code actually takes measurements, it makes sure to wait a correlation time, `2*tau_global`, between each measurement.

2.3 Bootstrap Error

Since we are calculating expectation values, we need to know the error on these values to gauge their accuracy. Monte Carlo simulations are basically an experiment on the computer, so we can think of this error as "statistical", as one does with experiments. Due to the inherent random nature of Monte Carlo simulations, there will be a lot of variation moving from one step to the next. This statistical error is analogous with thermal fluctuations in the physical world. All other sources of error are *systematic* errors. systematic error stems from the method we used to make the measurements. This kind of error affects the entire simulation. For example, ideally we would wait an infinite amount of time for the system to equilibrate, but this is not practical. By only waiting a finite number of steps, some systematic error will be introduced. That said, we can calculate the *statistical* error on our measurements using a technique well known in statistics as "bootstrapping" or "bootstrap error analysis".

To explain this method, we will calculate the error on a list of n *independent* energy measurements. Before actually beginning the error calculation, we calculate $\langle E \rangle$ by averaging this list of independent measurements, using each measurement only one time. To start the error calculation, we choose n out of the measurements list at random. Note that this allows us to choose duplicates, so it's possible to calculate the error using the same measurement more than once. Then, average those randomly chosen measurements. Let's call this new average E_i. E_i is now a new calculation of $\langle E \rangle$ except that it is using the resampled list, so E_i was calculated using duplicates. Then repeat this process a sufficient number of times until we have a list of many different E_i's. For our simulations, we resampled n times. So if we took 20000 measurements, we would resample the measurement list 20000 times, and then we would have 20000 values of E_i. Next, with our list of E_i's, we calculate the sigma by:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(E_i - \langle E \rangle\right)^2}. \tag{2.9}$$

This gives the overall statistical error on $\langle E \rangle$.

This past few sections detailed how one can perform a successful Monte Carlo simulation for a lattice. Now in order to calculate the Renyi Mutual Information, we must use a technique called a *replica trick*.

2.4 Replica Trick

To calculate the Renyi Mutual Information or Renyi Entropy, we employ a replica trick that allows us to calculate RMI from a sum of energy estimators, which are themselves measured through the Metropolis Algorithm. The second Renyi Mutual Information is given as follows:

$$I_2(X;Y) = S_2(\mathcal{A}) + S_2(\mathcal{B}) - S_2(\mathcal{A} \cup \mathcal{B}). \tag{2.10}$$

Three distinct energy quantities are collected to calculate $I_2(T)$, which center around a *replica trick*. The idea is to separate the lattice into two sections, $\mathcal{A}$ and $\mathcal{B}$. Spins in regions $\mathcal{A}$ of the two lattices are "strongly correlated." This means that whenever we choose a spin in these regions, that spin on both lattices must always update the same way. This means that spins in the regions $\mathcal{B}$ on the lattices flip independently, and spins in regions $\mathcal{A}$ always flip together. Since we are calculating the *second* Renyi Entropy, we only have two lattice replicas. Figure 2.3 illustrates this.

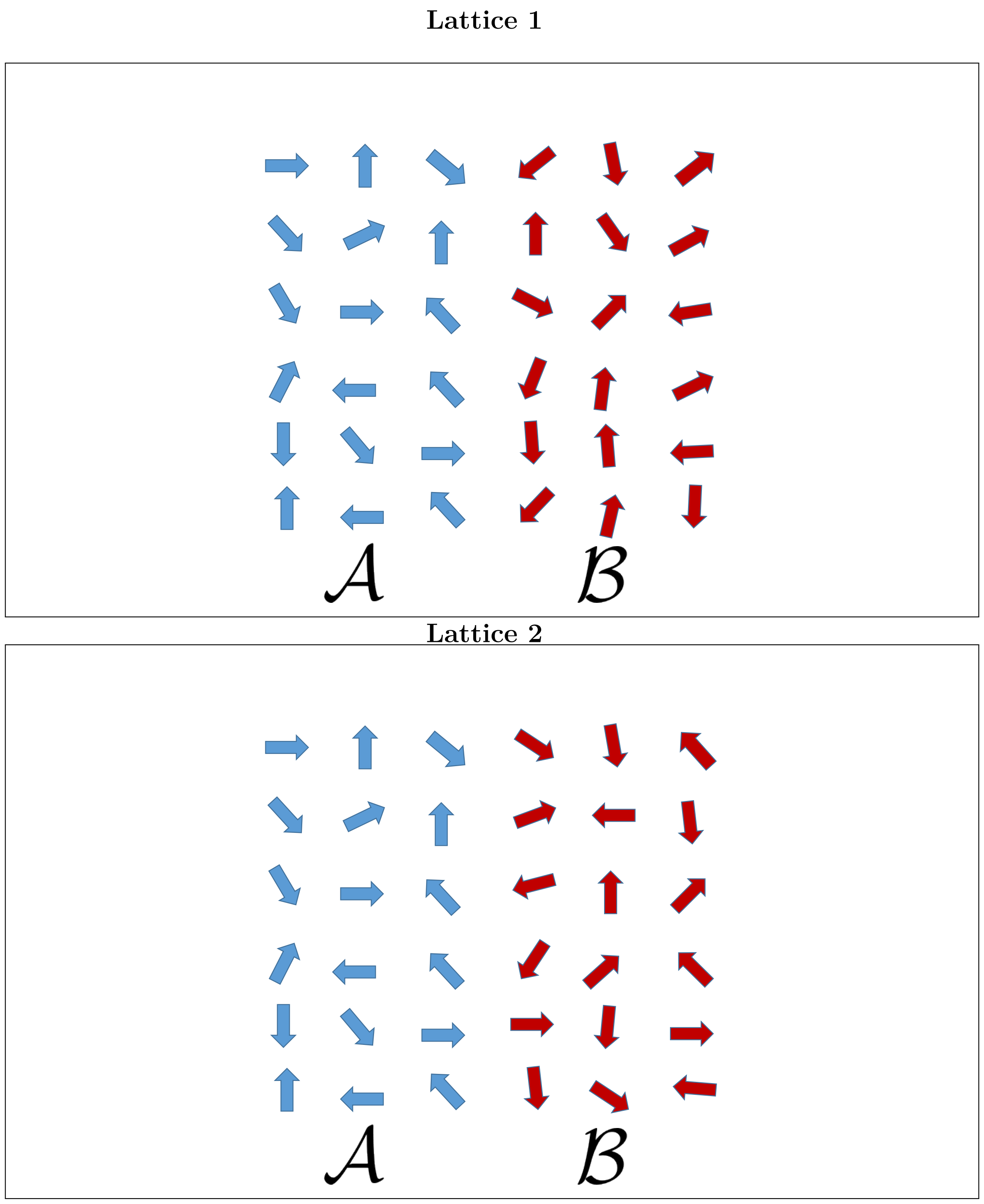


Figure 2.3: The lattices used in the replica trick

Mathematically, we are essentially creating two Hamiltonians from a single one. We show this now, by starting from the unreplicated lattice. Since we are calculating the *second* Renyi Entropy, the partition function of this single $L \times L$ lattice is given by:

$$\mathcal{Z}_1 = \sum_{\{S\}} e^{-\beta H} \tag{2.11}$$

Where H is the Hamiltonian of whatever model we are discussing. Now we take an

$L \times L$ lattice and divide it into two regions A and B. We want to find how much information region A remembers about region B as we trace over B. For a given fixed state in region A (i.e. specific fixed list of spins in region A), the probability to find such a state is

$$p_{I_A} = \mathcal{Z}_1^{-1}(\beta) \sum_{I_B} e^{-\beta H(I_A, I_B)}, \tag{2.12}$$

where we have divided our Hamiltonian into two regions A and B. The two regions can interact via the nearest neighbor spins on the boundary, which is captured in the calculation of Renyi Mutual Information. Again, since we are interested in the second Renyi Entropy, we need to find $p_{I_A}^2$ from equation (1.6), which is:

$$p_{I_A}^2 = \mathcal{Z}_1^{-2}(\beta) \left(\sum_{I_B} e^{-\beta H(I_A, I_B)} \right) \left(\sum_{J_B} e^{-\beta H(I_A, J_B)} \right). \tag{2.13}$$

The corresponding partition function for $p_{I_A}^2$ is obtained by tracing over all the states in A:

$$\mathcal{Z}_2[A, 2, \beta] = \sum_{I_A} p_{I_A}^2 = \mathcal{Z}_1^{-2}(\beta) \sum_{I_A, I_B, J_B} e^{-\beta (H(I_A, I_B) + H(I_A, J_B))} \tag{2.14}$$

So the Renyi Entropy can be described by

$$S_2(A) = -\log \mathcal{Z}_2[A, 2, \beta] + 2 \log \mathcal{Z}_1[\beta]. \tag{2.15}$$

We can also derive the Renyi Entropy using the expectation value of energy in a statistical system, which is given by

$$\langle E \rangle = \frac{\sum_n E_n e^{-\beta E_n}}{\sum_n e^{-\beta E_n}} = -\frac{\partial \log Z}{\partial \beta} = \frac{\partial S}{\partial \beta} \tag{2.16}$$

Integrating both sides with respect to β gives

$$S = \int_0^\beta \langle E \rangle d\beta. \tag{2.17}$$

Similarly, the Renyi Entropy is

$$S_2(A) = \int_0^\beta d\beta [\langle E \rangle_A - 2\langle E \rangle_0]. \tag{2.18}$$

where $\langle E \rangle_A$ is the total energy of the replica (when we trace over B), and $\langle E \rangle_0$ is the energy of the original system. It is from this definition of Renyi Entropy that we can define the Renyi Mutual Information, which measures, in a precise way, the information about A contained in B and vice versa:

$$I_2(A; B) = \int_0^\beta d\beta \quad [2\langle E \rangle_A(\beta) - \langle E \rangle_{A \cup B}(\beta) 2 \langle E \rangle_0(\beta)] \tag{2.19}$$

The quantity $\langle E \rangle_{A \cup B}(\beta)$ is the total energy estimator that corresponds to the Renyi entropy of the replica of the whole system $A \cup B$, i.e., it is equal to twice the energy of the original system, or equivalently, it is the energy of the original system at $T/2$.

Chapter 3

Codes and Algorithms

Now, we discuss the actual structure of the Monte Carlo simulations themselves. We ran simulations for variants of the Ising Model; the XY Spin Model with normal thermodynamics, and the XY Spin model with inverted thermodynamics (henceforth the QCD adjoint Model. This was the model that was linked to the QCD problem through the duality). The XY Spin model has no order parameter, and so it was a good test before moving on to the QCD adjoint model. The following is a description of the basic architecture used in the simulation code.

3.1 XY Model Code

The XY model is a lattice of spins that can all freely rotate on $[0, 2\pi]$. The Hamiltonian is:

$$H = -J \sum_{\langle i,j \rangle} \cos(\theta_i - \theta_j) \tag{3.1}$$

where J is a thermodynamical constant. This sum is a "nearest neighbor" sum for the i^{th} spin. The code is structured in three sections. Before the sections, the user defines the desired simulation parameters such as lattice size, independent measurements, and temperature range, among others. The first section defines three functions that input temperature T, and then run a Metropolis algorithm simulation to calculate the corresponding energy E of the lattice. The three functions calculate normal lattice energy $\langle E(T) \rangle_0$, region $A \cup B$ lattice energy $\langle E(T) \rangle_{A \cup B}$ and replica lattice energy $\langle E(T) \rangle_2$, respectively. A graph of these energies for a 16×16 lattice size can be seen in figure 3.1.

Each of these functions uses the Metropolis algorithm described in section 2.2, going through these steps:

1. Choose a random spin s_i on the lattice.

2. Generate a random number between 0 and 2π. This is the spin orientation we are *considering* flipping s_i to.

3. Calculate ΔE by summing the nearest neighbor energy around s_i according to the Hamiltonian of the system. If we are simulating the XY Model, then it is $H = -J \sum_{i,j} cos(\theta_i - \theta_j)$.

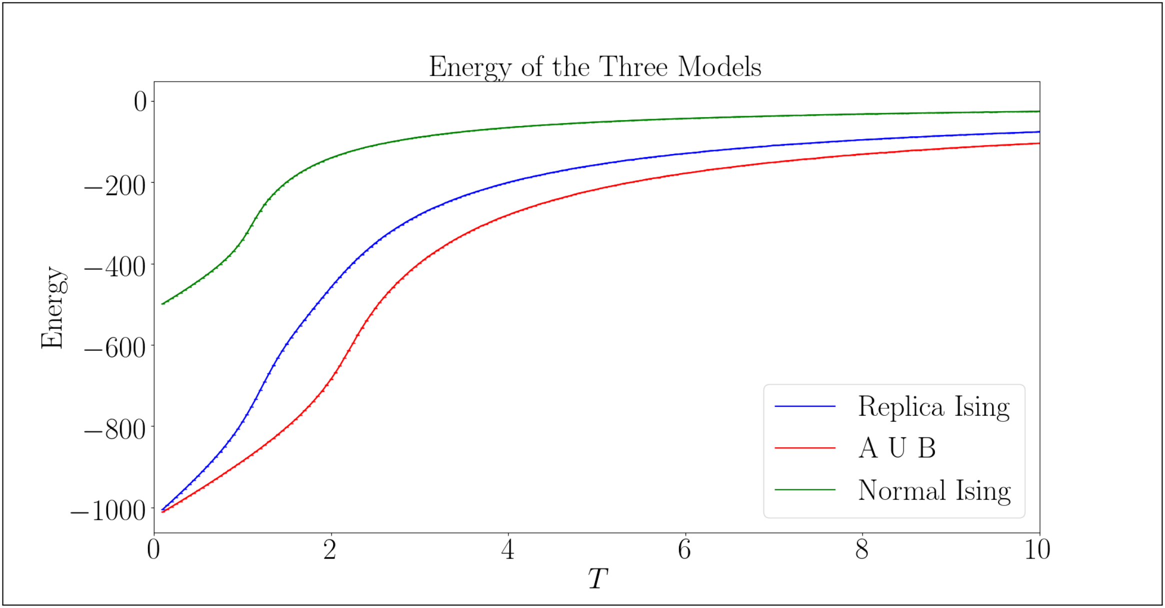


Figure 3.1: Replica Energy, $A \cup B$ energy, and unreplicated energy vs temperature for $N = 16$

4. Plug the ΔE from step 3 into the acceptance ratio. If the move is rejected, start over from step 1.

5. If the move is accepted, flip the spin and repeat from step 1.

Depending on what the user puts in for the number of independent measurements, this list will be repeated millions of times. We ran our highest accuracy simulations at 20000 measurements, which would repeat this list approximately 2 billion times for $N = 32$. This takes a couple days to complete, and to produce this data in a reasonable amount of time for various parameters, required heavy use of the Coeus computer cluster, at Portland State University. The parallel code needed to run this is detailed in chapter 4.

Then the code defines a function that uses the multiprocessing module to parallelize the task by distributing the preceding functions across the computer's cores. Then the code defines the highest level function (I will refer to this as the "control function") , which calls the previous parallel function for a specific temperature range, and then saves the output data to a file.

When one runs the code, the following happens: The control function creates a variable that will store all the different energies for different temperatures. Then a list of desired temperatures is generated, for example, temperatures between 0 and 4 with a step of $\Delta T = 0.05$. Then the parallel function will run the normal lattice energy function on all of those temperatures, recording the resulting energies, followed by the replica and $A \cup B$ energy functions. After all the energy calculations are completed, the program saves the specified temperature range of energies to a data file. This data file will later be aggregated with other temperature ranges to produce one cohesive data file. This is done so that the computer cluster can do the most efficient work (see chapter 4 for details).

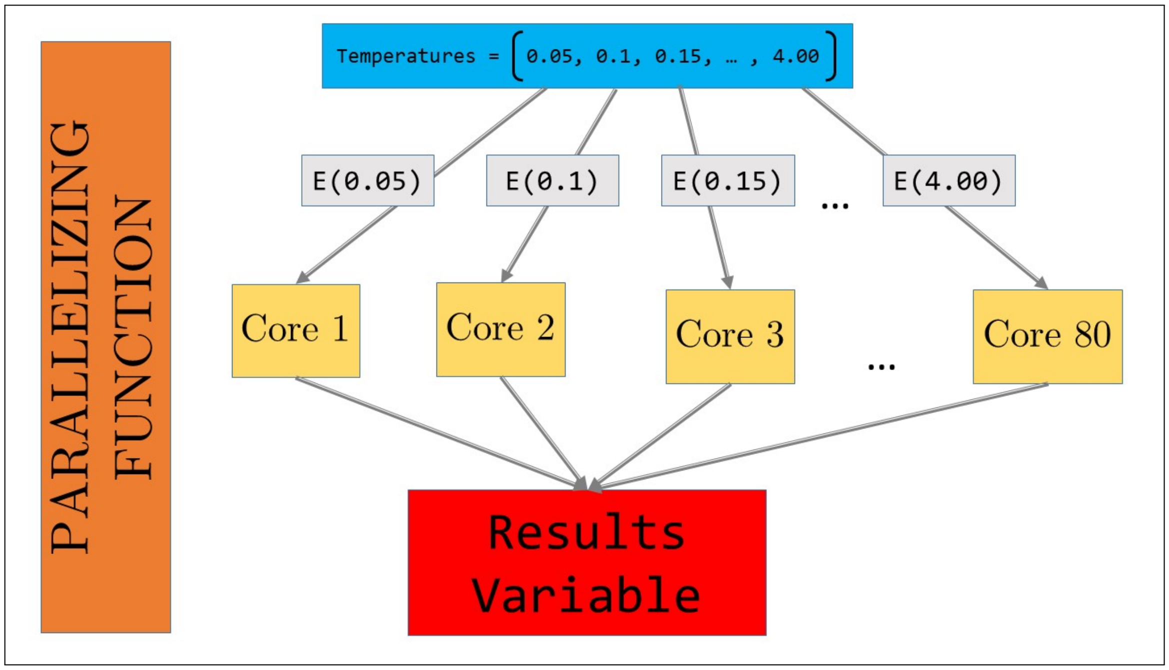


Figure 3.2: An illustration of how the code distributes work to computer cores

3.2 Calculating RMI

Once the data files were aggregated, we calculated RMI after the simulations completed using this formula:

$$\frac{I_2(T)}{\ell} = \sum_{T=i}^{T_f} \frac{2\langle E_i \rangle_A - 2\langle E_i \rangle + 4N^2}{T_i^2 \ell} \Delta T, \tag{3.2}$$

with error

$$\sigma_I(T) = \sum_{T=i}^{T_f} \sqrt{a^2 \sigma_{E_A}^2 + b^2 \sigma_{E^2}^2} \qquad a^2 = \left(\frac{2\Delta T}{T_i^2 \ell} \right)^2 \qquad b^2 = \left(-\frac{2\Delta T}{T_i^2 \ell} \right)^2 . \tag{3.3}$$

Here $\ell = 2 * N$ where N is the lattice size.

Figure 3.3 shows the results of this calculation for three lattice sizes. This particular graph matches rather closely the result of the same calculation in [32].

After we figured out the XY model, we moved onto running simulations of the QCD adjoint model.

3.3 QCD (Adj) Model Code

The Hamiltonian for the QCD (adj) model is:

$$H = -J \sum_{\langle i,j \rangle} \cos(\theta_i - \theta_j) + \tilde{y} \sum \cos(4\theta_i), \tag{3.4}$$

where the cosine function is summed over the nearest neighbor. For the purposes of the simulations, $\tilde{y}, J = 1$. The new cosine term and $\tilde{y}$ are the terms encoded with the QCD information. J contains the temperature information. It is important to

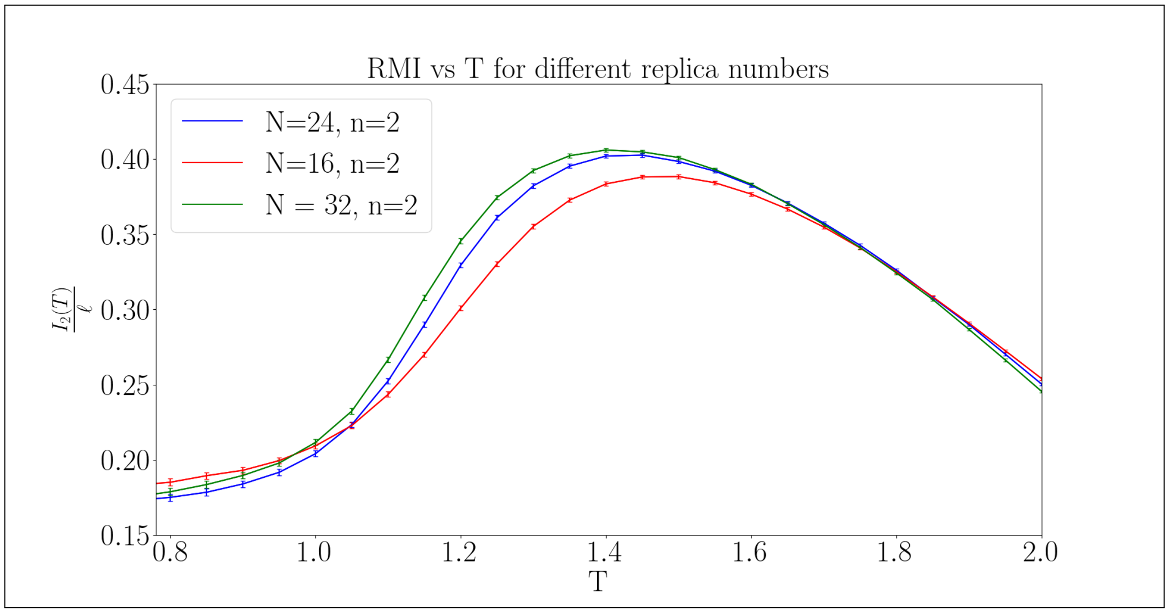


Figure 3.3: RMI for three different lattice sizes.This data took 20000 independent measurements.

note that since we are now talking about QCD, the thermodynamics are inverted. The boltzmann probability takes the form,

$$P(E_i) = \frac{e^{T*E_i}}{Z}. \tag{3.5}$$

This model was the main model we investigated in the paper, because it is linked to the strong force through the compactification duality. The Hamiltonian of this QCD (adj) model is equation (3.4). We tested what happened to the RMI with different values of $\tilde{y}$ and changing the 4 in $\tilde{y}\sum\cos(4\theta)$, to different values as well. The latter coefficient is henceforth referred to as p. When $p = 4$, we have the most direct link to the strong force, which is why it is important.

3.3.1 Layout

The layout of this code is virtually identical to the XY Model code described above: Three energy functions are distributed across CPUs for a particular temperature range and then the data is outputted to a data file. The difference is in the calculation of the energy for each energy function. There are two main differences, the addition of the $\tilde{y}$ term, and the acceptance ratio. The QCD thermodynamics change the acceptance ratio from

$$R = \exp(\frac{-\Delta \text{E}}{T})$$

to

$$R = \exp(-\Delta \text{E} * T).$$

In this code, in addition to the energy of the lattice, other quantities are measured to gain more understanding of the system.

3.3.2 Measured Quantities

The normal (un-replicated) energy function also took measurements of magnetization, heat capacity, and magnetic susceptibility. The most important of these quantities is susceptibility, because it is the closest thing the system has to an order parameter.

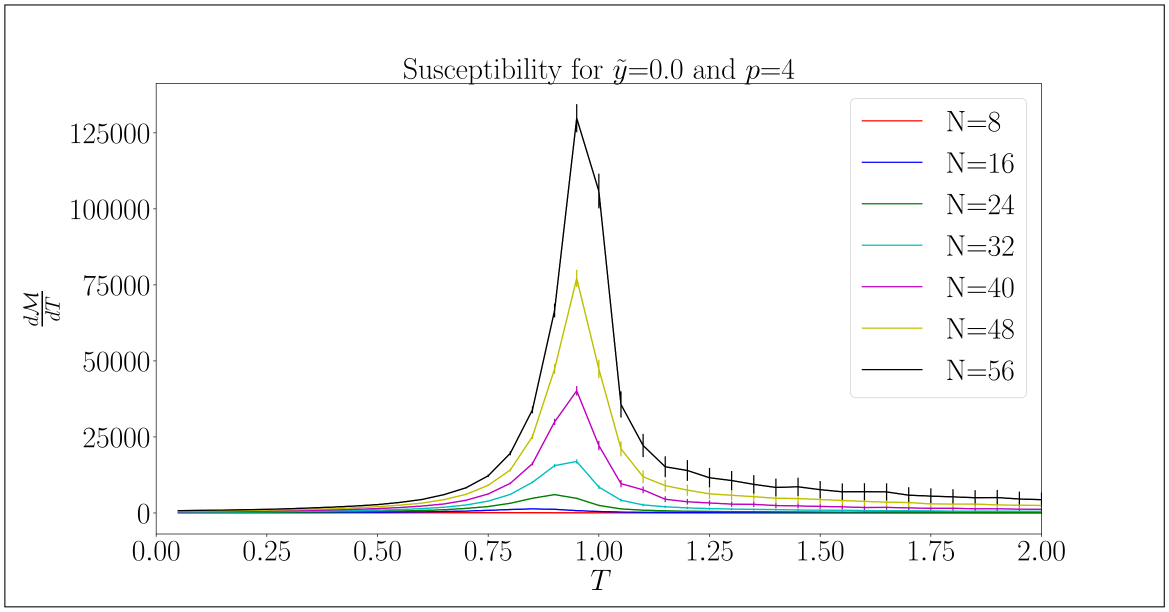


Figure 3.4: Susceptibility for QCD Model without the extra cosine term ($\tilde{y} = 0$)

Comparing figure 3.4 to figure 3.5 shows us there is a susceptibility peak at the second RMI crossing. This is important, because it is the second RMI crossing that predicts the phase transition of the QCD Model. The first crossing is an artifact of the replica trick, and doesn't actually tell us anything about the transition.

