Thermal Creation of Kink Anti-kink Pairs

Adam Furlong, Cillian Grall

May 2024

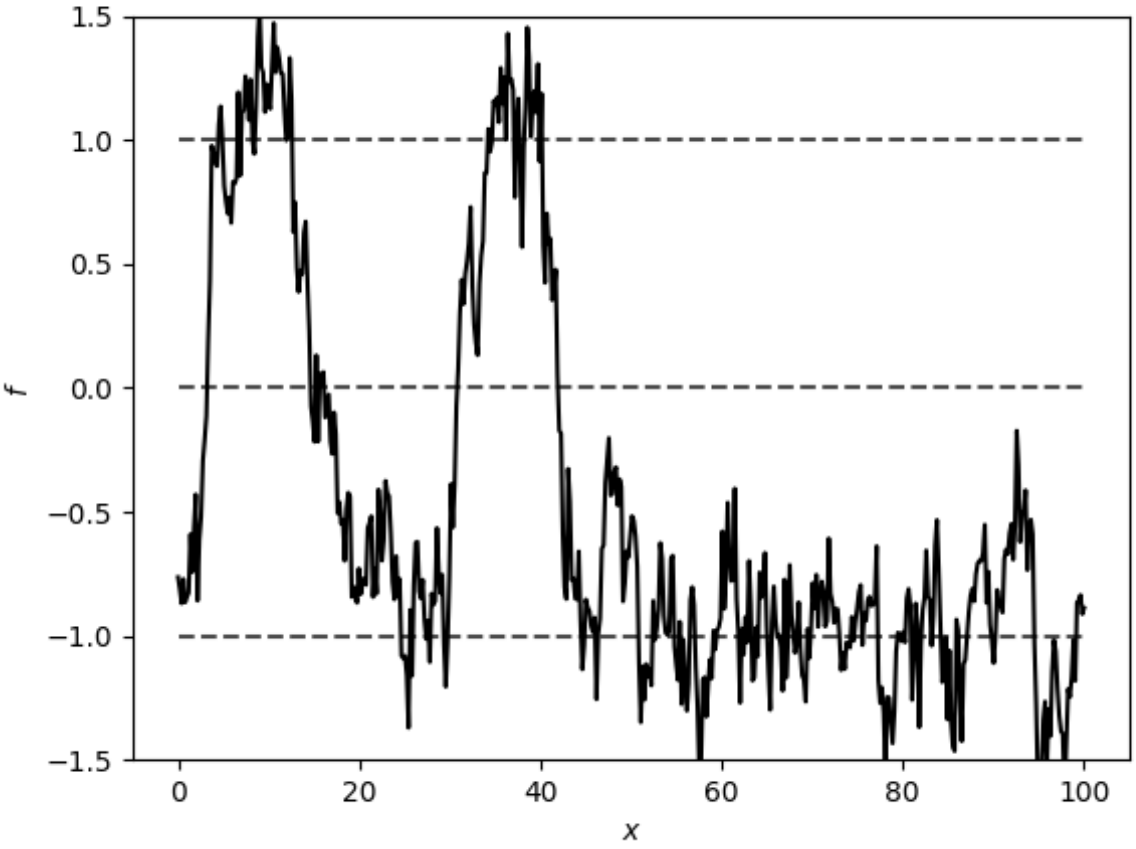


Figure 1: Example of a Field Configuration containing 2 Kink Anti-Kink pairs.

Contents

1 Introduction

1.1 The System

Consider a potential $V(f)$ with multiple local minima, and the wave equation describing the dynamics of a scalar field f subject to this potential:

$$\partial_{tt} f = \nu^2 \nabla^2 f - V'(f) \tag{1}$$

For simplicity we set the propagation velocity ν to unity. The Hamiltonian energy density is then given by:

$$\mathcal{H} = \frac{1}{2}(\partial_t f)^2 + \frac{1}{2}(\nabla f)^2 + V(f) \tag{2}$$

In the ground state, the field will sit in the lowest potential well. Due to the non-linearity of the field equation (1), if some energy is added, the system will develop thermal noise and evolve stochastically. Moreover, a small region of the field may cross into a different well of the potential. If such a configuration becomes stable and propagates through the field, it acts as a particle known as a *soliton*. However, if such a soliton meets an anti-soliton, they may annihilate. In Figure 1. we illustrate an example of such a thermally noisy (1+1) dimensional field, with two kink anti-kink pairs, where the potential minima are located at $f = 1, f = -1$.

In quantum theory, all particles are solutions to wave equations of the quantum fields. It is possible that in the extreme environment of the early Universe, some quantum field subject to such a multi-welled potential may have had sufficient thermal energy to create solitons. As the universe cooled, the field lost the thermal energy required to cause the soliton to transition back to the ground state. Assuming it didn't annihilate, such a solition would still wander the cosmos today. Conjectured examples include magnetic monopoles[1] and skyrmions[2], though none have been detected to date.

1.2 Aim of Project

In this report we consider a very simple example of this situation; a (1+1) dimensional field $f(x,t)$ subject to the double-welled potential $V(f)$:

$$V(f) = \frac{\lambda}{4}(f^2 - c^2)^2 \tag{3}$$

c^2 is a dimensionless quantity, representing the location of the minimum. It can also be interpreted as the square of the vacuum expectation value of the field. λ is a coupling constant to the field and has dimensions of mass squared.

The classical solution (kink) is given by ϕ. Its mass is given by M_s:

$$\phi(x) = c\tanh\left(c\sqrt{\frac{\lambda}{2}}x\right), \quad M_s = \sqrt{\frac{8\lambda}{9}}c^3 \tag{4}$$

We shall consider the case where $c = 1$ and $\lambda = \frac{1}{2}$. The field equations (1), total energy (2), kink solution and kink mass (4) are then:

$$\frac{\partial^2 f}{\partial t^2} = \frac{\partial^2 f}{\partial x^2} - \lambda f^3 + \lambda f \tag{5}$$

$$E = \int \frac{1}{2}\left(\frac{\partial f}{\partial t}\right)^2 + \frac{1}{2}\left(\frac{\partial f}{\partial t}\right)^2 + \frac{\lambda}{4}(f^2 - 1)^2 \tag{6}$$

$$\phi(x) = \tanh\left(\sqrt{\frac{\lambda}{2}}x\right) = \tanh\left(\frac{1}{2}x\right) \tag{7}$$

$$M_s = \sqrt{\frac{8\lambda}{9}} = \frac{2}{3} \tag{8}$$

For computation purposes, we consider f on a domain of length L with periodic boundary conditions imposed: $f(x) = f(x + L)$. We then discretise the domain into N nodes, evenly spaced by a distance $\Delta x = \frac{L}{N}$. We shall split time into evenly spaced timesteps of duration Δt.

Our aims are to:

1. Numerically simulate (5) using the finite difference method.
2. Numerically evaluate (6), ensuring the energy remains stable over time as the system evolves according to (5).
3. Prepare thermalised initial conditions over a wide range of energies using the Metropolis Hastings algorithm.
4. Track the number of kink anti-kink pairs, n, over time, over a range of energies. This requires creating an artificial definition of a kink anti-kink pair.
5. Ensure that for a range of energies, the expected number of pairs obeys the Boltzmann Law:

$$\langle n \rangle \sim \exp\left(\frac{M_s}{T}\right) \tag{9}$$

6. Investigate the energy dependence of the pair creation rate Γ and the pair creation time $\tau_0 = 1/\Gamma$.
7. Throughout we shall compare our results to those in Grigoriev and Rubakov[1].

2 Numerical Simulation

To numerically simulate the field equation (5), we shall use the finite difference method, outlined below. Its main advantage is that we only require to store two field configurations at any one time; we can calculate the next timestep from the previous two. This method is known to be stable for $\Delta t < \dfrac{\Delta x}{2}$, and so we choose $\Delta t = \dfrac{\Delta x}{4}$.

2.1 Finite Difference Method

In this method, we replace the derivatives in equation (1) with the following symmetric second derivative expressions:

$$\begin{aligned} \partial_{tt} f &\approx \frac{f(x,t-\Delta t) - 2f(x,t) + f(x,t+\Delta t)}{\Delta t^2} \\ \partial_{xx} f &\approx \frac{f(x-\Delta x,t) - 2f(x,t) + f(x+\Delta x,t)}{\Delta x^2} \end{aligned} \tag{10}$$

With these substitutions we solve for the next time step $f(x, t+\Delta t)$:

$$\begin{aligned} f(x,t+\Delta t) = &\left(2 - 2\frac{\Delta t^2}{\Delta x^2} + \lambda \Delta t^2\right) f(x,t) + \frac{\Delta t^2}{\Delta x^2}\left(f(x-\Delta x,t) + f(x+\Delta x,t)\right) \\ &- f(x,t-\Delta t) - \lambda \Delta t^2 f(x,t)^3 \end{aligned} \tag{11}$$

Relabelling:

$$\begin{aligned} f_{i+n,\text{old}} = f(x+n\Delta x, t-\Delta t) \qquad f_{i+n} = f(x+n\Delta x, t) \qquad f_{i+n,\text{new}} = f(x+n\Delta x, t+\Delta t) \\ C_1 = 2 - 2\frac{\Delta t^2}{\Delta x^2} + \lambda \Delta t^2 \qquad C_2 = \frac{\Delta t^2}{\Delta x^2} \qquad C_3 = -\lambda \Delta t^2 \end{aligned} \tag{12}$$

We can rewrite the update instruction (11) as:

$$f_{i,\text{new}} = -f_{i,\text{old}} + C_1 f_i + C_2\left(f_{i+1} + f_{i-1}\right) + C_3 f_i^3 \tag{13}$$

We shall consider three possible implementations of the finite difference method, choosing the fastest.

Method 1. Explicit Loop
The update instruction (13) can be implemented as a simple for-loop over the N nodes.

Method 2. Matrix Multiplication

Notice that all but the first and last terms in (13) are linear in the current state f. We can consider a linear operator M acting on the current state f, such that these linear terms are given by the matrix multiplication $M * f$. The update instruction (13) can be written and implemented as follows:

$$f_{\text{new}} = -f_{\text{old}} + M * f + C_3 f^3 \tag{14}$$

where M is the symmetric sparse matrix given by:

$$M_{i,k} = \begin{cases} C_1 & i = k \\ C_2 & i \equiv k \pm 1 \pmod N \\ 0 & \text{else} \end{cases} \tag{15}$$

Method 3. Rolling Array:
Notice instead that the terms f_{i+1}, f_{i-1} are simply the array f with the index shifted by one. The 'numpy' package has a module 'numpy.roll' that can shift arrays by an index like this. The update instruction (13) can be written and implemented as found in next_timestep.

$$f_{\text{new}} = -f_{\text{old}} + C_1 f + C_2 \left(f_{i\to i+1} + f_{i\to i-1}\right) + C_3 f_i^3 \tag{16}$$

where $f_{i\to i+1}$ and $f_{i\to i-1}$ are the current array f with the index shifted up and down by one respectively.

