Searching for a Connection between Maximum Entropy and the Arrow of Time

Aiden Lewis

Director: Ralph Chamberlin

April 2023

Arizona State University

Abstract

We implemented the well-known Ising model in one dimension as a computer program and simulated its behavior with four algorithms: (i) the seminal Metropolis algorithm; (ii) the microcanonical algorithm described by Creutz in 1983; (iii) a variation on Creutz's time-reversible algorithm allowing for bonds between spins to change dynamically; and (iv) a combination of the latter two algorithms in a manner reflecting the different timescales on which these two processes occur ("freezing" the bonds in place for part of the simulation). All variations on Creutz's algorithm were symmetrical in time, and thus reversible. The first three algorithms all favored low-energy states of the spin lattice and generated the Boltzmann energy distribution after reaching thermal equilibrium, as expected, while the last algorithm broke from the Boltzmann distribution while the bonds were "frozen." The interpretation of this result as a net increase to the system's total entropy is consistent with the second law of thermodynamics, which leads to the relationship between maximum entropy and the Boltzmann distribution.

Contents

1 Introduction

The passage of time is something the human mind has an inherent sense for. Imagine, for instance, a video of a building rising from a pile of rubble, or glass shards merging together in a seamless windowpane – of course, anyone would immediately recognise that the footage must be playing in reverse. Rubble cannot spontaneously assemble itself into a building, after all. This reflects a fundamental truth of the universe: time flows forwards, and forwards alone. Naturally, one might then expect that the laws of physics would also demonstrate this irreversibility in time; in reality, though, all microscopic laws are actually *symmetrical* in time. The emission of a photon, for example, is a perfect mirror of its absorption – equal amounts of energy and momentum are exchanged in both cases. Nowhere in the principles governing small-scale interactions like this is the arrow of time truly defined (Gold).

That symmetry, however, only exists in the microscopic contexts to which those principles apply. The universe as a whole, by contrast, is far too large and complex to be described completely by such simple laws. The countless interactions of the innumerable particles making up reality destroy that small-scale symmetry and replace it with the absolute one-way flow of time we observe. Describing systems like that requires the second law of thermodynamics. Crucially, the second law states that a system's total entropy cannot decrease over time – if two objects initially at different temperatures are brought into thermal contact, then they will eventually come to the same temperature and will not return to their original temperatures on their own ("Second Law"). This state of thermal equilibrium is the one for which the system is at its maximum entropy, and the second law of thermodynamics is the reason why that process of heat transfer is irreversible.

Taken on their own, both microscopic and macroscopic systems are accurate representations of reality. Taken together, however, the two seem to contradict one another: the second law of thermodynamics explicitly states that some processes are necessarily irreversible, while the microscopic laws of physics leave no room for asymmetry in time. This paper seeks to find a resolution to this conflict through an analysis of the well-known Ising model.

2 Theory

2.1 The Ising model

The Ising model is one of the most thoroughly-studied models in all of physics; accordingly, we know with reasonable certainty exactly how it should act in our simulation. It describes a magnet as a lattice of spins that can point either up or down, and the interactions between these spins dictate the model's behavior (Newman). The energy of the lattice is given by the Ising Hamiltonian, as below:

$$E = -J \sum_{\langle i,j \rangle} s_i s_j - B \sum_i s_i, \tag{1}$$

where E is the energy of the lattice's current state, J is the spin interaction energy, s is the orientation of a spin in the lattice ($+1$ for up and -1 for down), B is the strength of an external magnetic field, and i, j are the indices of neighboring spins in the lattice. To keep from over-counting, only spins' right neighbors are considered when performing the calculation. For the sake of simplicity, we keep the magnetic field strength at 0, as it is not relevant to our purposes. Simulations of the model are largely centered around placing energy-dependent conditions on whether or not a particular spin will be flipped. The maximum change in energy as a result of a spin-flip obeys the following relationship:

$$|\Delta E| = 2zJ, \tag{2}$$

where z is the lattice coordination number, the number of neighbors the spin being flipped has. Since our lattice is one-dimensional, $z = 2$ by default, though varying this number was a core aspect of one of the algorithms we used in the simulation. Furthermore, we imposed periodic boundary conditions on our starting lattice, i.e. made sure the first and last elements were neighbors of one another.

To understand Eq. 2, consider the following small 3-element lattice:

$$\uparrow \uparrow \uparrow$$

With interaction energy at 1, a value we maintained throughout the simulation, the total energy of this segment is $-1 - 1 - 1 = -3$, since aligned spins have energies of -1. If we flip the middle spin such that the lattice looks like this, though:

$$\uparrow \downarrow \uparrow$$

Now, there are two antialigned interactions and one aligned one, so the total energy becomes $1 + 1 - 1 = 1$. The total energy shift is 4, which is exactly what Eq. 2 says it should be. As such, it is perhaps worth thinking of the lattice less as a collection of spins and more as a

collection of spin-*pairs* – i.e., bonds – instead, at least as far as understanding how energy changes work in the Ising model. Importantly, though, note that Eq. 2 gives the *maximum* energy change – not all spin-flips will change the system's energy by 4, even while z is held at 2. As an example, consider the following 3-element lattice:

$$\uparrow\uparrow\downarrow$$

Its total energy is the same as the previous lattice (1), since it is essentially the same lattice but effectively shifted to the right by one unit. If we vary its middle spin such that its state becomes:

$$\uparrow\downarrow\downarrow$$

The energy of this state is $1 - 1 + 1 = 1$, exactly the same as before. This is much more apparent when looking at it through the spin-pair lens: both states have two antialigned spin-pairs and one aligned spin-pair, and all the spin-flip did was shift the high0energy bond one to the right. After all, $(1)(1) = 1 = (-1)(-1)$, so the orientation of an aligned pair is irrelevant to the total energy of the system. Consequently, a lattice filled entirely with up-spins has the exact same energy as one filled entirely with down-spins – this type of behavior is known as inverse dynamics.

2.2 The Metropolis algorithm

The Metropolis algorithm is a Monte Carlo method commonly used in simulations of the Ising model, first developed by Metropolis et al. in 1953 for use in hard-sphere gas simulations (Newman). In the context of the Ising model and our simulation of its behavior, its steps are as follows:

1. Select a site in the lattice at random.
2. Calculate the change in energy that would result from the site flipping its spin.
3. Check the sign of the energy change.
   (a) If the change in energy is negative or zero, accept the change and flip the spin.
   (b) If the change in energy is positive, accept the change with the following probability:
   $$P = e^{-J\beta\Delta E} \tag{3}$$
4. Repeat.

In Eq. 3, β is the thermodynamic beta $1/kT$, where k is the Boltzmann constant and T is system temperature. Put in plain English, Step 3 states that the system will *always* accept a transition to a lower or equal energy state, but only sometimes do so if the energy state

is higher. As a result, the Metropolis algorithm causes the Ising model to generally favor lower-energy states. Looking at the algorithm through the lens of conservation of energy, the system always has enough energy to accommodate a downward transition (where the lost energy is dissipated to the environment), so it always occurs. With an upward transition, though, the system needs to get the required energy from the arbitrarily large thermal bath the system is immersed in, which is not guaranteed. Taking the extreme temperature limits of Eq. 3 (when $\Delta E > 0$) helps illustrate this:

$$\lim_{T\to 0} \beta = \infty \rightarrow \lim_{T\to 0} P = \lim_{\beta\to\infty} e^{-J\beta\Delta E} = e^{-\infty} = 0$$

$$\lim_{T\to \infty} \beta = 0 \rightarrow \lim_{T\to \infty} P = \lim_{\beta\to 0} e^{-J\beta\Delta E} = e^{0} = 1$$

In other words, the system will never accept the energy increase if the thermal bath has no energy to give, and will always accept it if the bath has an infinite amount of energy to give. This behavior is exactly as we would expect from the principle of a thermal bath.

One of the key features of the Metropolis algorithm is that it must obey the condition of detailed balance – that is, the ultimate distribution of energies it exhibits after the system has come to thermal equilibrium (and therefore its maximum entropy state) must follow the probability distribution used to calculate the transition probabilities. The Metropolis algorithm uses the Boltzmann probability (Eq. 3) for this, so we should thus expect our simulation to generate the Boltzmann distribution.

2.3 The Creutz algorithm

The Metropolis algorithm is, at its core, a statistical method. The theory behind it relies on the concept of an ideal thermal bath, an approximation of reality that may not exist outside the domain of the thermodynamic limit, and the use of transition probabilities means that it is not reversible. The algorithm described by Creutz in 1983, by contrast, seeks to model the system's behavior in an entirely deterministic (and thus reversible) manner. Rather than a thermal bath, it uses a bath of demons as its source of upward transition energy. The demons can be thought of as a reservoir of kinetic energy, while the lattice of spins is a reservoir of potential energy. Under this algorithm, the overall system is microcanonical, but individual sites within the lattice are not. Rather, they exhibit canonical behavior, where the sites' temperatures depend on their respective demons. In this manner, they act as a sort of 'thermometer,' thus avoiding the multiplicity issue that affects the Metropolis algorithm.

When the system loses energy, a demon takes it on instead, and when it gains energy, it must come from a demon. In detail, the basic Creutz algorithm consists of the following steps:

1. Select a site in the lattice at random.
2. Calculate the change in energy that would result from the site flipping its spin.
3. Check the sign of the energy change.
   (a) If the change in energy is negative or zero, accept the change, flipping the spin and adding energy to its demon.
   (b) If the change in energy is positive, accept the change if and only if the demon has enough energy to accommodate it.
4. If the change occurs, transfer energy between the demon and system accordingly.
5. Repeat.

Evidently, it is nigh identical to the Metropolis algorithm, the only significant difference being the condition on upward transitions – it is deterministic, rather than probabilistic. Like Metropolis, the energy distribution generated by this algorithm also obeys the Boltzmann distribution; therefore, we know that both are valid methods for describing the behavior of the Ising model.