3.3.3 Calculating RMI

The Renyi Mutual Information for the QCD Model has different thermodynamics than the XY Model. This means the calculation of Renyi Entropy is different than the XY Model because it has inverted thermodynamics. Luckily for us, this meant we only needed to integrate from 0 to the desired T, instead of the desired T to T_{max}. This took a lower number of CPUs to get the data we needed, because we only needed to run simulations from $T = 0$ to $T = 4$ to get meaningful data. However, this inverted thermodynamics does have much longer equilibrium times. Overall, the QCD codes run faster than XY for the same RMI range. Here is the formula:

$$I_2(T) = \sum_{T_i=0}^{T} \Delta T \quad 2\langle E_i \rangle_2 - \langle E_i \rangle_{A\cup B} - 2\langle E_i \rangle_0. \tag{3.6}$$

Figure shows a calculation of RMI without the cosine term in (3.4)

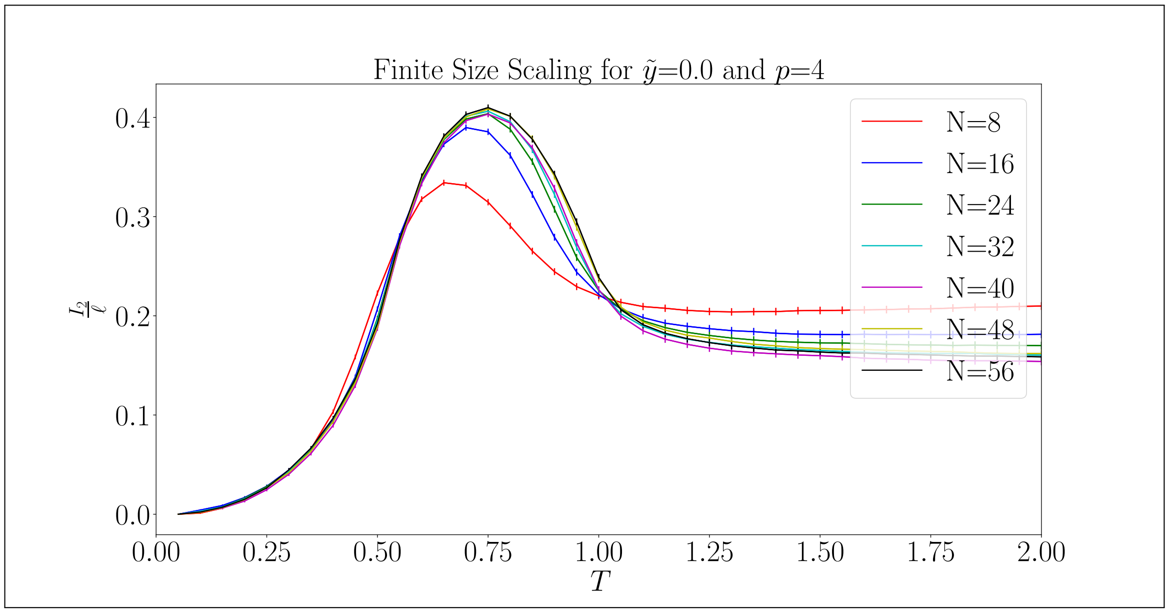


Figure 3.5: RMI for QCD with $\tilde{y} = 0$

Here are the results for different values of $\tilde{y}$. We can see that turning on the $\tilde{y}$ term causes some non trivial behavior.

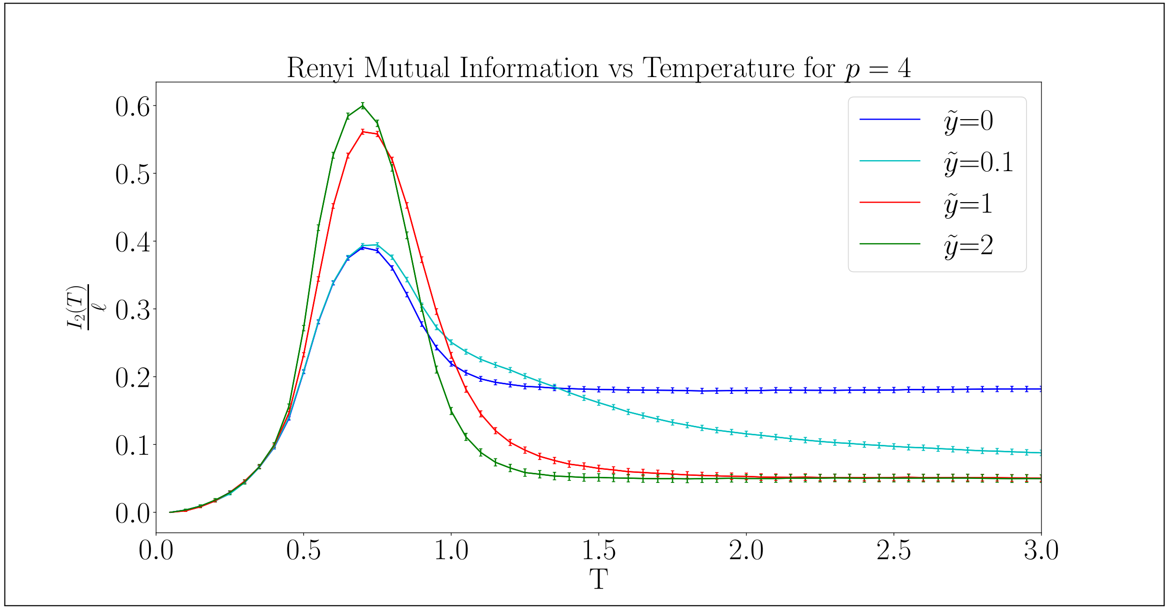


Figure 3.6: RMI for different values of $\tilde{y}$ and $p = 4$.

As you can see, ”turning on” the QCD term lowers the RMI after the crossing. This corresponds to the presence of strong force bosons. This next graph is the RMI for $\tilde{y} = 1$ and $p = 4$. This is the graph we were after this whole time:

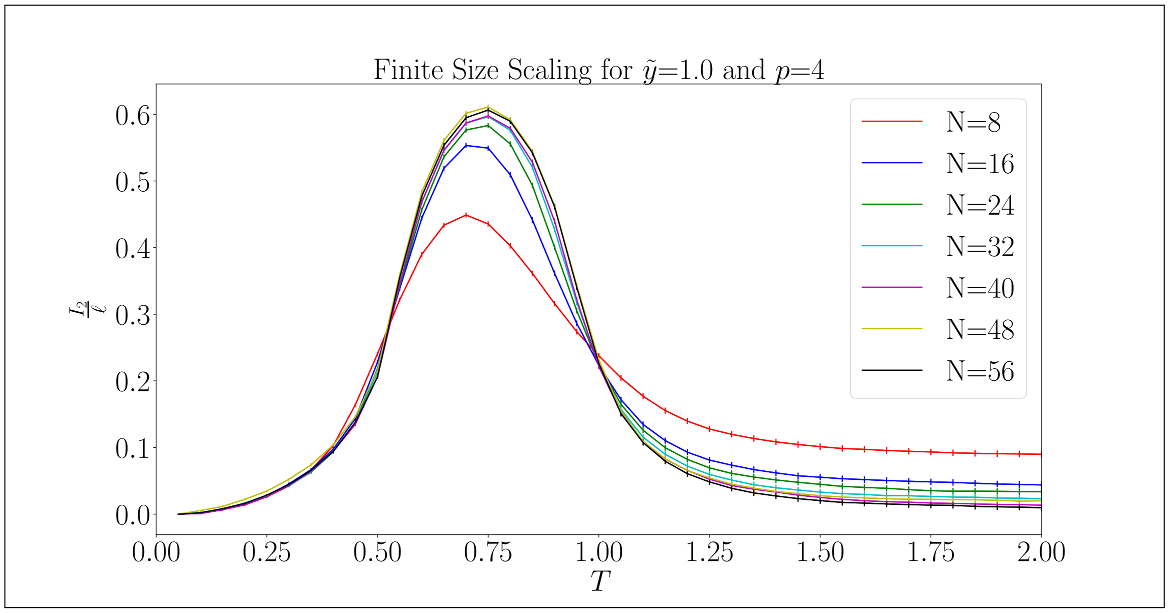


Figure 3.7: RMI for different lattice sizes with $\tilde{y} = 1$ and $p = 4$.

This graph took approximately 2400 computer cores, and 100000 CPU hours to complete. With the help of the Coeus cluster, this only took 3 days to complete. These results are discussed in more detail in section 5.

Chapter 4

Parallel Computing

During this research, we made heavy use of parallel computing architectures in order to hasten our data collection. The Coeus HPC cluster, operated by the Portland Institute of Computational Sciences was key to accomplishing this. As mentioned in the code description in section 3.1, the RMI simulation code has a ”multiprocessing” functionality that allows it to distribute itself across the cores of one computer. So it requires little effort to use all the cores of one machine, because the multiprocessing module does it for us. However, how would one make use of all the cores of multiple computers to further speed things up? This chapter aims to outline some of the logic behind parallel computing, using the Coeus cluster as an example.

4.1 Going Parallel

Let us proceed with an example. The typical parameters for our calculations involved running 20000 independent measurements, in a temperature range of $T = (0, 4)$ with a $\Delta T = 0.05$. If I wanted to calculate the energies for this temperature range then I have 79 total temperatures to complete (python doesn't include the end temperature when coding this range in). Keep in mind we must calculate the three energy types $(\langle E_0 \rangle, \langle E_2 \rangle, \langle E_{A \cup B} \rangle)$ for each of these temperatures, which makes the calculation take longer. If my computer has 28 available cores, then I will have to wait for 28 out of those 79 to complete, and then wait for the next 28, and then wait for the remaining 16 temperatures to finish. For 20000 independent measurements, this could take around a week depending on the lattice size. However, if I had 3 computers with 28 cores each, or 4 computers with 20 cores each, then I can run the same data for a third of the time. This is because each ”chunk” of temperatures will always take the same time to complete. For example, say one temperature takes 20 hours to complete. Then running one temperature on one core will take the same time as running 20 concurrent temperatures, since they are split up across multiple cores as in figure 4.1.

This means that if we can somehow run all the temperatures at the same time across multiple computers, the minimum time is just one temperature, instead of three as in the previous example! So where do we get the computers?

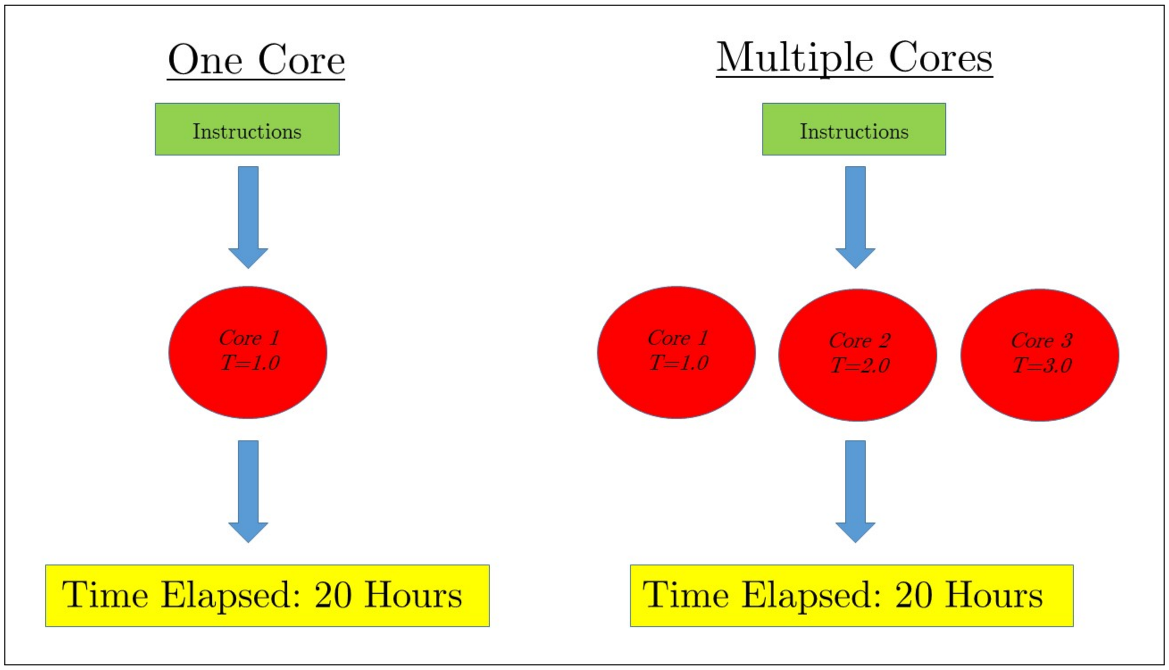


Figure 4.1: Running a temperature on one core takes the same time as running multiple temperatures concurrently on multiple cores

4.2 The Coeus Cluster

Enter the Coeus Cluster. A computer cluster is essentially a bunch of computers, or *nodes*, strung together to form a *super* computer. The cluster has a software controlling how each node talks to each other called a *scheduler*. To run a job, one needs to submit it to the scheduler. The scheduler then finds available nodes and distributes resources accordingly so that your job runs smoothly. Coeus uses a scheduler called SLURM, which will be further discussed below.

Most of the nodes on Coeus have 20 cores. Returning to our example, this means we can use a total of 4 nodes to complete all 79 of the temperatures, which would finish in the time that one temperature would take: 20 hours in this case. So we essentially increased productivity by 7900%, (in parallel programming circles this is called a 79X speedup).

So how does one actually tell the scheduler to run all these different temperatures on different nodes? With SLURM, there a few ways to do this, but we will only discuss two of them. The first is to use a language called MPI, or *Message Passing Interface*. This is a computer language built for unleashing the full power of supercomputers that is written in C++. However, for an individual who doesn't already know C this is a tall task to learn from the ground up. Especially for an individual writing a thesis and pressed for time to get the data needed for their research (that's me). So for our research we did not use MPI, but rather a "brute force method" involving SLURM and a lot of Python scripts.

4.3 Brute Force Parallel Computing

In order to run a job with SLURM, one must create a job submission (or *batch* script, and then submit it to the scheduler. A typical batch script looks like this:

```bash
#!/bin/bash
#SBATCH --job-name=N,n=32,2_0.0to2.0
#SBATCH --output=RMI_QCD0.0to2.0.txt
#SBATCH --nodes 1
#SBATCH --cpus-per-task=20
#SBATCH --ntasks-per-node 1
#SBATCH --time=4-00:00:00
#SBATCH --mem-per-cpu=MaxMemPerCPU
#SBATCH --partition medium
module load Python/python3.6.1
srun python3.6 RMI_QCD_Cluster.py 0.0 2.0
```

If the above's filename was "`QCD1_submit.sh`" then one would then submit this in a bash shell like so:

```
sh QCD1_submit.sh
```

This batch script would submit one job to the cluster. However, we can't submit a job that would run more than 20 temperatures, or else those extra temperatures would double the time it takes to run the whole simulation. We can submit the first 20 temperatures, and then the next 20 as seperate jobs through seperate batch scripts. The only downside to this is that in order to calculate RMI, we need all the energies in one data file, because $I_2(T)$ is calculated through integrating from T_min to T. So after running all the separate jobs, we need to bring all the data back into one file to process them. What we will do is this:

1. Submit the first 20 temperatures to the cluster in a first batch file
2. Submit the second 20 temperatures to the cluster in a second batch file
3. Submit the third 20 temperatures...and so on until we have submitted enough jobs so that all 79 temperatures are running concurrently.
4. Aggregate all the result files from these various runs into one data file
5. Perform desired RMI calculations

Of course, this method requires writing a bunch of batch scripts, and then submitting them every time one wants to run a simulation. This can get labor intensive, so I wrote additional codes to both generate the batch scripts (in appendix C), and a code to aggregate all the different data files (appendix C.3). It is indeed a lot of work to properly distribute all the temperatures to the cluster, and that's why I have dubbed this the "Brute Force method". This process is how most of the simulations were run. I also wrote a code entitled "Control Center.py" that allowed me to submit 7 lattice sizes, with 4 different parameters each, a total of 28 simulations (and 112 jobs), very easily, which is included in appendix C.1.

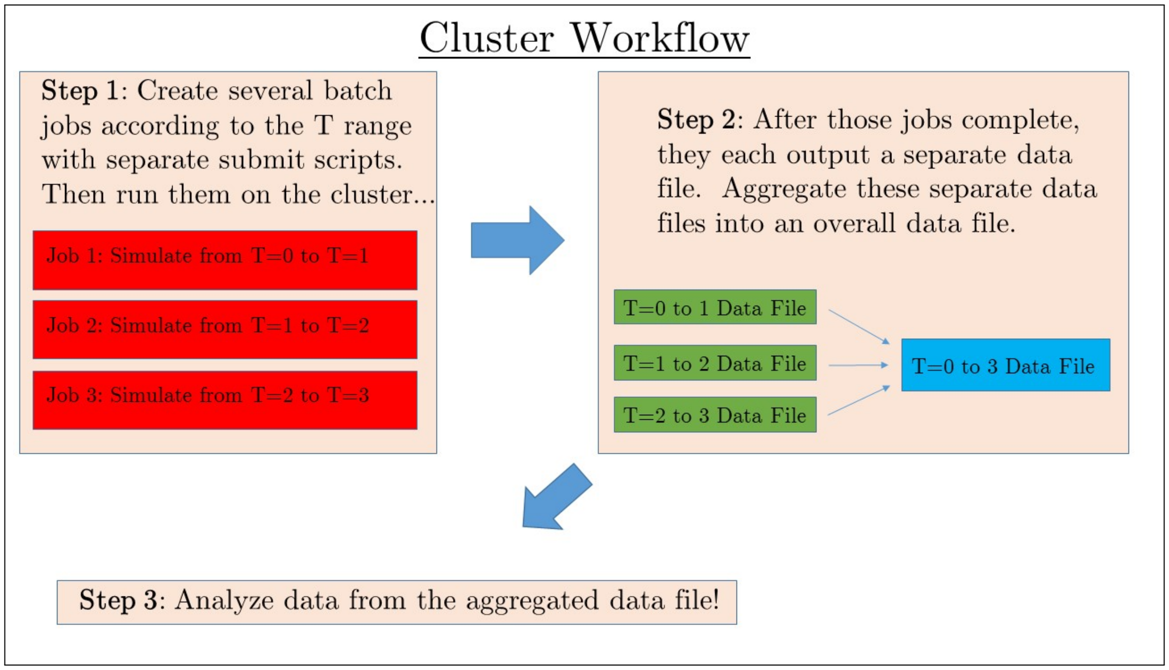

Cluster Workflow
Step 1: Create several batch jobs according to the T range with separate submit scripts. Then run them on the cluster...
Job 1: Simulate from T=0 to T=1
Job 2: Simulate from T=1 to T=2
Job 3: Simulate from T=2 to T=3
Step 2: After those jobs complete, they each output a separate data file. Aggregate these separate data files into an overall data file.
T=0 to 1 Data File
T=1 to 2 Data File
T=2 to 3 Data File
T=0 to 3 Data File
Step 3: Analyze data from the aggregated data file!

Figure 4.2: The various tasks needed to complete a *full* simulation job on the cluster

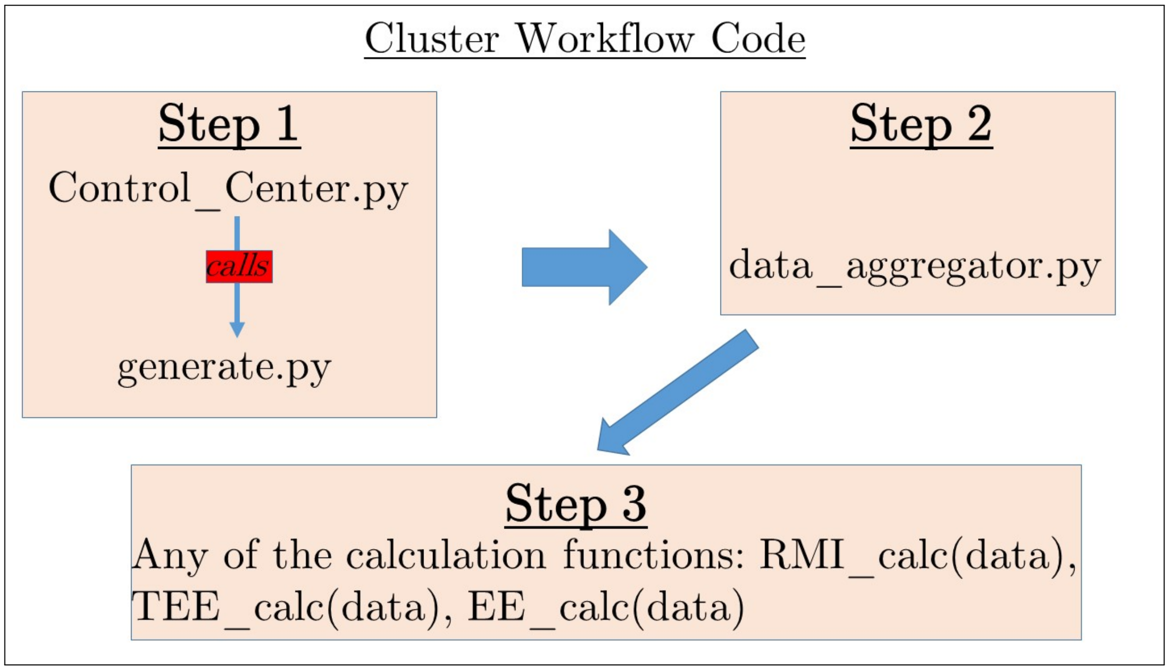


Figure 4.3: The various codes used to complete the tasks needed to complete a *full* simulation job on the cluster. This code is all included in the appendices.

Chapter 5

Results, Analysis, and Summary

5.1 Renyi Mutual Information and the Strong Force

To review, we ran Monte Carlo simulations of the XY model with the Hamiltonian

$$H = -J\sum_{\langle i,j\rangle}\cos(\theta_i - \theta_j) + \tilde{y}\sum\cos(p\theta_i) \tag{5.1}$$

over different temperatures, and measured the energy at each temperature. Then we calculated the Renyi Mutual Information at each temperature, and graphed different lattice sizes together. We did this for different p values, the results of which are below.

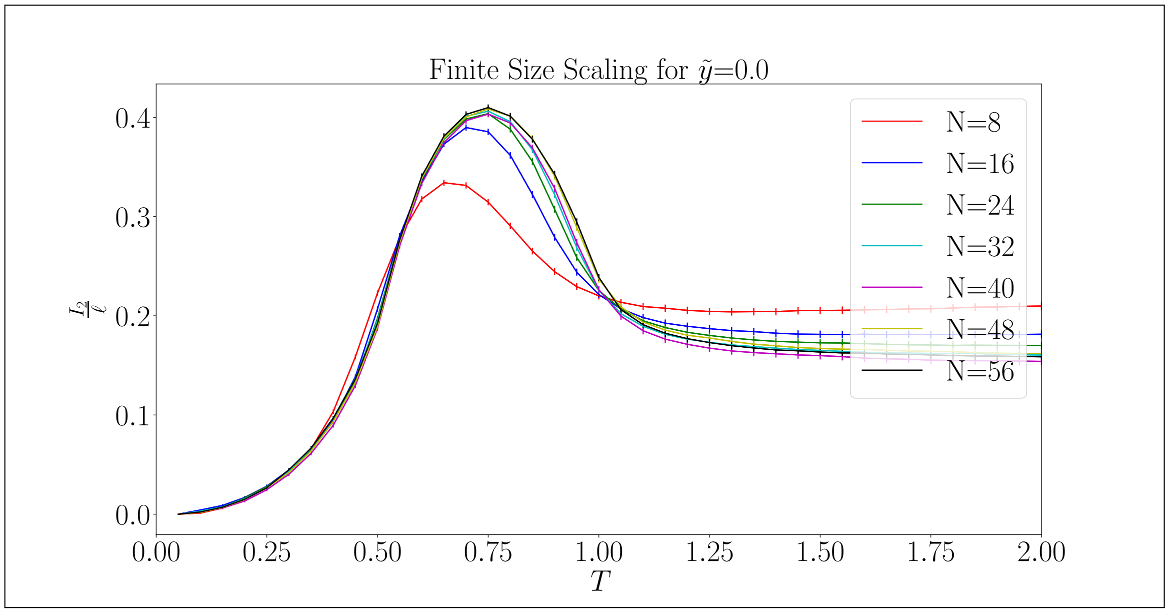


Figure 5.1: RMI vs T with $p = 0$.

This is the RMI with $\tilde{y} = 0$. This has essentially "turned off" the QCD coupled term and removed the strong force from the system, except for the inverted thermodynamics. This system does not have a thermodynamic order parameter, and the RMI measurement shows a crossing at the temperature $T \approx 0.99$ So the RMI has detected a phase transition despite there being no order parameter.

Next, we tested $\tilde{y} = 1.0$, with $p = 1$.

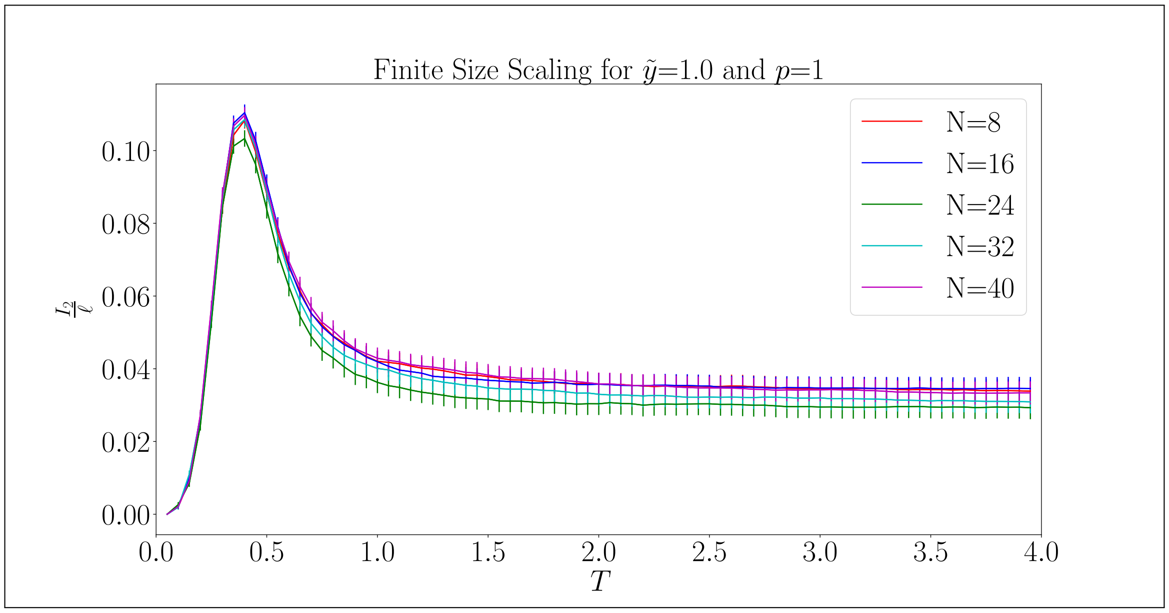


Figure 5.2: RMI of the lattices $N = 8$ to $N = 56$ with $p = 1$.