Here we compare the speeds of our three implementations. We prepare field configurations at temperature $T = 1$ using the Metropolis-Hastings Algorithm as set out in Section 3. The evolution of the fields is traced for 10^5 timesteps, with the times and speeds recorded in the table below. We shall often be running simulations for longer time periods than this, so we think it necessary to choose the fastest of the three methods; the Rolling Array.

Method	Time Taken (seconds)	Time-Steps per second
Explicit Loop	23.9	4178
Matrix	6.74	14837
Rolling Array	3.05	32800

2.2 Energy

We now wish to evaluate the total energy of our system according to (6). To take full advantage of the benefits of the finite difference method, we wish to calculate energy from only two points in time. Using a backward first derivative to replace $\partial_t f$ and a symmetric first derivative to replace $\partial_x f$, the energy expression (6) then becomes the following sum over the N nodes.

$$E = \sum_{i=1}^{N} \left[\frac{1}{2} \left(\frac{f_i - f_{\text{old},i}}{\Delta t} \right)^2 + \frac{1}{2} \left(\frac{f_{i+1} - f_{i-1}}{2\Delta x} \right)^2 + \frac{\lambda}{4} (f_i^2 - 1)^2 \right] \tag{17}$$

Notice there are three terms in the summation; a kinetic term, an interaction term and a potential term. We will be interested how the energy is distributed between them:

Kinetic Term: $$K = \frac{1}{2\Delta t} \sum_{i=1}^{N} \left(f_i - f_{\text{old},i}\right)^2 \tag{18}$$

Interaction Term: $$I = \frac{1}{4\Delta x^2} \sum_{i=1}^{N} \frac{1}{2} \left(f_{i+1} - f_{i-1}\right)^2 = \frac{1}{4\Delta x^2} \sum_{i=1}^{N} f_i (f_i - f_{1-2}) \tag{19}$$

Potential Term: $$P = \frac{\lambda}{4} \sum_{i=1}^{N} (f_i^2 - 1)^2 \tag{20}$$

Each of these can be implemented as sums over numpy arrays. The total energy is simply the sum of the three terms. These are how we are going to determine the energy distribution of our field from the "energy"function.

3 Initial Conditions

We now wish to evaluate the energy output, but first we need to generate viable initial conditions. To do this we created a function we call Fourier Initial Conditions. This code was used to prepare the starting configuration of the system to be as close to thermal equilibrium as possible whilst having relatively short computation time. It accomplishes this by evaluating the proper thermal amplitudes of the spectral components of f, multiplies them by random phases to imitate the thermal noise, and saves the result in an array. It then takes a fourier transform of the array and then assigns the real part of the result to the arrays f and f_{old}. The energy of the system was controlled by a scaling variable.

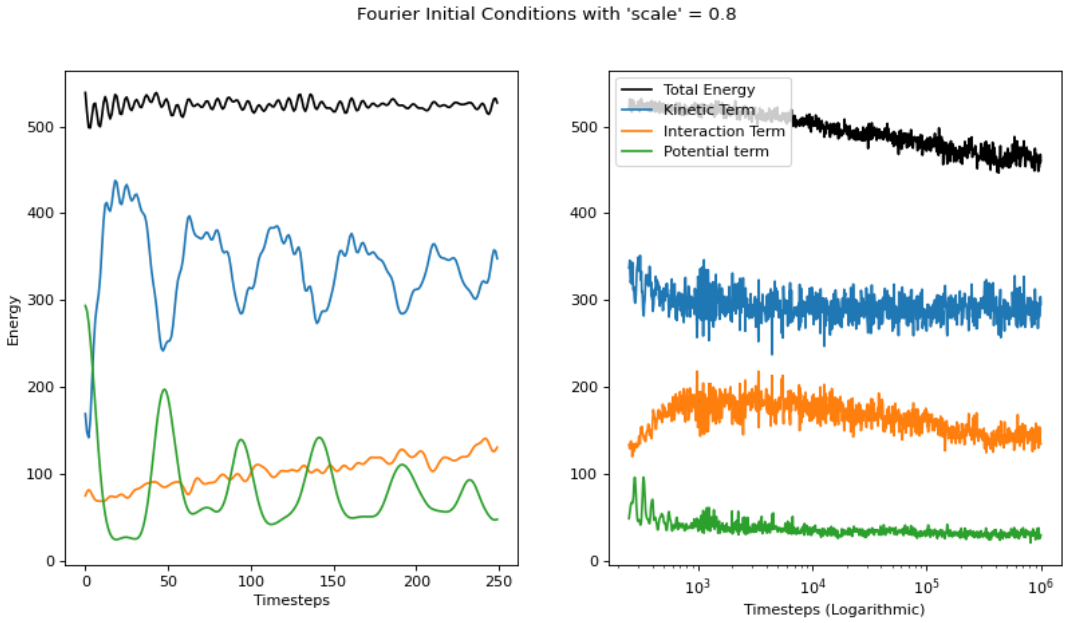


Figure 2: Time dependence of energy distribution. Energy terms are measured every timestep for 250 timesteps, then 1000 measurements are spread logarithmically until 10^6 timesteps

From the graph on the left in Figure (2) we see the total energy in the energy distribution outputted by the Fourier Initial Conditions seems to be relatively stable. However we can see that all terms in the distribution deviate significantly from their initial values and continuously oscillate. From the graph of the right in Figure (2) we can see that the energy appears to stabilise somewhat after approximately 500 timesteps. However there is a significant amount of variation in the energy. The biggest issue is that we can clearly see a gradual decrease in energy of the system over time. This means the energy is not conserved. It is this gradual decrease and oscillation in the energy distribution that makes kink counting difficult.

In place of this, we developed code to initialise that state of the system using the Markov Chain Monte Carlo - Heat Bath algorithm.

3.1 Metropolis Hastings Algorithm - Formulation

We need to develop a set of initial conditions f, f_{old} of energy E with a stable energy distribution.

Consider the ground state field configuration where f and f_{old} are constant at the same potential minimum. Now to each node in turn, attach it to a heat bath, of temperature T. This will allow this node to fluctuate, and when the heat bath is disconnected, the node will be left with value z according to the probability distribution $p(z)$.

$$p(z) = \frac{\exp(-E(z)/T)}{Z}, \qquad Z = \int_{-\infty}^{\infty} \exp(-E(y)/T)\mathrm{d}y \tag{21}$$

$E(z)$ is the energy associated with the field when this node has value z. $\exp(-E(z)/T)$ is the Boltzmann factor and Z is the canonical partition function.

Suppose we create an algorithm to mimic this effect. For a node k, we change the value of the the field at that node from y to z according to a probability distribution $P(z|y)$. After passing all N nodes, we evolve the system by a time-step according to (3). We iterate this process until we see that the energy distributions stabilises. This is an example of a Monte Carlo Markov Chain; the system is moved through the space of possible states by this random process, ending up in more probable states.

The following properties are sufficient to ensure the algorithm will approach an equilibrium state[4]:

1. P a probability distribution.

$$P(z|y) \geq 0 \ \forall y, z, \quad \int P(z|u)\mathrm{d}u = 1 \tag{22}$$

2. Detailed balance.

$$\frac{P(z|y)}{P(y|z)} = \frac{p(z)}{p(y)} \tag{23}$$

We can think of $P(z|y)$ as two separate operations; proposing a state z to move to, and then accepting this change. The proposal probability distribution $G(z|y)$ is the conditional probability of proposing to change the node value of z, given its current value y. The acceptance probability distribution $A(z|y)$ is the conditional probability of accepting this change. Then we write:

$$P(z|y) = G(z|y)A(z|y) \tag{24}$$

With $\Delta E = E(z) - E(y)$, the previous conditions (21), (22) become:

1. G a probability distribution:

$$G(z|y) \geq 0 \ \forall y, z, \quad \int G(z|u)\mathrm{d}u = 1 \tag{25}$$

2. Metropolis-Hastings Choice:

$$A(z|y) = \min\left(1, \frac{G(y|z)}{G(z|y)} \exp(-\Delta E/T)\right) \tag{26}$$

It is worth emphasising that any such choice of $G(z|y)$ will approach an equilibrium state. Typically, as $G(z|y)$ better approximates p(z), the algorithm will converge in fewer iterations. We shall consider three possible implementations, choosing the fastest.

3.2 Energy of a Node

We first investigate the energy change caused by moving a node as in the Metropolis Hastings Algorithm. This will speed up calculations and enable some approximations of $p(z)$.