The primary focus of this paper is on a variation of the Creutz algorithm wherein the bonds between different sites on the lattice are allowed to be broken. Creutz himself discussed a very similar modification to his original algorithm in 1985, but ours differs from his in that demons are not fixed to site locations; rather, the status of a bond is used in determining the interaction energy between neighboring sites. Two sites connected by a broken bond do not interact, so their associated energy is 0 regardless of the orientations of their respective spins. In mathematical terms, this effectively amounts to varying the lattice coordination number z in Eq. 2 between its maximum value of 2 and its minimum value of 0 within the lattice. To show what this means on a physical level, recall the small 3-element lattice we described in section 2.1:

$$\uparrow\downarrow\uparrow$$

As before, its energy in this state is 1. If, however, we break the bond linking the middle and last spins as follows:

$$\uparrow\downarrow\,|\,\uparrow$$

The total energy of this state is then $1 + 0 - 1 = 0$; and if we break the middle spin's other bond, as below:

$$\uparrow \;|\; \downarrow \;|\; \uparrow$$

Then the down-spin effectively ceases to interact with any of its neighbors. Indeed, any changes to its orientation have no bearing on the ultimate energy of the state. In Eq. 2, the above two states are described by $z = 1$ and $z = 0$ for the former and latter respectively.

With regards to altering the Creutz algorithm, bond making and breaking takes the form of an additional step after the spin-flip is performed. The condition it obeys when determining whether to make or break a bond mirrors that used to decide the spin-flip operation: if the demon has enough energy to accommodate the change, it will happen, and energy will be transferred accordingly. As before, the kinetic-energy-like demon has a minimum energy of 0, so nothing will change if it does not have enough energy. Like the other two algorithms, the energy distribution generated by this "modified Creutz algorithm" also follows the Boltzmann distribution, confirming its validity as a method of simulating the behavior of the Ising model.

Our simulations of the 'default' Creutz algorithm and its bond-breaking variation were implemented with reversibility incorporated. Before running the simulation, we generated a random list of sites for the algorithm to visit, which it traversed the length of for the first half and then again in reverse for the second half. With the bond variation specifically, we also switched the order of operations for the second half so as to maintain reversibility (i.e. spin-bond for the first, bond-spin for the second). Furthermore, we simulated them both simultaneously in a mixed approach based on thirds – bond formation was allowed for the first and last thirds, but frozen in place for the middle third. During this part of the simulation, the system behaved according to the standard Creutz algorithm, but the lattice had by that point an effectively random arrangement of bonds. The theoretical basis of this alteration lies primarily in the idea of differing timescales – the bond- and spin-flips occur at the same rate for the first and last thirds, but for the middle third, the latter is much faster than the former. In this, we are not seeking to describe a specific physical system *per se*, but rather a conceptual one in which bond formation occurs at a variable timescale. As an example, similar behavior has been observed in the magnetic dynamics of glass molecules (Tracht *et al.*), so we know that the underlying assumptions of this simulation are still physically reasonable.

The purpose of this variable-timescale simulation is to study the energy distribution it generates over time. It is found that the algorithms dictating the behavior of the model in the first and final thirds, on their own, generate distributions in agreement with the Boltzmann distribution – in other words, that they inhabit their respective states of maximum

entropy. Accordingly, the adherence of the energies during the middle third to the Boltzmann distribution serves as a measure of the effect of frozen bond formation on the system's maximum entropy; to quantify this, we analysed the ratios of energy level probabilities over time:

$$\frac{p_{n+1}}{p_n} = e^{-J\beta\Delta E}, \tag{4}$$

where $n+1$ and n are the indices of two energy states (lower and higher respectively), ΔE is the change in energy between those states, J is interaction energy, and β is the thermodynamic beta (Müller). Eq. 4 can be rewritten into the following form:

$$\ln p_{n+1} - \ln p_n = -\beta = \Delta \ln p \tag{5}$$

ΔE was replaced with 1 here, since we are only concerned with adjacent energy levels. $\Delta \ln p$ serves as shorthand to replace the somewhat more cumbersome expression $\ln p_{n+1} - \ln p_n$. By plotting this quantity over time, we can numerically describe how well the energies adhere to the Boltzmann distribution. Indeed, $\Delta \ln p$ is related to the local temperature averaged over adjacent energy levels.

3 Methodology

The model and all associated algorithms were implemented entirely within the programming language Python, primarily in order to make use of its object-oriented nature. By taking this approach, we were able to write code with an emphasis on generality – having dedicated objects for the lattice, its spin sites, and the bonds between them allowed us to create a number of basic properties and equations that saw ubiquitous use throughout the program. This made its core structure quite flexible, which was highly valuable while the project was still in its formative days. As an example, the very first iteration of the code implemented only the basic Ising model and Metropolis algorithm. We could have represented the lattice as an array of integers, as most implementations do; instead, though, we created a 'Lattice' object containing an array of 'Sites,' with each class having various fundamental methods built into them. As such, when we expanded its scope and included the Creutz algorithm, the structure we had already created was versatile enough to accommodate it. Further research on this subject could benefit from our code in much the same way.

This has the added benefit of increased readability: by creating different layers of abstractions of physical concepts, we were able to write the code in such a way that even somebody unfamiliar with the principles of Python, or even coding as a whole, would see the actual physics being done. As an example, consider the following hypothetical line of code:

```
E = sum([spins[i]*spins[(i+1)%len(spins)]*bonds[i] for i in range(len(
    spins))])
```

If spins and bonds were stored as separate arrays of integers, this would be one way to calculate the lattice's total energy. To somebody looking at the code for the first time, though, this would hardly be easy to see. In our code, by contrast, the same would be achieved by:

```
E = lat.energy
```

There is little ambiguity in the above line. Ultimately, the close adherence of the program's fundamental structure to the design of the Ising model allowed us to keep the physics of the program at the center and spend less time on code-writing minutiae.

One consideration we paid particular attention to was computation time. To get useful results from statistical algorithms like Metropolis, large lattice sizes need to be simulated for long periods of time, and that can get quite computationally intensive. Nested loops are especially time-consuming – if, for instance, one has a loop of length 5 that contains a loop of length 8, adding one to the length of the outer loop adds 8 extra sets of operations. A simulation making frequent use of such nested loops is much more liable to take a prohibitively long time to run than one without them, so we made sure to avoid using them wherever possible. The most significant examples of this were the energy calculations within the algorithms. The `energy` function referenced above loops through the entire lattice, which we ran at size 500, so using it for every single Monte Carlo timestep would have dramatically increased the computation time of the simulation. Instead, we stored and manually adjusted a running total of the energy and calculated changes to it using a slight variation on Eq. 2, a simple arithmetic operation. This allowed us to run much larger simulations for far longer than we would have been able to under the nested-loop approach, greatly increasing the accuracy and usefulness of our results.

The full code, as well as a link to the project's Github repository, can be found in the appendices of Section 8.

4 Results

The same initial lattice was used for the simulations of the first three sections in order to maintain consistency.

4.1 The Metropolis algorithm

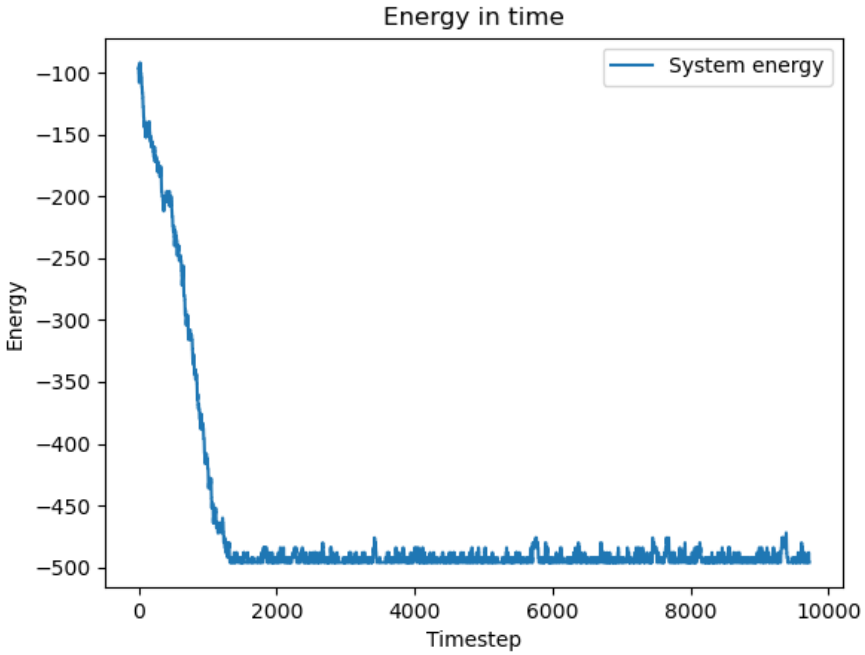


Figure 1, Energy over time of the Ising model.

The temperature used in the simulation was $1/J\beta = 2.5$. All future references to temperature have been scaled to the Boltzmann constant in this way. Essentially, the system started at an infinite temperature (with a randomly-generated set of spin alignments) and then was immediately cooled to this temperature.

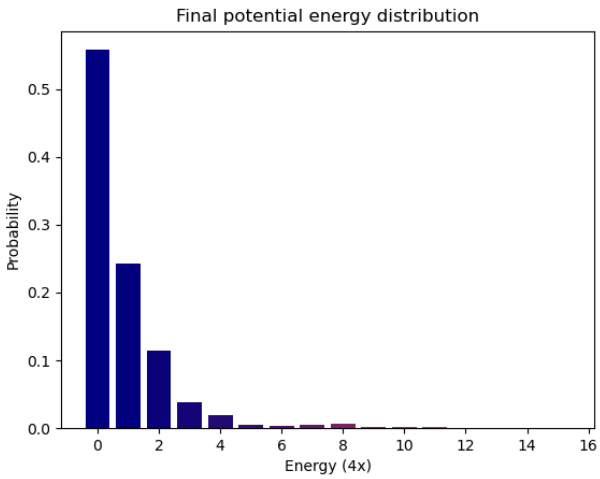


Figure 2.1, Final energy distribution generated by the simulation (non-linear).