This situation has no order parameter, and the RMI detects this. The RMI shows no crossing, and the Susceptibility shows no discontinuity, proving there is no thermodynamic order parameter.

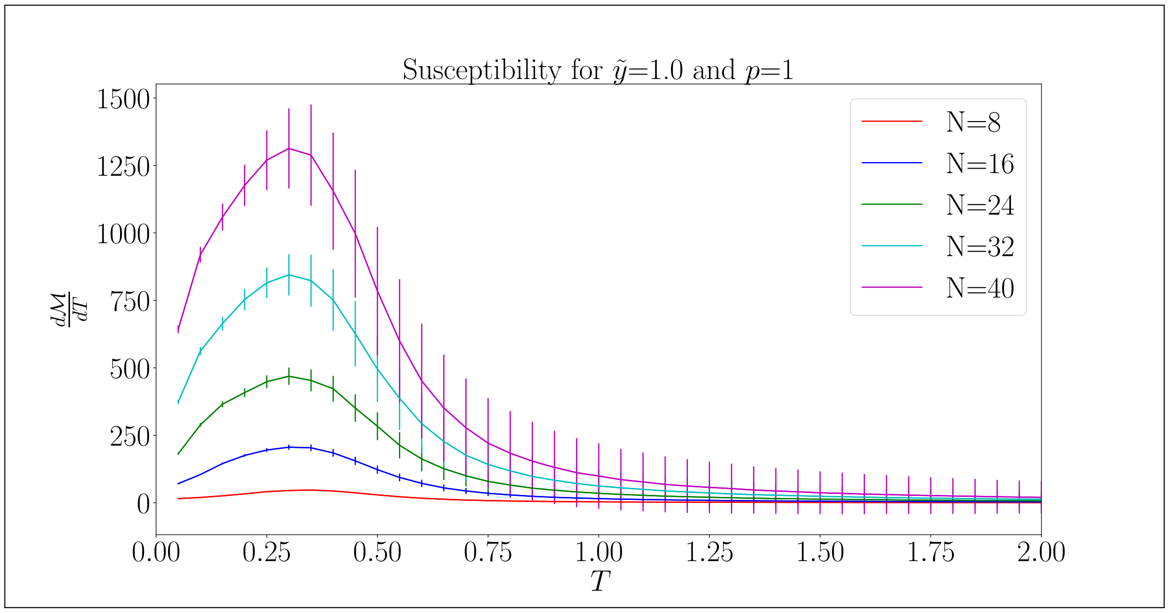


Figure 5.3: Susceptibility of the lattices $N = 8$ to $N = 56$ with $p = 1$.

This means that the deconfinement happens gradually. Now it makes sense to talk about this in the context of deconfinement. The lack of a crossing means the confinement transition will happen gradually, instead of all at once.

This can be seen in the $p = 4$ case as well. Recall this is the situation with benign quarks.

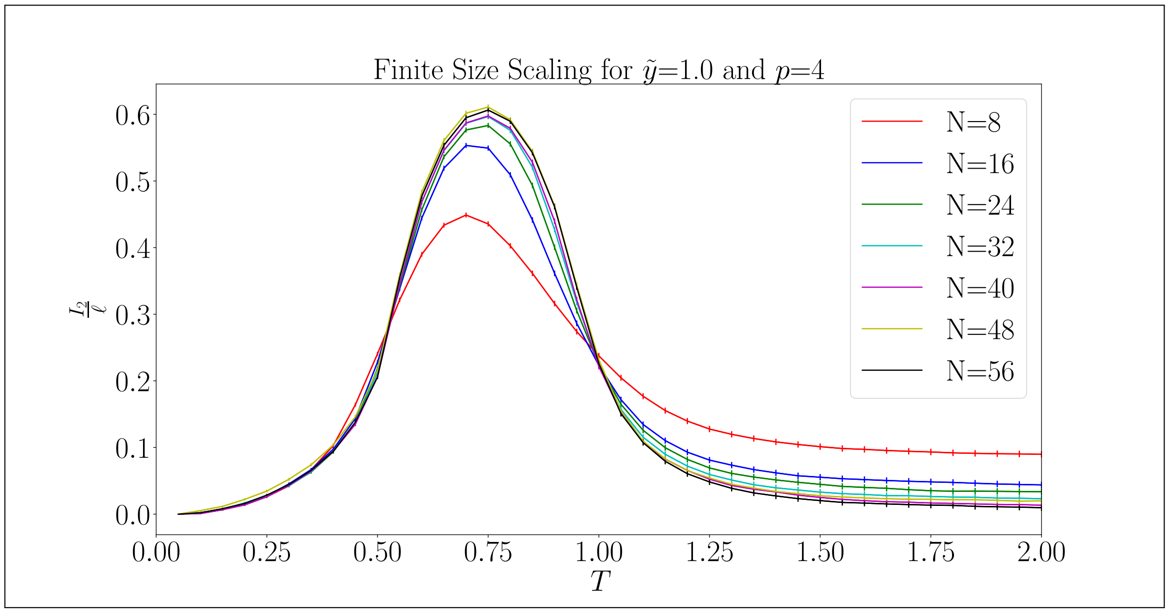


Figure 5.4: RMI of the lattices $N = 8$ to $N = 56$ with $p = 4$.

As we can see there is a very clear crossing around the transition temperature, $T \approx 0.99$. Thus we can conclude there is a phase transition, and that it happens abruptly, as opposed to gradually.

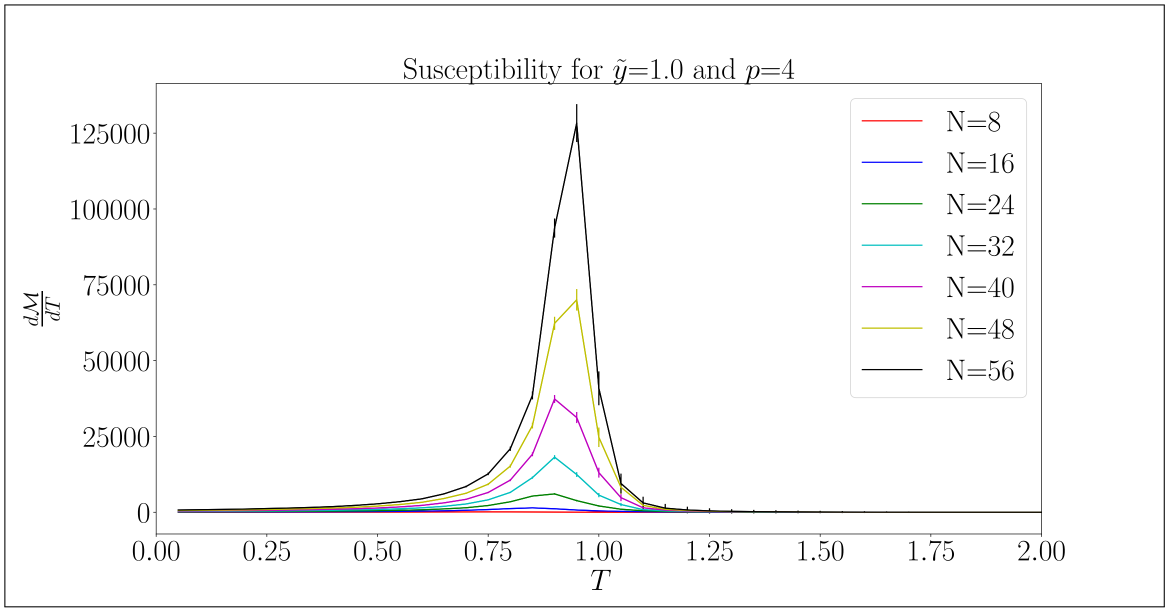


Figure 5.5: Susceptibility of the lattices $N = 8$ to $N = 56$ with $p = 4$.

If we compare RMI to the thermodynamic order parameter susceptibility, we can see that RMI exhibits the crossing at the same point as the susceptibility discontinuity. The order parameter here shows a discontinuity around the same temperature (the transition temperature) $T \approx 0.99$. As $N \to \infty$, this discontinuity would become asymptotic, signifying a phase transition. This means that the phase transition happens suddenly, due to the discontinuous nature, and the RMI registers this as a crossing of different lattice sizes. To learn about our system using RMI, we have to simulate multiple different lattice sizes.

Thus we conclude that Renyi Mutual Information can act as a universal order parameter, and can provide insight into the behavior of the nature of the deconfinement phase transition. In addition the methods I used to calculate the RMI and simulate the XY model are scalable, and available for use by other interested parties. The codes are included in the appendices for your reference. The “Control Center” package can be loaded on to any cluster with the SLURM scheduler, and then you can run these simulations on a massively parallel architecture. Please refer to the manual included in the appendices for user manuals on each code.

Chapter 6

Future Directions

This research was a very good introduction to understanding Renyi Mutual Information, and there are many more things we can do in the future. We can run simulations with higher lattice numbers, with 3,4, or 5 replica lattices and then examine the behavior of the system.

A more thorough investigation into a quantity called topological entanglement entropy would be interesting as well. This quantity might offer some more insight into the phase transition.

On the code side of things, there are a few next steps that would be helpful. I'd like to eventually build a python package full of all the functions and scripts I created, so that they can be used by others and improved upon. I have a first draft of a function module called "entanglement" for some calculation functions.

I'd also like to create Control Center codes (appendix C.1) compatible with other cluster scheduler software besides SLURM.

Bibliography

[1] Cesar A. Agon, Matthew Headrick, Daniel L. Jafferis, and Skyler Kasko. Disk entanglement entropy for a Maxwell field. *Phys. Rev.*, D89(2):025018, 2014.

[2] Mohamed M. Anber. The abelian confinement mechanism revisited: new aspects of the Georgi-Glashow model. *Annals Phys.*, 341:21–55, 2014.

[3] Mohamed M. Anber, Scott Collier, Erich Poppitz, Seth Strimas-Mackey, and Brett Teeple. Deconfinement in $\mathcal{N} = 1$ super Yang-Mills theory on $\mathbb{R}^3 \times \mathbb{S}^1$ via dual-Coulomb gas and "affine" XY-model. *JHEP*, 11:142, 2013.

[4] Mohamed M. Anber and Erich Poppitz. Microscopic Structure of Magnetic Bions. *JHEP*, 06:136, 2011.

[5] Mohamed M. Anber and Erich Poppitz. New nonperturbative scales and glueballs in confining supersymmetric gauge theories. 2017.

[6] Mohamed M. Anber, Erich Poppitz, and Brett Teeple. Deconfinement and continuity between thermal and (super) Yang-Mills theory for all gauge groups. *JHEP*, 09:040, 2014.

[7] Mohamed M. Anber, Erich Poppitz, and Mithat Unsal. 2d affine XY-spin model/4d gauge theory duality and deconfinement. *JHEP*, 04:040, 2012.

[8] Sinya Aoki, Takumi Iritani, Masahiro Nozaki, Tokiro Numasawa, Noburo Shiba, and Hal Tasaki. On the definition of entanglement entropy in lattice gauge theories. *JHEP*, 06:187, 2015.

[9] Ibrahima Bah, Alberto Faraggi, Leopoldo A. Pando Zayas, and Cesar A. Terrero-Escalante. Holographic entanglement entropy and phase transitions at finite temperature. *Int. J. Mod. Phys.*, A24:2703–2728, 2009.

[10] Pinaki Banerjee, Atanu Bhatta, and B. Sathiapalan. Sine-Gordon Theory : Entanglement entropy and holography. *Phys. Rev.*, D96(12):126014, 2017.

[11] James M. Bardeen, B. Carter, and S. W. Hawking. The Four laws of black hole mechanics. *Commun. Math. Phys.*, 31:161–170, 1973.

[12] Jacob D. Bekenstein. Black holes and entropy. *Phys. Rev.*, D7:2333–2346, 1973.

[13] V. L. Berezinsky. Destruction of long range order in one-dimensional and two-dimensional systems having a continuous symmetry group. 1. Classical systems. *Sov. Phys. JETP*, 32:493–500, 1971. [Zh. Eksp. Teor. Fiz.59,907(1971)].

[14] D. Boyanovsky and R. Holman. Critical behavior and duality in extended Sine-Gordon theories. *Nucl. Phys.*, B358:619–653, 1991.

[15] Daniel Boyanovsky. Field Theoretical Renormalization and Fixed Point Structure of a Generalized Coulomb Gas. *J. Phys.*, A22:2601–2614, 1989.

[16] P. V. Buividovich and M. I. Polikarpov. Numerical study of entanglement entropy in SU(2) lattice gauge theory. *Nucl. Phys.*, B802:458–474, 2008.

[17] Curtis G. Callan, Jr. and Frank Wilczek. On geometric entropy. *Phys. Lett.*, B333:55–61, 1994.

[18] Constantine Callias. Index Theorems on Open Spaces. *Commun. Math. Phys.*, 62:213–234, 1978.

[19] H. Casini and M. Huerta. Entanglement entropy in free quantum field theory. *J. Phys.*, A42:504007, 2009.

[20] Horacio Casini, Marina Huerta, and Jose Alejandro Rosabal. Remarks on entanglement entropy for gauge fields. *Phys. Rev.*, D89(8):085012, 2014.

[21] Csaba Csaki, Yuri Shirman, John Terning, and Michael Waterbury. Twisted Sisters: KK Monopoles and their Zero Modes. 2017.

[22] N. Michael Davies, Timothy J. Hollowood, and Valentin V. Khoze. Monopoles, affine algebras and the gluino condensate. *J. Math. Phys.*, 44:3640–3656, 2003.

[23] William Donnelly, Ben Michel, and Aron Wall. Electromagnetic Duality and Entanglement Anomalies. *Phys. Rev.*, D96(4):045008, 2017.

[24] William Donnelly and Aron C. Wall. Do gauge fields really contribute negatively to black hole entropy? *Phys. Rev.*, D86:064042, 2012.

[25] Gerald V. Dunne, Ian I. Kogan, Alex Kovner, and Bayram Tekin. Deconfining phase transition in (2+1)-dimensions: The Georgi-Glashow model. *JHEP*, 01:032, 2001.

[26] Christopher Eling, Yaron Oz, and Stefan Theisen. Entanglement and Thermal Entropy of Gauge Fields. *JHEP*, 11:019, 2013.

[27] Mitsutoshi Fujita, Tatsuma Nishioka, and Tadashi Takayanagi. Geometric Entropy and Hagedorn/Deconfinement Transition. *JHEP*, 09:016, 2008.

[28] Sudip Ghosh, Ronak M Soni, and Sandip P. Trivedi. On The Entanglement Entropy For Gauge Theories. *JHEP*, 09:069, 2015.

[29] David J. Gross, Robert D. Pisarski, and Laurence G. Yaffe. QCD and Instantons at Finite Temperature. *Rev. Mod. Phys.*, 53:43, 1981.

[30] Martin Hasenbusch. The two-dimensional xy model at the transition temperature: a high-precision monte carlo study. *Journal of Physics A: Mathematical and General*, 38(26):5869, 2005.

[31] Johannes Helmes, Jean-Marie Stéphan, and Simon Trebst. Rényi entropy perspective on topological order in classical toric code models. *Phys. Rev. B*, 92:125144, Sep 2015.

[32] Jason Iaconis, Stephen Inglis, Ann B. Kallin, and Roger G. Melko. Detecting classical phase transitions with renyi mutual information. *Phys. Rev. B*, 87:195134, May 2013.

[33] R. Jackiw and C. Rebbi. Solitons with Fermion Number 1/2. *Phys. Rev.*, D13:3398–3409, 1976.

[34] Jorge V. Jose, Leo P. Kadanoff, Scott Kirkpatrick, and David R. Nelson. Renormalization, vortices, and symmetry breaking perturbations on the two-dimensional planar model. *Phys. Rev.*, B16:1217–1241, 1977.

[35] Daniel N. Kabat. Black hole entropy and entropy of entanglement. *Nucl. Phys.*, B453:281–299, 1995.

[36] L. P. Kadanoff. Lattice Coulomb Gas Representations of Two-Dimensional Problems. *J. Phys.*, A11:1399–1417, 1978.

[37] Alexei Kitaev and John Preskill. Topological entanglement entropy. *Phys. Rev. Lett.*, 96:110404, Mar 2006.

[38] Igor R. Klebanov, David Kutasov, and Arvind Murugan. Entanglement as a probe of confinement. *Nucl. Phys.*, B796:274–293, 2008.

[39] Ian I. Kogan and Alex Kovner. Monopoles, vortices and strings: Confinement and deconfinement in (2+1)-dimensions at weak coupling. 2002.

[40] Zohar Komargodski, Tin Sulejmanpasic, and Mithat Ünsal. Walls, anomalies, and deconfinement in quantum antiferromagnets. *Phys. Rev.*, B97(5):054418, 2018.

[41] J. M. Kosterlitz and D. J. Thouless. Ordering, metastability and phase transitions in two-dimensional systems. *J. Phys.*, C6:1181–1203, 1973.

[42] Yuri V. Kovchegov and D. T. Son. Critical temperature of the deconfining phase transition in (2+1)-d Georgi-Glashow model. *JHEP*, 01:050, 2003.

[43] Thomas C. Kraan and Pierre van Baal. Monopole constituents inside SU(n) calorons. *Phys. Lett.*, B435:389–395, 1998.

[44] Nicolas Laflorencie. Quantum entanglement in condensed matter systems. *Phys. Rept.*, 646:1–59, 2016.

[45] P. Lecheminant, Alexander O. Gogolin, and Alesander A. Nersesyan. Criticality in selfdual sine-Gordon models. *Nucl. Phys.*, B639:502–523, 2002.

[46] Ki-Myeong Lee and Piljin Yi. Monopoles and instantons on partially compactified D-branes. *Phys. Rev.*, D56:3711–3717, 1997.

[47] Michael Levin and Xiao-Gang Wen. Detecting topological order in a ground state wave function. *Phys. Rev. Lett.*, 96:110405, Mar 2006.

[48] Jinfeng Liao and Edward Shuryak. Strongly coupled plasma with electric and magnetic charges. *Phys. Rev.*, C75:054907, 2007.

[49] Max A. Metlitski and Tarun Grover. Entanglement Entropy of Systems with Spontaneously Broken Continuous Symmetry. 2011.

[50] Y. Nakagawa, A. Nakamura, S. Motoki, and V. I. Zakharov. Entanglement entropy of SU(3) Yang-Mills theory. *PoS*, LAT2009:188, 2009.

[51] David R. Nelson and J. M. Kosterlitz. Universal Jump in the Superfluid Density of Two-Dimensional Superfluids. *Phys. Rev. Lett.*, 39:1201–1205, 1977.

[52] M. E. J. Newman and G. T. Barkema. *Monte Carlo Methods in Statistical Physics.* Clarendon Press, 2011.

[53] Mark E. J. Newman. *Computational Physics.* Createspace, 2013.

[54] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press, New York, NY, USA, 10th edition, 2011.

[55] Tatsuma Nishioka and Tadashi Takayanagi. AdS Bubbles, Entropy and Closed String Tachyons. *JHEP*, 01:090, 2007.

[56] Tom M. W. Nye and Michael A. Singer. An L**2 index theorem for Dirac operators on S**1 x R**3. *Submitted to: J. Funct. Anal.*, 2000.

[57] Alexander M. Polyakov. Quark Confinement and Topology of Gauge Groups. *Nucl. Phys.*, B120:429–458, 1977.

[58] Erich Poppitz and M. Erfan Shalchian T. String tensions in deformed Yang-Mills theory. *JHEP*, 01:029, 2018.

[59] Erich Poppitz and Mithat Unsal. Index theorem for topological excitations on R**3 x S**1 and Chern-Simons theory. *JHEP*, 03:027, 2009.

[60] Djordje Radicevic. Notes on Entanglement in Abelian Gauge Theories. 2014.

[61] E. Rastelli, S. Regina, and A. Tassi. Monte carlo simulation of a planar rotator model with symmetry-breaking fields. *Phys. Rev. B*, 69:174407, May 2004.

[62] Shinsei Ryu and Tadashi Takayanagi. Holographic derivation of entanglement entropy from AdS/CFT. *Phys. Rev. Lett.*, 96:181602, 2006.

[63] Hiroyuki Shimizu and Kazuya Yonekura. Anomaly constraints on deconfinement and chiral phase transition. 2017.

[64] Dusan Simic and Mithat Unsal. Deconfinement in Yang-Mills theory through toroidal compactification with deformation. *Phys. Rev.*, D85:105027, 2012.

[65] Sergey N. Solodukhin. Remarks on effective action and entanglement entropy of Maxwell field in generic gauge. *JHEP*, 12:036, 2012.

[66] Benjamin Svetitsky and Laurence G. Yaffe. Critical Behavior at Finite Temperature Confinement Transitions. *Nucl. Phys.*, B210:423–447, 1982.

[67] Gerard 't Hooft. On the Quantum Structure of a Black Hole. *Nucl. Phys.*, B256:727–745, 1985.

[68] Mithat Unsal. Magnetic bion condensation: A New mechanism of confinement and mass gap in four dimensions. *Phys. Rev.*, D80:065001, 2009.

[69] Mithat Unsal and Laurence G. Yaffe. Center-stabilized Yang-Mills theory: Confinement and large N volume independence. *Phys. Rev.*, D78:065035, 2008.

[70] Karel Van Acoleyen, Nick Bultinck, Jutho Haegeman, Michael Marien, Volkher B. Scholz, and Frank Verstraete. The entanglement of distillation for gauge theories. *Phys. Rev. Lett.*, 117(13):131602, 2016.

[71] Alexander Velytsky. Entanglement entropy in d+1 SU(N) gauge theory. *Phys. Rev.*, D77:085021, 2008.

[72] G. Vidal, J. I. Latorre, E. Rico, and A. Kitaev. Entanglement in quantum critical phenomena. *Phys. Rev. Lett.*, 90:227902, Jun 2003.

[73] X. G. Wen. *Quantum field theory of many-body systems: From the origin of sound to an origin of light and electrons.* 2004.

[74] Michael M. Wolf, Frank Verstraete, Matthew B. Hastings, and J. Ignacio Cirac. Area laws in quantum systems: Mutual information and correlations. *Phys. Rev. Lett.*, 100:070502, Feb 2008.

[75] B. Zeng, X. Chen, D.-L. Zhou, and X.-G. Wen. Quantum Information Meets Quantum Matter – From Quantum Entanglement to Topological Phase in Many-Body Systems. *ArXiv e-prints*, August 2015.

[76] Ariel R. Zhitnitsky. Entropy, Contact Interaction with Horizon and Dark Energy. *Phys. Rev.*, D84:124008, 2011.

[77] Jean Zinn-Justin. Quantum field theory and critical phenomena. *Int. Ser. Monogr. Phys.*, 113:1–1054, 2002.

[78] J. B. Zuber and C. Itzykson. Quantum Field Theory and the Two-Dimensional Ising Model. *Phys. Rev.*, D15:2875, 1977.

Appendices

Appendix A

XY Model Code

This code collected data for the basic XY Spin model with Hamiltonian:

$$H = -J \sum_{\langle i,j \rangle} \cos(\theta_i - \theta_j).$$

The following sections are a manual for running this code.

A.1 Adjusting Parameters

To adjust the lattice size of the simulation, enter a value in the variable `N_global` (line 9). The only lattice sizes this code will run for are 16, 24, 32, 64. To add a new lattice size, you must complete an equilibrium test to determine the equilibrium time of the lattice size you are running. This process will be discussed in section A.2.

To adjust the number of independent measurements (see section 2.2.2 for explanation), change the variable `E_measurements` (line 23 in listing A.2). These are the only two parameters for this code.

A.2 Running

If the `N_global` you select doesn't have an associated `tau_global`, this means you need to run an equilibrium test. To do this, you need to modify the `XY_E(T)` function to output energy for each Monte Carlo step, and add each energy reading to an array. It is easiest to copy paste the `XY_E(T)` function to a new python script so we can graph the energy reading vs Monte Carlo steps. For your convenience I have included line 28 in listing A.2 to test the equilibrium time for each temperature. Once you have copied the function into a new python file make sure to include the `import` statements from lines 1-5. Then run the `XY_E(T)` function for various temperatures, and you will see the equilibrium time for each one (see section 2.2.1 for details).

To run this code for a specified temperature range, open the command line (if you're on a linux system and command prompt on windows), and navigate to the directory that has your script in it. Then enter the temperature range as command line arguments:

```
C:\[Path to Script]> python XY_RMI.py 0.0 2.0 0.05
```

Which would then run a simulation from $T = 0$ to $T = 2$ with temperature steps of 0.05; a total of 40 temperatures. While the simulation is running, you should see lines on the screen that say:

```
N=32; Normal XY-Model at T=0.05
N=32; Normal XY-Model at T=0.1
N=32; Normal XY-Model at T=0.15
N=32; Normal XY-Model at T=0.2
N=32; Normal XY-Model at T=0.25
...
```

If you were running this on the cluster, you would have to submit a batch script like the one in listing 4.3 in section 4.3.

Moreover, if you were running multiple jobs on the cluster in the manner outlined in section 4.3 then you would have to aggregate the multiple data output files together into one file in order to properly calculate RMI using the code in appendix C.3.

A.3 Data Analysis

This code will output data files in the same folder that this script is located. This code will output a data file that has several arrays in it. Each array will contain important data. This is the order of the arrays in the code:

1. Temperature plot
2. Energy of the replica lattice
3. Variance of the replica energy
4. Energy of the $\mathcal{A} \cup \mathcal{B}$ lattice
5. Variance of the $\mathcal{A} \cup \mathcal{B}$ energy
6. Energy of the normal lattice
7. Variance of the normal energy
8. Renyi Mutual Information for the simulation
9. Variance on the RMI for the simulation

To analyze any of this data, you can access each array by loading the array using the `loadtxt` function in Python like this:

Listing A.1: A data analysis code block that assigns each array of data to a variable name for later manipulation.

```python
Data = loadtxt(fname='simulation_output.txt')
T_plot = Data[0]
# Gathers the replica data
E_replica = Data[1]
sigma_replica = Data[2]

# Gathers A_U_B data
E_A_U_B = Data[3]
sigma_A_U_B = Data[4]

```

```
# Gathers the normal data
E_normal = Data[5]
sigma_normal = Data[6]

RMI = Data[7]
RMI_sigma = Data[8]
```

You can then calculate whatever quantity you want from these variables. By using the energy variables for example, you can calculate RMI again using the code in section E.1.1, or you could calculate the derivative of the RMI. You could also create graphs of the RMI, like the following for N=32 and 20K measurements.