Consider the previous field configuration f_{old} and the current field configuration f. Let the k^{th} node of f be variable, and let its value be known as z, fixing all other nodes. There is one kinetic term (18), two interaction terms (19) and one potential term (20) that are explicitly dependent on z. Collecting all other terms into the constant S_k, we can write an expression for the energy as a function of z.

$$E(z) = \frac{1}{2\Delta t^2}(z - f_{k,\text{old}})^2 + \frac{1}{4\Delta x^2} z(z - f_{k-2}) + \frac{1}{4\Delta x^2} f_{k+2}(f_{k+2} - z) + \frac{\lambda}{4}(z^2 - 1)^2 + S_k \tag{27}$$

We can expand this into a quartic in terms of z:

$$E(z) = \frac{\lambda}{4} z^4 + \left(\frac{1}{2\Delta t^2} + \frac{1}{4\Delta x^2} - \frac{\lambda}{2} \right) z^2 + \left(-\frac{f_{k,\text{old}}}{\Delta t^2} - \frac{f_{k+2} + f_{k-2}}{4\Delta x^2} \right) z + \left(S_k + \frac{f_{k,\text{old}^2}}{2\Delta t^2} + \frac{f_{k+2}^2}{4\Delta x^2} + \frac{\lambda}{4} \right) \tag{28}$$

Relabelling the coefficients:

$$p = \frac{2}{\lambda}\left(\frac{1}{2\Delta t^2} + \frac{1}{4\Delta x^2} - \frac{\lambda}{2} \right), \qquad q = \frac{1}{\lambda}\left(-\frac{f_{k,\text{old}}}{\Delta t^2} - \frac{f_{k+2} + f_{k-2}}{4\Delta x^2} \right), \qquad C = \left(S_k + \frac{f_{k,\text{old}}^2}{2\Delta t^2} + \frac{f_{k+2}^2}{4\Delta x^2} + \frac{\lambda}{4} \right)$$
$$E(z) = \lambda\left(\frac{1}{4} z^4 + \frac{1}{2} p z^2 + q z \right) + C \tag{29}$$

For our choice of constants, $p = 864.0752 > 0$ is a positive constant. All dependence of the energy on other node values is then contained within q and C. In the Metropolis Hastings Algorithm, we are only interested in energy differences, and so we may neglect to evaluate the constant C. In particular, energy differences can be expressed:

$$\Delta E = E(z) - E(y) = \lambda\left(\frac{1}{4}\left(z^4 - y^4\right) + \frac{1}{2} p\left(z^2 - y^2\right) + q(z - y) \right) \tag{30}$$

It is also possible to locate energy minima. These are solutions of the equation $E'(z) = 0$, which is a depressed cubic in z:

$$E'(z) = \lambda\left(z^3 + pz + q\right) \tag{31}$$

Since there is no quadratic term and the cubic and linear coefficients are positive, we know by Cardano's Formula that there is only one real root z_0 and it is given analytically by:

$$z_0 = \sqrt[3]{-\frac{q}{2} + \sqrt{D}} + \sqrt[3]{-\frac{q}{2} - \sqrt{D}}, \qquad D = \frac{q^2}{4} + \frac{p^3}{27} \tag{32}$$

We have located the single minimum of the energy functions $E(z)$, which corresponds to the location of unique maximum of the probability distribution $p(z)$.

3.3 Proposed Implementations

Now armed with these expressions for the energy of a node, we consider three possible implementations.

1. Exact Proposal

The simplest choice of G is the probability distribution $p(z)$ itself. This simplifies the acceptance probability (26) to unity.

$$G(z|y) = p(z), \qquad A(z|y) = 1 \tag{33}$$

This is known as the standard Heat Bath regime. In cases where it is computationally inexpensive to pull random numbers from this distribution, it converges in very low time. Examples of this are systems where $p(z)$ is a Gaussian distribution, so the energy is quadratic, or systems with finite states. A famous bipartite example of the latter is the Ising model.

In cases where the probability distribution $p(z)$ is not so simple, the process of generating random numbers from $p(z)$ involves using the inverse of the cumulative distribution function F(z):

$$F(z) = \int_{-\infty}^{z} p(u)\mathrm{d}u \tag{34}$$

This involves generating a univariate random number r and then solving the equation $h_r(z) = F(z) - r$ to find the corresponding value of z. However, this task is not straightforward. The derivative of $h_r(z)$ corresponds to the probability distribution $p(z)$, which tends to be close to zero for most z values. Consequently, without constraining the root, conventional root-finding methods like Newton-Raphson may fail to converge. Fortunately, we can identify the point where the derivative of $h_r(z)$ is maximal, denoted as z_0. Using this information, we could potentially establish bounds for the root.

Unfortunately, there is a larger problem. In our case, $E(z)$ is quartic. This means evaluating these integrals is computationally expensive, especially in the large numbers required by a root-finding procedure. As such, we did not observe this method converging in a reasonable computation time.

2. Approximate Proposal

Choices of $G(z|y)$ that better approximate $p(z)$ typically converge in less iterations. After ruling out $G(z|y) = p(z)$, we now attempt to choose $G(z|y) \sim p(z)$.

Since we know z_0 is the single peak of $p(z)$, a natural choice is a Gaussian distribution centred at z_0. The proposal and acceptance probabilities can then be expressed as:

$$G(z|y) = g(z) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(z-z_0)^2}{2\sigma^2}\right), \quad A(z|y) = \exp\left(\frac{(z-z_0)^2 - (y-z_0)^2}{2\sigma^2}\right) \exp(-\Delta E/T) \tag{35}$$

There is a free parameter in this approach; the standard deviation σ. We find that a σ that is constant with respect to temperature can not cause timely convergence for both low and high temperatures. We wish to find an expression for σ in terms of T that does not require calculating the computationally expensive integrals (21), (34).

One way to do this is to equate the logarithmic derivatives at some point $u \neq z_0$, (else both would be 0):

$$\begin{aligned} \frac{p'(u)}{p(u)} &= \frac{g'(u)}{g(u)} \\ -\beta E'(u) &= \frac{-1}{\sigma^2}(u - z_0) \\ \sigma &= \sqrt{\frac{u - z_0}{\beta E'(u)}} \end{aligned} \tag{36}$$

This choice of σ will mean $g(z)$ approximates the behaviour of $p(z)$ in the region of a point u. The natural point to choose is the peak z_0, so we define σ by the limit:

$$\sigma = \lim_{u \to z_0} \sqrt{\frac{u - z_0}{E'(u)}T} = \sqrt{\frac{T}{E''(z_0)}} = \sqrt{\frac{T}{\lambda(3z_0^2 + p)}} \tag{37}$$

In the cases of small temperatures, for example $T \sim 1$, we expect $3z_0^2 << p$, which would further simplify the expression to:

$$\sigma \approx \sqrt{\frac{T}{\lambda p}} \approx 0.05\sqrt{T} \tag{38}$$

Testing for temperatures $T \sim 1$, we found this algorithm converges within a few hundred iterations, with behaviour similar to that which will be seen in Figure 3.

3. Symmetric Proposal

The third choice we consider is not an approximation for $p(z)$ but rather it is the choice for the proposal distribution to be symmetric, ie for $G(z|y) = G(y|z)$. Due to its computational simplicity, this has become the standard choice. The natural choice is a Gaussian distribution centred at y.

$$G(z|y) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(z-z_0)^2}{2\sigma^2}\right), \quad A(z|y) = \min\left(1, \exp(-\Delta E/T)\right) \tag{39}$$

Once again, σ is a free parameter. Inspired by our previous calculations, we choose $\sigma = 0.05\sqrt{T}$.

3.4 Exact Implementation

Our implementation of the Metropolis Hastings Algorithm proceeds as follows:

1. For the kth node of f, call its value y.
2. Propose a value z to assign to this node, where z is a random number pulled from the normal distribution centered at y with standard deviation $0.05\sqrt{T}$.
3. Accept this change if $r < \exp(-\Delta E/T)$, where r is a uni-variate random number.
4. Perform the above for each of the N nodes, randomly permuted. This is to avoid any bias that could arise from going through the nodes left to right each time.
5. Update the system according to (5).
6. Iterate the above until the energy of the system has stabilized.

This is a simple and computationally inexpensive method. We find that our algorithm converges for temperatures $0 < T \leq 10^4$ within 100 iterations, about a second of computation time. This is a great improvement over the Fourier Initial Conditions, which never seem to fully thermalise.

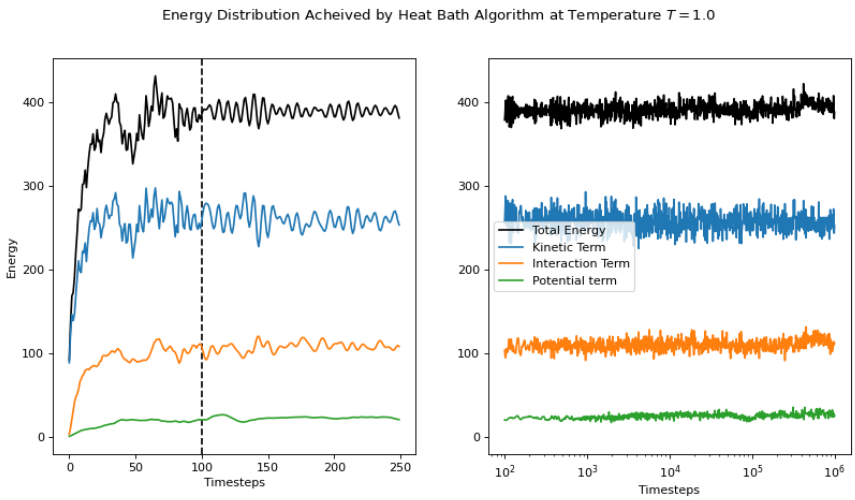


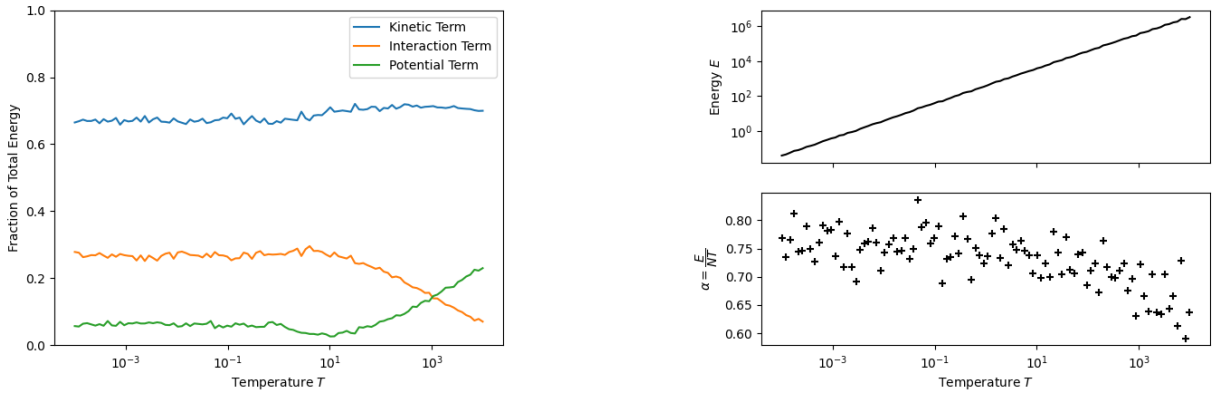
Figure 3: Time dependence of energy distribution. Energy terms are measured every timestep for 250 timesteps, then 1000 measurements are spread logarithmically until 10^6 timesteps.

We can see that there is far less instability in all terms of the distribution, thus creating less thermal noise which will considerably improve kink counting. From the right graph of Figure (3) we see that the energy of

the system is approximately conserved (within a 10% margin), in contrast to the damped energy output from the Fourier Initial Conditions code. We also notice that the ratio between terms is stable over time periods up to 10^6 time-steps. We have achieved a thermalised state at this temperature.