The values on the x-axis represent units of 4 energy, as that is the energy step-size used by the Ising model when the lattice coordination number z is held constant at 2. Its adherence to the Boltzmann distribution is easier to see when plotting the natural logarithms of the probabilities:

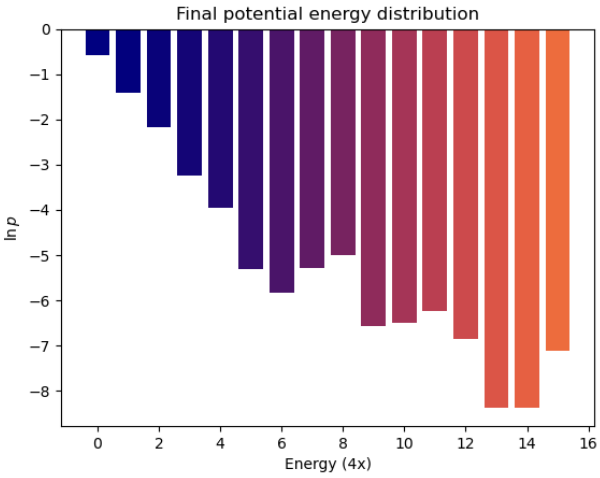


Figure 2.2, Final energy distribution generated by the simulation.

The values for lower energy states roughly follow a linear relationship, as we would expect. The slope of this line, given by Eq. 5, gives rise to the temperature of the lattice.

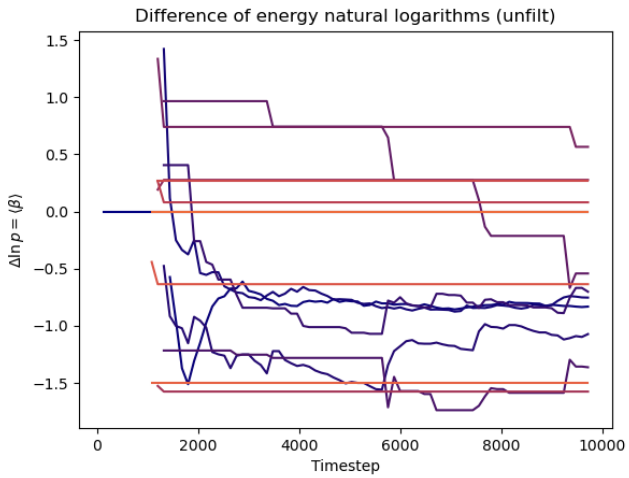


Figure 3.1, Differences in the energy distribution's natural logarithms over time (unfiltered).

The straight, high-energy lines in Fig. 3.1 (shown in bright orange; dark blue represents a low energy state) are a consequence of insignificant statistical data – the lattice rarely occupies states of high energy, so their relative values in the histogram stay roughly constant over time and produce the outlier lines seen on the graph. As they do not represent statistically significant findings, the high-energy datapoints have been excluded from our analysis.

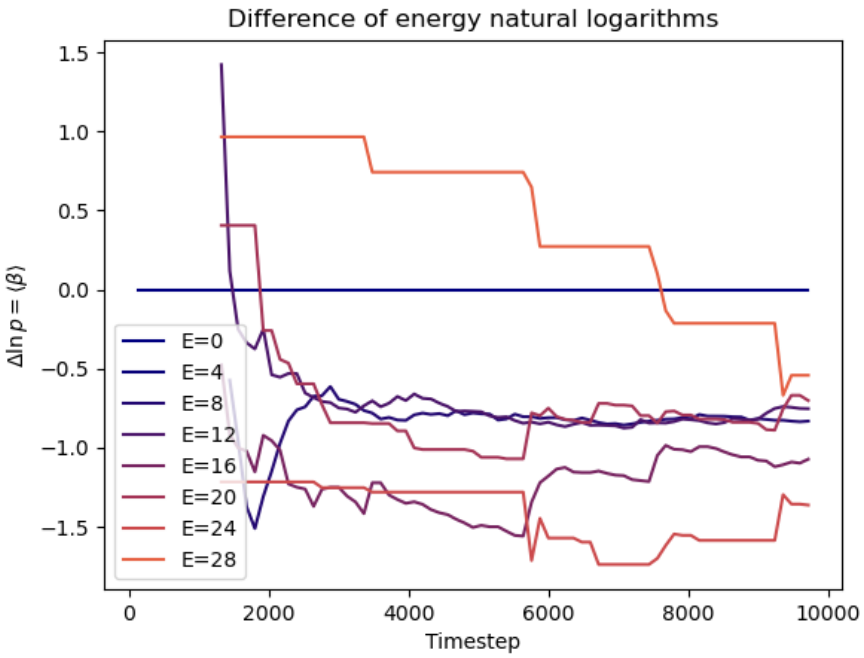


Figure 3.2, Differences in the energy distribution's natural logarithms over time.

With the outliers filtered out, we can use Eq. 5 with the data from the last half of the simulation (the point by which the system's distribution had largely stabilised) to calculate the temperature as a check of our data. The results are tabulated below:

Energy	**Temperature**
4	—
8	4.889
12	4.859
16	3.395
20	4.782
24	2.604
28	-36.113

Averaging these values, excluding the meaningless negative one, we find that the temperature of the system is $1/J\beta = 4.1 \pm 1.1$. Though the expected temperature value of 2.5 lies outside this range, it is still within two standard deviations of the experimental value generated by the simulation, so the discrepancy is not severe.

4.2 The Creutz algorithm

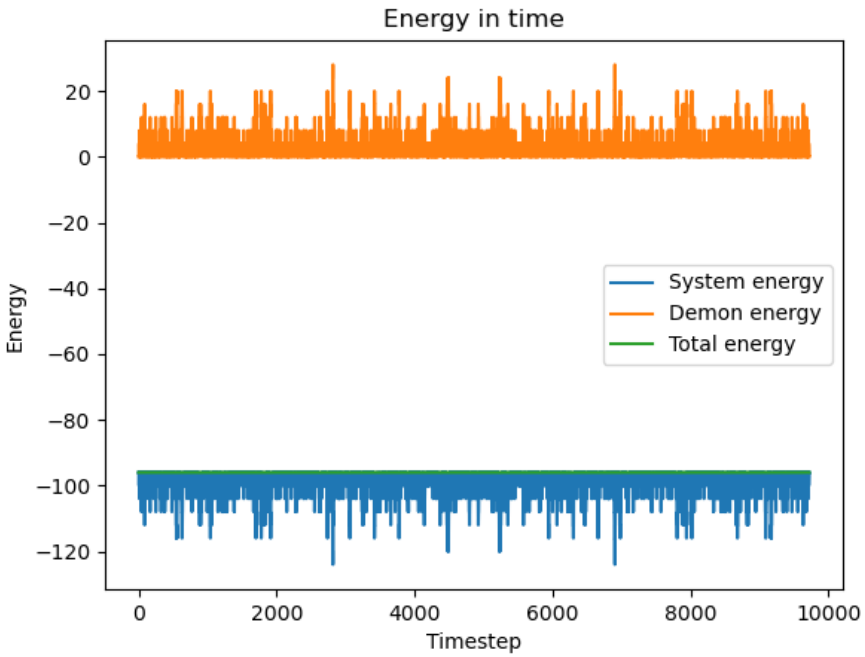


Figure 4, Energy over time of the Ising model.

It is easy to tell by eye that the algorithm is fully reversible, but to confirm this observation, our code checked the final lattice against its initial state just to ensure that the two were identical. Though it is technically possible that the two happened to be equivalent purely by chance, there are order 10^{150} possible configurations of a size-500 lattice like ours. Accordingly, the likelihood of this happening is so low as to be effectively impossible, so we are reasonably sure of our check's validity.

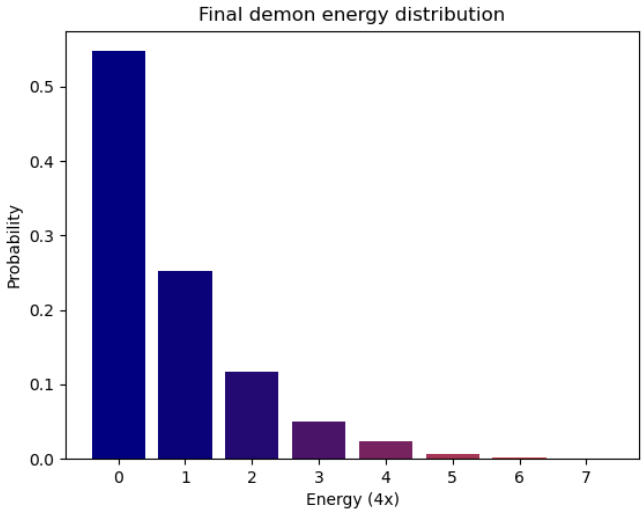


Figure 5.1, Final energy distribution generated by the simulation (non-linear).

The energy values on the x-axis have been scaled as before.