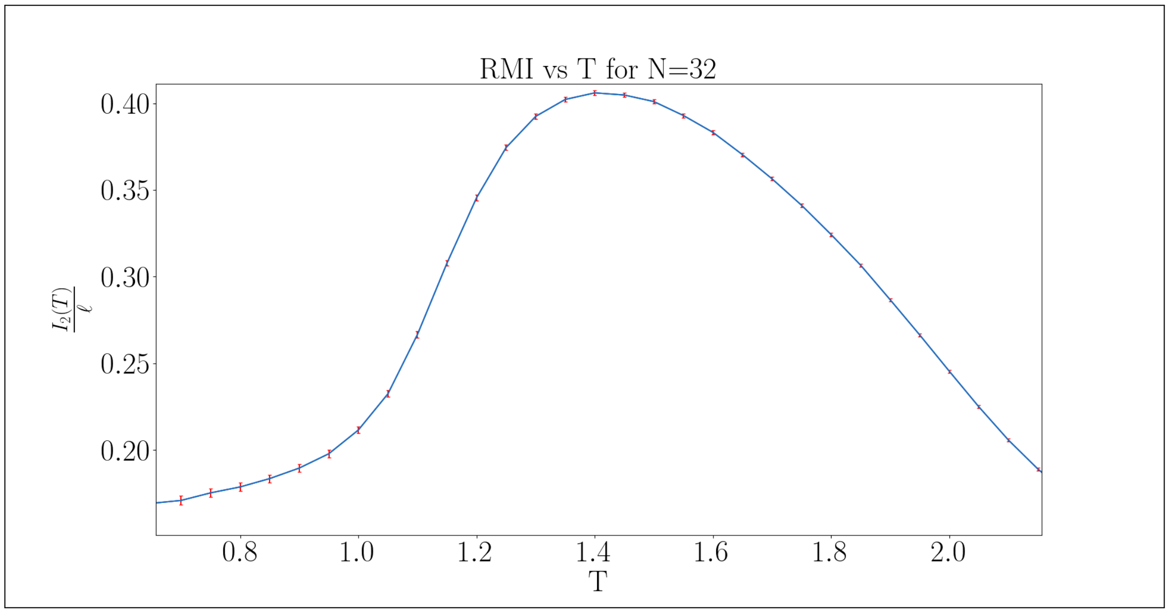


Figure A.1: RMI for N=32 and M=20000

Now here is the code:

Listing A.2: XY Spin Model Code

```
from numpy import ones, arange, sqrt, array, savetxt, vstack, zeros
from math import exp, pi, cos
from random import random, randrange
from multiprocessing import Pool
import time, sys, datetime

date = datetime.date.today()

N_global = 16
if N_global == 16:
    tau_global = 10240
    tau_after = 1000
if N_global == 24:
    tau_global = 14000
    tau_after = 1200
if N_global == 32:
```

```python
    tau_global = 21000
    tau_after = 4000
if N_global == 64:
    tau_global = 55000
    tau_after = 10000

E_measurements = 20000


# Normal XY
def XY_E(T):
        equilibrium_test = 'no' # change this to yes to produce a graph of energy vs MC steps.
    global N_global, E_measurements, tau_after
    J = 1
    N = N_global # The lattice size: NxN
    tau = tau_global # The equilibration time
    if T > 20:
        tau = tau_after
    BM = E_measurements # Number of independent measurements for the bootstrap analysis
    steps = 2 * tau * BM # Number of times the program will run
    E = -2 * (N * N) # Initial Value of Energy since all spins start pointed up at \theta_i = 0.0
    L = zeros([N, N], float) # Generates the lattice where each entry is a value of \theta_i

    print("N=", N, "; Normal XY-Model at T=", T)
    if equilibrium_test == 'yes':
        E_plot = [] # this line is for equilibrium tests. Make sure to  comment out if running a simulation
    expE = 0.0 # Expectation value of E
    measurements = [] # List of Measurements
    # Main Monte Carlo cycle
    for x in range(steps + 1):
        i = randrange(0, N)
        j = randrange(0, N) # Picks a random starting location

        # Decides an anticipated spin amount
        L_update = random() * 2 * pi
        # Calculates change in energy that would occur if this spin was accepted
        dE = 0.0
        # Starts calculating the nearest neighbor sum at location L[ i -1 , j]
        neighbor = i - 1
        if neighbor > -1:
            dE += cos(L_update - L[neighbor, j]) - cos(L[i, j] - L[neighbor, j]) # Checks if the neighbor is within the
```

```
                ↪ lattice
        else:
            dE += cos(L_update - L[N - 1, j]) - cos(L[i, j] - L[N - 1, j]) # Periodic boundary conditions
        neighbor = i + 1
        if neighbor < N:
            dE += cos(L_update - L[neighbor, j]) - cos(L[i, j] - L[neighbor, j])
        else:
            dE += cos(L_update - L[0, j]) - cos(L[i, j] - L[0, j])
        neighbor = j - 1
        if neighbor > -1:
            dE += cos(L_update - L[i, neighbor]) - cos(L[i, j] - L[i, neighbor])
        else:
            dE += cos(L_update - L[i, N - 1]) - cos(L[i, j] - L[i, N - 1])
        neighbor = j + 1
        if neighbor < N:
            dE += cos(L_update - L[i, neighbor]) - cos(L[i, j] - L[i, neighbor])
        else:
            dE += cos(L_update - L[i, 0]) - cos(L[i, j] - L[i, 0])
        dE *= -J
        # Calculates whether L[i,j] rotates
        R = exp(-dE / T)
        if R > 1 or random() < R:
            L[i, j] = L_update
            E += dE # / (N * N)

        if equilibrium_test == 'yes':
                E_plot.append(E)
        if x != 0 and x % (2 * tau) == 0: # If the program didn't just start, and we are a correlation time since the last measurement, take a measurement
            expE += E
            measurements.append(E) # Adds the measurement to the list
    expE /= BM

    # The Bootstrap Error Analysis
    resample = BM # times to repeat re-sampling
    B_i = [] # for the calculation of <B> and sigma
    for y in range(resample):
        B = 0.0
        for z in range(int(BM)):
            n = randrange(0, BM)
            B += measurements[n]
        B /= BM
        B_i.append(B)

    # Now to calculate the Bootstrap sigma
    sigma_sigma = 0.0
    for w in range(resample):
        sigma_sigma += (B_i[w] - expE) ** 2
    sigma_sigma /= resample
    sigma_bootstrap = sqrt(sigma_sigma)

    if equilibrium_test == 'yes':
        plot(E_plot)
        show()
    return [T, expE, sigma_bootstrap] # This will create a results
        ↪ matrix which can be plotted


def XY_A_U_B(T):
    global N_global, E_measurements, tau_after
    J = 1
    N = N_global # The lattice size: NxN
    tau = tau_global # The equilibration time
    if T > 20:
        tau = tau_after
    BM = E_measurements # Number of independent measurements for the
        ↪ bootstrap analysis
    steps = 2 * tau * BM # Number of times the program will run
    E = -4 * (N * N) # Initial Value of Energy since all spins start
        ↪ pointed up at \theta_i = 0.0
    L = zeros([N, N], float) # Generates the lattice where each entry
        ↪ is a value of \theta_i

    print("N=", N, "; A-union-B XY-Model at T=", T)

    expE = 0.0 # Expectation value of E
    measurements = [] # List of Measurements
    # Main Monte Carlo cycle
    for x in range(steps + 1):
        i = randrange(0, N)
        j = randrange(0, N) # Picks a random starting location

        # Decides an anticipated spin amount
        L_update = random() * 2 * pi
        # Calculates change in energy that would occur if this spin
            ↪ was accepted
        dE = 0.0
        # Starts calculating the nearest neighbor sum at location L[ i
            ↪ -1 , j]
        neighbor = i - 1
        if neighbor > -1:
            dE += cos(L_update - L[neighbor, j]) - cos(
                L[i, j] - L[neighbor, j]) # Checks if the neighbor is within the lattice
        else:
            dE += cos(L_update - L[N - 1, j]) - cos(L[i, j] - L[N - 1, j]) # Periodic boundary conditions
        neighbor = i + 1
        if neighbor < N:
            dE += cos(L_update - L[neighbor, j]) - cos(L[i, j] - L[neighbor, j])
        else:
            dE += cos(L_update - L[0, j]) - cos(L[i, j] - L[0, j])
        neighbor = j - 1
        if neighbor > -1:
            dE += cos(L_update - L[i, neighbor]) - cos(L[i, j] - L[i, neighbor])
        else:
            dE += cos(L_update - L[i, N - 1]) - cos(L[i, j] - L[i, N - 1])
        neighbor = j + 1
        if neighbor < N:
            dE += cos(L_update - L[i, neighbor]) - cos(L[i, j] - L[i, neighbor])
        else:
            dE += cos(L_update - L[i, 0]) - cos(L[i, j] - L[i, 0])
        dE *= 2 * -J
        # Calculates whether L[i,j] rotates
        R = exp(-dE / T)
        if R > 1 or random() < R:
            L[i, j] = L_update
            E += dE # / (N * N)

        if x != 0 and x % (2 * tau) == 0: # If the program didn't just start, and we are a correlation time since the last measurement, take a measurement
            expE += E
            measurements.append(E) # Adds the measurement to the list
    expE /= BM

    # The Bootstrap Error Analysis
    resample = BM # times to repeat re-sampling
    B_i = [] # for the calculation of <B> and sigma
    for y in range(resample):
        B = 0.0
        for z in range(int(BM)):
            n = randrange(0, BM)
            B += measurements[n]
        B /= BM
        B_i.append(B)

    # Now to calculate the Bootstrap sigma
    sigma_sigma = 0.0
    for w in range(resample):
        sigma_sigma += (B_i[w] - expE) ** 2
    sigma_sigma /= resample
    sigma_bootstrap = sqrt(sigma_sigma)

    return [T, expE, sigma_bootstrap] # This will create a results
        ↪ matrix which can be plotted


def XY_Replica_E(T):
    global N_global, E_measurements, tau_global, tau_after
    J = 1
    N = N_global # The lattice size: NxN
    # A test to make things quicker; higher temperatures equilibrate
        ↪ faster
    tau = tau_global
    if T > 20:
        tau = tau_after
    BM = E_measurements # Number of independent measurements for the
        ↪ bootstrap analysis
    steps = 2 * tau * BM # Number of times the program will run
    E = -4 * (N * N) # Initial Value of Energy since all spins start
        ↪ pointed up at \theta_i = 0.0
    boundary = N // 2
    L1 = zeros([N, N], float) # Lattice 1 where each entry is a value
        ↪ of \theta_i
    L2 = zeros([N, N], float) # Lattice 2
    A_1 = L1[:, 0:boundary]
    A_2 = L2[:, 0:boundary]
    B_1 = L1[:, boundary: N]
    B_2 = L2[:, boundary: N]

    print("N=", N, "; Replica XY-Model at T=", T)

    expE = 0.0 # Expectation value of E
    measurements = [] # List of Measurements
    # Main Monte Carlo cycle
    for x in range(steps + 1):
        i = randrange(0, N)
        j = randrange(0, N) # Picks a random starting location

        # Decides an anticipated spin amount
        L1_update = random() * 2 * pi
        L2_update = random() * 2 * pi

        if j < boundary:
            L1_update = L2_update
        # Calculates change in energy that would occur if this spin was accepted
        dE = 0.0
        # Starts calculating the nearest neighbor sum at location L[ i -1 , j]
        neighbor = i - 1
        if neighbor > -1:
            dE += cos(L1_update - L1[neighbor, j]) - cos(L1[i, j] - L1[neighbor, j])
            dE += cos(L2_update - L2[neighbor, j]) - cos(L2[i, j] - L2[neighbor, j])
        else:
            dE += cos(L1_update - L1[N - 1, j]) - cos(L1[i, j] - L1[N - 1, j]) # Periodic boundary conditions
            dE += cos(L2_update - L2[N - 1, j]) - cos(L2[i, j] - L2[N - 1, j])
        neighbor = i + 1
        if neighbor < N:
            dE += cos(L1_update - L1[neighbor, j]) - cos(L1[i, j] - L1[neighbor, j])
            dE += cos(L2_update - L2[neighbor, j]) - cos(L2[i, j] - L2[neighbor, j])
        else:
            dE += cos(L1_update - L1[0, j]) - cos(L1[i, j] - L1[0, j])
            dE += cos(L2_update - L2[0, j]) - cos(L2[i, j] - L2[0, j])
        neighbor = j - 1
        if neighbor > -1:
            dE += cos(L1_update - L1[i, neighbor]) - cos(L1[i, j] - L1[i, neighbor])
            dE += cos(L2_update - L2[i, neighbor]) - cos(L2[i, j] - L2[i, neighbor])
        else:
            dE += cos(L1_update - L1[i, N - 1]) - cos(L1[i, j] - L1[i, N - 1])
            dE += cos(L2_update - L2[i, N - 1]) - cos(L2[i, j] - L2[i, N - 1])
        neighbor = j + 1
        if neighbor < N:
            dE += cos(L1_update - L1[i, neighbor]) - cos(L1[i, j] - L1[i, neighbor])
            dE += cos(L2_update - L2[i, neighbor]) - cos(L2[i, j] - L2[i, neighbor])
        else:
            dE += cos(L1_update - L1[i, 0]) - cos(L1[i, j] - L1[i, 0])
            dE += cos(L2_update - L2[i, 0]) - cos(L2[i, j] - L2[i, 0])
        dE *= -J

        # Calculates whether L[i,j] rotates
        R = exp(-dE / T)
```

```python
        if R > 1 or random() < R:
            L1[i, j] = L1_update
            L2[i, j] = L2_update
            E += dE # / (N * N)
        if x != 0 and x % (2 * tau) == 0:
            expE += E
            measurements.append(E) # Adds the measurement to the list
    expE /= BM

    # The Bootstrap Error Analysis
    resample = BM # times to repeat re-sampling
    B_i = [] # for the calculation of <B> and sigma
    for y in range(resample):
        B = 0.0
        for z in range(int(BM)):
            n = randrange(0, BM)
            B += measurements[n]
        B /= BM
        B_i.append(B)

    # Now to calculate the Bootstrap sigma
    sigma_sigma = 0.0
    for w in range(resample):
        sigma_sigma += (B_i[w] - expE) ** 2
    sigma_sigma /= resample
    sigma_bootstrap = sqrt(sigma_sigma)

    # This is a test to make sure that A_1 and A_2 are indeed being updated the same.
    equivalence_test = 'no'
    if equivalence_test == 'yes':
        matches = 0.0
        for columns in range(0, boundary):
            for rows in range(N):
                if A_1[rows, columns] == A_2[rows, columns]:
                    matches += 1
        if matches == N * boundary:
            print("A_1 and A_2 are the same!")
        else:
            print("We messed up somewhere :(")

    return [T, expE, sigma_bootstrap] # This will create a results matrix which can be plotted


def vary_temps_RMI(T_min, T_max, T_step, graph='no'):
    if T_min == 0:
        temps = arange(T_min + T_step, T_max, T_step)
    else:
        temps = arange(T_min, T_max, T_step)

    # I have to separate the core mapping to prevent a memory error
    cores = Pool()
    result1 = cores.map(XY_Replica_E, temps)
    cores.close()
    cores.join()

    cores = Pool()
    result2 = cores.map(XY_A_U_B, temps)
    cores.close()
    cores.join()

    cores = Pool()
    result3 = cores.map(XY_E, temps)
    cores.close()
    cores.join()

    replica = array(result1)
    A_U_B = array(result2)
    normal = array(result3)

    # Both Ising models are at the same temperature so,
    T_plot = normal[:, 0] # Takes the first column of the results matrix
    #
    # Replica XY
    E_replica = replica[:, 1] # Second column
    sigma_replica = replica[:, 2] # Third column

    # Normal XY
    E_A_U_B = A_U_B[:, 1] # Second column
    sigma_A_U_B = A_U_B[:, 2] # Third column

    E_normal = normal[:, 1]
    sigma_normal = normal[:, 2]

    if graph == 'yes':
        plot(T_plot, E_replica, 'b', label='Replica XY')
        plot(T_plot, E_A_U_B, 'r', label='A U B')
        plot(T_plot, E_normal, 'g', label='Normal XY')

        title("Energy of the Three Models", fontsize=16)
        xlabel(r"$T$", fontsize=16)
        ylabel("Energy", fontsize=16)
        xlim(0, 10)
        legend()
        show()
    return T_plot, E_replica, sigma_replica, E_A_U_B, sigma_A_U_B,
        ↪ E_normal, sigma_normal


def RMI_calc(T_min, T_max, T_step, save_data='no', graph='no'):
    t1 = time.time()

    Data = vary_temps_RMI(T_min, T_max, T_step)

    T_plot = Data[0]
    # Gathers the replica data
    E_replica = Data[1]
    sigma_replica = Data[2]
    # Gathers the normal data
    E_A_U_B = Data[3]
    sigma_A_U_B = Data[4]

    E_normal = Data[5]
    sigma_normal = Data[6]
    # Calculating RMI for each T
    print('Working on Renyi Mutual Information...')
    count = len(E_A_U_B)

    RMI_plot = []
    RMI_sigma_plot = []
    deltaT = T_step
    # Calculates the RMI and the sigma for each RMI(T)
    for i in range(count):
        RMI = 0.0
        sigma_sigma_i = 0.0
        for j in range(i, count):
            term_j = deltaT * (2 * (E_replica[j]) - (E_A_U_B[j]) - 2 *
                ↪  E_normal[j]) / ((T_plot[j]) ** 2)
            RMI += term_j
            # Now to propagate the error from each E measurement...
            sigma_sigma_j = ((2 * deltaT) / ((T_plot[j] ** 2) *
                ↪ N_global * 2)) ** 2 * (sigma_replica[j] ** 2) + (
                ↪ deltaT / ((
                T_plot[j] ** 2) * N_global * 2)) ** 2 * (sigma_A_U_B[j]
                    ↪  ** 2) + ((2 * deltaT) / ((T_plot[j] ** 2) *
                    ↪ N_global * 2)) ** 2 * (sigma_normal[j] ** 2)
            sigma_sigma_i += sigma_sigma_j
        sigma_i = sqrt(sigma_sigma_i)
        RMI /= 2 * N_global
        RMI_plot.append(RMI)
        RMI_sigma_plot.append(sigma_i)
        if i % 100 == 0:
            print('Calculating RMI for T =', i * T_step)

    if save_data == 'yes':
        RMI_data = array([RMI_plot, RMI_sigma_plot])
        Data = array(Data)
        Data_txt = vstack((Data, RMI_data))
        t_elapse = (time.time() - t1) / 3600
        savetxt('RMI XY;{0};{1},{2},{3},{4},{5}.txt'.format(date,
           ↪ E_measurements, T_min, T_max, T_step, N_global),
           ↪ Data_txt, header='This data took {0:.3f} hours and was
           ↪ recorded on {1}. This was run on the PSU Cluster.'.
           ↪ format(t_elapse, datetime.datetime.today()))


    return T_plot, RMI_plot, RMI_sigma_plot


if __name__ == '__main__':
    t_start = time.time()

    # Main Program
    T_min = float(sys.argv[1].split(',')[0])
    T_max = float(sys.argv[2].split(',')[0])
    T_step = float(sys.argv[3].split(',')[0])

    RMI_calc(T_min, T_max, T_step, save_data='yes')

    # End of Main Program

    t_elapse = (time.time() - t_start) / 3600
    print("Full Program done in {0:.3f} hours".format(t_elapse))
```

Appendix B

QCD adjoint XY Model Code

This was the main code I ran to collect all the data used for the paper. It used this Hamiltonian:

$$H = -J \sum_{\langle i,j \rangle} \cos(\theta_i - \theta_j) + \tilde{y} \sum \cos(p\theta_i) \tag{B.1}$$

This code is fed arguments through the console when initiating. This feature made running it on the cluster much easier, because I could call batches of this code with a wide variety of parameters simultaneously.

B.1 Running

There is no initial parameter set up this time, because all parameters are entered when running it in the command prompt. This makes running many different simulations easier, but adds steps to running just one simulation. The command line arguments are ordered as follows:

1. Argument 1 = lattice size N
2. Argument 2 = the ΔT
3. Argument 3 = number of measurements to take
4. Argument 4 = starting temperature
5. Argument 5 = ending temperature
6. Argument 6 = value of $\tilde{y}$ (see (B.1))
7. Argument 7 = value of p (see (B.1))

Enter in the command line:

```
> python RMI_QCD_Cluster.py [N] [T_step] [measurements] [T_start] [T_end] [y_tilde] [p]
```

So to execute this script, open the command prompt. Navigate to the directory of the script, and then enter a command like so:

```
> python RMI_QCD_Cluster.py 32 0.05 20000 0.0 1.0 1.0 4
```

Which would run a simulation for these particular parameters.

B.2 Data Analysis

This code outputs a file with all the desired data we want. The data is organized like so:

1. Temperature plot
2. Energy of the replica lattice
3. Variance of the replica energy
4. Energy of the $\mathcal{A} \cup \mathcal{B}$ lattice
5. Variance of the $\mathcal{A} \cup \mathcal{B}$ energy
6. Energy of the normal lattice
7. Variance of the normal energy
8. Magnetization of the lattice
9. Variance of the Magnetization
10. Heat capacity
11. Variance on the heat capacity
12. Susceptibility
13. Variance on susceptibility

Like the previous section, a code that assigns each array to a variable will allow you to analyze the desired data. However notice that this code doesn't output the RMI. This is because usually this code was run in the method of section 4.3. Once again, if this was run on the cluster, an aggregation code must be run in order to output one file. If ran in this manner, the code will output the data files to a folder titled "Data from RMI QCD; 2017-12-13; 20000, 0.05, 32, n=2, y 0.0, theta=4", which will contain all the files for each temperature range.

For example, if I ran one simulation from 0.0 to 4.0, then there would be four data files in this output folder that would need to be aggregated. This aggregation code is listed in appendix C.3.