Another great advantage of this code is that due to the long-term stability of the total energy. Should we need to calculate the average energy over a run, a small number of evaluations would be sufficient.

We think it natural to expect the energy achieved by our algorithm to obey $E \propto NT$. We are also curious if the ratio achieved between terms is dependent on temperature. These are investigated by applying the algorithm over a wide range of temperatures.



(a) Energy Terms averaged over 150 timesteps

(b) Energy - Temperature Relationship

Figure 4: Metropolis-Hastings Energy Distribution and Conservation

Examining Figure (a), we see the ratio between terms in the distribution does change with temperature. In particular the potential term grows in strength and the interaction term decreases as the field climbs the steep potential well. For reasons that we will justify later, only temperatures $T < 10$ are of any interest. In this region, the ratio between terms seems independent of temperature.

Now examining Figure (b), we are curious if the relationship between energy achieved by the HBA is proportional to temperature as it should be. The linearity of this graph proves this. Plotting over a wide range of temperatures demonstrates just how reliable this algorithm is!

Indeed we find the energy temperature relation obeys:

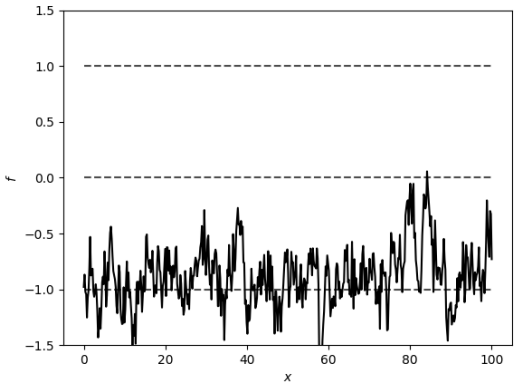
$$E = \alpha NT, \quad \alpha \sim 0.75 \tag{40}$$

Unfortunately α seems to be a random number from 0.65 to 0.8, that is approximately 0.75 for low temperatures. This means we can never exactly control the energy output from the algorithm. On average, it provides 70% to 80% of our requested energy. Two states prepared at the same temperature may often differ in energy by up to 30%. The behaviour of the field depends on the energy, and so it may be misleading to plot over temperature. For this reason, any plot from now on will be over energy. The initial conditions will be prepared by applying the Metropolis Hastings algorithm at a temperature chosen with the aim to achieve the required energy.

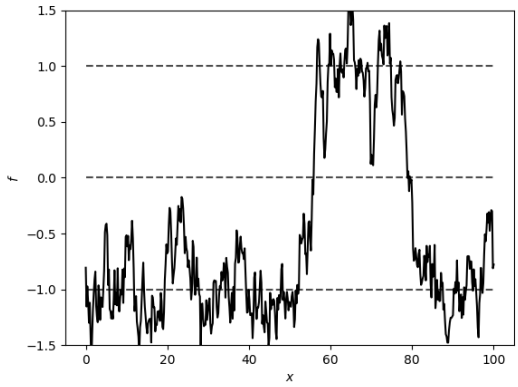
4 Kinks and Anti-Kinks

Now that we are confident in the stability of our initial conditions, we turn our attention to detecting solitons. In the case of our field configuration, a soliton that is in a positive well, and its surroundings in the negative well, we shall call it a 'kink'. In the reversed case, an 'anti-kink'.

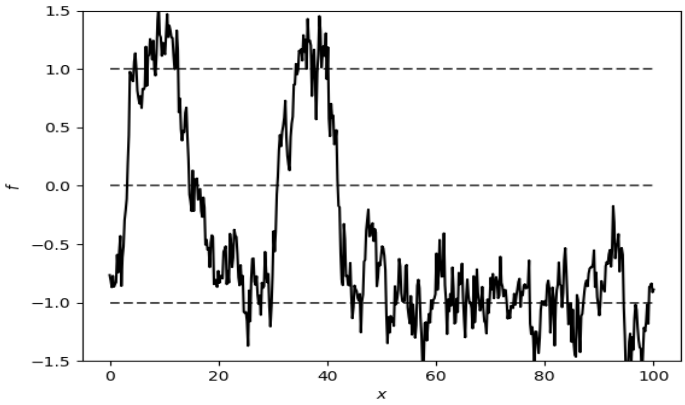
Here we provide graphs showing field configurations containing 0,1,2 kinks.



(a) $n = 0$



(b) n=1

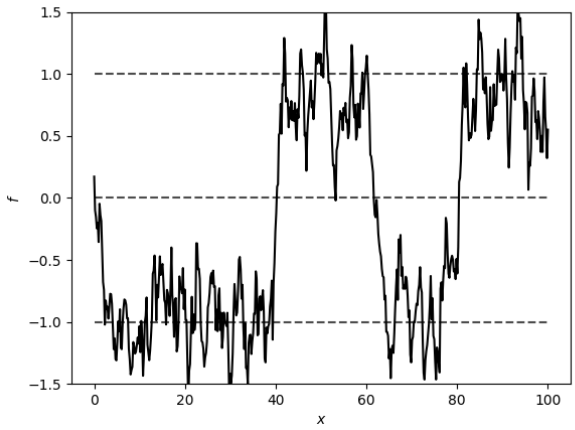


(c) $n = 2$

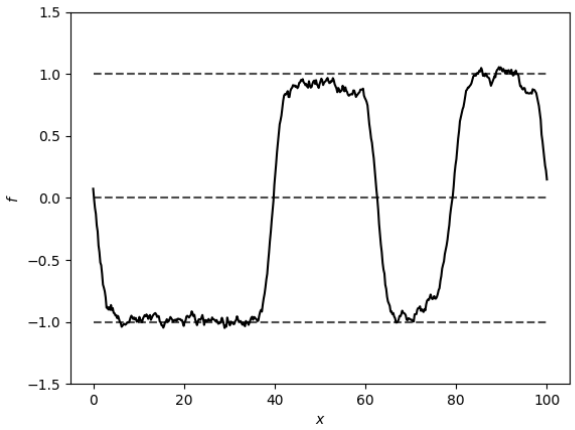
Figure 5: Field Configurations prepared at termperature $T = 1$ exhibiting different pair numbers

4.1 Frozen Solitons

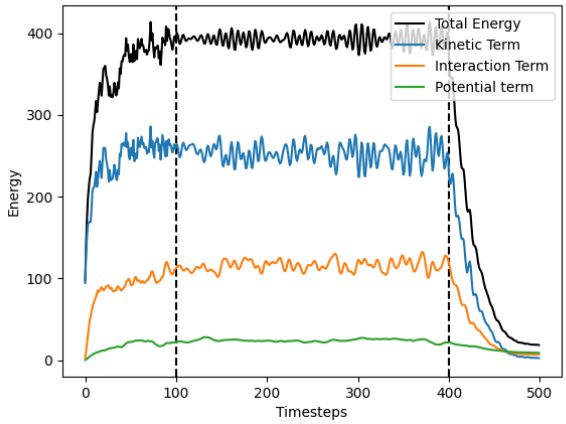
To aid in motivating a definition of a kink we are going to consider heating and cooling a system. We do this by attaching the ground state of the system to a heat bath at temperature T_1 so that the system has enough thermal energy to cross the potential barrier. We then attach the system to a different heat bath with a much lower temperature T_2 (ie.$T_2 \ll T_1$). In some cases we observe that a region of the field can become trapped in the opposing well to the environment and no longer have the required energy to cross back over the potential barrier. This will be indicated by the energy ratio of the system not being conserved in this cooling, in particular only the kinetic and interaction terms will lose energy.



(a) Field at 400 timesteps - before cooling



(b) Field at 500 timesteps - after cooling



(c) Time Dependence of Energy

Figure 6: The system is in contact to a heat bath of temperature $T_1 = 1$ from 0 to 100 timesteps, and to a heat bath of temperature $T_2 = 0.01$ from 400 to 500 timesteps.

From this figure we can see that we have indeed created a pair of "frozen" kinks; the field has been reduced to an energy where no kinks should be present, however there is indeed 2 kinks present. These frozen kinks have some properties of interest. They seem to have a well-defined structure, being wide and tall. They also seem separated; in between them is an anti-kink. We shall use these properties to motivate our own definition.

There is a physical significance to this idea. Consider a field subject to the multi-welled potential. In the early universe, the field would have been heated, possibly to such a degree that sections of the field cross into another minimum of the potential. During the rapid expansion and cooling of the universe, the field will lose thermal energy. There may have been certain solitons that were "frozen" in the field and now lack the required thermal energy to cross back over. If such a soliton were created, then it could still be traversing the universe today, provided it did not meet its end in an annihilation event with another soliton. There are hypothesised examples of such solitons, such as magnetic monopoles[1] and skyrmions[2].

In the case where the potential has minima of different heights, it is possible for the field to have cooled into a higher potential well than the ground state. Such a field would then seem to have a positive vacuum energy. It has been hypothesised that the Higgs field may be in such a false vacuum.[5]

It is of interest then, to study soliton formation in these heated fields. It is necessary to introduce some artificial definition of a kink, inspired by the properties of this 'frozen' kink, such as some width, height and separation from other kinks.

4.2 Motivating a Definition

To effectively identify kinks in a field configuration, we first need to establish a precise definition of what constitutes a kink. In [1],Grigoriev and Rubakov laid out their definition of a kink; a region between two consecutive zero crossings, of a minimum width $w_{\text{kink}} = 5$ nodes, where the field exceeds a minimum height $h_{\text{kink}} = 0.5$. We shall use the properties noticed in the frozen kink, and the criteria imposed in [1] to motivate our own definition.

First we investigate the energy dependence of the average number of zero-crossings; a node where the field differs in sign from the previous node. We investigate this by preparing initial conditions, according to our Metropolis Hastings Algorithm, over a wide range of temperatures. We then trace the evolution for 10^6 timesteps. Every 10 timesteps, we identify the number of zero-crossings in the field. We shall commonly measure quantities this often, calling it a 'frame'. The results are illustrated in Figure(7).