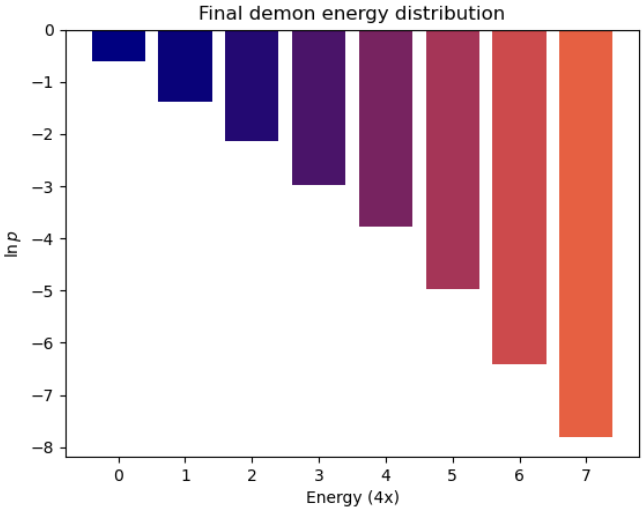


Figure 5.2, Final energy distribution generated by the simulation.

As expected, the natural logarithms of the probabilities follow an approximately straight line.

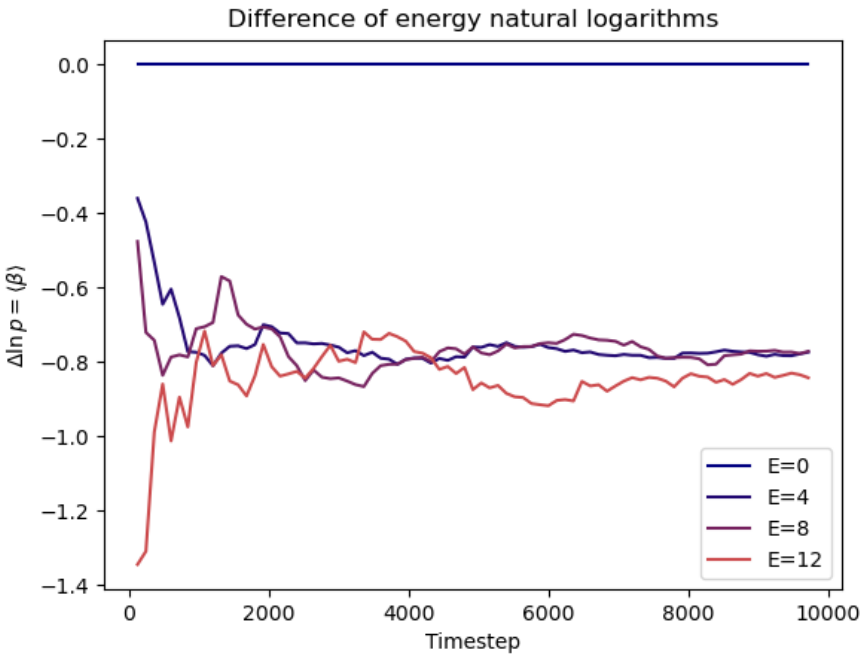


Figure 6, Differences in the energy distribution's natural logarithms over time.

From the parts of the above data where the system was at equilibrium, we found the temperatures corresponding to each energy level to be as follows:

Energy	**Temperature**
4	5.179
8	5.226
12	4.649

Averaging these, we find the system's temperature to be $kT = 5.0 \pm 0.3$. Although this value does differ notably from the temperature found for the same lattice under the Metropolis algorithm, their one-standard-deviation ranges do still intersect, so the two results are nonetheless in reasonable agreement with one another, especially given the relatively small number of data-points generated by this part of the simulation.

4.3 Creutz algorithm (bond formation)

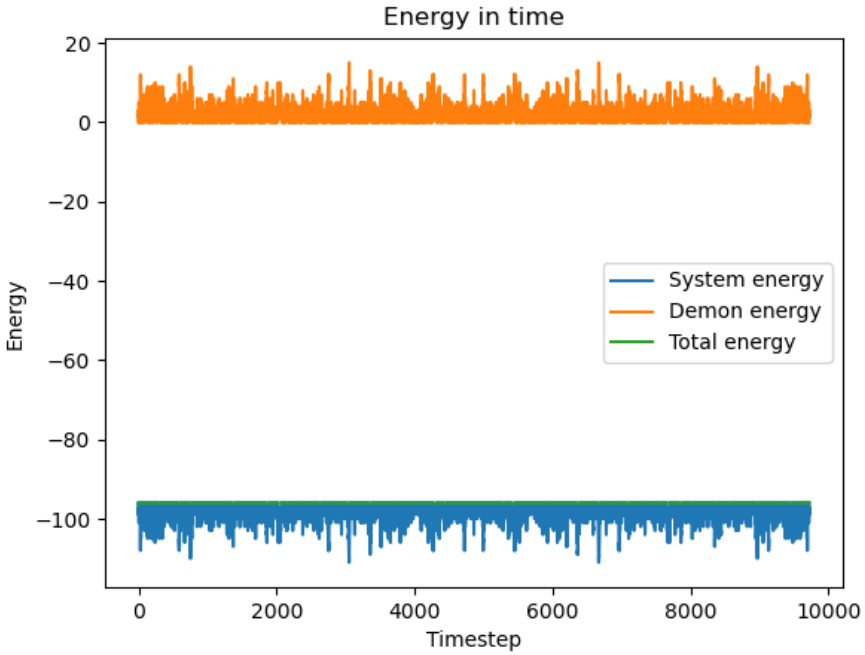


Figure 7, Energy over time of the Ising model.

Qualitatively, the above results look quite similar to their counterpart in Fig. 4, but the range of energy states the system inhabits is noticeably smaller in this case. Due to the bond formation, though, the system changes energies in steps of 1 rather than 4, so this is to be expected. Proper reversibility was confirmed in the same way as in the above section.

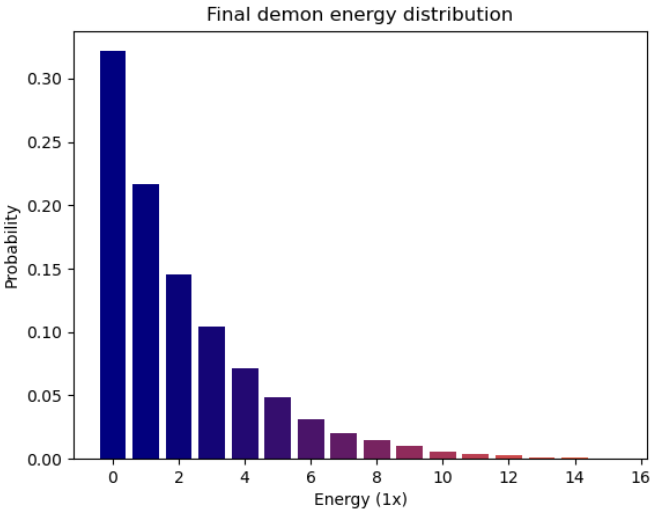


Figure 8.1, Final energy distribution generated by the simulation (non-linear).

Unlike in the prior cases, the energies of the distribution have not been scaled, again due to the change in energy step size.

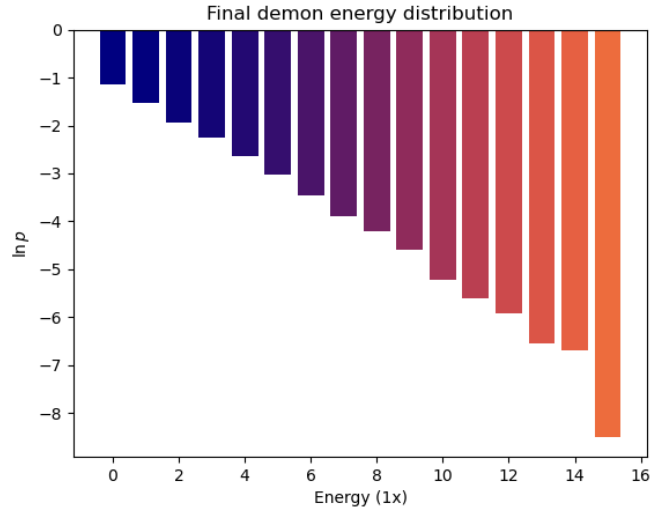


Figure 8.2, Final energy distribution generated by the simulation.

The linear relationship between energy and probability natural logarithm is quite clear in this case, largely due to the aforementioned lack of scaling.

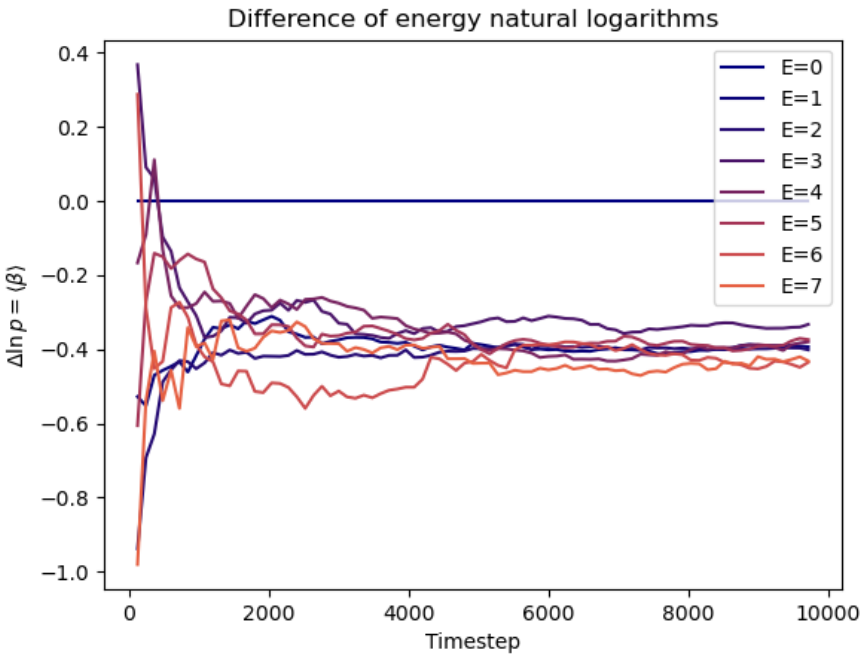


Figure 9, Differences in the energy distribution's natural logarithms over time (filtered).