Listing B.1: The main code I ran

```python
from numpy import ones, arange, sqrt, array, savetxt, vstack, zeros
from math import exp, pi, cos, sin
from random import random, randrange
from multiprocessing import Pool
import time, sys, os, datetime, numpy
date = datetime.date.today()

N_global = int(sys.argv[1].split(',')[0])
T_step = float(sys.argv[2].split(',')[0])
if N_global == 8:
    tau_global = 9000
```

```python
if N_global == 16:
    tau_global = 14000
if N_global == 24:
    tau_global = 40000
if N_global == 32:
    tau_global = 55000
if N_global == 40:
    tau_global = 75000
if N_global == 48:
    tau_global = 125000
if N_global == 56:
    tau_global = 150000
E_measurements = int(sys.argv[3].split(',')[0])

y_tilde = float(sys.argv[6].split(',')[0])
theta_coefficient = int(sys.argv[7].split(',')[0])


# Normal XY
def QCD_E(T):
    global N_global, E_measurements, tau_after, y_tilde, theta_coefficient
    J = 1
    N = N_global # The lattice size: NxN
    tau = tau_global # The equilibration time

    BM = E_measurements # Number of independent measurements for the bootstrap analysis
    steps = 2 * tau * BM # Number of times the program will run
    E = -2 * (N * N) - y_tilde * (N * N) # Initial Value of Energy since all spins start pointed up at \theta_i = 0.0
    m_1 = N*N # Initial value of magnetization
    m_2 = 0
    L = zeros([N, N], float) # Generates the lattice where each entry is a value of \theta_i

    print("N=", N, "; Normal QCD-Model at T=", T)

    expE = 0.0 # Expectation value of E
    expE_E = 0.0
    expM = 0.0
    expM_M = 0.0
    measurements = [] # List of Measurements
    E_E_measurements = []
    M_measurements = []
    M_M_measurements = []
    # Main Monte Carlo cycle
    for x in range(steps + 1):
        i = randrange(0, N)
```

```
            j = randrange(0, N) # Picks a random starting location

            # Decides an anticipated spin amount
            L_update = random() * 2 * pi
            # Calculates change in energy that would occur if this spin
               ↪ was accepted
            dE = 0.0
            # Starts calculating the nearest neighbor sum at location L[ i
               ↪ -1 , j]
            neighbor = i - 1
            if neighbor > -1:
                dE += cos(L_update - L[neighbor, j]) - cos(L[i, j] - L[
                   ↪ neighbor, j]) # Checks if the neighbor is within the
                   ↪  lattice
            else:
                dE += cos(L_update - L[N - 1, j]) - cos(L[i, j] - L[N - 1,
                   ↪  j]) # Periodic boundary conditions
            neighbor = i + 1
            if neighbor < N:
                dE += cos(L_update - L[neighbor, j]) - cos(L[i, j] - L[
                   ↪ neighbor, j])
            else:
                dE += cos(L_update - L[0, j]) - cos(L[i, j] - L[0, j])
            neighbor = j - 1
            if neighbor > -1:
                dE += cos(L_update - L[i, neighbor]) - cos(L[i, j] - L[i,
                   ↪ neighbor])
            else:
                dE += cos(L_update - L[i, N - 1]) - cos(L[i, j] - L[i, N -
                   ↪  1])
            neighbor = j + 1
            if neighbor < N:
                dE += cos(L_update - L[i, neighbor]) - cos(L[i, j] - L[i,
                   ↪ neighbor])
            else:
                dE += cos(L_update - L[i, 0]) - cos(L[i, j] - L[i, 0])
            dE *= -J
            dE += y_tilde * (cos(theta_coefficient * L[i, j]) - cos(
               ↪ theta_coefficient * L_update))

            # Calculates whether L[i,j] rotates
            R = exp(-dE * T)
            if R > 1 or random() < R:
                m_1 = m_1 + cos(L_update) - cos(L[i, j])
                m_2 = m_2 + sin(L_update) - sin(L[i, j])
                L[i, j] = L_update
                E += dE # / (N * N)
        # this is where all the various measurements are taken
            if x != 0 and x % (2 * tau) == 0:
```

```
            expE += E
            EE = E * E
            expE_E += EE
            M = sqrt(m_1**2 + m_2**2)
            expM += M
            MM = M * M
            expM_M += MM
            measurements.append(E) # Adds the measurement to the list
            M_measurements.append(M)
            E_E_measurements.append(EE)
            M_M_measurements.append(MM)
    expE /= BM
    expM /= BM
    expE_E /= BM
    expM_M /= BM

    # The Bootstrap Error Analysis
    resample = BM # times to repeat re-sampling
    B_i = [] # for the calculation of <B> and sigma
    M_i = []
    E_E_i = []
    M_M_i = []
    for y in range(resample):
        B = 0.0
        M_error = 0.0
        E_E_error = 0.0
        M_M_error = 0.0
        for z in range(int(BM)):
            n = randrange(0, BM)
            B += measurements[n]
            E_E_error += measurements[n]
            M_error += M_measurements[n]
            M_M_error += M_M_measurements[n]
        B /= BM
        M_error /= BM
        E_E_error /= BM
        M_M_error /= BM
        B_i.append(B)
        E_E_i.append(E_E_error)
        M_i.append(M_error)
        M_M_i.append(M_M_error)

    # Now to calculate the Bootstrap sigma
    sigma_sigma = 0.0
    sigma_sigma_M = 0.0
    sigma_sigma_E_E = 0.0
    sigma_sigma_M_M= 0.0
    for w in range(resample):
        sigma_sigma += (B_i[w] - expE) ** 2
        sigma_sigma_M += (M_i[w] - expM) ** 2
        sigma_sigma_E_E += (E_E_i[w] - expE_E) ** 2
        sigma_sigma_M_M += (M_M_i[w] - expM_M) ** 2

    sigma_sigma /= resample
    sigma_sigma_M /= resample
    sigma_sigma_E_E /= resample
    sigma_sigma_M_M /= resample

    sigma_bootstrap = sqrt(sigma_sigma)
    sigma_bootstrap_M = sqrt(sigma_sigma_M)
    sigma_bootstrap_EE = sqrt(sigma_sigma_E_E)
    sigma_bootstrap_MM = sqrt(sigma_sigma_M_M)

    heatcap = expE_E - (expE * expE)
    sigma_heatcap = sqrt((sigma_bootstrap_EE**2) + ((2 * heatcap * sigma_bootstrap) / expE)**2)

    susceptibility = expM_M - (expM * expM)
    sigma_susceptibility = sqrt((sigma_bootstrap_MM**2) + ((2 * susceptibility * sigma_bootstrap_M) / expM)**2)

    return [T, expE, sigma_bootstrap, M, sigma_bootstrap_M, heatcap, sigma_heatcap, susceptibility, sigma_susceptibility] # This will create a results matrix which can be plotted


def QCD_A_U_B(T):
    global N_global, E_measurements, tau_after
    J = 1
    N = N_global # The lattice size: NxN
    tau = tau_global # The equilibration time

    BM = E_measurements # Number of independent measurements for the bootstrap analysis
    steps = 2 * tau * BM # Number of times the program will run
    E = -4 * (N * N) - 2 * y_tilde * (N * N) # Initial Value of Energy since all spins start pointed up at \theta_i = 0.0
    L = zeros([N, N], float) # Generates the lattice where each entry is a value of \theta_i

    print("N=", N, "; A-union-B QCD-Model at T=", T)

    expE = 0.0 # Expectation value of E
    measurements = [] # List of Measurements
    # Main Monte Carlo cycle
    for x in range(steps + 1):
        i = randrange(0, N)
        j = randrange(0, N) # Picks a random starting location

        # Decides an anticipated spin amount
        L_update = random() * 2 * pi
        # Calculates change in energy that would occur if this spin was accepted
        dE = 0.0
        # Starts calculating the nearest neighbor sum at location L[ i -1 , j]
        neighbor = i - 1
        if neighbor > -1:
            dE += cos(L_update - L[neighbor, j]) - cos(
                L[i, j] - L[neighbor, j]) # Checks if the neighbor is within the lattice
        else:
            dE += cos(L_update - L[N - 1, j]) - cos(L[i, j] - L[N - 1, j]) # Periodic boundary conditions
        neighbor = i + 1
        if neighbor < N:
            dE += cos(L_update - L[neighbor, j]) - cos(L[i, j] - L[neighbor, j])
        else:
            dE += cos(L_update - L[0, j]) - cos(L[i, j] - L[0, j])
        neighbor = j - 1
        if neighbor > -1:
            dE += cos(L_update - L[i, neighbor]) - cos(L[i, j] - L[i, neighbor])
        else:
            dE += cos(L_update - L[i, N - 1]) - cos(L[i, j] - L[i, N - 1])
        neighbor = j + 1
        if neighbor < N:
            dE += cos(L_update - L[i, neighbor]) - cos(L[i, j] - L[i, neighbor])
        else:
            dE += cos(L_update - L[i, 0]) - cos(L[i, j] - L[i, 0])
        dE *= 2 * -J
        dE += 2 * y_tilde * (cos(theta_coefficient * L[i, j]) - cos(theta_coefficient * L_update))

        # Calculates whether L[i,j] rotates
        R = exp(-dE * T)
        if R > 1 or random() < R:
            L[i, j] = L_update
            E += dE # / (N * N)

        if x != 0 and x % (2 * tau) == 0:
            expE += E
            measurements.append(E) # Adds the measurement to the list
    expE /= BM
```

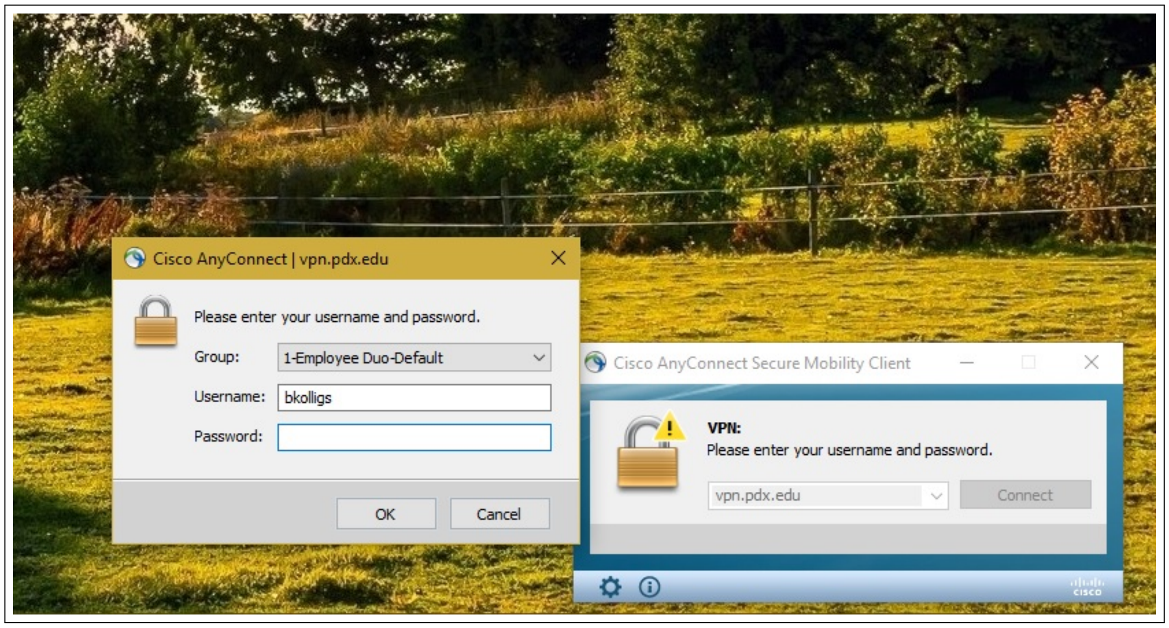
```python

    # The Bootstrap Error Analysis
    resample = BM # times to repeat re-sampling
    B_i = [] # for the calculation of <B> and sigma
    for y in range(resample):
        B = 0.0
        for z in range(int(BM)):
            n = randrange(0, BM)
            B += measurements[n]
        B /= BM
        B_i.append(B)

    # Now to calculate the Bootstrap sigma
    sigma_sigma = 0.0
    for w in range(resample):
        sigma_sigma += (B_i[w] - expE) ** 2
    sigma_sigma /= resample
    sigma_bootstrap = sqrt(sigma_sigma)

    return [T, expE, sigma_bootstrap] # This will create a results matrix which can be plotted


def QCD_Replica_E(T):
    global N_global, E_measurements, tau_global, tau_after
    J = 1
    N = N_global # The lattice size: NxN
    tau = tau_global

    BM = E_measurements # Number of independent measurements for the bootstrap analysis
    steps = 2 * tau * BM # Number of times the program will run
    E = -4 * (N * N) - 2 * y_tilde * (N * N) # Initial Value of Energy  since all spins start pointed up at \theta_i = 0.0
    boundary = N // 2
    L1 = zeros([N, N], float) # Lattice 1 where each entry is a value of \theta_i
    L2 = zeros([N, N], float) # Lattice 2
    A_1 = L1[:, 0:boundary]
    A_2 = L2[:, 0:boundary]
    B_1 = L1[:, boundary: N]
    B_2 = L2[:, boundary: N]

    print("N=", N, "; Replica QCD-Model at T=", T)

    expE = 0.0 # Expectation value of E
    measurements = [] # List of Measurements
    # Main Monte Carlo cycle
    for x in range(steps + 1):
        i = randrange(0, N)
        j = randrange(0, N) # Picks a random starting location

        # Decides an anticipated spin amount
        L1_update = random() * 2 * pi
        L2_update = random() * 2 * pi

        if j < boundary:
            L1_update = L2_update
        # Calculates change in energy that would occur if this spin was accepted
        dE = 0.0
        # Starts calculating the nearest neighbor sum at location L[ i -1 , j]
        neighbor = i - 1
        if neighbor > -1:
            dE += cos(L1_update - L1[neighbor, j]) - cos(L1[i, j] - L1[neighbor, j])
            dE += cos(L2_update - L2[neighbor, j]) - cos(L2[i, j] - L2[neighbor, j])
        else:
            dE += cos(L1_update - L1[N - 1, j]) - cos(L1[i, j] - L1[N - 1, j]) # Periodic boundary conditions
            dE += cos(L2_update - L2[N - 1, j]) - cos(L2[i, j] - L2[N - 1, j])
        neighbor = i + 1
        if neighbor < N:
            dE += cos(L1_update - L1[neighbor, j]) - cos(L1[i, j] - L1[neighbor, j])
            dE += cos(L2_update - L2[neighbor, j]) - cos(L2[i, j] - L2[neighbor, j])
        else:
            dE += cos(L1_update - L1[0, j]) - cos(L1[i, j] - L1[0, j])
            dE += cos(L2_update - L2[0, j]) - cos(L2[i, j] - L2[0, j])
        neighbor = j - 1
        if neighbor > -1:
            dE += cos(L1_update - L1[i, neighbor]) - cos(L1[i, j] - L1[i, neighbor])
            dE += cos(L2_update - L2[i, neighbor]) - cos(L2[i, j] - L2[i, neighbor])
        else:
            dE += cos(L1_update - L1[i, N - 1]) - cos(L1[i, j] - L1[i, N - 1])
            dE += cos(L2_update - L2[i, N - 1]) - cos(L2[i, j] - L2[i, N - 1])
        neighbor = j + 1
        if neighbor < N:
            dE += cos(L1_update - L1[i, neighbor]) - cos(L1[i, j] - L1[i, neighbor])
            dE += cos(L2_update - L2[i, neighbor]) - cos(L2[i, j] - L2[i, neighbor])
        else:
            dE += cos(L1_update - L1[i, 0]) - cos(L1[i, j] - L1[i, 0])
            dE += cos(L2_update - L2[i, 0]) - cos(L2[i, j] - L2[i, 0])
        dE *= -J
        dE += y_tilde * (cos(theta_coefficient * L1[i, j]) - cos(theta_coefficient * L1_update)) + y_tilde * (cos(theta_coefficient * L2[i, j]) - cos(theta_coefficient * L2_update))

        # Calculates whether L[i,j] rotates
        R = exp(-dE * T)
        if R > 1 or random() < R:
            L1[i, j] = L1_update
            L2[i, j] = L2_update
            E += dE # / (N * N)
        if x != 0 and x % (2 * tau) == 0:
            expE += E
            measurements.append(E) # Adds the measurement to the list
    expE /= BM

    # The Bootstrap Error Analysis
    resample = BM # times to repeat re-sampling
    B_i = [] # for the calculation of <B> and sigma
    for y in range(resample):
        B = 0.0
        for z in range(int(BM)):
            n = randrange(0, BM)
            B += measurements[n]
        B /= BM
        B_i.append(B)

    # Now to calculate the Bootstrap sigma
    sigma_sigma = 0.0
    for w in range(resample):
        sigma_sigma += (B_i[w] - expE) ** 2
    sigma_sigma /= resample
    sigma_bootstrap = sqrt(sigma_sigma)

    # This is a test to make sure that A_1 and A_2 are indeed being updated the same.
    equivalence_test = 'no'
    if equivalence_test == 'yes':
        matches = 0.0
        for columns in range(0, boundary):
            for rows in range(N):
                if A_1[rows, columns] == A_2[rows, columns]:
                    matches += 1
        if matches == N * boundary:
            print("A_1 and A_2 are the same!")
        else:
            print("We messed up somewhere :(")

    return [T, expE, sigma_bootstrap] # This will create a results
        ↪ matrix which can be plotted


def vary_temps_RMI(T_min, T_max, T_step):
    if T_min == 0:
        temps = arange(T_min + T_step, T_max, T_step)
    else:
        temps = arange(T_min, T_max, T_step)

    # I have to separate the core mapping to prevent a memory error
    cores = Pool()
    result1 = cores.map(QCD_Replica_E, temps)
    cores.close()
    cores.join()

    cores = Pool()
    result2 = cores.map(QCD_A_U_B, temps)
    cores.close()
    cores.join()

    cores = Pool()
    result3 = cores.map(QCD_E, temps)
    cores.close()
    cores.join()

    replica = array(result1)
    A_U_B = array(result2)
    normal = array(result3)

    # Both Ising models are at the same temperature so,
    T_plot = normal[:, 0] # Takes the first column of the results
        ↪ matrix
    #
    # Replica lattice
    E_replica = replica[:, 1] # Second column
    sigma_replica = replica[:, 2] # Third column

    # Normal lattice
    E_A_U_B = A_U_B[:, 1] # Second column
    sigma_A_U_B = A_U_B[:, 2] # Third column

    E_normal = normal[:, 1]
    sigma_normal = normal[:, 2]
```

```python

    magnetization = normal[:, 3]
    sigma_mag = normal[:, 4]

    heatcap = normal[:, 5]
    sigma_heatcap = normal[:, 6]

    susceptibility = normal[:, 7]
    sigma_susceptibility = normal[:, 8]

    return T_plot, E_replica, sigma_replica, E_A_U_B, sigma_A_U_B,
       ↪ E_normal, sigma_normal, magnetization, sigma_mag, heatcap,
       ↪ sigma_heatcap, susceptibility, sigma_susceptibility


def RMI_calc(T_min, T_max, T_step, save_data='no'):
    t1 = time.time()

    Data = vary_temps_RMI(T_min, T_max, T_step)

    T_plot = Data[0]
    # Gathers the replica data
    E_replica = Data[1]
    sigma_replica = Data[2]
    # Gathers the normal data
    E_A_U_B = Data[3]
    sigma_A_U_B = Data[4]

    E_normal = Data[5]
    sigma_normal = Data[6]

    if save_data == 'yes':
        Data = array(Data)
        t_elapse = (time.time() - t1) / 3600
        folder_path = '/home/bkolligs/Control_Center/QCD_Model/
           ↪ Finished_Data/'
        folder_name = 'Data from RMI QCD; {0}; {1}, {2}, {3}, n=2, y
           ↪ ~{4}, theta={5}'.format(date, E_measurements, T_step,
           ↪ N_global, y_tilde, theta_coefficient)
        if not os.path.exists(folder_path + folder_name):
            os.makedirs(folder_path + folder_name)
        savetxt('{8}{6}/RMI QCD; {0}; {1}, {2}, {3}, {4}, {5}, n=2, y
           ↪ ~{7}, theta={9}.txt'.format(date, E_measurements, T_min,
           ↪  T_max, T_step, N_global, folder_name, y_tilde,
           ↪ folder_path, theta_coefficient), Data, header='This data
           ↪  took {0:.3f} hours and was recorded on {1}. This was
           ↪ run on the PSU Cluster.'.format(t_elapse, datetime.
           ↪ datetime.today()))

    return T_plot


if __name__ == '__main__':
    t_start = time.time()

    # Main Program
    T_min = float(sys.argv[4].split(',')[0])
    T_max = float(sys.argv[5].split(',')[0])

    RMI_calc(T_min, T_max, T_step, save_data='yes')

    # End of Main Program

    t_elapse = (time.time() - t_start) / 3600
    print("Full Program done in {0:.3f} hours".format(t_elapse))
```

Appendix C

Parallel Computing Architecture Code

First you must create a PSU Odin account. Make sure to set up the ”Duo” activation, because this is required to log in to the VPN to access the cluster. ”Duo” is a two-factor authentication system, which means you can’t log in with just your Odin ID. Next, to get started on the cluster, you must contact whoever is in charge of it and request an account to be made (at the time of writing this the contact is William Garrick: will@pdx.edu). Alternatively, you can fill out the PICS Request form discussed on this page: *https://sites.google.com/pdx.edu/research-computing/faqs/request-account?authuser=0*. Next install the Cisco Any Connect Secure Mobility Client *https://www.cisco.com/c/en/us/products/security/anyconnect-secure-mobility-client/index.html*. This is the VPN that will allow you to connect to the cluster.


Cisco AnyConnect | vpn.pdx.edu

Please enter your username and password.

Group: 1-Employee Duo-Default

Username: bkolligs

Password:

OK Cancel

Cisco AnyConnect Secure Mobility Client

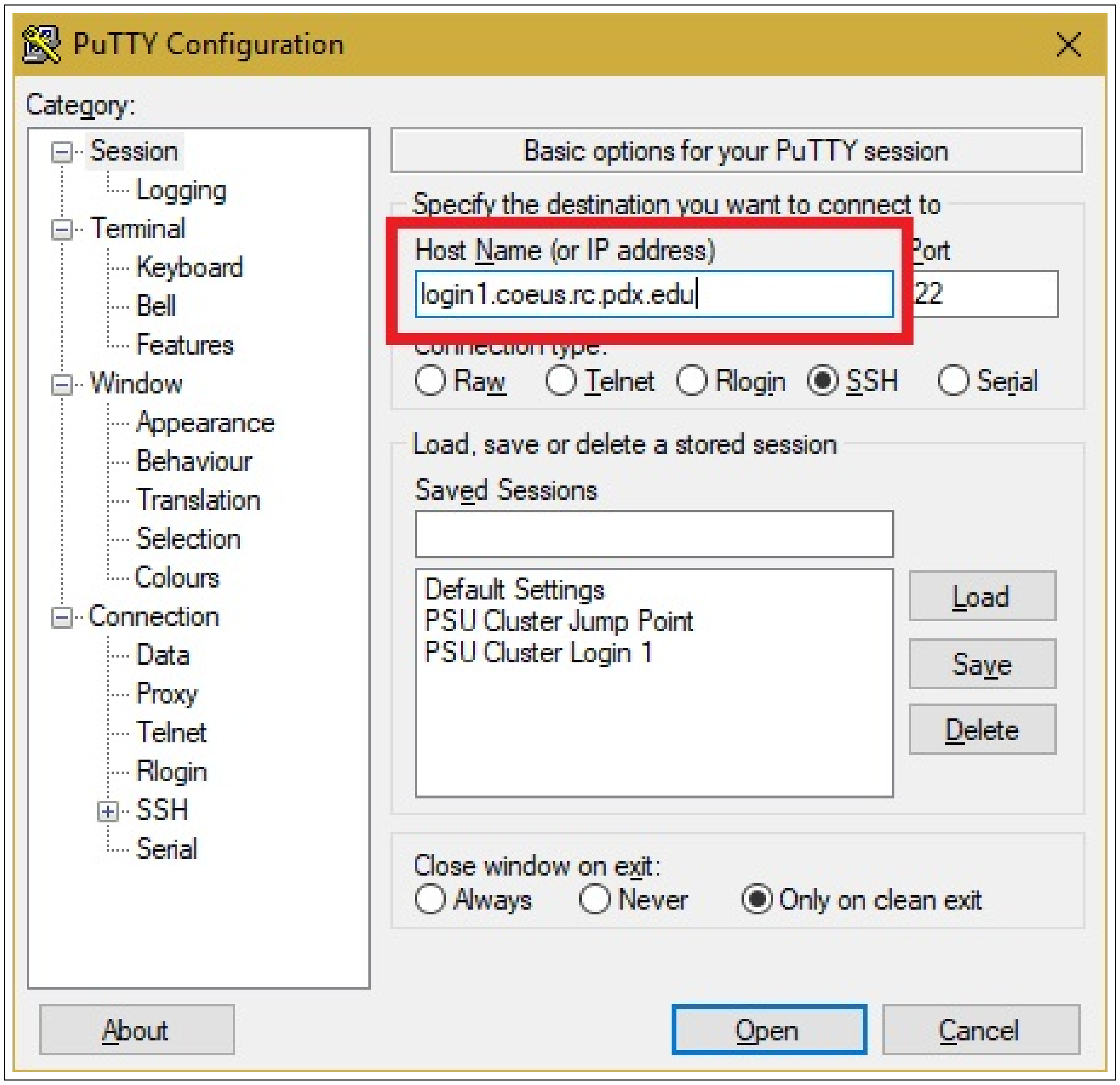
VPN:
Please enter your username and password.

vpn.pdx.edu Connect

Figure C.1: Enter your Odin ID login information. Then you will be prompted with a Duo message somewhere, depending on how you set it up. I set up Duo to send a push notification to my phone, an option which required me to download the Duo app.

Enter “vpn.pdx.edu” and click connect.

Now to connect to the Coeus cluster itself, we need to use an SSH connection. Mac computers will have this installed already in the terminal, but Windows users must download the “PuTTY” client https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html.



Specify the destination you want to connect to

Host Name (or IP address): login1.coeus.rc.pdx.edu

Port: 22

Connection type: ○ Raw ○ Telnet ○ Rlogin [✓ SSH] ○ Serial

Load, save or delete a stored session

Saved Sessions

Default Settings
PSU Cluster Jump Point
PSU Cluster Login 1

Close window on exit: ○ Always ○ Never [✓ Only on clean exit]

Figure C.2: In the Host Name field enter “login1.coeus.rc.pdx.edu”. Note that you can only connect to this host name if you are connected to PSU’s network using the VPN.

In the host name field, enter “login1.coeus.rc.pdx.edu”. A terminal will open up that will prompt you for your Odin ID. If this is your first time logging in to the cluster, you will be prompted with a few questions. Answer “y” or “yes” to all of them.

Then when you get to your home screen, enter in

```
> touch ~/.actrun
```

This will finish the setup of your home directory. After you complete the initial set up, log out and back in again. Now you are good to go!

Since we hope to run things on the cluster, we need a way to transfer files to and from our computer. This is done using a file transfer protocol. Mac users are encouraged to research how to do this with their terminals. Windows users may download a program called "WinSCP" *https://winscp.net/eng/download.php*. This program is very easy to use and you can just drag and drop folders from your computer to the cluster using the *commander* interface.

C.1 Control Center Code

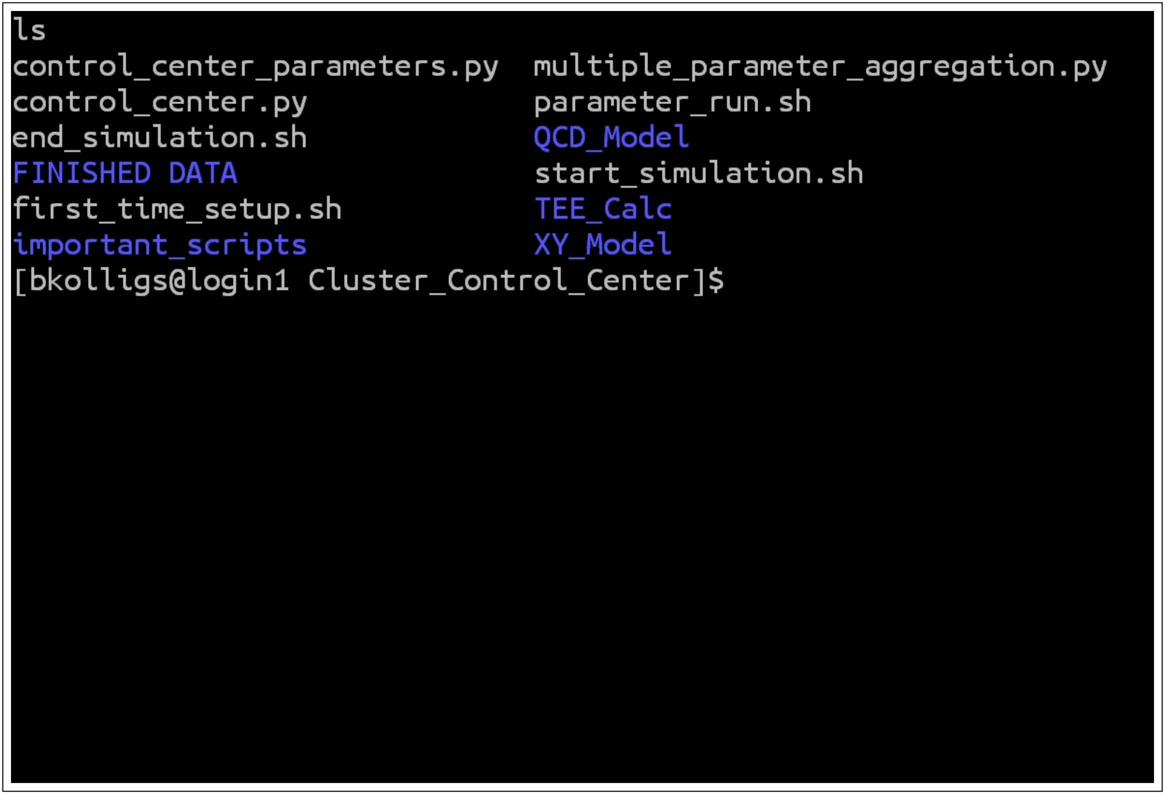
Now for the cluster code. This "Control_Center.py" is the main code I ran on the cluster. It allowed me to run simulations without navigating to manually submit all the jobs I wanted to run which saved hours of labor. This is the main interface you should submit jobs to the cluster in. This code is compatible with the XY model, the QCD model, and calculating Topological Entanglement entropy (TEE).