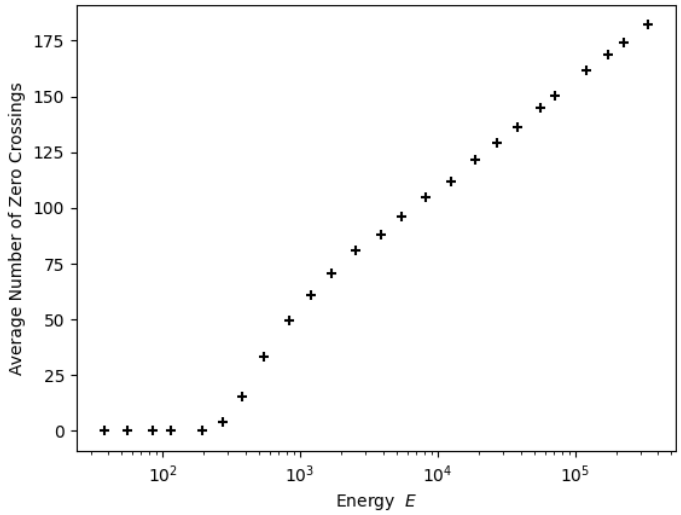


Figure 7: Energy dependence of the average number of zero-crossings. Zero at low temp then logarithmic

As one would expect we see that for low energies, here $E < 250$, we see that there are no zero crossings. This implies the number of kinks is 0 for these energies. This gives us a lower bound on what energies to use when we wish to generate kinks. We also notice that after this lower energy bound, the number of zeros rapidly increases, growing logarithmically thereafter.
For example, we see that at an energy level of 10^4, where there are on average 80 zero-crossings This means the average spacing between zero-crossings decreases to below 5. This reduction in spacing raises concerns about the longevity of detected kinks. A gap between zero-crossings that is too short may correspond to a region that is about to cross over into the opposing potential minimum, and so will likely not survive for long. This will cause the number of kinks to fluctuate often over time, as these short gaps frequently appear and disappear, making it more difficult to identify creation events later.

To resolve this, we would like to impose a minimum width onto our kink candidates. We shall use $w_{\text{kink}} = 20$ nodes. We define a "wide gap" to be a region between two zero-crossings that are separated by at least w_{kink} nodes. We investigate the energy dependence of the average number of wide gaps in the same way as we did zero-crossings.

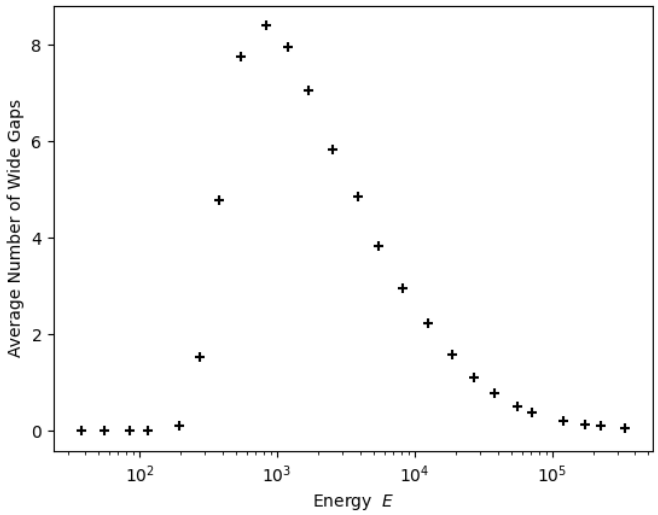


Figure 8: Average Number of Wide Gaps Γ against Energy E

We observe that, like the number of zero-crossings, the number of wide gaps is zero for low energies, $E < 250$. We also observe a peak at $E \sim 1000$, as the zero-crossings become so dense as to drop the average distance between them. Our primary area of interest will therefore be within the range $250 < E < 1000$, where the number of wide gaps is growing rapidly.

Another concerning configuration is a region between two zero-crossings where the field value is close to zero. It is under risk of being pulled across to the opposing potential minimum. Like we introduced a width requirement, we now introduce a height requirement for kink cadidates, to prevent the kink number from fluctuating too much. A kink is then a region, of width at least w_{kink}, between two zero-crossings, where the field has height at least h_{kink}. An anti-kink is a similar region, but with value at most $-h_{\text{kink}}$. We shall use $h_{\text{kink}} = 0.5$, as was done in [1]. This ensures kinks are sufficiently deep within their potential well and are safe from immediately crossing.

We also need to note that kinks and antikinks may only exist in pairs. Thus, counting kinks and counting kink-antikink pairs should be equivalent. However, in almost any algorithm we can create, there are cases where it will detect more kinks than anti-kinks and vice-versa. In theory for every kink there should be an anti-kink, and so we only consider the pair number n, the smaller of the two detected amounts.
Consider the pathological field configuration in Figure(9). The short fluctuations at $x = 20, 40, 60, 80$ split the single kink and anti-kink into 3. Our criteria up to now tell us that there are 3 kink anti-kink pairs in this configuration, though we know that in a short period of time these fluctuations will disappear, reducing the number back to one. We wish to ignore this fluctuation, and so we impose the separation condition; between any two kinks is an anti-kink, and vice-versa. With this condition, we can correctly say there is a single pair in Figure (9), preventing the kink number from fluctuating.

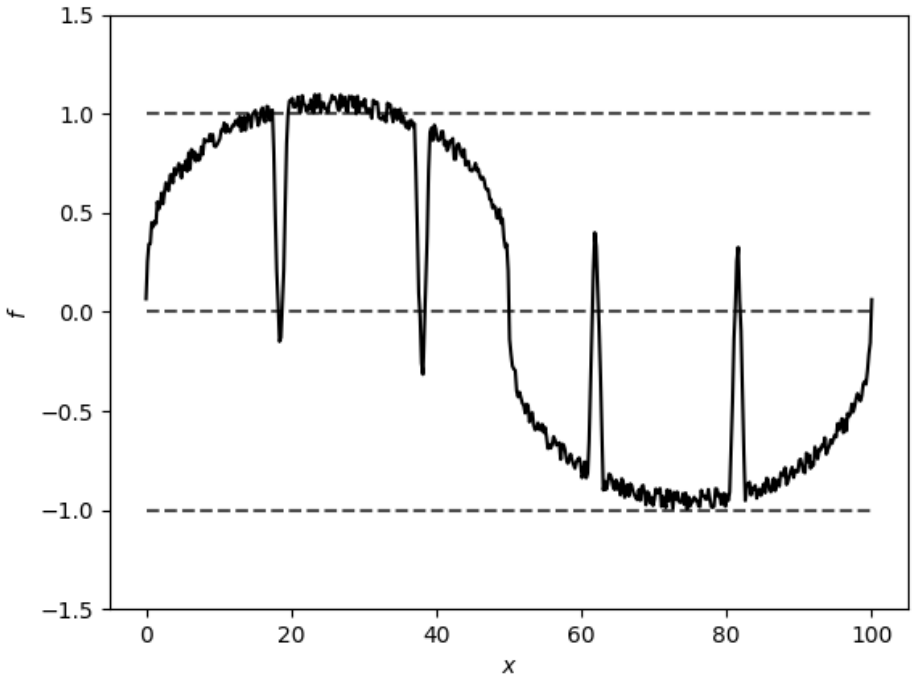


Figure 9: Pathological Field Configuration (Constructed Artificially). Motivates the inclusion of a separation condition.

4.3 Pair Number n

Now that we have a general notion of what our kinks will look like we wish to set a certain criterion to create a more definite formulation of a kink in order to avoid over/under counting.

We wish a kink to have a minimum width, height, separation and life-time. The procedue for calculating the pair number n of a field using "kink counting code" can be explain as follows:

1. Identify the zero-crossings, splitting the domain into the regions between them, called blocks.
2. The field sign in a block will be constant. Check if a frame contains a continuous region of length at least w_{kink} where the field is of absolute value at least h_{kink}. A positive region satsifying this is a 'kink'; a negative one an 'anti-kink'
3. To enforce the separation condition, if the last wave identified was a kink, only check if the current block contains an anti-kink, and vice versa. If nothing has yet been identified, check if the current block contains either.
4. Set the pair number n to be the lower of the number of kinks and anti-kinks identified.

We now prepare two fields and count the pair number once a frame using this procedure. We keep track of the cumulative sum over time. We see that for the lower temperature, this cumulative sum is often flat; this corresponds to there not being any pairs in the field. We plot both until they reach 10,000 pair detections. It takes far longer for the lower temperature to reach this target, telling us it has a lower pair number on average.

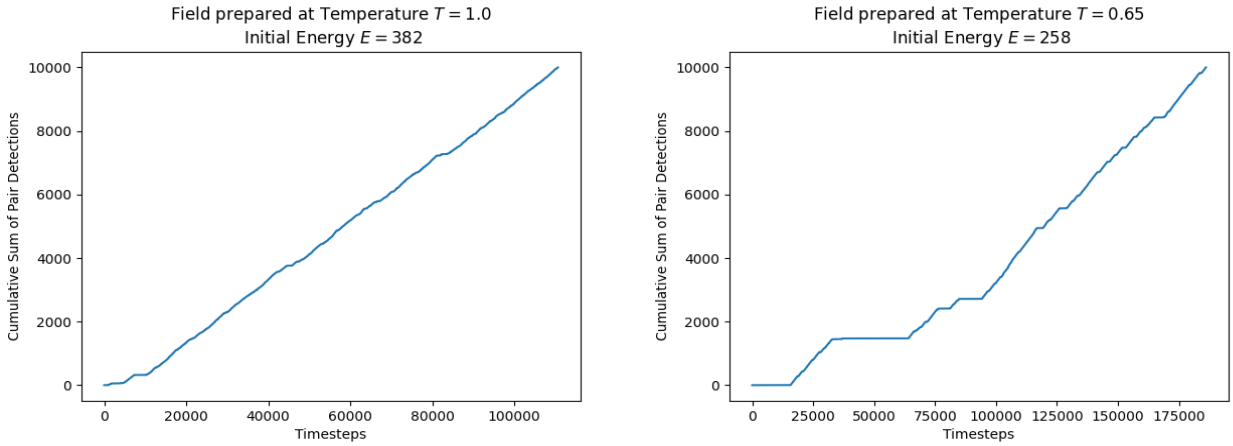


Figure 10: Cumulative Pair Detections at different temperatures

We now plot, in the same way as average zero-crossings and wide gaps, the average pair number over a wide range of temperatures.