Using the data generated above, we calculated the temperature of the lattice under the Creutz algorithm with bond formation enabled. The results are tabulated below:

Energy	**Temperature**
1	2.511
2	2.522
3	2.998
4	2.460
5	2.586
6	2.403
7	2.236

Averaging the data, we found its temperature to be $kT = 2.5 \pm 0.2$. This value is significantly lower than what we found under the static-bond Creutz algorithm and is instead in much better agreement with the results of the Metropolis algorithm (the expected temperature value in particular). Considering the small sample size used to generate the former result, though, we believe that the value produced by this simulation is the more statistically meaningful one.

4.4 Creutz algorithm (variable timescale)

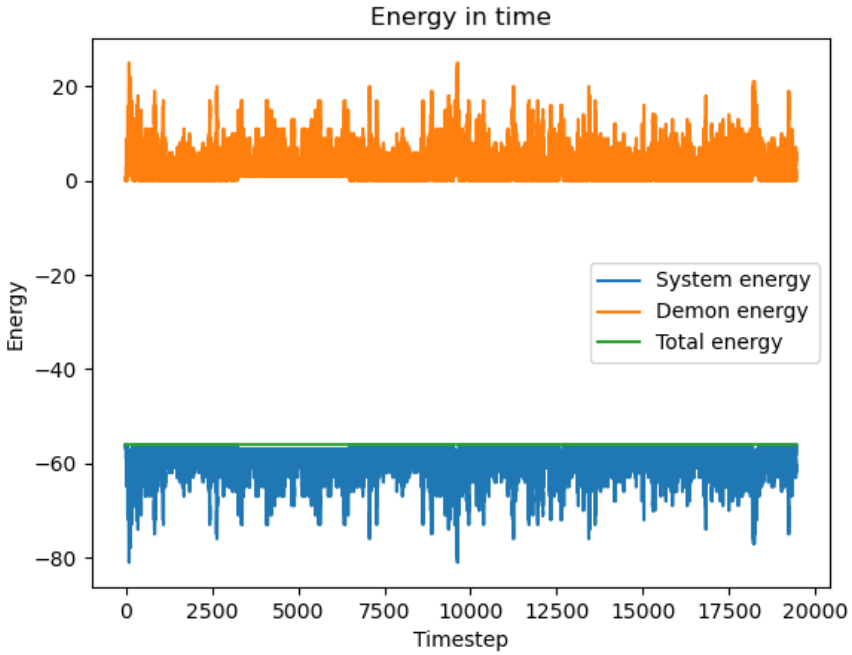


Figure 10, Energy over time of the Ising model.

The way in which this particular simulation was run differs somewhat from the previous ones. First of all, a different starting lattice configuration with a lower initial energy was used. The behavior unique to the variable timescale approach is much clearer in this case, though it can still be observed when running the simulation on the original lattice. They have been included in Appendix 8.1. Secondly, the simulation's runtime is doubled – the sites the algorithm chooses to operate on for the first half are generated as normal, with a list of random indices that is then reversed, but the second half is composed of an entirely independent list of random sites. The reversibility of the first half can be seen visually in Fig. 10, though the nature of this 'time extension' renders the program's built-in safety check ineffective. We chose to do this in order to more clearly demonstrate the behavior of the model at long times, as shown in Fig. 12.2.

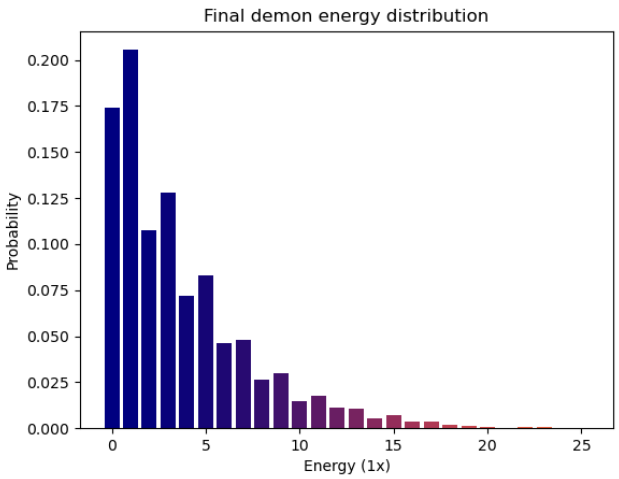


Figure 11.1, Final energy distribution generated by the simulation (non-linear).

Evidently, the distribution generated by the algorithm does not obey the Boltzmann distribution; as such, it is impossible to calculate the lattice's temperature, thereby removing the primary point of (quantitative) comparison between the various algorithms. It is for this reason that we elected to instead focus on a different lattice structure that clearly displayed this unique behavior.

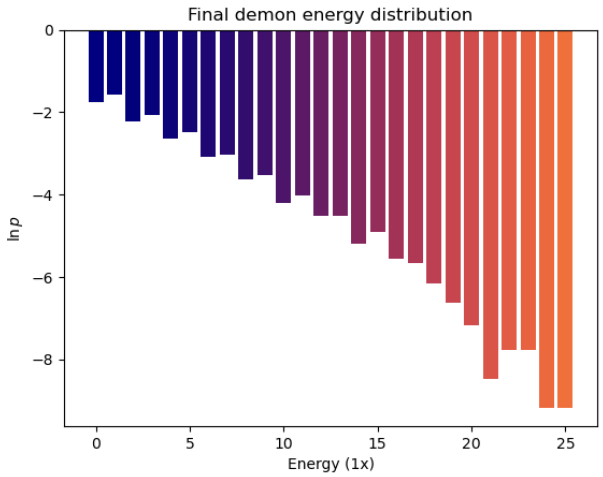


Figure 11.2, Final energy distribution generated by the simulation.

The effect is not as obvious when the probability natural logarithms are plotted, but it is nonetheless apparent that the relationship between $\ln p$ and E is not linear (as we would expect it to be, if the probabilities obeyed the Boltzmann distribution).

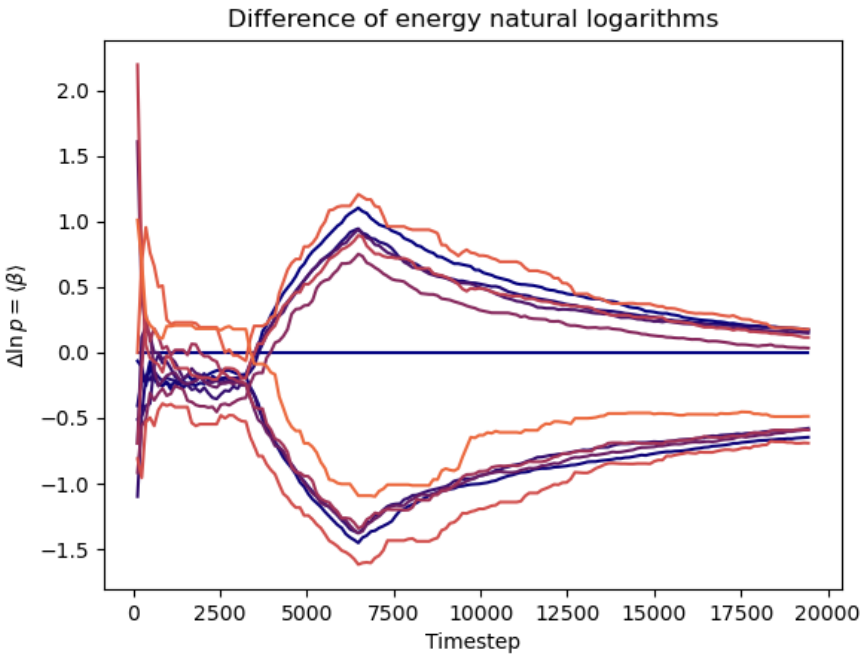


Figure 12, Differences in the energy distribution's natural logarithms over time.

The different thirds of the simulation are quite visible on the logarithm difference graph – during the first third (where bonds can be formed and broken freely), it behaves as in the other simulations using the Creutz algorithm, but during the second (where the instantaneous bond-state at that specific timestep becomes frozen in), their values diverge rapidly. Once bond formation is reenabled during the last third, they gradually start to come back together at long times. This behavior is easier to see intuitively with an animation of the energy distribution's evolution in time (link embedded).

5 Discussion

5.1 Results of the Metropolis algorithm simulation

The Ising model, when governed by the Metropolis algorithm, behaved exactly as we expected it would. It favors low-energy states and generates the Boltzmann distribution upon reaching its equilibrium state, just as the condition of detailed balance requires it to. It is worth noting, however, that the simulation did not actually yield exponential behavior for the spins. This is because the multiplicity of an energy macrostate, i.e. the number of

lattice configurations that produce a particular energy value, skews the distribution it generates. Theoretically, one could solve this problem by simply dividing each element of the distribution by the multiplicities of their corresponding macrostates; however, this option is highly impractical for a computer simulation like this. In a one-dimensional lattice of length 500, the size we used for all our simulations, there are 500 different spins that can either point up or down. Thus, the total number of unique configurations the lattice can occupy is $2^{500} \approx 3.27e150$. Computers generally do not handle operations involving such incredibly large numbers very well, so we elected to simply focus on the energy of the system instead. This accomplishes much the same thing as the multiplicity adjustment while also being far more reliable, easy to implement, and easy to compute.

On the matter of the temperature, the discrepancy between the experimental and expected values is likely a consequence of the probabilistic nature of the Metropolis algorithm. Within only the 6 data-points we generated, there were temperatures ranging between well under 3 and almost as high as 5, which is a significant degree of variation. It is likely that, had we run the simulation for a longer period of time, the values would have stabilised somewhat; however, the simulation was simply too computationally-intensive for that to be practical. Increasing the number of timesteps by any notable amount caused it to take a prohibitively long time to run. Nonetheless, given that the implementation of the Metropolis algorithm was largely intended to serve as a means of confirming that our implementation of the Ising model was accurate, we do not find this to be especially concerning.