Note that this code will work on any cluster that uses the SLURM scheduler. To use it with a different scheduler, you will have to edit the "generate.py" file to generate batch scripts in the correct format, and also modify line 78-79 of the code which runs the "run_it_all.sh" bash script that submits multiple SLURM jobs at once.

C.1.1 Running

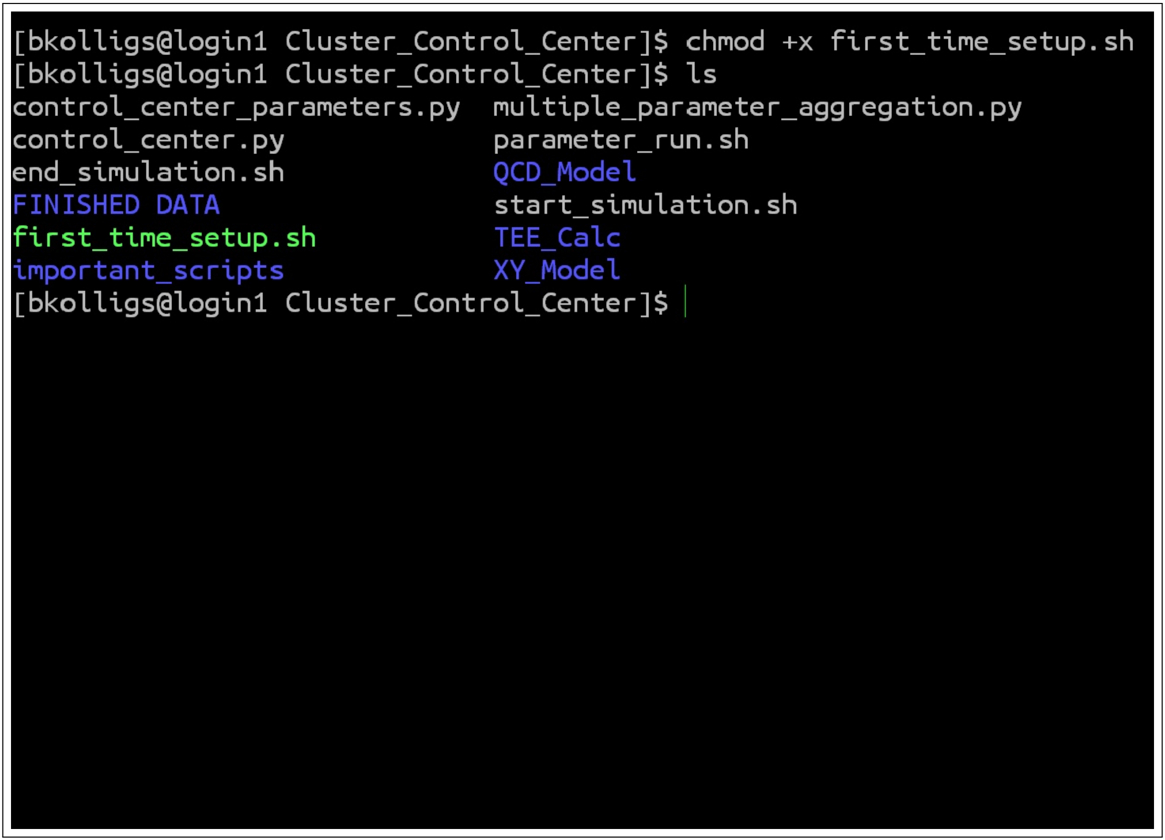
This code will automatically submit one batch of jobs for one simulation, or many batches of jobs for many simulations. First you must load a copy of the Control Center folder onto the cluster. Email me (benjaminkolligs@lclark.edu) or Mohamed Anber (manber@lclark.edu) for a functional copy of this folder.

The first thing to do after copying control center onto the cluster is to run the first time setup code, which is described in the following graphics C.3, C.4, C.5:



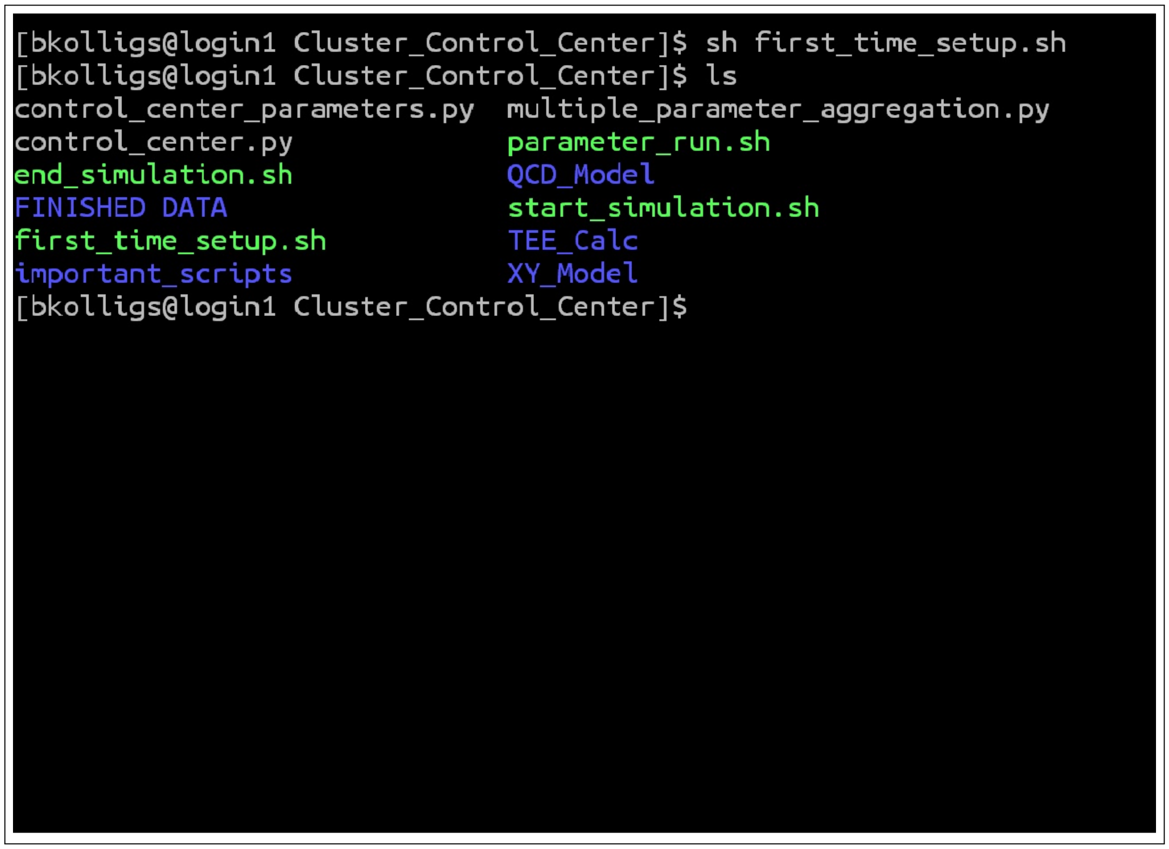
```
ls
control_center_parameters.py  multiple_parameter_aggregation.py
control_center.py             parameter_run.sh
end_simulation.sh             QCD_Model
FINISHED DATA                 start_simulation.sh
first_time_setup.sh           TEE_Calc
important_scripts             XY_Model
[bkolligs@login1 Cluster_Control_Center]$
```

Figure C.3: Step 1: The intial contents of the control center after copying the folder to your directory on the cluster.



```
[bkolligs@login1 Cluster_Control_Center]$ chmod +x first_time_setup.sh
[bkolligs@login1 Cluster_Control_Center]$ ls
control_center_parameters.py  multiple_parameter_aggregation.py
control_center.py             parameter_run.sh
end_simulation.sh             QCD_Model
FINISHED DATA                 start_simulation.sh
first_time_setup.sh           TEE_Calc
important_scripts             XY_Model
[bkolligs@login1 Cluster_Control_Center]$
```

Figure C.4: Step 2: Enter the command `chmod +x first_time_setup.sh`. This makes “first_time_setup.sh” executable. After entering this command, the file name be highlighted green to show it is now executable.



```
[bkolligs@login1 Cluster_Control_Center]$ sh first_time_setup.sh
[bkolligs@login1 Cluster_Control_Center]$ ls
control_center_parameters.py  multiple_parameter_aggregation.py
control_center.py             parameter_run.sh
end_simulation.sh             QCD_Model
FINISHED DATA                 start_simulation.sh
first_time_setup.sh           TEE_Calc
important_scripts             XY_Model
[bkolligs@login1 Cluster_Control_Center]$
```

Figure C.5: Step 3: Now run the setup script by entering `sh first_time_setup.sh`. This makes all the necessary scripts executable. After running the first time setup script, you should now see several files highlighted green.

Now to initialize this program and run a simulation, navigate to the `Control_Center` folder. Then type

```
> sh start_simulation.sh
```

First, the program will ask you whether or not you want to input the parameters manually. If you say ’yes’, then the program will ask you to enter each required parameter. Follow the remaining prompts. Note that when the program asks you to input a partition, larger lattice sizes will take longer amounts of time. The partition names and time limits are as follows:

1. ’medium’: ’4-00:00:00’,
2. ’long’: ’20-00:00:00’,
3. ’phi’: ’20-00:00:00’,
4. ’allcpu’:’4:00:00’,
5. ’short’: ’2:00:00’

Enter one of the partition names and the program will automatically assign time limits to the simulation.

The program will also ask you for a “starting T” and “ending T”, as well as a temperature step size. This is different from `T_step` which was asked earlier. This essentially describes how many jobs you want to run, and what temperature ranges each job will simulate. For example, say I wanted to run a simulation from $T = 0$ to $T = 4$ with a $\Delta T = 0.05$. This is a total of 79 temperatures. If I'm running on the PSU Cluster, (which you probably are), then each node in the medium and long partitions has 20 cores. So to most efficiently run my simulation, I want to submit 4 jobs, three that will run 20 temperatures, and one that will run the last 19. The way to do this is be entering a starting T of 0, ending T of 4, and temperature step size of 1. This essentially creates four jobs, that will run from $T = 0$ to 1, $T = 1$ to 2, $T = 2$ to 3, $T = 3$ to 4. Running sub-simulations in these ranges is the most efficient way to utilize the nodes of the cluster.

We can also think of this in a "dimensional analysis" way of thinking. The cluster runs $0.05K$ per core. A node has 20 cores. So in a sense, this "temperature step size" parameter is the K/node that the cluster runs.

$$\frac{0.05K}{\text{core}} * \frac{20}{1}\frac{\text{core}}{\text{node}} = \frac{0.05K * 20}{\text{node}} = \frac{1K}{\text{node}}. \tag{C.1}$$

This means that you can always calculate the "temperature step size" T_{STEP} with the simple equation,

$$T_{\text{STEP}} = C\Delta T, \tag{C.2}$$

where C is the number of computer cores, and ΔT is the step size of the overall temperature range.

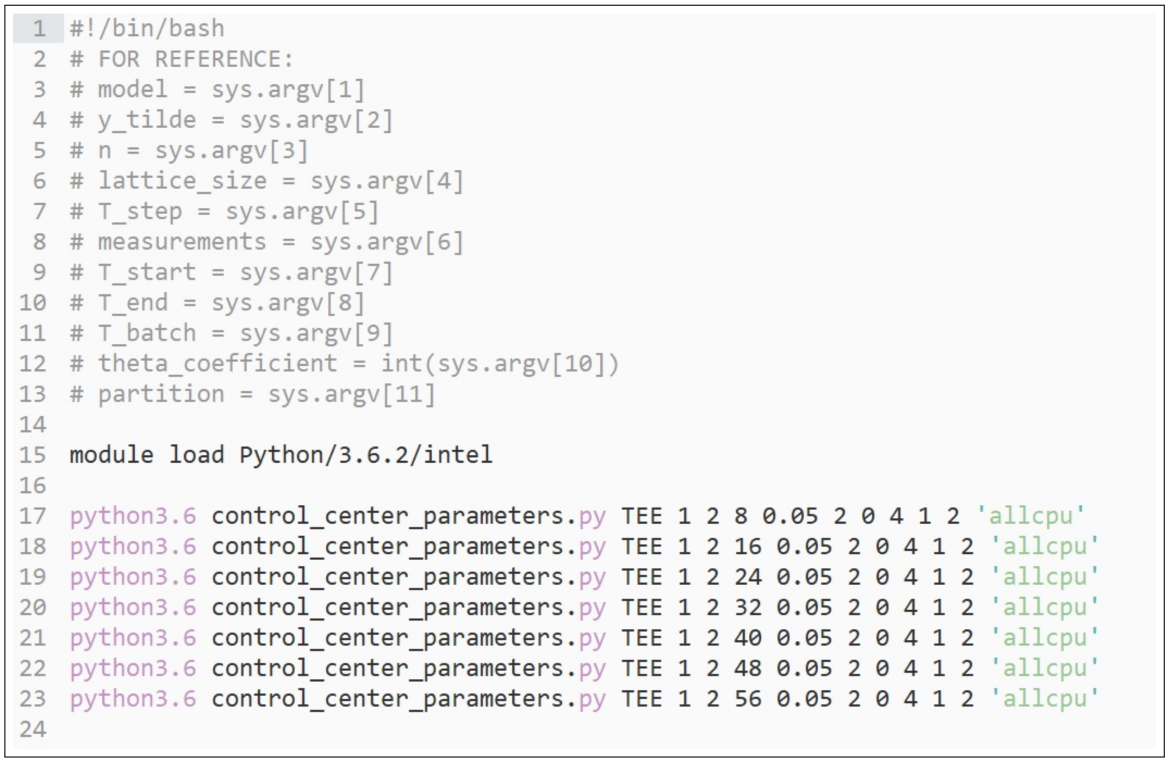
After this, you will be asked a couple more questions, and finally whether or not you want to start the simulation. If you answer 'yes', then the code will automatically submit the jobs you requested. Then wait until these jobs are finished. Type 'squeue' to check the SLURM queue to see how long your jobs have been running. After the jobs finish (be it hours or days) then navigate back to the Control Center folder, and execute the end simulation script with:

```
> sh end_simulation.sh
```

This will ask you what model to aggregate data for, because since we submitted four jobs, each of which that only runs $\approx \frac{20}{79}$ of the total temperatures we have four data files left at the end. This end simulation script will automatically aggregate these and deposit the aggregated file in the "FINISHED DATA" folder.

Running Multiple Parameter Simulations at Once

If you answer 'no' to the very first question the program asks you about inputting parameters manually, then the Control Center will read parameters from the "parameter_run.sh" file. This is a file that lists many simulation parameters and then submits jobs for all of them. To show you the power of this code, I routinely submitted 112 jobs just by typing 'no', using this method. Below is an example of how this file should be formatted:



```bash
#!/bin/bash
# FOR REFERENCE:
# model = sys.argv[1]
# y_tilde = sys.argv[2]
# n = sys.argv[3]
# lattice_size = sys.argv[4]
# T_step = sys.argv[5]
# measurements = sys.argv[6]
# T_start = sys.argv[7]
# T_end = sys.argv[8]
# T_batch = sys.argv[9]
# theta_coefficient = int(sys.argv[10])
# partition = sys.argv[11]

module load Python/3.6.2/intel

python3.6 control_center_parameters.py TEE 1 2 8 0.05 2 0 4 1 2 'allcpu'
python3.6 control_center_parameters.py TEE 1 2 16 0.05 2 0 4 1 2 'allcpu'
python3.6 control_center_parameters.py TEE 1 2 24 0.05 2 0 4 1 2 'allcpu'
python3.6 control_center_parameters.py TEE 1 2 32 0.05 2 0 4 1 2 'allcpu'
python3.6 control_center_parameters.py TEE 1 2 40 0.05 2 0 4 1 2 'allcpu'
python3.6 control_center_parameters.py TEE 1 2 48 0.05 2 0 4 1 2 'allcpu'
python3.6 control_center_parameters.py TEE 1 2 56 0.05 2 0 4 1 2 'allcpu'

```

Figure C.6: There is a particular order the parameters must be in, displayed in the top.

The parameter order is as follows:

1. Model name ('QCD', 'TEE', 'XY')
2. $\tilde{y}$ in equation (B.1)
3. Lattice number n
4. ΔT
5. Number of independent measurements
6. Starting temperature (explained in previous section)
7. Ending temperature
8. Job temperature step size
9. p in equation (B.1)
10. The partition to run the jobs on

As you can see, there is another script called "control_center_parameters.py" that will take these parameters as arguments and automate the parameter setting process you went through in the previous section when you input parameters manually. Depending on how many jobs you run, you still finish aggregating them all by calling "end_simulation.sh". Once you finish aggregating then you can analyze the data by graphing and calculating quantities with some of the functions in the "entanglement" module I created, or with your own!

Happy programming!

Listing C.1: The 'Control_Center.py' code

```
import os, io
print("STEP 1: Enter basic simulation parameters below...")
if input("Would you like to input the parameters manually? ") == 'yes
    ↪ ':
    # gets the model variety
    models = {'QCD': 'QCD_Model', 'XY': 'XY_Model', 'TEE': 'TEE_Calc'}

    model = input("Enter model to simulate: ")
    while model not in models:
        print("Valid models are: ", list(models.keys()))
        model = input("Invalid model name. Enter model to simulate: ")

    if model == 'XY':
        y_tilde = 0
    else:
        y_tilde = int(input("Enter y_tilde: "))

    # gets lattice number
    valid_n = [2, 3, 4]
    n = int(input("Enter replica number: "))
    while n not in valid_n:
        n = int(input("Invalid replica number. Enter replica number: "
            ↪ ))

    # gets the lattice size
    lattice_size = int(input("Lattice size: "))
    while lattice_size % 8 != 0:
        lattice_size = int(input("Enter a multiple of 8 as a lattice
            ↪ size: "))

    # gets T_step
    T_step = float(input("Enter T_step: "))

    # gets measurements
    measurements = int(input("Enter monte carlo measurements: "))

    # gets theta coefficient
    if model == 'XY':
        theta_coefficient = 0
    else:
        theta_coefficient = int(input("Enter a theta coefficient: "))

    # partition on cluster
    partitions = {'medium': '4-00:00:00', 'long': '20-00:00:00', 'phi'
        ↪ : '20-00:00:00', 'allcpu':'4:00:00', 'short': '2:00:00'}
    partition = input("Enter partition to run simulation on: ")
    while partition not in partitions:
```

```python
        print("Valid partitions are: ", list(partitions.keys()))
        partition = input("Invalid partition. Enter cluster partition: ")

    time_limit = partitions[partition]


else:
    os.system("sh parameter_run.sh")
    quit()

calculation_file = "N={0},{1}M,{2}dT,n={3}".format(lattice_size, measurements, T_step, n)
path = os.path.dirname(os.path.realpath(__file__))

print("\nSTEP 2: Ready to populate the calculation file: ", calculation_file)
T_start = float(input("Enter a starting T: "))
T_end = float(input("Enter an ending T: "))
T_batch = float(input("Enter a temperature stepsize for T = {0}-{1}: ".format(T_start, T_end)))
# runs generate.py to populate the calculation file full of batch scripts
os.system('python3.6 important_scripts/generate.py {0} {1} {2} {3} "{4}" "{5}" "{6}" {7} {8} {9} {10} {11} {12} {13}'.format(lattice_size, measurements, T_step, n, path, model, models[model], y_tilde, T_start, T_end, T_batch, theta_coefficient, partition, time_limit))

print("\nCalculation file populated!")
while input("\nSTEP 3: Is the monte carlo script up to date? ") != 'yes':
    print("Update the script and enter 'yes' to continue...")

os.system('chmod +x {0}/{1}/{2}dT/{3}/run_it_all.sh'.format(path, models[model], T_step, calculation_file))

condition = input("Would you like to run the simulation now? ")

while condition != 'yes':
    if condition == 'no':
        break

    else:
        print("Enter 'yes' to run simulation")
if condition == 'yes':
    os.system('sh {0}/{1}/{2}dT/{3}/run_it_all.sh'.format(path, models[model], T_step, calculation_file))
```

C.2 Batch Script Generator Code

This second code was used to divide the temperatures I needed to run into manageable chunks for the cluster. It is compatible with the SLURM scheduler. This code is to be used with the above `Control_Center.py` file. It should be placed in a directory called "important_scripts" in the same directory as the `Control_Center.py` file.

Listing C.2: 'generate.py' code used for running jobs on the cluster

```
import os, io, numpy, sys, shutil
# T_min is the starting T, T_cutoff is the ending T. step_size is the
   ↪  RMI temp step size, and chunk is how many temperatures a
   ↪ singular call of RMI_Cluster.py should calculate.
N = int(sys.argv[1])
measurements = int(sys.argv[2])
T_size = float(sys.argv[3])
n = int(sys.argv[4])
path = sys.argv[5]
model = sys.argv[6]
model_folder = sys.argv[7]
y_tilde = float(sys.argv[8])
theta_coefficient = int(sys.argv[12])
partition = sys.argv[13]
time_limit = sys.argv[14]

# The Script name format depending on the model
scripts = {'XY': 'RMI_{0}_Cluster_n={1}.py'.format(model, n), 'QCD':
   ↪ 'RMI_{0}_Cluster_n={1}.py'.format(model, n), 'TEE': '
   ↪ Multiple_Shapes.py'}

packed_folder = 'N={0},{1}M,{2}dT,n={3}'.format(N, measurements,
   ↪ T_size, n)
whole_path = '{0}/{1}/{2}dT/{3}'.format(path, model_folder, T_size ,
   ↪ packed_folder)
folder_path = '{0}/{1}/{2}dT/{3}'.format(path, model_folder, T_size ,
   ↪  packed_folder)

if not os.path.exists(folder_path):
    os.makedirs(folder_path)
else:
    shutil.rmtree(folder_path) #removes all the subdirectories!
    os.makedirs(folder_path)


def float_array(T_min, T_max, T_step, check_num='yes'):
    number = int(round((T_max - T_min), 4) / T_step)
    if check_num == 'yes':
        print("Going to try this as num: ", number)
    array = numpy.linspace(T_min, T_max, number, endpoint=False)
    return array
```

```python


def temperature_spreader(T_min, T_cutoff, step_size):
    range_list = []
    for T in float_array(T_min, T_cutoff, step_size, check_num='no'):
        if T == list(float_array(T_min, T_cutoff, step_size, check_num='no'))[-1]:
            range_list.append((T, T_cutoff))
        else:
            range_list.append((T, T + step_size))
    return range_list

T_start = float(sys.argv[9])
T_end = float(sys.argv[10])
T_batch = float(sys.argv[11])

temp_list = temperature_spreader(T_start, T_end, T_batch)

print("T_min, T_max pairs: \n", numpy.array(temp_list))

script_number = 1
for temp_pair in temp_list:
    with io.open('{0}/{4}dT/{1}/{2}{3}_submit.sh'.format(model_folder, packed_folder, model, script_number, T_size), 'w', newline='\n') as batchscript:
        write_T_min = round(temp_pair[0], 1)
        write_T_max = round(temp_pair[1], 1)
        # This generates the batch script!
        batchscript.write(
            "#!/bin/bash \n#SBATCH --job-name=N,n={0},{1}_{2}to{3} \n#SBATCH --output={9}/RMI_{4}{2}to{3}.txt \n#SBATCH --nodes 1 \n#SBATCH --cpus-per-task=20 \n#SBATCH --ntasks-per-node 1 \n#SBATCH --time={11} \n#SBATCH --mem-per-cpu=MaxMemPerCPU \n#SBATCH --partition {10} \nsrun python3.6 {13}/{5}/Master\ Codes/{14} {0} {6} {7} {2} {3} {8} {12} {13}".format(N, n, write_T_min, write_T_max, model, model_folder, T_size, measurements, y_tilde, whole_path, partition, time_limit, theta_coefficient, path, scripts[model]))
    script_number += 1

# creates a script that allows me to run all the scripts
with io.open('{0}/{1}dT/{2}/run_it_all.sh'.format(model_folder, T_size, packed_folder), 'w', newline='\n') as run_file:
    run_file.write('#!/bin/bash\n')
    for script_number in range(1, len(temp_list) + 1):
        run_file.write('sbatch {2}/{0}{1}_submit.sh \nsleep 0.01\n'.format(model, script_number, whole_path))
```

```

# creates a script that allows me to cancel all the scripts
with io.open('{0}/{1}dT/{2}/emergency_halt.sh'.format(model_folder,
    ↪ T_size, packed_folder), 'w', newline='\n') as run_file:
    run_file.write('#!/bin/bash\n')
    for temp_pair in (temp_list):
        T_min = round(temp_pair[0], 1)
        T_max = round(temp_pair[1], 1)
        run_file.write('scancel N,n={0},{1}_{2}to{3} \nsleep 0.01\n'.
            ↪ format(N, n, T_min, T_max))

# Summary of task
print("\nCreated a folder named \n'{0}/{1}/{2}' \nwith {3} 'sbatch'
    ↪ scripts inside.".format(model_folder, T_size, packed_folder,
    ↪ script_number))
```

C.3 Aggregation Code

This code is for after a simulation runs, it is to be used with the Control Center code in the previous section. This code takes multiple data files over a spread of temperatures and aggregates them back into one data file, as described in section 4.3.

C.3.1 Running

To use this code, usually you would just type `sh end_simulation.sh` in the Control_Center folder. However, if you want to just aggregate a single data folder, then you must fill in the appropriate parameters for the data you are aggregating on lines 3-8. This version is a bit different than the version used in the Control_Center code. Then enter the date of the data in the `data_of_data` variable. This is the date that is in the outputted data file folder. For example, if you ran the QCD code in appendix B, you would get a folder titled "Data from RMI QCD; 2017-12-13; 20000, 0.05, 32, n=2, y 0.0, theta=4".

The `model_input` variable are the directories that the above folder is stored in. It is convenient to keep all the different data files in one directory, in this case 'QCD Model Raw Data'. `model_output` is the directory that the code will deposit the aggregated data file.

Note that you must create the "QCD Model Raw Data" and "QCD Model Aggregate Data" files manually. When finished, this code will output one data file with a name like so: "RMI QCD; 2017-12-13; 20000, 0.0, 4.0, 0.05, 32, n=2, y 0.0, theta=4.txt" After this aggregation, the above file will function normally as a complete data file, and can be used for calculations.