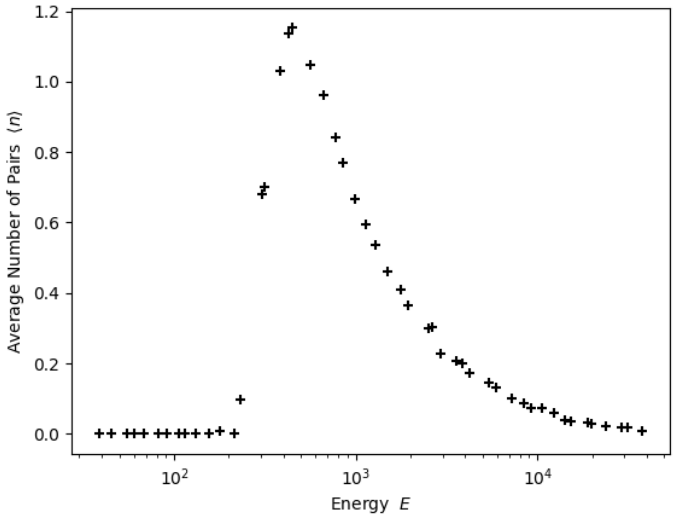


Figure 11: Energy Dependence of the Average Pair Number $\langle n \rangle$

We can see from this graph that for $E < 20$ the average number of pairs is 0, as we expected. There is a rapid increase in the average number of pairs being created in $250 < E < 500$. Then when $E > 500$ we see the average number of pairs decrease. This decrease in pair production is caused by our algorithm breaking down. This is due to a few factors, the main one being, at high energies the number of kinks being created is so high and since there is limited space, they no longer satisfy the width condition we had defined.

It is predicted[1] that the pair number should satisfy the Boltzmann Law (9). Using our value of M_s (8), and the temperature-energy relation (40) this becomes:

$$\langle n \rangle = C \exp\left(\frac{2N\alpha}{3E}\right) \tag{41}$$

where C is a proportionality factor. It is clear from Figure 11. that this will not hold for energies $E > 10^3$, but we can show that it does hold over a short range of energies, before the zero-crossings become too frequent.

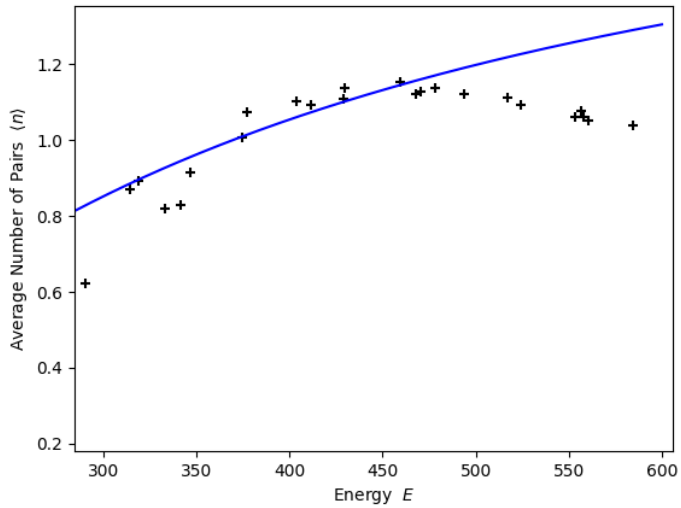


Figure 12: Energy Dependence of the Average Pair Number $\langle n \rangle$
The blue line is the prediction (41) with $C = 2$, $\alpha = 0.75$.

Again we find that for $E \leq 250$ that the average number of pairs is 0, as the system does not yet have the energy required to created pairs. For $250 \leq E \leq 450$ we find the average number of pairs does grow roughly according the the approximation. For $E > 450$, as we saw before, this is the region where our algorithm breaks down. This explains why we see the deviation from the prediction.

5 Pair Creations

Though we have tried to create a robust pair-counting algorithm, it is still possible for our algorithm to interpret a fluctuation of the field as a pair. Also, it is often possible for a kink and an anti-kink to pass through each other, and soon re-emerge without annihilating. These events cause short term fluctuations in the pair number over time, as seen in fFigure (13). There are however, increases that happen at 100,600 and $\sim$ 1200 which survive for a long time. These are called creation events. A long-term decrease is called an annihilation event. We are interested in how often these events occur. In order to calculate this, we must first create a procedure to remove the short-term fluctuations.

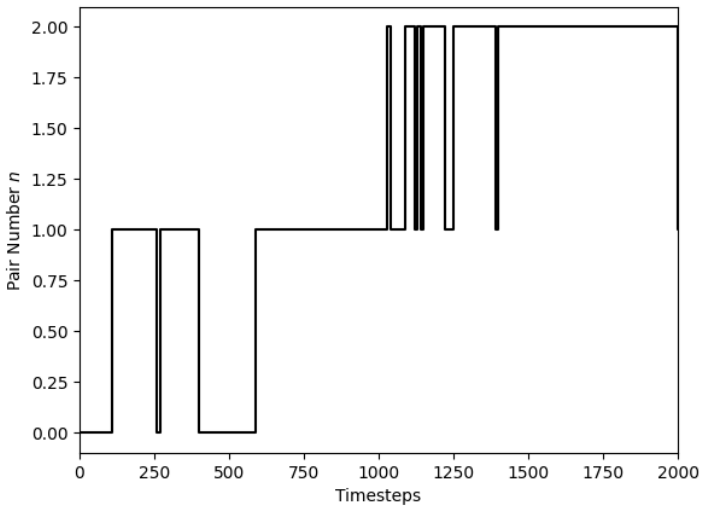


Figure 13: Time dependence of the Pair Number n. Prepared at $T = 1$.
Short fluctuations are noticed at $t \approx 250, 1100, 1200, 1400$ timesteps

5.1 Fluctuation Removal Procedure

Say we are interested in how many creations happened in $t_{\max}$ frames. To do this, we run a simulation for a little bit longer, calling the extra time a 'buffer period'. Every frame, calculate the pair number n saving the values in an array. After we have this array, remove fluctuations left to right, of all durations up to d_{kink}, (in frames).

1. For each duration $d < d_{\text{kink}}$, go through the array left to right. If an element is not equal to its predecessor, set it equal to the array element d to the right. This removes all fluctuations of duration d frames from the main section of the array; there may be some in the buffer section.
2. Repeating this for all $d < d_{\text{kink}}$, we will be left only with changes to the pair number that persist for at least d_{kink} frames; a creation or annihilation event.
3. Summing the positive changes that occur within the $t_{\max}$ frames will be the number of creations in that time period. From this we can calculate the creation rate.

To implement this procedure, we need to know the buffer period required. Notice that the first round, $d = 1$, will pull data from 1 frame past the $t_{\max}$ frames. $d = 2$ will pull a further 2 and so on up to, but not including d_{kink}. The buffer time necessary, in frames, will then be given by:

$$\sum_{d=1}^{d_{\text{kink}}-1} d = \frac{1}{2} d_{\text{kink}}(d_{\text{kink}} - 1) \tag{42}$$

Since we chose $d_{\text{kink}} = 50$ timesteps $= 5$ frames, we get the buffer period to be 20 frames. This is no extra computational effort, but we should be careful not to consider 'creations' within this buffer period.

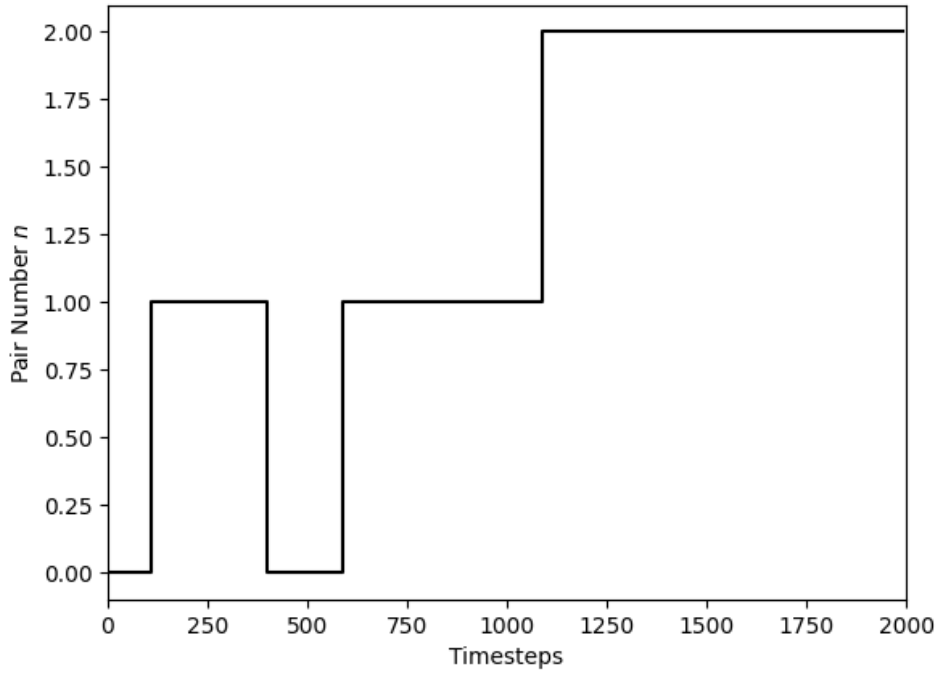


Figure 14: Smoothed version of Fig. 13. Short term fluctuations have been removed according to the Fluctuation Removal Procedure. It is now apparent 3 creation events occur; at $t = 100, 600, 1100$.

5.2 Creation Time τ_0

Now that we can calculate the number of creation rates that occurred in a time period, we can define the creation time τ_0 as the length of the time period, in time units, divided by the number of creations. A time unit is $1/\Delta t$ timesteps. τ_0 is then the expected time taken for a pair creation event to occur.

It is predicted in [1] that for a short range of energies, the creation time τ_0 obeys:

$$\tau_0 = C\frac{c^2}{LT}\exp\left(\frac{M_s}{T}\right) \tag{43}$$

For our values of $c = 2, M_s = 2/3$, (8), and the temperature-energy relationship (4), the prediction reads:

$$\tau_0 = C\frac{N\alpha}{LE}\exp\left(\frac{2N\alpha}{3E}\right) \tag{44}$$

As will be seen in Figure (15), we can fit our data well with the above formula for $C = 1400$, for a short range of energies. Grigoriev and Rubakov report $C = 26$ suits their data best. It is possible that we have miscalculated their prediction. This miscalculation may be a result of the use of dimensionless quantities in [3].