Ultimately, the quality of the Metropolis algorithm's results indicates that the Ising model as we have represented it is accurate to the theory. We can therefore confidently conclude that the results generated by the simulations using the Creutz algorithm are, in fact, meaningful.

5.2 Results of the Creutz algorithm simulations, with and without bond formation

The Creutz algorithm in its 'default' form performed in accordance with all expectations, as it both generated the Boltzmann distribution and was reversible in time. As for the temperatures we calculated from the data, they were somewhat higher than we would have expected from the results of the Metropolis algorithm, but not worryingly so. Most of the values were within the observed range of experimental results from Metropolis, and since there were only three in total, we do not believe that any statistically significant conclusions can be drawn from them.

The algorithm performed similarly well when bond formation and breaking was allowed, also generating the Boltzmann distribution and being fully time-symmetrical. The temperature values we found from analysing the results of Fig. 9.2 were lower than we would have expected from the 'default' Creutz simulation's outputs, but – again – a sample size of 3 is too small to say much of anything meaningful. As the Metropolis algorithm, our benchmark for accuracy, produced much closer temperatures than the static-bond Creutz algorithm did, we are fairly sure that our results are meaningful.

5.3 Results of the Creutz algorithm simulation with variable timescale

Despite the fact that the two algorithms used during the simulation both generate the Boltzmann distribution, taken together, they very clearly do not. Rather, depending on the energy of the lattice at the moment of bond freezing, the model exclusively inhabits states of either even or odd energies for the middle third. In our case, those were the odd energy states, but another simulation might equally well have shown the opposite. At any rate, Fig. 11 shows that the algorithm does not produce results that follow the Boltzmann distribution, and Fig. 12.2 shows how the different epochs of the simulation correspond with changes in its adherence to the distribution. Evidently, the introduction of a static set of broken bonds to the lattice causes this to occur; the divergences in $\Delta \ln p$ appear and grow during this period, while they gradually begin to realign at long times. It is likely that, at $t \to \infty$, the energies would return to their original form of the Boltzmann distribution.

According to the second law of thermodynamics, the total entropy of a closed system can never increase. In fact, for a reversible process, it remains constant ("Second Law"). Analysing Fig. 12.2 from this perspective, our results become quite interesting. Despite the variable-timescale Creutz algorithm being fully reversible and deterministic, its entropy seems to decrease. When the lattice of the Ising model is in thermal equilibrium, its (unique) state of maximum entropy, algorithms for simulating it like Metropolis and Creutz generate the Boltzmann distribution; we have demonstrated this fact ourselves. Since the energy distribution was, for the first third of the simulation, in the Boltzmann distribution, we know that its entropy was at its maximum value at that point. The fact that it subsequently falls *out* of the Boltzmann distribution during the second third then indicates that the system's net entropy has actually decreased. That is the only possible explanation that remains consistent with the second law of thermodynamics. As the system then returns to the Boltzmann distribution at long times, it seems to now be returning to its initial equilibrium state of maximum entropy. The total entropy value has been decreased by a temporary freeze in the bond structure of the lattice.

As promising as these results are, however, we cannot make any definitive assertions about the relationship between the system's entropy, the second law of thermodynamics, and reversibility in time at the moment. Though they are *consistent* with what we would expect from the second law, we cannot state conclusively that the system's total entropy has decreased without actually directly calculating it ourselves. It would be very valuable to continue research into this subject with a specific focus on quantitatively measuring the lattice's entropy over time. If it were found to indeed decrease while frozen and subsequently increase, then we would have good reason to believe that we have found evidence for the arrow of time.

6 Conclusions

Ultimately, we are quite satisfied with the results of our research. Our implementation of the Ising model and the algorithms used in simulating it all behaved as we expected them to on their own. Furthermore, combining some of them together in the variable-timescale approach produced highly encouraging results that are very indicative of the connection between maximum entropy and the arrow of time we sought to find. There are a number of areas where the simulation and algorithms could be improved and expanded upon, particularly with regards to entropy calculations and larger simulation times/sizes, and further research along these lines would be well worth conducting.

Tabulated below are the results we found for the lattice's temperatures under the various algorithms:

Metropolis	**Creutz (static bonds)**	**Creutz (dynamic bonds)**
4.1 ± 1.1 *(Expected: 2.5)*	5.0 ± 0.3	2.5 ± 0.2

7 References

1. M. Creutz, "Deterministic Ising Dynamics" (May 1985), *Annals of Physics*, vol. 167, pp. 62-72.

2. M. Creutz, "Microcanonical Monte Carlo Simulation" (Feb. 1983), *Physical Review Letters*, vol. 50, no. 19

3. T. Gold, "Why Time Flows: The Physics of Past & Future" (2003), *Daedalus*, vol. 132, no. 2, On Time (p. 37-40). Retrieved from `https://www.jstor.org/stable/20027838`.

4. R. Müller, "The Boltzmann factor: A simplified derivation," Technische Universität Braunschweig. Retrieved from `https://www.tu-braunschweig.de/index.php?eID=dumpFile&t=f&f=138377&token=ce78f1a73be3528669c0a5a4a6675d0e3284b02e`. Accessed March 2023.

5. M.E.J. Newman, G.T. Barkema, *Monte Carlo Methods in Statistical Physics* (1999), Clarendon Press.

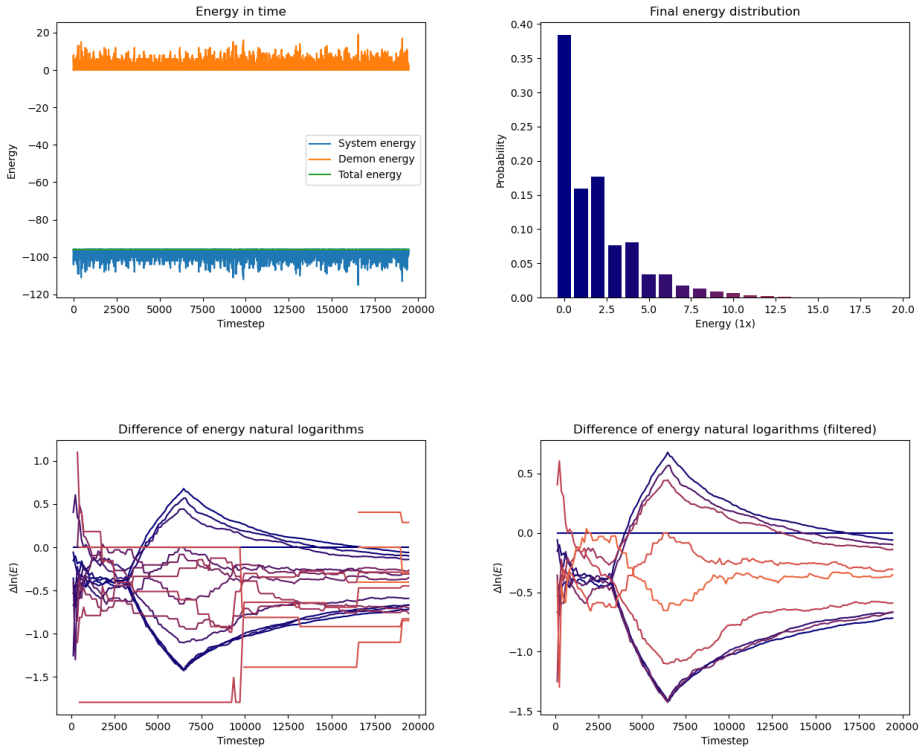
6. "Second Law of Thermodynamics," National Aeronautics and Space Administration (2021). Retrieved from `https://www.grc.nasa.gov/www/k-12/airplane/thermo2.html`. Accessed March 2023.

7. U. Tracht *et al*, "Length Scale of Dynamic Heterogeneities at the Glass Transition Determined by Multidimensional Nuclear Magnetic Resonance" (Sept. 1998), *Physical Review Letters*, vol. 81, no. 13.

8 Appendices

All code can also be found on GitHub.

8.1 Results from Creutz algorithm (variable timescale) for the original lattice



8.2 'Bond' object

```
class Bond:
'''
Represents the bond between two sites in the lattice.

### Attributes
bd: Whether or not the bond exists
'''
bd = None

def __init__(self, bu: bool=True):
    '''
```

```
        Initialises the bond.

        ### Parameters
        bu: Whether or not the bond exists
        '''
        self.bd = bu

    def mk(self):
        '''
        Makes the bond.
        '''
        self.bd = True

    def brk(self):
        '''
        Breaks the bond.
        '''
        self.bd = False

    def flip(self):
        '''
        Makes a broken bond or breaks a made bond.
        '''
        self.bd = not self.bd

    def __int__(self):
        '''
        Represents the bond as an integer.
        '''
        return int(self.bd)

    def __add__(self, b2):
        '''
        Allows the '+' operator to be used with bond objects.

        ### Parameters
        b2: The subject of the addition. Can be any type capable of being
            cast as an integer
        '''
        return int(self) + int(b2)

    __radd__ = __add__ # (Allows addition from the other direction, i.e. 5
        + b instead of b + 5)

    def __mul__(self, b2):
        '''
        Allows the '*' operator to be used with bond objects.
```

```
        ### Parameters
        b2: The subject of the multiplication. Can be any type capable of
            being cast as an integer
        '''
        return int(self) * int(b2)

    __rmul__ = __mul__ # (Allows multiplication from the other direction, i
        .e. 5*b instead of b*5)

    def __bool__(self) -> bool:
        '''
        Represents the bond as a boolean.
        '''
        return self.bd

    def __repr__(self):
        '''
        Represents the bond as a string, specifically for console printing
            purposes.
        '''
        return str(self.bd)
```