Listing C.3: This is the code that aggregates different temperature ranges together after the simulations all complete.

```
1 import os, numpy, time, math, pylab, datetime
2 # parameters for calculating RMI
3 lattice_size = 32
```

```
T_step = 0.05
n = 2
y_tilde = 1
theta = 4
measurements = 2
# This finds the correct files to aggregate
date_of_data = '2018-01-28'
model_input = 'QCD Model Raw Data'
model_output = 'QCD Model Aggregate Data'
model = 'QCD'
path = os.path.dirname(os.path.realpath(__file__))

data_directory = '{0}/{1}/Data from RMI QCD; {2}; {8}, {3}, {4}, n
    ↪ ={5}, y~{6}, theta={7}'.format(path, model_input, date_of_data,
    ↪  T_step, lattice_size, n, y_tilde, theta, measurements)


# master plots for bringing everything together
Master_T_plot = []
Master_E_replica = []
Master_sigma_replica = []
Master_E_A_U_B = []
Master_sigma_A_U_B = []
Master_E_normal = []
Master_sigma_normal = []

derived_quant = 'no'
Master_cap = []
Master_sigma_cap = []
Master_susc = []
Master_sigma_susc = []

magnetization = 'no'
Master_mag = []
Master_sigma_mag = []


contents = os.listdir(data_directory)
print(contents)


# reorders the listdir to go from 0 to 100 temperature.
def ordering(data_name):
    name_split = data_name.split(',')
    T_min = float(name_split[1])
    T_max = float(name_split[2])
    rank = T_min + T_max
    return float(rank)

```

```

# calculates RMI
def RMI_calc(Data, N_global, T_step, graph='no'):
    global date, n
    alpha = n
    t1 = time.time()
    T_plot = Data[0]
    # Gathers the replica data
    E_replica = Data[1]
    sigma_replica = Data[2]
    # Gathers the normal data
    E_A_U_B = Data[3]
    sigma_A_U_B = Data[4]

    E_normal = Data[5]
    sigma_normal = Data[6]
    # Calculating RMI for each T
    print('Working on Renyi Mutual Information...')
    count = len(E_A_U_B)

    RMI_plot = []
    RMI_sigma_plot = []
    deltaT = T_step
    # Calculates the RMI and the sigma for each RMI(T)
    for i in range(count):
        RMI = 0.0
        sigma_sigma_i = 0.0
        for j in range(i, count):
            term_j = deltaT * (2 * (E_replica[j]) - (E_A_U_B[j]) -
                ↪ alpha * E_normal[j]) / ((T_plot[j]) ** 2)
            RMI += term_j
            # Now to propagate the error from each E measurement...
            sigma_sigma_j = ((2 * deltaT) / ((T_plot[j] ** 2) *
                ↪ N_global * 2)) ** 2 * (sigma_replica[j] ** 2) + (
                ↪ deltaT / ((
                T_plot[j] ** 2) * N_global * 2)) ** 2 * (sigma_A_U_B[j]
                    ↪  ** 2) + ((2 * deltaT) / ((T_plot[j] ** 2) *
                    ↪ N_global * 2)) ** 2 * (sigma_normal[j] ** 2)
            sigma_sigma_i += sigma_sigma_j
        sigma_i = math.sqrt(sigma_sigma_i)
        RMI /= 2 * N_global
        RMI_plot.append(RMI)
        RMI_sigma_plot.append(sigma_i)
        if i % 100 == 0:
            print('Calculating RMI for T =', i * T_step)

    if graph == 'yes' or graph == 'plot':
        pylab.plot(T_plot, RMI_plot, 'b', linewidth=3)
        pylab.errorbar(T_plot, RMI_plot, yerr=RMI_sigma_plot, capsize
            ↪ =2, ecolor='r')
        pylab.title(r'RMI vs $T$; $T_{step}$' + ' = {0};'.format(
            ↪ T_step) + ' $T_{max}$' + ' = 100 ', fontsize=16)
        pylab.xlabel(r'$T$', fontsize=16)
        pylab.ylabel(r'$\frac{I_2(T)}{\ell}$', fontsize=16)
        pylab.xlim(0, 10)
        # pylab.ylim(0, 0.5)
        t_elapse = (time.time() - t1) / 60
        print('Done in {0:.3f} minutes'.format(t_elapse))
        if graph == 'plot':
            pylab.show()


    return T_plot, RMI_plot, RMI_sigma_plot


def RMI_calc_QCD(Data, N_global, T_step, graph='no'):
    global date, n
    alpha = n
    t1 = time.time()
    T_plot = Data[0]
    # Gathers the replica data
    E_replica = Data[1]
    sigma_replica = Data[2]
    # Gathers the normal data
    E_A_U_B = Data[3]
    sigma_A_U_B = Data[4]

    E_normal = Data[5]
    sigma_normal = Data[6]
    # Calculating RMI for each T
    print('Working on Renyi Mutual Information...')
    count = len(E_A_U_B)

    RMI_plot = []
    RMI_sigma_plot = []
    deltaT = T_step
    # Calculates the RMI and the sigma for each RMI(T)
    for i in range(count):
        RMI = 0.0
        sigma_sigma_i = 0.0
        for j in range(0, i):
            term_j = deltaT * (2 * (E_replica[j]) - (E_A_U_B[j]) -
                ↪ alpha * E_normal[j])
            RMI += term_j
            # Now to propagate the error from each E measurement...
            sigma_sigma_j = ((2 * deltaT) / (N_global * 2)) ** 2 * (
                ↪ sigma_replica[j] ** 2) + (deltaT / ( N_global * 2))
                ↪ ** 2 * (sigma_A_U_B[j] ** 2) + ((alpha * deltaT) / (
                ↪  N_global * 2)) ** 2 * (sigma_normal[j] ** 2)
            sigma_sigma_i += sigma_sigma_j
        sigma_i = math.sqrt(sigma_sigma_i)
        RMI /= 2 * N_global
        RMI_plot.append(RMI)
        RMI_sigma_plot.append(sigma_i)
        if i % 100 == 0:
            print('Calculating RMI for T =', i * T_step)

    if graph == 'yes' or graph == 'plot':
        pylab.plot(T_plot, RMI_plot, 'b', linewidth=3)
        pylab.errorbar(T_plot, RMI_plot, yerr=RMI_sigma_plot, capsize
            ↪ =2, ecolor='r')
        pylab.title(r'RMI vs $T$; $T_{step}$' + ' = {0};'.format(
            ↪ T_step) + ' $T_{max}$' + ' = 100 ', fontsize=16)
        pylab.xlabel(r'$T$', fontsize=16)
        pylab.ylabel(r'$\frac{I_2(T)}{\ell}$', fontsize=16)
        pylab.xlim(0, 10)
        # pylab.ylim(0, 0.5)
        t_elapse = (time.time() - t1) / 60
        print('Done in {0:.3f} minutes'.format(t_elapse))
        if graph == 'plot':
            pylab.show()

    return T_plot, RMI_plot, RMI_sigma_plot


# iterates over all chunk files
for data_files in sorted(contents, key=ordering):
    print(data_files)
    prefix = data_files.split(',')[0]
    suffix = ',' + data_files.split(',')[3] + ',' + data_files.split('
        ↪ ,')[4] + ',' + data_files.split(',')[5] + ',' + data_files.
        ↪ split(',')[6] + ',' + data_files.split(',')[7]
    T_max = float(data_files.split(',')[2])

    data_chunk = numpy.loadtxt('{0}/{1}'.format(data_directory,
        ↪ data_files))
    # Gathers temperatures and adds them to the master list
    T_plot = list(data_chunk[0])
    Master_T_plot += T_plot
    # gathers the different energies and adds them to the master list
    E_replica = list(data_chunk[1])
    sigma_replica = list(data_chunk[2])
    Master_E_replica += E_replica
    Master_sigma_replica += sigma_replica

    E_A_U_B = list(data_chunk[3])
    sigma_A_U_B = list(data_chunk[4])
    Master_E_A_U_B += E_A_U_B
    Master_sigma_A_U_B += sigma_A_U_B

    E_normal = list(data_chunk[5])
    sigma_normal = list(data_chunk[6])
    Master_E_normal += E_normal
    Master_sigma_normal += sigma_normal

    if magnetization == 'yes':
        mag = list(data_chunk[7])
        sigma_mag = list(data_chunk[8])
        Master_mag += mag
        Master_sigma_mag += sigma_mag
    if derived_quant == 'yes':
        cap_chunk = list(data_chunk[9])
        sigma_cap = list(data_chunk[10])
        Master_cap += cap_chunk
        Master_sigma_cap += sigma_cap

        susc_chunk = list(data_chunk[11])
        sigma_susc = list(data_chunk[12])
        Master_susc += susc_chunk
        Master_sigma_susc += sigma_susc

if magnetization == 'yes':
    pre_aggregate_data = numpy.array([Master_T_plot, Master_E_replica,
        ↪  Master_sigma_replica, Master_E_A_U_B, Master_sigma_A_U_B,
        ↪ Master_E_normal, Master_sigma_normal, Master_mag,
        ↪ Master_sigma_mag, Master_cap, Master_sigma_cap, Master_susc
        ↪ , Master_sigma_susc], float)
    if derived_quant == 'yes':
        pre_aggregate_data = numpy.array(
        [Master_T_plot, Master_E_replica, Master_sigma_replica,
            ↪ Master_E_A_U_B, Master_sigma_A_U_B, Master_E_normal,
         Master_sigma_normal, Master_cap, Master_sigma_cap,
             ↪ Master_susc, Master_sigma_susc], float)
else:
    pre_aggregate_data = numpy.array(
        [Master_T_plot, Master_E_replica, Master_sigma_replica,
            ↪ Master_E_A_U_B, Master_sigma_A_U_B, Master_E_normal,
         Master_sigma_normal, Master_mag, Master_sigma_mag, Master_cap
             ↪ , Master_sigma_cap, Master_susc,
         Master_sigma_susc], float)

print("The T_max is: ", T_max)
confirmation = numpy.arange(0 + T_step, T_max, T_step)
print(suffix)
final_filename = prefix + ', 0.0, {}'.format(T_max) + suffix
print(final_filename)
```

```python
if len(confirmation) == len(Master_T_plot):
    print("The experimental and control T_plots have the same length! Calculating RMI...")

    if model == 'QCD':
        RMI_data = RMI_calc_QCD(pre_aggregate_data, lattice_size, T_step)
    else:
        RMI_data = RMI_calc(pre_aggregate_data, lattice_size, T_step)

    RMI = RMI_data[1]
    RMI_sigma = RMI_data[2]
    RMI_add = numpy.array([RMI, RMI_sigma])
    aggregate_data = numpy.vstack((pre_aggregate_data, RMI_add))

    print("Saving a file...")

    numpy.savetxt(model_output + '/' + final_filename, aggregate_data, header='This data was aggregated on {}'.format(datetime.datetime.today()))

    print("Saved a file: {0}/{1}".format(model_output, final_filename))
else:
    print("\nThe experimental and control T_plots are different lengths! File save suspended!")
    print("It should be {0} but it is {1} instead.".format(len(confirmation), len(Master_T_plot)))
    print("E_normal length = ", len(Master_E_normal), "\nE_replica length = ", len(Master_E_replica), "\nE_AUB length = ", len(Master_E_A_U_B))

    condition = input("Continue with file save?\n")
    if condition == 'yes' or condition == 'y' or condition == 'Yes':
        print("Calculating RMI...")
        RMI_data = RMI_calc(pre_aggregate_data, lattice_size, T_step)
        RMI = RMI_data[1]
        RMI_sigma = RMI_data[2]
        RMI_add = numpy.array([RMI, RMI_sigma])
        aggregate_data = numpy.vstack((pre_aggregate_data, RMI_add))

        print("Saving a file...")

        numpy.savetxt(model_output + '/' + final_filename, aggregate_data,
                      header='This data was aggregated on {}'.format(datetime.datetime.today()))
        print("Saved a file: {0}/{1}".format(model_output, final_filename))
```

Appendix D

Topological Entanglement Entropy Code

This code is the first draft of a Monte Carlo simulation that would calculate the quantity known as topological entanglement entropy (TEE). To use this code, enter the parameters in the command line arguments. The order of these are as follows:

1. Lattice size N
2. ΔT
3. Independent measurement number
4. Starting temperature
5. Ending temperature
6. $\tilde{y}$ from the Hamiltonian B.1
7. p from B.1
8. The output path for the data. This is a path that the code will save the output data to.

Then to call the function in the command prompt, enter the starting and stopping temperatures like so:

```
> python TEE_QCD.py 0.0 1.0
```

```python
from numpy import ones, arange, sqrt, array, savetxt, vstack, zeros
from math import exp, pi, cos, sin
from random import random, randrange
from multiprocessing import Pool
import time, sys, os, datetime, numpy
date = datetime.date.today()

N_global = int(sys.argv[1].split(',')[0])
T_step = float(sys.argv[2].split(',')[0])
output_path = sys.argv[8].split(',')[0]