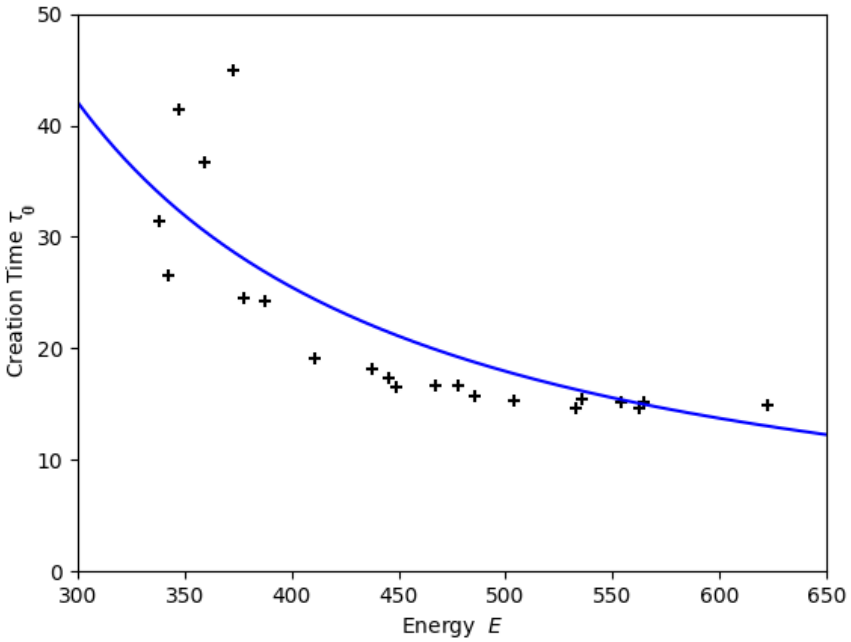


Figure 15: Energy Dependence of the Creation Time. The blue line is (40) with $C = 1400, \alpha = 0.75$

Here we observe the creation time vs energy. As we expected, low energy systems correspond to kink creation events taking longer. As the energy of the system increases the kink creation time decreases.

5.3 Creation Rate Γ

The creation rate $\Gamma = \tau_0^{-1}$ is the number of creation events in a time period divided by that time in time units.

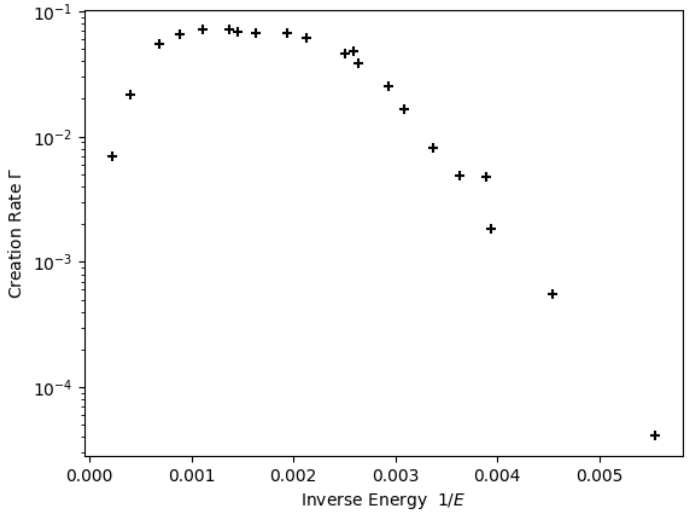


Figure 16: Kink Creation log(Γ) vs 1/E

Here we can see the graph of creation rate against inverse energy on a semi-logarithmic scale. From the graph we observe that for for high inverse energies, $0.0028 \leq 1/E \leq 0.005$, which correspond to low energies, the kink creation rate grows roughly exponentially. We then see the kink creation events begin to slow for lower inverse energies, corresponding to higher energies, $0.001 < 1/E < 0.0028$ and then decrease rapidly in the region $0 < 1/E < 0.001$. This only provides more evidence that our algorithm is only valid for a certain energy range and that it breaks down for high energy systems.

6 Conclusions

To summarise the method, we used a rolling array to implement the finite difference method to evolve the system, coupled with the Metropolis Hastings Algorithm to initialise the state of the system in order to study the dynamics of kink-antikink creation.

We found that for $E < 225$, that there are no kink-antikink pairs being created, due to the system not have the required thermal energy to cross over the potential well.
This interval of no creation events is then followed by an exponential growth in creation events in the region $225 < E < 450$ for energy. Any energies higher than the upper bound of $E \sim 450$ we find that the algorithm begins to break down as due to the high energy of the system, there are too many zero crossings, and so the gaps are not big enough resulting in them failing the width condition.

The main issue that arises at high temperatures when zero-crossings become too dense, so the gaps between them become shorter. By increasing the number of nodes N, this may be alleviated somewhat.
There remains room for further investigation, by extending the algorithm to higher dimensions, in particular (3 + 1) dimensions, this is where the most interesting solitons are present, such as the afor mentioned monopoles[1] and skyrmions[2]. The Finite Difference method,in particular our Rolling Array implementation, generalises well to higher dimesions, as does the Metropolis Hastings Algorithm.
We could also investigate the effect different values for λ, the coupling constant, and N, the number of nodes. Our code as currently written cannot quite handle a change in N. This would change the number of timesteps in a time unit, and spatial distnace between nodes. Our code uses $d_{\text{kink}} = 50$ timesteps, corresponding to about 2.5 time units, and $w_{\text{kink}} = 20$ nodes, corresponding to about 4 length units. The second values are physically important, not the first. Thus any change to N is changing our proxy definition of a kink; and so the results at different values of N cannot be compared fairly.

7 References

[1]G. 't Hooft, Nucl. Phys. B79 (1974) 276; A.M. Polyakov, Pisma ZhETF 20 (1974) 430
[2]T.H.R. Skyrme, Nucl. Phys. 31 (1962) 556; L.D. Faddeev, in Proc. Int. Conf. on Non-local field theories, Alushta, 1976
[3] Creutz, M. (1983). Quarks, Gluons and Lattices (Chapter 18).
[4] D. Yu. Grigoriev and V. A. Rubakov, "Soliton Pair Creation at Finite Temperatures. Numerical Study in (1+1) Dimensions," Nucl. Phys. B 299 (1988) 67.
[5]Kusenko, APS. (2015). Are We on the Brink of the Higgs Abyss? Physics, 8, 108. DOI:10.1103/Physics.8.108

8 Appendix 1 - Main Code

8.1 Constants, Coefficients Miscellanea

```
"""
List of Coefficients and Constants
L, N, lamb, dx, dt, frame_space
"""
import numpy as np
import matplotlib.pyplot as plt


# system variables
L = 100                         # length of 'rod' / 'string'
N = 512                         # number of nodes
lamb = 0.5                      # potential coefficient

# units
dx = L / N                      # spatial distance between nodes
dt = dx / 4                     # time-step between states
frame_space = 10                # time between measurements (timesteps)


# coefficients
C_2 = dt**2 / dx**2
C_3 = -lamb * dt**2
C_1 = 2 - 2 * C_2 - C_3
```

8.2 Finite Difference Method Implementation

8.2.1 Explicit For Loop

```
    def Update_1(f_old, f):
    f_new = - f_old + c * f**3
    for i in range(N):
        f_new[i-1] += a*f[i-1]+b*( f[i-2] + f[i] )
    return f, f_new
```

8.2.2 Matrix Multiplication

```
M=np.zeros(N*N).reshape((N,N))
for i in range(N):
    M[i,i-1] = b
    M[i,i]   = a
    M[i-1,i] = b


def Update_2(f_old, f):
    return f, - f_old + np.matmul(M, f) + c*f*f*f
```

8.2.3 Discretization

```
"""
Defines functions regarding the Discretisation of Field Equation (5)
and Energy Equation (6):

next_timestep
next_frame
energy

Defines the variables:

L, N, lamb, dx, dt, frame_space
"""
import numpy as np
import matplotlib.pyplot as plt


# system variables
L = 100                     # length of domain
N = 512                     # number of nodes
lamb = 0.5                  # potential coefficient

# units
dx = L / N                  # spatial distance between nodes
dt = dx / 4                 # time-step between states
frame_space = 10            # time between measurements (timesteps)


# coefficients
C_2 = dt**2 / dx**2
C_3 = -lamb * dt**2
C_1 = 2 - 2 * C_2 - C_3


def next_timestep(f_old, f):
    """
    Given the previous and current field configurations, 'f_old' and 'f',
    updates the field configurations to the next timestep
    according to equation (13)

    In particular uses the Rolling Array equation (16)

    Returns
    -------
    f :     current field configuration
    f_new : next field configuration
    """
    return f, (-f_old + C_1 * f +
            C_2 * (np.roll(f, 1) + np.roll(f, -1)) +
            C_3 * f ** 3)


def next_frame(f_old, f):
    """
    Given the previous and current field configurations, 'f_old' and 'f',
    updates the field configurations to the next frame
    using 'next_timestep'

    Returns
    -------
    f :     current field configuration
    f_new : next field configuration
    """
    # until next frame
    for _ in range( frame_space ):
        # update the system by one timestep
        f_old, f = next_timestep(f_old, f)

    return f_old, f


def energy(f_old, f):
    """
    Given the previous and current field configurations, 'f_old' and 'f',
    calculates the Kinetic, Interaction, Potential and Total Energy
```

```
    according to equations (17), (18), (19), (20)

    Returns
    -------
    K :  kinetic term
    I :  interaction term
    P :  potential term
    E :  total energy
    """
    K = np.sum( ( f - f_old )**2 ) / (2 * dt * dt)
    I = np.sum( f * ( f - np.roll( f, 2 ) ) ) / (4 * dx * dx)
    P = lamb * np.sum( ( f * f - 1 )**2 ) / 4
    E = K + I + P
    return K, I, P, E
```