8.3 'Site' object

```
from bonds import Bond

class Site:
    '''
    Represents an individual site within the lattice.

    ### Attributes
    st: The direction the site points. 'True' is up, 'False' is down
    ln: The site to its left ('left neighbor')
    lb: The bond connecting it to its left neighbor
    rn: The site to its right ('right neighbor')
    rb: The bond connecting it to its right neighbor
    ind: The index of the site within the lattice
    J: Interaction energy
    B: Magnetic field strength

    (NB: J and B are not meant to be changed from their default values!)
    '''
    st = False
    ln = None
    lb = None
    rn = None
    rb = None
```

```
ind = None
J = 1
B = 0

def __init__(self, ud: bool, ln: 'Site'=None, lb: Bond=None, rn: 'Site'
    =None, rb: Bond=None, ind: int=None, J: float=1, B: float=0):
    '''
    Initialises the site.

    ### Parameters
    ud: The direction the site points. `True` is up, `False` is down
    ln: The site to its left ('left neighbor')
    lb: The bond connecting it to its left neighbor
    rn: The site to its right ('right neighbor')
    rb: The bond connecting it to its right neighbor
    ind: The index of the site within the lattice
    J: Interaction energy
    B: Magnetic field strength
    '''
    self.st = ud
    self.ln = ln
    self.lb = lb
    self.rn = rn
    self.rb = rb
    self.ind = ind
    self.J = J
    self.B = B

def flip(self):
    '''
    Flips the orientation of the site.
    '''
    self.st = not self.st

def setNs(self, l: 'Site', r: 'Site'):
    '''
    Sets the site's neighbors.

    ### Parameters
    l: The site's future left neighbor
    r: The site's future right neighbor
    '''
    self.ln = l
    self.rn = r

@property
def E(self) -> int:
    '''
```

```
        Returns the energy of the site.
        '''
        return -(self.J*self*(self.rn*self.rb + self.ln*self.lb) + self.B*
            self)

    def __repr__(self):
        '''
        Represents the site as a string, specifically for console printing
            purposes.
        '''
        return str(self)

    def __int__(self) -> int:
        '''
        Represents the site as a number. Positive is up, negative is down.
        '''
        if self.st: return 1
        else: return -1

    def __add__(self, s2):
        '''
        Allows the '+' operator to be used with site objects.

        ### Parameters
        s2: The subject of the addition. Can be any type capable of being
            cast as an integer
        '''
        return int(self) + int(s2)

    __radd__ = __add__ # (Allows addition from the other direction, i.e. 5
        + s instead of s + 5)

    def __mul__(self, s2):
        '''
        Allows the '*' operator to be used with site objects.

        ### Parameters
        s2: The subject of the multiplication. Can be any type capable of
            being cast as an integer
        '''
        return int(self) * int(s2)

    __rmul__ = __mul__ # (Allows multiplication from the other direction, i
        .e. 5*s instead of s*5)

    def __str__(self) -> str:
        '''
        Represents the site's direction with a single text character.
```

```
        '''
        if self.st: return '^'
        else: return 'v'
```

8.4 'Lattice' object

```
import numpy as np
from sites import Site
from bonds import Bond

class Lattice:
    '''
    The (1D) lattice of spins. This is the core of the model.

    ### Attributes
    lat: The array of sites
    J: Interaction energy
    B: Magnetic field strength
    E: System energy
    dE: Demon energy
    size: Lattice size, i.e. length in this case

    (NB: J and B are not meant to be changed from their default values!)
    '''
    lat = np.empty(0, dtype=Site)
    J = 1
    B = 0
    E = 0
    dE = 0
    size = 0

    def __init__(self, size: int=500, intEn: float=1, demonEn: float=0,
        magF: float=0, dir: bool=None, align: float=None):
        '''
        Initialises the lattice.

        ### Parameters
        size: Lattice size
        intEn: Interaction energy
        demonEn: Initial demon energy
        magF: Magnetic field strength
        dir: Initial direction of all spins (`True` is up, `False` is down)
        align: Ratio of up- to down-spins
        '''
        self.J = intEn
        self.B = magF
        self.size = size
```

```
        self.lat = np.empty(size, dtype=Site)
        altSpin = True
        for s in range(self.size): # Creates the initial set of spins &
            bonds
            spin = True
            if dir is not None: spin = dir
            else: spin = altSpin
            if s%2 == 1: altSpin = not altSpin
            self.lat[s] = Site(ud=spin, ind=s, J=self.J, B=self.B)
            self.lat[s].rb = Bond()
        for s in range(self.size): # Joins the spins and bonds together
            properly
            self.lat[s].setNs(self.lat[s-1], self.lat[(s+1)%self.size])
            self.lat[s].lb = self.lat[s-1].rb
        if align is not None: # Generate directions based on 'align'
            align = align%1
            for s in range(self.size): self.lat[s].st = np.random.choice([
                False, True], p=[1-align, align])
        self.E = self.energy
        self.dE = demonEn

    @property
    def energy(self) -> int:
        '''
        Calculates the instantaneous energy of the system.
        (NB: Avoid using inside loops! The lattice can get quite large, so
            running this frequently will make the simulation take much
            longer.
        Store and update the energy dynamically instead!)

        ### Returns
        tot: Total energy of system
        '''
        tot = 0
        for s in self.lat: tot += -self.J*s*s.rn*s.rb - self.B*s
        return tot

    def metropolis(self, site: Site, temp: float=5, *args, **kwargs):
        '''
        Decides whether or not to update a particular site according to the
             Metropolis algorithm.

        ### Parameters
        site: The site to be checked
        '''
        # 'temp' is measured relative to the Boltzmann constant, i.e. kT =
            temp
        b = 1/temp
```

```
        diffE = 2*(self.J*site*(site.rn + site.ln) + self.B*site) #
            Calculates difference in pre- and post-flip energy
        doFlip = True
        if diffE > 0:
            probOfFlip = np.e**(-b*(diffE))
            doFlip = np.random.choice([False, True], p=[1-probOfFlip,
                probOfFlip])
        if doFlip:
            # site.flip()
            self.trans(diffE)
            if abs(self.E) >= self.size: self.trans(-diffE)

    def demon(self, site: Site, brk: bool=True, rev: bool=False, **kwargs):
        '''
        Decides whether or not to update a particular site according to the
             Creutz algorithm.

        ### Parameters
        site: The site to be checked
        brk: Allow/disallow bond breaking/making
        rev: Whether or not the system is currently reversing
        '''
        order = [self.flip, self.bonds]
        if not brk: order.pop()
        if rev: order = np.flip(order)
        [f(site) for f in order]

    def flip(self, site: Site):
        '''
        Decides whether or not to flip the site. Will always occur if the
            demons have enough energy for it.

        ### Parameters
        site: The site to be checked
        '''
        diffE = -2*site.E
        if diffE <= self.dE:
            self.trans(diffE)
            site.flip()

    def bonds(self, site: Site):
        '''
        Decides whether or not to break the site's (right) bond. Will
            always occur if the demons have enough energy for it.

        ### Parameters
        site: The site to be checked
        '''
```

```
            initE = site.E
            site.rb.flip()
            diffE = site.E - initE
            if diffE <= self.dE: self.trans(diffE)
            else: site.rb.flip()

        def trans(self, E: int):
            '''
            Transfers energy from the demons to the system.

            ### Parameters
            E: The quantity of energy to be transferred over
            '''
            self.E += E
            self.dE -= E

        def __repr__(self):
            '''
            Represents the lattice as a string. '|' indicates a broken bond.
            (NB: Used only when printing to the console - this is not a general
                 string method!)

            ### Returns
            rv: The string representation of the lattice
            '''
            rv = ''
            for s in self.lat:
                rv += str(s)
                if s.rb: rv += ' '
                else: rv += '|'
            return rv
```