```

```python
if N_global == 8:
    tau_global = 10000
if N_global == 16:
    tau_global = 15000
if N_global == 24:
    tau_global = 60000
if N_global == 32:
    tau_global = 100000
if N_global == 40:
        tau_global = 150000
if N_global == 48:
    tau_global = 200000
if N_global == 56:
    tau_global = 210000

E_measurements = int(sys.argv[3].split(',')[0])
print("Using {} Measurements".format(E_measurements))

y_tilde = float(sys.argv[6].split(',')[0])
theta_coefficient = int(sys.argv[7].split(',')[0])


def region(i_start, i_end):
    return list(range(int(i_start), int(i_end)))


def QCD_E(T):
    global N_global, E_measurements, tau_after, y_tilde,
       ↪ theta_coefficient
    kappa = 4 * pi
    #J = - (8 * T) / (pi * kappa)
    J = 1
    N = N_global # The lattice size: NxN
    tau = tau_global # The correlation time
    # if T > 20:
    # tau = tau_after
    BM = E_measurements # Number of independent measurements for the
       ↪ bootstrap analysis
    steps = 2 * tau * BM # Number of times the program will run
    E = -2 * (N * N) - y_tilde * (N * N) # Initial Value of Energy
       ↪ since all spins start pointed up at \theta_i = 0.0
    m_1 = N*N # Initial value of magnetization
    m_2 = 0
    L = zeros([N, N], float) # Generates the lattice where each entry
       ↪ is a value of \theta_i

    # JT = J / T # The parameter J divided by T (temperature)
       ↪ Multiplying dE by this will potentially save time,
    # but if this is done, make sure to multiply dE by T again when
```

```python
doing E += dE/(N*N)

    print("N=", N, "; Normal QCD XY-Model at T=", T)

    expE = 0.0 # Expectation value of E
    expM = 0.0
    measurements = [] # List of Measurements
    M_measurements = []
    # Main Monte Carlo cycle
    for x in range(steps + 1):
        i = randrange(0, N)
        j = randrange(0, N) # Picks a random starting location

        # Decides an anticipated spin amount
        L_update = random() * 2 * pi
        # Calculates change in energy that would occur if this spin was accepted
        dE = 0.0
        # Starts calculating the nearest neighbor sum at location L[ i -1 , j]
        neighbor = i - 1
        if neighbor > -1:
            dE += cos(L_update - L[neighbor, j]) - cos(L[i, j] - L[neighbor, j]) # Checks if the neighbor is within the lattice
        else:
            dE += cos(L_update - L[N - 1, j]) - cos(L[i, j] - L[N - 1, j]) # Periodic boundary conditions
        neighbor = i + 1
        if neighbor < N:
            dE += cos(L_update - L[neighbor, j]) - cos(L[i, j] - L[neighbor, j])
        else:
            dE += cos(L_update - L[0, j]) - cos(L[i, j] - L[0, j])
        neighbor = j - 1
        if neighbor > -1:
            dE += cos(L_update - L[i, neighbor]) - cos(L[i, j] - L[i, neighbor])
        else:
            dE += cos(L_update - L[i, N - 1]) - cos(L[i, j] - L[i, N - 1])
        neighbor = j + 1
        if neighbor < N:
            dE += cos(L_update - L[i, neighbor]) - cos(L[i, j] - L[i, neighbor])
        else:
            dE += cos(L_update - L[i, 0]) - cos(L[i, j] - L[i, 0])
        dE *= -J
        dE += y_tilde * (cos(theta_coefficient * L[i, j]) - cos(
            ↪ theta_coefficient * L_update))

        # Calculates whether L[i,j] rotates
        R = exp(-dE * T)
        if R > 1 or random() < R:
            m_1 = m_1 + cos(L_update) - cos(L[i, j])
            m_2 = m_2 + sin(L_update) - sin(L[i, j])
            L[i, j] = L_update
            E += dE # / (N * N)

        if x != 0 and x % (2 * tau) == 0:
            expE += E
            M = sqrt(m_1**2 + m_2**2)
            expM += M
            # print("at x = ",x," ", expE)
            measurements.append(E) # Adds the measurement to the list
            M_measurements.append(M)
    expE /= BM
    expM /= BM
    # print(expE)

    # The Bootstrap Error Analysis
    resample = BM # times to repeat re-sampling
    B_i = [] # for the calculation of <B> and sigma
    M_i = []
    for y in range(resample):
        B = 0.0
        M_error = 0.0
        for z in range(int(BM)):
            n = randrange(0, BM)
            B += measurements[n]
            M_error += M_measurements[n]
        B /= BM
        M_error /= BM
        B_i.append(B)
        M_i.append(M_error)

    # Now to calculate the Bootstrap sigma
    sigma_sigma = 0.0
    sigma_sigma_M = 0.0
    for w in range(resample):
        sigma_sigma += (B_i[w] - expE) ** 2
        sigma_sigma_M += (M_i[w] - expM) ** 2
    sigma_sigma /= resample
    sigma_sigma_M /= resample
    sigma_bootstrap = sqrt(sigma_sigma)
    sigma_bootstrap_M = sqrt(sigma_sigma_M)

    return [T, expE, sigma_bootstrap, M, sigma_bootstrap_M] # This
```

```python
will create a results matrix which can be plotted


# The parallel sections
def Region_1(T):
    global N_global, E_measurements, tau_global, tau_after
    J = 1
    N = N_global # The lattice size: NxN
    # A test to make things quicker; higher temperatures equilibrate faster
    tau = tau_global

    BM = E_measurements # Number of independent measurements for the bootstrap analysis
    steps = 2 * tau * BM # Number of times the program will run
    E = -4 * (N * N) - 2 * y_tilde * (N * N) # Initial Value of Energy  since all spins start pointed up at \theta_i = 0.0
    boundary = N // 2
    L1 = zeros([N, N], float) # Lattice 1 where each entry is a value of \theta_i
    L2 = zeros([N, N], float) # Lattice 2
    A_1 = [list(range(int(N/8), int(N/8 + .75 * N))), list(range(int(N/8), int(N/8 + N/4)))]
    A_2 = [list(range(int(N/8), int(N/8) + int(.75 * N))), list(range(int(N/8 + N/2), int(3*N/8 + N/2)))]
    B_1 = L1[:, boundary: N]
    B_2 = L2[:, boundary: N]

    lattice_test = 'no'
    if lattice_test == 'yes':
        print("PERFORMING LATTICE TEST...DO NOT COLLECT DATA!")
        for i in range(N):
            for j in range(N):
                if i in A_1[0] and j in A_1[1]:
                    L1[i, j] = 1
                    L2[i, j] = 1
                if i in A_2[0] and j in A_2[1]:
                    L1[i, j] = 2
                    L2[i, j] = 2
        print(L1)
        print(L2)

    print("N=", N, "; Region 1 at T=", T)

    expE = 0.0 # Expectation value of E
    measurements = [] # List of Measurements
    # Main Monte Carlo cycle
    for x in range(steps + 1):
        i = randrange(0, N)
        j = randrange(0, N) # Picks a random starting location

        # Decides an anticipated spin amount
        L1_update = random() * 2 * pi
        L2_update = random() * 2 * pi

        if i in A_1[0] and j in A_1[1]:
                    L1_update = L2_update
        if i in A_2[0] and j in A_2[1]:
                    L1_update = L2_update
        # Calculates change in energy that would occur if this spin was accepted
        dE = 0.0
        # Starts calculating the nearest neighbor sum at location L[ i -1 , j]
        neighbor = i - 1
        if neighbor > -1:
            dE += cos(L1_update - L1[neighbor, j]) - cos(L1[i, j] - L1[neighbor, j])
            dE += cos(L2_update - L2[neighbor, j]) - cos(L2[i, j] - L2[neighbor, j])
        else:
            dE += cos(L1_update - L1[N - 1, j]) - cos(L1[i, j] - L1[N - 1, j]) # Periodic boundary conditions
            dE += cos(L2_update - L2[N - 1, j]) - cos(L2[i, j] - L2[N - 1, j])
        neighbor = i + 1
        if neighbor < N:
            dE += cos(L1_update - L1[neighbor, j]) - cos(L1[i, j] - L1[neighbor, j])
            dE += cos(L2_update - L2[neighbor, j]) - cos(L2[i, j] - L2[neighbor, j])
        else:
            dE += cos(L1_update - L1[0, j]) - cos(L1[i, j] - L1[0, j])
            dE += cos(L2_update - L2[0, j]) - cos(L2[i, j] - L2[0, j])
        neighbor = j - 1
        if neighbor > -1:
            dE += cos(L1_update - L1[i, neighbor]) - cos(L1[i, j] - L1[i, neighbor])
            dE += cos(L2_update - L2[i, neighbor]) - cos(L2[i, j] - L2[i, neighbor])
        else:
            dE += cos(L1_update - L1[i, N - 1]) - cos(L1[i, j] - L1[i, N - 1])
            dE += cos(L2_update - L2[i, N - 1]) - cos(L2[i, j] - L2[i, N - 1])
        neighbor = j + 1
        if neighbor < N:
            dE += cos(L1_update - L1[i, neighbor]) - cos(L1[i, j] - L1
[i, neighbor])
            dE += cos(L2_update - L2[i, neighbor]) - cos(L2[i, j] - L2[i, neighbor])
        else:
            dE += cos(L1_update - L1[i, 0]) - cos(L1[i, j] - L1[i, 0])
            dE += cos(L2_update - L2[i, 0]) - cos(L2[i, j] - L2[i, 0])
        dE *= -J
        dE += y_tilde * (cos(theta_coefficient * L1[i, j]) - cos(theta_coefficient * L1_update)) + y_tilde * (cos(theta_coefficient * L2[i, j]) - cos(theta_coefficient * L2_update))

        # Calculates whether L[i,j] rotates
        R = exp(-dE * T)
        if R > 1 or random() < R:
            L1[i, j] = L1_update
            L2[i, j] = L2_update
            E += dE # / (N * N)
        if x != 0 and x % (2 * tau) == 0:
            expE += E
            # print("at x = ",x," ", expE)
            measurements.append(E) # Adds the measurement to the list

    expE /= BM
    # print(expE)

    # The Bootstrap Error Analysis
    resample = BM # times to repeat re-sampling
    B_i = [] # for the calculation of <B> and sigma
    for y in range(resample):
        B = 0.0
        for z in range(int(BM)):
            n = randrange(0, BM)
            B += measurements[n]
        B /= BM
        B_i.append(B)

    # Now to calculate the Bootstrap sigma
    sigma_sigma = 0.0
    for w in range(resample):
        sigma_sigma += (B_i[w] - expE) ** 2
    sigma_sigma /= resample
    sigma_bootstrap = sqrt(sigma_sigma)

    # This is a test to make sure that A_1 and A_2 are indeed being updated the same.
    equivalence_test = 'no'
    if equivalence_test == 'yes':
        matches = 0.0
        for columns in range(N):
            for rows in range(N):
                if L1[rows, columns] == L2[rows, columns]:
                    matches += 1
                # if rows in A_1[0] and columns in A_1[1]:
                # if L1[rows, columns] == L2[rows, columns]:
                # matches += 1
                # if rows in A_2[0] and columns in A_2[1]:
                # if L1[rows, columns] == L2[rows, columns]:
                # matches += 1
        print(matches)
        if matches == 2 * int(.75 * N * N/4):
            print("A_1 and A_2 match!")
        else:
            print("We messed up somewhere :(")

    return [T, expE, sigma_bootstrap] # This will create a results matrix which can be plotted


# The big U
def Region_2(T):
    global N_global, E_measurements, tau_global, tau_after
    J = 1
    N = N_global # The lattice size: NxN
    # A test to make things quicker; higher temperatures equilibrate faster
    tau = tau_global

    BM = E_measurements # Number of independent measurements for the bootstrap analysis
    steps = 2 * tau * BM # Number of times the program will run
    E = -4 * (N * N) - 2 * y_tilde * (N * N) # Initial Value of Energy  since all spins start pointed up at \theta_i = 0.0
    boundary = N // 2
    L1 = zeros([N, N], float) # Lattice 1 where each entry is a value of \theta_i
    L2 = zeros([N, N], float) # Lattice 2

    A_1 = [region(N/8, N/8 + .75 * N), region(N/8 + N/2, 3*N/8 + N/2) + region(N/8, N/8 + N/4)]
    A_2 = [region(N/8, 3*N/8), region(3*N/8, 5*N/8)]

    # A_2 = [list(range(int(N/8), int(N/8 + .75 * N))), list(range(int(N/8), int(N/8 + N/4))) + list(range(int(N/8 + N/2), int(3*N/8 + N/2)))]

    lattice_test = 'no'
    if lattice_test == 'yes':
        print("PERFORMING LATTICE TEST...DO NOT COLLECT DATA!")
        for i in range(N):
            for j in range(N):
                if i in A_1[0] and j in A_1[1]:
                    L1[i, j] = 1
                    L2[i, j] = 1
                if i in A_2[0] and j in A_2[1]:
                    L1[i, j] = 2
                    L2[i, j] = 2
        print(L1)
        print(L2)

    print("N=", N, "; Region 2 at T=", T)

    expE = 0.0 # Expectation value of E
    measurements = [] # List of Measurements
    # Main Monte Carlo cycle
    for x in range(steps + 1):
        i = randrange(0, N)
        j = randrange(0, N) # Picks a random starting location

        # Decides an anticipated spin amount
        L1_update = random() * 2 * pi
        L2_update = random() * 2 * pi

        if i in A_1[0] and j in A_1[1]:
            L1_update = L2_update
        if i in A_2[0] and j in A_2[1]:
            L1_update = L2_update
        # Calculates change in energy that would occur if this spin
            ↪ was accepted
        dE = 0.0
        # Starts calculating the nearest neighbor sum at location L[ i
            ↪ -1 , j]
        neighbor = i - 1
        if neighbor > -1:
            dE += cos(L1_update - L1[neighbor, j]) - cos(L1[i, j] - L1
                ↪ [neighbor, j])
            dE += cos(L2_update - L2[neighbor, j]) - cos(L2[i, j] - L2
                ↪ [neighbor, j])
        else:
            dE += cos(L1_update - L1[N - 1, j]) - cos(L1[i, j] - L1[N
                ↪ - 1, j]) # Periodic boundary conditions
            dE += cos(L2_update - L2[N - 1, j]) - cos(L2[i, j] - L2[N
                ↪ - 1, j])
        neighbor = i + 1
        if neighbor < N:
            dE += cos(L1_update - L1[neighbor, j]) - cos(L1[i, j] - L1
                ↪ [neighbor, j])
            dE += cos(L2_update - L2[neighbor, j]) - cos(L2[i, j] - L2[neighbor, j])
        else:
            dE += cos(L1_update - L1[0, j]) - cos(L1[i, j] - L1[0, j])
            dE += cos(L2_update - L2[0, j]) - cos(L2[i, j] - L2[0, j])
        neighbor = j - 1
        if neighbor > -1:
            dE += cos(L1_update - L1[i, neighbor]) - cos(L1[i, j] - L1[i, neighbor])
            dE += cos(L2_update - L2[i, neighbor]) - cos(L2[i, j] - L2[i, neighbor])
        else:
            dE += cos(L1_update - L1[i, N - 1]) - cos(L1[i, j] - L1[i, N - 1])
            dE += cos(L2_update - L2[i, N - 1]) - cos(L2[i, j] - L2[i, N - 1])
        neighbor = j + 1
        if neighbor < N:
            dE += cos(L1_update - L1[i, neighbor]) - cos(L1[i, j] - L1[i, neighbor])
            dE += cos(L2_update - L2[i, neighbor]) - cos(L2[i, j] - L2[i, neighbor])
        else:
            dE += cos(L1_update - L1[i, 0]) - cos(L1[i, j] - L1[i, 0])
            dE += cos(L2_update - L2[i, 0]) - cos(L2[i, j] - L2[i, 0])
        dE *= -J
        dE += y_tilde * (cos(theta_coefficient * L1[i, j]) - cos(theta_coefficient * L1_update)) + y_tilde * (cos(theta_coefficient * L2[i, j]) - cos(theta_coefficient * L2_update))

        # Calculates whether L[i,j] rotates
        R = exp(-dE * T)
        if R > 1 or random() < R:
            L1[i, j] = L1_update
            L2[i, j] = L2_update
            E += dE # / (N * N)
        if x != 0 and x % (2 * tau) == 0:
            expE += E
            # print("at x = ",x," ", expE)
            measurements.append(E) # Adds the measurement to the list

    expE /= BM
    # print(expE)

    # The Bootstrap Error Analysis
    resample = BM # times to repeat re-sampling
    B_i = [] # for the calculation of <B> and sigma
    for y in range(resample):
```

```python
        B = 0.0
        for z in range(int(BM)):
            n = randrange(0, BM)
            B += measurements[n]
        B /= BM
        B_i.append(B)

    # Now to calculate the Bootstrap sigma
    sigma_sigma = 0.0
    for w in range(resample):
        sigma_sigma += (B_i[w] - expE) ** 2
    sigma_sigma /= resample
    sigma_bootstrap = sqrt(sigma_sigma)

    # This is a test to make sure that A_1 and A_2 are indeed being updated the same.
    equivalence_test = 'no'
    if equivalence_test == 'yes':
        matches = 0.0
        for columns in range(N):
            for rows in range(N):
                if L1[rows, columns] == L2[rows, columns]:
                    matches += 1
                # if rows in A_1[0] and columns in A_1[1]:
                # if L1[rows, columns] == L2[rows, columns]:
                # matches += 1
                # if rows in A_2[0] and columns in A_2[1]:
                # if L1[rows, columns] == L2[rows, columns]:
                # matches += 1
        print(matches)
        if matches == 2 * int(.75 * N * N/4) + N*N/16:
            print("A_1 and A_2 match!")
        else:
            print("We messed up somewhere :(")

    return [T, expE, sigma_bootstrap] # This will create a results matrix which can be plotted


# The nice box
def Region_3(T):
    global N_global, E_measurements, tau_global, tau_after
    J = 1
    N = N_global # The lattice size: NxN
    # A test to make things quicker; higher temperatures equilibrate faster
    tau = tau_global

    BM = E_measurements # Number of independent measurements for the
```

```python
        bootstrap analysis
    steps = 2 * tau * BM # Number of times the program will run
    E = -4 * (N * N) - 2 * y_tilde * (N * N) # Initial Value of Energy since all spins start pointed up at \theta_i = 0.0
    boundary = N // 2
    L1 = zeros([N, N], float) # Lattice 1 where each entry is a value of \theta_i
    L2 = zeros([N, N], float) # Lattice 2

    A_1 = [region(N/8, N/8 + .75 * N), region(N/8 + N/2, 3*N/8 + N/2) + region(N/8, N/8 + N/4)]
    A_2 = [region(N/8, 3*N/8) + region(N - 3*N/8, N - N/8), region(3*N/8, 5*N/8)]

    # A_2 = [list(range(int(N/8), int(N/8 + .75 * N))), list(range(int(N/8), int(N/8 + N/4))) + list(range(int(N/8 + N/2), int(3*N/8 + N/2)))]

    lattice_test = 'no'
    if lattice_test == 'yes':
        print("PERFORMING LATTICE TEST...DO NOT COLLECT DATA!")
        for i in range(N):
            for j in range(N):
                if i in A_1[0] and j in A_1[1]:
                    L1[i, j] = 1
                    L2[i, j] = 1
                if i in A_2[0] and j in A_2[1]:
                    L1[i, j] = 2
                    L2[i, j] = 2
        print(L1)
        print(L2)
        print(A_2)

    print("N=", N, "; Replica Region 3 at T=", T)

    expE = 0.0 # Expectation value of E
    measurements = [] # List of Measurements
    # Main Monte Carlo cycle
    for x in range(steps + 1):
        i = randrange(0, N)
        j = randrange(0, N) # Picks a random starting location

        # Decides an anticipated spin amount
        L1_update = random() * 2 * pi
        L2_update = random() * 2 * pi

        if i in A_1[0] and j in A_1[1]:
            L1_update = L2_update
        if i in A_2[0] and j in A_2[1]:
            L1_update = L2_update
        # Calculates change in energy that would occur if this spin was accepted
        dE = 0.0
        # Starts calculating the nearest neighbor sum at location L[ i -1 , j]
        neighbor = i - 1
        if neighbor > -1:
            dE += cos(L1_update - L1[neighbor, j]) - cos(L1[i, j] - L1[neighbor, j])
            dE += cos(L2_update - L2[neighbor, j]) - cos(L2[i, j] - L2[neighbor, j])
        else:
            dE += cos(L1_update - L1[N - 1, j]) - cos(L1[i, j] - L1[N - 1, j]) # Periodic boundary conditions
            dE += cos(L2_update - L2[N - 1, j]) - cos(L2[i, j] - L2[N - 1, j])
        neighbor = i + 1
        if neighbor < N:
            dE += cos(L1_update - L1[neighbor, j]) - cos(L1[i, j] - L1[neighbor, j])
            dE += cos(L2_update - L2[neighbor, j]) - cos(L2[i, j] - L2[neighbor, j])
        else:
            dE += cos(L1_update - L1[0, j]) - cos(L1[i, j] - L1[0, j])
            dE += cos(L2_update - L2[0, j]) - cos(L2[i, j] - L2[0, j])
        neighbor = j - 1
        if neighbor > -1:
            dE += cos(L1_update - L1[i, neighbor]) - cos(L1[i, j] - L1[i, neighbor])
            dE += cos(L2_update - L2[i, neighbor]) - cos(L2[i, j] - L2[i, neighbor])
        else:
            dE += cos(L1_update - L1[i, N - 1]) - cos(L1[i, j] - L1[i, N - 1])
            dE += cos(L2_update - L2[i, N - 1]) - cos(L2[i, j] - L2[i, N - 1])
        neighbor = j + 1
        if neighbor < N:
            dE += cos(L1_update - L1[i, neighbor]) - cos(L1[i, j] - L1[i, neighbor])
            dE += cos(L2_update - L2[i, neighbor]) - cos(L2[i, j] - L2[i, neighbor])
        else:
            dE += cos(L1_update - L1[i, 0]) - cos(L1[i, j] - L1[i, 0])
            dE += cos(L2_update - L2[i, 0]) - cos(L2[i, j] - L2[i, 0])
        dE *= -J
        dE += y_tilde * (cos(theta_coefficient * L1[i, j]) - cos(theta_coefficient * L1_update)) + y_tilde * (cos(
            ↪ theta_coefficient * L2[i, j]) - cos(theta_coefficient *
            ↪ L2_update))

        # Calculates whether L[i,j] rotates
        R = exp(-dE * T)
        if R > 1 or random() < R:
            L1[i, j] = L1_update
            L2[i, j] = L2_update
            E += dE # / (N * N)
        if x != 0 and x % (2 * tau) == 0:
            expE += E
            # print("at x = ",x," ", expE)
            measurements.append(E) # Adds the measurement to the list

    expE /= BM
    # print(expE)

    # The Bootstrap Error Analysis
    resample = BM # times to repeat re-sampling
    B_i = [] # for the calculation of <B> and sigma
    for y in range(resample):
        B = 0.0
        for z in range(int(BM)):
            n = randrange(0, BM)
            B += measurements[n]
        B /= BM
        B_i.append(B)

    # Now to calculate the Bootstrap sigma
    sigma_sigma = 0.0
    for w in range(resample):
        sigma_sigma += (B_i[w] - expE) ** 2
    sigma_sigma /= resample
    sigma_bootstrap = sqrt(sigma_sigma)

    # This is a test to make sure that A_1 and A_2 are indeed being
        ↪ updated the same.
    equivalence_test = 'no'
    if equivalence_test == 'yes':
        matches = 0.0
        for columns in range(N):
            for rows in range(N):
                if L1[rows, columns] == L2[rows, columns]:
                    matches += 1
                # if rows in A_1[0] and columns in A_1[1]:
                # if L1[rows, columns] == L2[rows, columns]:
                # matches += 1
                # if rows in A_2[0] and columns in A_2[1]:
                # if L1[rows, columns] == L2[rows, columns]:
```

```python
                # matches += 1
        print(matches)
        if matches == 2 * int(.75 * N * N/4) + N*N/8:
            print("A_1 and A_2 match!")
        else:
            print("We messed up somewhere :(")

    return [T, expE, sigma_bootstrap] # This will create a results
        ↪ matrix which can be plotted


def vary_temps_RMI(T_min, T_max, T_step):
    if T_min == 0:
        temps = arange(T_min + T_step, T_max, T_step)
    else:
        temps = arange(T_min, T_max, T_step)

    # I have to separate the core mapping to prevent a memory error
    cores = Pool()
    result1 = cores.map(Region_1, temps)
    cores.close()
    cores.join()

    cores = Pool()
    result2 = cores.map(Region_2, temps)
    cores.close()
    cores.join()

    cores = Pool()
    result3 = cores.map(Region_3, temps)
    cores.close()
    cores.join()

    cores = Pool()
    result4 = cores.map(QCD_E, temps)
    cores.close()
    cores.join()

    shape_1 = array(result1)
    shape_2 = array(result2)
    shape_3 = array(result3)
    normal = array(result4)

    # Both Ising models are at the same temperature so,
    T_plot = normal[:, 0] # Takes the first column of the results
        ↪ matrix

    E_shape_1 = shape_1[:, 1] # Second column
    sigma_shape_1 = shape_1[:, 2] # Third column

    E_shape_2 = shape_2[:, 1] # Second column
    sigma_shape_2 = shape_2[:, 2] # Third column

    E_shape_3 = shape_3[:, 1] # Second column
    sigma_shape_3 = shape_3[:, 2] # Third column

    E_normal = normal[:, 1]
    sigma_normal = normal[:, 2]

    return T_plot, E_shape_1, sigma_shape_1, E_shape_2, sigma_shape_2,
        ↪ E_shape_3, sigma_shape_3, E_normal, sigma_normal


def Topological_Entropy(T_min, T_max, T_step, save_data='no'):
    global output_path
    t1 = time.time()

    Data = vary_temps_RMI(T_min, T_max, T_step)

    T_plot = Data[0]

    if save_data == 'yes':
        Data = array(Data)
        t_elapse = (time.time() - t1) / 3600
        folder_path = '{0}/TEE_Calc/Finished_Data/'.format(output_path
            ↪ )
        folder_name = 'Data from TEE QCD; {0}; {1}, {2}, {3}, n=2, y
            ↪ ~{4}, theta={5}'.format(date, E_measurements, T_step,
            ↪ N_global, y_tilde, theta_coefficient)
        if not os.path.exists(folder_path + folder_name):
            os.makedirs(folder_path + folder_name)
        savetxt('{8}{6}/RMI TEE; {0}; {1}, {2}, {3}, {4}, {5}, n=2, y
            ↪ ~{7}, theta={9}.txt'.format(date, E_measurements, T_min,
            ↪  T_max, T_step, N_global, folder_name, y_tilde,
            ↪ folder_path, theta_coefficient), Data, header='This data
            ↪  took {0:.3f} hours and was recorded on {1}. This was
            ↪ run on the PSU Cluster.'.format(t_elapse, datetime.
            ↪ datetime.today()))

    return T_plot


if __name__ == '__main__':
    t_start = time.time()

    # Main Program
    T_min = float(sys.argv[4].split(',')[0])
    T_max = float(sys.argv[5].split(',')[0])

    Topological_Entropy(T_min, T_max, T_step, save_data='yes')

    # End of Main Program

    t_elapse = (time.time() - t_start) / 3600
    print("Full Program done in {0:.3f} hours".format(t_elapse))
```

Appendix E

Codes of Derived Quantities

These codes are the functions used to calculate various quantities once I obtain the aggregated data files. These quantities are calculated using the three types of energy.

E.1 Renyi Mutual Information

E.1.1 RMI for the XY Model

This first formula is used for calculating RMI of the XY Model, and uses the normal thermodynamics. RMI is calculated as follows:

$$I_2(T) = \sum_{T_i=T}^{T_{max}} \Delta T \frac{2\langle E_i \rangle_2 - \langle E_i \rangle_{A\cup B} - 2\langle E_i \rangle_0}{T_i^2}.$$

Running

This is a python function. It has to be embedded into an existing python script.

This function takes an argument called `Data`. This is a variable that must be a data file from the XY simulation code in appendix A. You must use numpy to assign this variable like so: `Data = loadtxt(data_file.txt)`. Then, enter lattice size `N_global` and ΔT `T_step`. The keyword argument `graph = 'no'` can be changed to `'yes'` to automatically graph the RMI. This function returns an array that holds:

1. Temperature plot
2. RMI
3. Variance on RMI

Listing E.1: Renyi Mutual Information calculation function for XY Model

```python
from numpy import loadtxt, sqrt, array, vstack, savetxt
from pylab import plot, show, title, xlabel, ylabel, xlim, ylim,
    ↪ errorbar, legend, rcParams, rc
import time, numpy

def RMI_calc(Data, N_global, T_step, graph='no'):
    global date, n
    alpha = n
    t1 = time.time()
    T_plot = Data[0]
    # Gathers the replica data
    E_replica = Data[1]
    sigma_replica = Data[2]
    # Gathers the normal data
    E_A_U_B = Data[3]
    sigma_A_U_B = Data[4]

    E_normal = Data[5]
    sigma_normal = Data[6]
    # Calculating RMI for each T
    print('Working on Renyi Mutual Information...')
    count = len(E_A_U_B)

    RMI_plot = []
    RMI_sigma_plot = []
    deltaT = T_step
    # Calculates the RMI and the sigma for each RMI(T)
    for i in range(count):
        RMI = 0.0
        sigma_sigma_i = 0.0
        for j in range(i, count):
            term_j = deltaT * (2 * (E_replica[j]) - (E_A_U_B[j]) -
                ↪ alpha * E_normal[j]) / ((T_plot[j]) ** 2)
            RMI += term_j
            # Now to propagate the error from each E measurement...
            sigma_sigma_j = ((2 * deltaT) / ((T_plot[j] ** 2) *
                ↪ N_global * 2)) ** 2 * (sigma_replica[j] ** 2) + (
                ↪ deltaT / ((
                T_plot[j] ** 2) * N_global * 2)) ** 2 * (sigma_A_U_B[j]
                    ↪  ** 2) + ((2 * deltaT) / ((T_plot[j] ** 2) *
                    ↪ N_global * 2)) ** 2 * (sigma_normal[j] ** 2)
            sigma_sigma_i += sigma_sigma_j
        sigma_i = math.sqrt(sigma_sigma_i)
        RMI /= 2 * N_global
        RMI_plot.append(RMI)
        RMI_sigma_plot.append(sigma_i)
        if i % 100 == 0:
            print('Calculating RMI for T =', i * T_step)

    if graph == 'yes' or graph == 'plot':
        pylab.plot(T_plot, RMI_plot, 'b', linewidth=3)
        pylab.errorbar(T_plot, RMI_plot, yerr=RMI_sigma_plot, capsize
            ↪ =2, ecolor='r')
        pylab.title(r'RMI vs $T$; $T_{step}$' + ' = {0};'.format(
            ↪ T_step) + ' $T_{max}$' + ' = 100 ', fontsize=16)
        pylab.xlabel(r'$T$', fontsize=16)
        pylab.ylabel(r'$\frac{I_2(T)}{\ell}$', fontsize=16)
        pylab.xlim(0, 10)
        # pylab.ylim(0, 0.5)
        t_elapse = (time.time() - t1) / 60
        print('Done in {0:.3f} minutes'.format(t_elapse))
        if graph == 'plot':
            pylab.show()


    return T_plot, RMI_plot, RMI_sigma_plot
```

E.1.2 RMI for the QCD Model

Due to the inverted thermodynamics, the RMI for QCD is calculated differently. This is actually much easier to simulate due to this. The formula is as follows:

$$I_2(T) = \sum_{T_i=0}^{T} \Delta T \quad 2\langle E_i \rangle_2 - \langle E_i \rangle_{A \cup B} - 2\langle E_i \rangle_0.$$

Running

This code uses the import statements from the previous code listing. This is a python function. It has to be embedded into an existing python script.

This function takes an argument called `Data`. This is a variable that must be a data file from the QCD simulation code in appendix B. You must use numpy to assign this variable like so: `Data = loadtxt(data_file.txt)`. Then, enter lattice size `N_global` and ΔT `T_step`. The keyword argument `graph = 'no'` can be changed to `'yes'` to automatically graph the RMI. This function returns an array that holds:

1. Temperature plot
2. RMI
3. Variance on RMI

Listing E.2: Renyi Mutual Information calculation function for QCD Model

```
def RMI_calc_QCD(Data, N_global, T_step, graph='no'):
    global date, n
    alpha = n
    t1 = time.time()
    T_plot = Data[0]
    # Gathers the replica data
    E_replica = Data[1]
    sigma_replica = Data[2]
    # Gathers the normal data
    E_A_U_B = Data[3]
    sigma_A_U_B = Data[4]

    E_normal = Data[5]
    sigma_normal = Data[6]
    # Calculating RMI for each T
    print('Working on Renyi Mutual Information...')
    count = len(E_A_U_B)

    RMI_plot = []
    RMI_sigma_plot = []
    deltaT = T_step
    # Calculates the RMI and the sigma for each RMI(T)
    for i in range(count):
        RMI = 0.0
        sigma_sigma_i = 0.0
        for j in range(0, i):
            term_j = deltaT * (2 * (E_replica[j]) - (E_A_U_B[j]) -
                ↪ alpha * E_normal[j])
            RMI += term_j
            # Now to propagate the error from each E measurement...
            sigma_sigma_j = ((2 * deltaT) / (N_global * 2)) ** 2 * (
                ↪ sigma_replica[j] ** 2) + (deltaT / ( N_global * 2))
                ↪ ** 2 * (sigma_A_U_B[j] ** 2) + ((alpha * deltaT) / (
                ↪  N_global * 2)) ** 2 * (sigma_normal[j] ** 2)
            sigma_sigma_i += sigma_sigma_j
        sigma_i = math.sqrt(sigma_sigma_i)
        RMI /= 2 * N_global
        RMI_plot.append(RMI)
        RMI_sigma_plot.append(sigma_i)
        if i % 100 == 0:
            print('Calculating RMI for T =', i * T_step)

    if graph == 'yes' or graph == 'plot':
        pylab.plot(T_plot, RMI_plot, 'b', linewidth=3)
        pylab.errorbar(T_plot, RMI_plot, yerr=RMI_sigma_plot, capsize
            ↪ =2, ecolor='r')
        pylab.title(r'RMI vs $T$; $T_{step}$' + ' = {0};'.format(
            ↪ T_step) + ' $T_{max}$' + ' = 100 ', fontsize=16)
        pylab.xlabel(r'$T$', fontsize=16)
        pylab.ylabel(r'$\frac{I_2(T)}{\ell}$', fontsize=16)
        pylab.xlim(0, 10)
        # pylab.ylim(0, 0.5)
        t_elapse = (time.time() - t1) / 60
        print('Done in {0:.3f} minutes'.format(t_elapse))
        if graph == 'plot':
            pylab.show()

    return T_plot, RMI_plot, RMI_sigma_plot
```

E.2 Renyi Entropy

This is the Renyi Entropy for the QCD Model, calculated with the following formula:

$$S_2(T) = \sum_{T_i=0}^{T} \Delta T \quad 2\langle E_i \rangle_2 - 2\langle E_i \rangle_0.$$

Running

This is a python function. It has to be embedded into an existing python script. When calculating Renyi Entropy to use for calculating TEE, the $\langle E_i \rangle_2$ changes to $\langle E_i \rangle_{regionshape}$ The difference between this and the XY are as in section E.1, with inverted thermodynamics.

To change this to calculate XY Model RMI, simply adjust the for loop that comes before the energy calculation. Recall that Topological Entanglement Energy is a sum of Renyi Entropies for different shaped replica regions.

When calculating the Renyi Entropy for the QCD Model, the variable `E_interest` ↪ Is referring to `E_replica`, but when we need Renyi Entropy for the TEE calculation, `E_interest` refers to the different region shape energies. This function returns an array that holds:

1. EE
2. Variance on EE

Listing E.3: Renyi Entropy calculation function

```python
def EE_calc(T_plot, E_interest, sigma_E_interest, E_normal,
    ↪ sigma_normal, T_step, n):
    count = len(T_plot)
    deltaT = T_step

    S_plot = []

    S_sigma = []
    for i in range(count):
        S_A = 0.0
        sigma_sigma_i = 0.0

        for j in range(0, i):
            term_j = deltaT * ((E_interest[j]) - (n * E_normal[j]))
            S_A += term_j
            # error propagation:
            sigma_sigma_j = (deltaT * deltaT * (sigma_E_interest[j]**2
                ↪ + 4 * (sigma_normal[j]**2)))
            sigma_sigma_i += sigma_sigma_j
        sigma_i = numpy.sqrt(sigma_sigma_i)
        S_plot.append(S_A)
        S_sigma.append(sigma_i)
    pylab.plot(T_plot, S_plot, 'b')
    pylab.errorbar(T_plot, S_plot, yerr=S_sigma, ecolor='r')
    pylab.show()
    return S_plot, S_sigma
```

E.3 Topological Entanglement Entropy

This function calculates the TEE. The function detailed in E.2 must be defined above this function for it to work. The TEE formula is:

$$TEE_2(T) = S_i = -S_{shape1}[T] + 2 * S_{shape2}[T] - S_{shape3}[T].$$

Where S is the Renyi Entropy of each shape.

E.4 Running

This is a python function. It must be embedded in an existing python script with the imports of the previous codes.

The two arguments are `Data, T_step`. `T_step` is simply the ΔT. This is a variable that must be a data file from the TEE simulation code in appendix D. You must use numpy to assign this variable like so: `Data = loadtxt(data_file.txt)`. This function returns an array of:

1. Temperature plot
2. TEE
3. Variance on TEE

Listing E.4: Topological Entanglement Entropy calculation function

```
def TEE_calc(Data, T_step):
    global date, n
    t1 = time.time()
    T_plot = Data[0]
    # Gathers the replica data
    E_shape_1 = Data[1]
    sigma_shape_1 = Data[2]

    # Gathers the normal data
    E_shape_2 = Data[3]
    sigma_shape_2 = Data[4]

    E_shape_3 = Data[5]
    sigma_shape_3 = Data[6]

    E_normal = Data[7]
    sigma_normal = Data[8]
    print("Calculating EE for... ")
    # calculates the Renyi Entropy
    print("Shape 1...")
    shape_1_data = EE_calc(T_plot, E_shape_1, sigma_shape_1, E_normal, sigma_normal, T_step, 2)
    print("Shape 2...")
    shape_2_data = EE_calc(T_plot, E_shape_2, sigma_shape_2, E_normal, sigma_normal, T_step, 2)
    print("Shape 3...")
    shape_3_data = EE_calc(T_plot, E_shape_3, sigma_shape_3, E_normal, sigma_normal, T_step, 2)

    S_shape_1 = shape_1_data[0]
    S_shape_1_sigma = shape_1_data[1]
    S_shape_2 = shape_2_data[0]
    S_shape_2_sigma = shape_2_data[1]
    S_shape_3 = shape_3_data[0]
    S_shape_3_sigma = shape_3_data[1]


    # Calculating TEE for each T
    print('Working on Topological Entanglement Entropy...')
    count = len(T_plot)
    TEE_plot = []
    TEE_sigma_plot = []
    for T in range(count):
        S_i = -S_shape_1[T] + 2 * S_shape_2[T] - S_shape_3[T]
        TEE_plot.append(S_i)
        sigma = numpy.sqrt(S_shape_1_sigma[T]**2 + 4*S_shape_2_sigma[T]**2 + S_shape_3_sigma[T]**2)
        TEE_sigma_plot.append(sigma)
    pylab.plot(T_plot, TEE_plot, 'b')
    pylab.errorbar(T_plot, TEE_plot, yerr=TEE_sigma_plot, ecolor='r')
    pylab.show()
    prime = derivative(T_plot, TEE_plot)[1]
    return T_plot, TEE_plot, TEE_sigma_plot
```