8.5 Function library

```
'''
A general-purpose collection of functions used in processing the simulation
    's outputs.
'''

import sys
import os
import numpy as np
import lattice as lat
import matplotlib.pyplot as plt
import random as rand
import pickle as pkl
from numpy import cos, pi
```

```
def longest(l: list) -> int:
    '''
    Finds the length of the longest element in a list (excluding trailing
        zeros).

    ### Parameters
    l: The list of iterables

    ### Returns
    maxLen: The longest length in l
    '''
    maxLen = -1
    for a in l:
        a = np.trim_zeros(a, 'b')
        if a.size > maxLen: maxLen = a.size
    return maxLen

def memAdd(o):
    '''
    Prints the memory address of a given object.

    ### Parameters
    o: The object
    '''
    print(hex(id(o)))

def genRandList(l: int, size: int, rev: bool=True) -> np.ndarray:
    '''
    Generates a list of random positions for the simulation to run through.

    ### Parameters
    l: The length of the desired list
    size: The size of the lattice
    rev: Whether or not the system is supposed to reverse

    ### Returns
    rL: The randomly-generated list
    '''
    rL = np.array([int(rand.random()*size) for i in range(l)])
    if rev: rL = np.append(rL, np.flip(rL))
    return rL

def procHex(*args):
    '''
    Turns a sequence of numbers (between 0 and 256) into a hexadecimal
        color code.
    Typically, only 3 inputs should be used (RGB), but a fourth is
```

```
        supported (RGBA).

    ### Parameters
    *args: The sequence of numbers
    '''
    return '#'+''.join([f'0{hex(s)[2:]}'[-2:] for s in args])

def getColor(i: float):
    '''
    Calculates RGB color based on position in a custom gradient and returns
         it as a hexadecimal color code.

    ### Parameters
    i: A decimal between 0 and 1 representing position within the gradient.
    '''
    r = int(-120*cos(pi*i) + 120)
    g = int(-120*cos(0.5*pi*i) + 120)
    b = int(70*cos(0.75*pi*(0.5*i + 0.25)) + 70)
    return procHex(r,g,b)

def dispGrad():
    '''
    Displays the gradient described in `getColor()` as an image.
    '''
    grad = np.linspace(0, 1, 100)
    for x in grad:
        plt.plot([x for z in np.zeros_like(grad)], grad, 'o', color=
            getColor(x+.0))
    plt.savefig('grad.png')
    plt.show()

def printDist(dist: np.ndarray, xax: str='t'):
    '''
    Prints a given distribution to the console.

    ### Parameters
    dist: The distribution to be printed
    xax: The variable on the x-axis of the distribution. Default is time,
        set to 'e' for energy
    '''
    name = 'Time block'
    if xax == 'e': name = 'Energy'
    [print(f'{name} {a}: {dist[a]}') for a in range(dist.shape[0])]

def norm(dist: np.ndarray):
    '''
    Normalises a given distribution.
```

```
    ### Parameters
    dist: The distribution to be normalised
    '''
    return dist/sum(dist)

def genLats(n: int=10, s: int=500, a: float=0.7):
    '''
    Generates several .lat files containing lattice objects.

    ### Parameters
    n: The number of lattices to generate
    s: The size of the lattices
    a: The alignment fraction to use (0-1)
    '''
    checkDir('lats')
    sys.setrecursionlimit(100000)
    for i in range(n):
        l = lat.Lattice(size=s, align=a)
        with open(f'lats/{l.energy}.lat', 'wb') as f: pkl.dump(l, f)

def load(l: lat.Lattice):
    '''
    'Load a lattice from a file,' sort of. It doesn't actually 'load'
        anything, per se, but it's useful for typing purposes (i.e.
    telling Python that the object loaded from the file is, in fact, a '
        Lattice').

    ### Parameters
    l: The lattice object to load.
    '''
    return l

def checkDir(*args):
    '''
    Make sure the directory in question does, in fact, exist. Used in file
        saving.

    ### Parameters
    args: The directory(ies) to check. Works within the directory of the
        project.
    '''
    for dir in args:
        if not os.path.exists(dir): os.mkdir(dir)
```

8.6 Main file for running the program

```
'''
```

```
Written for Python 3.9.12
Using Anaconda for packages

Author: Aiden Lewis
'''

from lib import *

def run(temp=2.5):      # 'temp' is used only in the Metropolis algorithm
    ### SPECIFY INPUT PARAMETERS HERE ###
    size = 500          # Sets lattice size
    doDemons = True     # Decides whether to use the Creutz or the
        Metropolis algorithm (Creutz is 'True')
                            # (If 'False', all other conditions will be
                                updated accordingly - they only matter if
                                it's 'True')

    doFreeze = True     # Turn bond freezing on/off for the middle third
    doBrks = True       # Decides whether or not to allow bond formation/
        breaking
    time = 20           # Change this and only this if you want to run it
        for more/less time

    ### INITIAL SETUP ###
    genLats(s=size)
    simLat = 'main'
    timeExtension = 0
    blockCount = 81 # This should never need changing, but if it is, keep
        it at a multiple of 3 so bond formation/breaking works properly
    if doDemons and (doBrks and doFreeze): simLat = 'cvar'
    with open(f'{simLat}.lat', 'rb') as f: l = pkl.load(f)
    l = load(l)
    if not doDemons: l.dE = -size - l.energy
    initL = l.__repr__()
    numSteps = 6*blockCount*time
    name = 'metro'
    algr = l.metropolis
    eLim = 16
    if doDemons:
        name = 'creutz'
        algr = l.demon
    else:
        doBrks = False
    if not doBrks: doFreeze = True
    if doBrks and doFreeze:
        eLim = 50
        timeExtension = 1
    order = genRandList(int(numSteps/2), size)
```

```
rev = False
eDiffBlock = 6*time
eDiffsUnp = []
if timeExtension: order = np.array([np.append(order, genRandList(
    numSteps, size, rev=False)) for i in range(timeExtension)]).flatten
    ()
energies = np.zeros_like(order, dtype=float)
energies = np.append(energies, 0.0)
demonEnergies = np.zeros_like(energies, dtype=float)
dEdist = np.zeros_like(energies, dtype=float)
energies[0] = l.energy
demonEnergies[0] = l.dE
dEblock = np.zeros_like(dEdist, dtype=float)
sys.setrecursionlimit(10000)
factor = 4
if doBrks: factor = 1
with open(f'init-{name}.lat', 'wb') as f: pkl.dump(l, f)
checkDir(f'blocks-{name}')

### SIMULATION LOOP ###
for i in range(order.size):
    if i >= numSteps/2: rev = True
    if i >= numSteps/2 - numSteps/6 and i < numSteps/2 + numSteps/6:
        brk = not doFreeze
    else: brk = doBrks
    algr(l.lat[order[i]], brk=brk, rev=rev, temp=temp)
    energies[i+1] = l.E
    demonEnergies[i+1] = l.dE
    dEblock[abs(int(l.dE/factor))] += 1
    if i%eDiffBlock == 0 and i != 0:
        eDiffsUnp.append(norm(dEblock.copy()))
        dEblock_trim = np.trim_zeros(dEblock, 'b')
        ind = int(i/eDiffBlock)
        clrs = [getColor(e/dEblock_trim.size) for e in range(
            dEblock_trim.size)]
        plt.bar(range(dEblock_trim.size), norm(dEblock_trim), color=
            clrs)
        plt.xlabel('Energy')
        plt.ylabel('Probability')
        plt.title(f'Distribution at time block {ind}')
        plt.savefig(f'blocks-{name}/{ind}.png')
        plt.close()
with open(f'fin-{name}.lat', 'wb') as f: pkl.dump(l, f)
if doDemons and l.__repr__() == initL: print('Reversibility condition
    upheld!')
elif doDemons and not timeExtension: print('Reversibility condition
    failed!')
```

```
### ENERGY LOGARITHM DIFFERENCE PROCESSING ###
eDiffsUnp.append(norm(dEblock))
eDiffsUnp = np.array(eDiffsUnp)
dEdist = dEblock
eDiffsNZ = []
lgt = longest(eDiffsUnp)
if lgt > eLim: lgt = eLim
for e in eDiffsUnp:
    dEdist += e
    e[lgt] = '7'
    eDiffsNZ.append(np.trim_zeros(e, 'b')[:lgt])
eDiffsNZ = np.array(eDiffsNZ)
dEdist = np.trim_zeros(dEdist)
eDiffsLn = np.log(eDiffsNZ)
eDiffs = []
for e in eDiffsLn:
    diff = np.zeros(e.size)
    for i in np.arange(1, e.size):
        diff[i] = e[i] - e[i-1]
        if abs(diff[i]) == np.inf or i-1 < 0: diff[i] = None
    eDiffs.append(diff)
eDiffs = np.array(eDiffs)

### RESULTS PLOTTING ###
# Energy over time
plt.plot(energies, label='System energy')
if doDemons:
    plt.plot(demonEnergies, label='Demon energy')
    plt.plot(demonEnergies + energies, label='Total energy')
plt.legend()
plt.ylabel('Energy')
plt.xlabel('Timestep')
plt.title('Energy in time')
plt.savefig(f'energy-{name}.png')
plt.close()

# Final energy distribution (ln)
lbl = 'demon energy'
if not doDemons: lbl='potential energy'
dEdist = dEdist[:eLim]
colors = [getColor(e/dEdist.size) for e in range(dEdist.size)]
dEdist2 = np.log(norm(dEdist))
plt.bar(range(dEdist2.size), dEdist2, color=colors)
plt.ylabel(r'$\ln p$')
plt.xlabel(f'Energy ({factor}x)')
plt.title(f'Final {lbl} distribution')
plt.savefig(f'edist-{name} (ln).png')
plt.close()
```

```
    # Final energy distribution (not ln)
    dEdist = norm(dEdist)
    plt.bar(range(dEdist.size), dEdist, color=colors)
    plt.ylabel('Probability')
    plt.xlabel(f'Energy ({factor}x)')
    plt.title(f'Final {lbl} distribution')
    plt.savefig(f'edist-{name}.png')
    plt.close()

    # Energy differences
    tblocks = np.arange(1, (2*timeExtension + (not timeExtension))*
        blockCount+1)*eDiffBlock
    eDiffsT = np.transpose(eDiffs)
    [plt.plot(tblocks, eDiffsT[e], color=getColor(e/len(eDiffsT))) for e in
         range(len(eDiffsT))]
    plt.xlabel('Timestep')
    plt.ylabel(r'$\Delta\ln p = \langle\beta\rangle$')
    plt.title('Difference of energy natural logarithms (unfilt)')
    plt.savefig(f'ediffs-{name}.png')
    plt.close()

    # Energy differences (filtered)
    tblocks = np.arange(1, (2*timeExtension + (not timeExtension))*
        blockCount+1)*eDiffBlock
    eDiffsT = np.transpose(eDiffs)
    t1 = 0
    for e in range(int(len(eDiffsT)/2)):
        plt.plot(tblocks, eDiffsT[e], color=getColor(2*e/len(eDiffsT)),
            label=f'E={factor*e}')
        if not timeExtension:
            temp = -factor/np.mean((eDiffsT[e])[int(len(eDiffsT[e])/2):])
            if e == 1: t1 = temp
            print(f'Temperature (E={factor*e}):\t{temp}')
    plt.xlabel('Timestep')
    plt.ylabel(r'$\Delta\ln p = \langle\beta\rangle$')
    plt.title('Difference of energy natural logarithms')
    plt.savefig(f'ediffs-{name}-filtered-noleg.png')
    plt.legend()
    plt.savefig(f'ediffs-{name}-filtered.png')
    plt.close()

    return t1

if __name__ == '__main__':
    run()
```