8.3 Initial Conditions

```
"""
Defines functions to implement the Metropolis Hastings Algorithm:

linear_coefficient
energy_diff
prob_accept
heat_bath_iteration
heat_bath
"""
from Discretisation import np, N, lamb, dx, dt, next_timestep


# Quadratic coefficient (lambda * p / 2) of the Node Energy Polynomial (29)
quad = 1 / (2 * dt * dt) + 1 / (4 * dx * dx) - 0.5*lamb


def linear_coefficient(f_old, f, k):
    """
    Linear coefficient (lambda * q) of the Node Energy Polynomial (29)
    """
    return - f_old[k] / (dt * dt) - (f[k-2] + f[(k+2)%N]) / (4 * dx * dx)


def energy_diff(f_old, f, k, z, y):
    """
    Calculates the Energy Difference according to Equation (30)
    using 'linear_coefficient'

    Behaves better factorised.
    """
    lin = linear_coefficient(f_old, f, k)
    D_E = (z-y) * (  (z+y) * ( lamb * (z*z+y*y) / 4 + quad )  + lin )
    return  D_E


def prob_accept(f_old, f, T, k, z, y):
    """
    Calculates the Acceptance Probability A(z|y) according to Equation (39)
    using 'energy_diff'
    """
    Diff_E = energy_diff(f_old, f, k, z, y)
    return np.exp(- Diff_E / T)


def heat_bath_iteration(f_old, f, T, sigma):
    """
    Updates the field configuration 'f'
    according to one iteration of the Metropolis Hastings Algorithm
    using 'prob_accept'
    """
    # Attach a Heat Bath
    # For each node, randomly ordered
    for k in np.random.permutation(N):
        y = f[k]

        # Propose a new value according to G(z|y)
        z = np.random.normal(y, sigma)

        # Accept change according to A(z|y)
        r = np.random.rand()
        if r < prob_accept(f_old, f, T, k, z, y):
            f[k] = z

    return f


def heat_bath(T, iter_max=100, sigma_factor=0.05):
    """
    Prepares a thermalised state of tmperature 'T'
    by applying 'iter_max' iterations of the Metropolis Hastings Algorithm
    using 'heat_bath_iteration'
    """
    # standard deviation
    sigma = sigma_factor * np.sqrt(T)
```

```
    # prepare the ground state
    f_old = -np.ones(N)
    f = -np.ones(N)

    # For a number of iterations
    # Evolve in contact with a Heat Bath
    for iter_num in range(iter_max):
        f = heat_bath_iteration(f_old, f, T, sigma)

        # evolve by a timestep
        f_old, f = next_timestep(f_old, f)

    return f_old, f
```

8.4 Kink Counting Code

```
"""
Defines functions to identify kink anti-kink pairs and creations
as described in sections 4 and 5:

zero_crossings
zeros_and_wide_gaps
kink_in_block
anti_kink_in_block
pairs
smooth
creations
creation_rates

Defines the variables:
w_kink, h_kink, d_kink, d_kink_frame, buff_frame
"""
from Discretisation import np, N, dx, dt, frame_space
from Initial_Conditions import heat_bath

# kink variables
w_kink = 20                 # minimum width of kink (nodes)
h_kink = 0.5                # minimum height of a kink
d_kink = 50                 # minimum duration of kink (time-steps)

# can also be expressed in frames
d_kink_frame = d_kink // frame_space

# a buffer period required to use smooth properly
buff_frame = d_kink_frame * (d_kink_frame - 1)//2


def zero_crossings( f ):
    """
    Identifies the zero-crossings of 'f'
    Returns the indices of the array 'f' where the array value
    differs in sign to its predecessor

    Assumes array is non-zero
    """
    return np.where( f * np.roll(f,1) < 0)[0]


def zeros_and_wide_gaps( f ):
    """
    Returns number of zero-crossings and wide gaps
    as described in section 4.2
    using 'zero_crossings''
    """
    zeros = zero_crossings( f )

    #   if no zero-crossings, return no wide gaps
    z = len(zeros)
    if z == 0:
        return 0, 0

    #   else split into blocks
    #   search for a block that is a wide gap
    gap_count = 0


    #   define boundary section
    block = np.append( np.arange(zeros[-1], N), np.arange(0, zeros[0]))

    # if passes width requirement, increase count
    if len(block) >= w_kink:
        gap_count += 1

    #   every other section
    for i in range( 1, z ):

        #   define section
        block = np.arange(zeros[i-1], zeros[i])

        # if passes width requirement, increase count
```

```
        if len(block) >= w_kink:
            gap_count += 1

    return z, gap_count


def kink_in_block( block, f ):
    """
    Determines if there is a kink present in the block of 'f';
    if it passes the width and height conditions.

    Block: indices of 'f' where the sign of 'f' is constant
    f : the current field configuration
    """
    high_count = 0

    # for each node
    for i in block :

        # if passes height requirement, then increase counter
        if f[i] > h_kink:
            high_count += 1

            # if passes width requirement, then there is a kink
            if high_count >= w_kink:
                return True

        # if fails height requirement, then reset counter
        else:

            # reset counter
            high_count = 0

    # if at end no kink was detected, then there is no kink
    return False


def anti_kink_in_block( block, f ):
    """
    Determines if there is an anti-kink present in the block of 'f';
    if it passes the width and height conditions.

    Block: indices of 'f' where the sign of 'f' is constant
    f : the current field configuration
    """
    deep_count = 0

    # for each node
    for i in block :

        # if passes height requirement, incerase counter
        if f[i] < - h_kink:
            deep_count += 1

            # if passes width requirement, then there is an anti - kink
            if deep_count >= w_kink:
                return True

        # if fails height requirement, reset counter
        else:
            deep_count = 0

    # if at end no anti - kink was detected, then there is no anti - kink
    return False


def pairs( f ):
    """
    Calculates the pair number 'n'
    Using the procedure described in 4.3
    using 'zero_crossings', 'kink_in_block' and 'anti_kink_in_block'

    """
    # identify all zero-crossings
    zeros = zero_crossings( f )
```

```
    #   if no zero-crossings, return no pairs
    z = len(zeros)
    if z == 0:
        return 0

    #   else split into blocks
    #   search for a block containing a kink

    # set counters to zero
    kink_count = 0
    anti_kink_count = 0

    # we are not yet looking for anything in particular
    kink_search = False
    anti_kink_search = False

    for i in range( z ):

        # define block

        if i == 0:
            # due to periodic boundary conitions,
            # boundary block is defined differently
            block = np.append( np.arange(zeros[-1], N), \
                               np.arange(0, zeros[0]))
        else:
            block = np.arange(zeros[i-1], zeros[i])


        if kink_search == True:
            #   search for kink in block

            if kink_in_block( block, f ) == True:
                #   if there is, count it, and now search for anti-kink
                kink_count += 1
                kink_search = False
                anti_kink_search = True

        if anti_kink_search == True:
            #   search for anti-kink in block

            if anti_kink_in_block( block, f ) == True:
                #   if there is, count it, and now search for kink
                anti_kink_count += 1
                kink_search = True
                anti_kink_search = False

        else:
            # search for both
            # only happens until first detection

            if kink_in_block( block, f ) == True:
                #   if there is, count it, and now search for anti-kink
                kink_count += 1
                anti_kink_search = True

            if anti_kink_in_block( block, f ) == True:
                #   if there is, count it, and now search for kink
                anti_kink_count += 1
                kink_search = True

    #   return the minimum; the number of pairs
    return min( kink_count, anti_kink_count )

#   Kink-count Smoothing
def smooth(array, tmax_frame):
    """
    Removes fluctuations of duration less than 'd_min' from an array
    according to the procedure described in section 5.1
    """
    #   for each duration 'd' up to the minimum acceptable
    for d in range(1, d_kink_frame ):
        #   remove, left to right, all fluctuations of duration 'n'

        #   for each node before the respective buffer
        for k in range(1, len(array) - d * (d + 1)//2 ):
```

```
            #   if a value doesn't match its predecessor
            if array[k] != array[k - 1]:
                #   set it to the value 'n' nodes in the future
                array[k] = array[k + d]

    return array[: tmax_frame ]


def creations(array, tmax_frame):
    """
    Calculates the number of creations that occurred in this time-frame
    given the array of pair numbers over (tmax_frame + buff_frame) frames
    using the procedure described in section 5.1
    using 'smooth'
    """
    s_array = smooth(array, tmax_frame)
    #   count the upward steps
    diff = [max(y - x, 0) for x, y in zip(s_array[:-1], s_array[1:])]
    amount = sum( diff )
    return amount


def creation_rates( pair_array, tmax_frame ):
    """
    Calculates the creation time 'tau' and creation rate 'Gamma'
    given the array of pair numbers over (tmax_frame + buff_frame) frames
    using the procedure described in section 5.1
    using 'creations'
    """
    # we measure these quantities in proper time
    # 1 frame is 'frame_space' timesteps
    # 1 timestep is 'dt' time units
    # therefore one frame is 'frame_space' * dt time units
    tmax_units = frame_space * dt * tmax_frame
    creation_amount = creations(pair_array, tmax_frame)

    tau =  tmax_units / creation_amount
    Gamma = creation_amount / tmax_units
    return Gamma, tau
```

8.5 Test Functions for Plots

```
"""
Defines functions used in the majority of plots:

heat_bath_T_test
zeros_and_wide_gaps_test
pairs_test
Gamma_and_tau_test
"""
from Discretisation import np, plt, N, next_timestep, next_frame, energy
from Initial_Conditions import heat_bath
from Kinks_and_Creations import zeros_and_wide_gaps, pairs, \
                            creation_rates, buff_frame

def heat_bath_T_test(T, iter_max, sigma_factor):
    """
    Prepares initial condition of temperature 'T'
    according to the Metropolis-Hastings Algorithm
    using 'heat_bath'
    Tracks Energy Distribution over the next 'iter_max' timesteps,
    returns averages
    using 'energy'

    Returns
    -------
    Kf_avg :  average kinetic fraction
    If_avg :  average interaction fraction
    Pf_avg :  average potential fraction
    E_avg :   average total energy
    alpha :   energy-temperature proportionality constant
    """
    # initialise state heat bath algorithm
    f_old, f = heat_bath(T, sigma_factor=sigma_factor)

    # Initialize arrays
    Kf_array = np.zeros( iter_max )
    If_array = np.zeros( iter_max )
    Pf_array = np.zeros( iter_max )
    E_array = np.zeros( iter_max )

    # For a number of iterations
    # Evolve independent of Heat Bath
    # Measuring Energy
    for iter_num in range(iter_max):

        # Once per iteration, measure energies and store
        K, I, P, E = energy(f_old, f)
        Kf_array[iter_num] = K/E
        If_array[iter_num] = I/E
        Pf_array[iter_num] = P/E
        E_array[iter_num] = E

        # Evolve by one time-step
        f_old, f = next_timestep(f_old, f)


    #   calculate means since disconnection from heat bath
    Kf_avg = np.mean((Kf_array))
    If_avg = np.mean((If_array))
    Pf_avg = np.mean((Pf_array))
    E_avg = np.mean(E_array)

    #   calculate the constant of proportionality
    alpha = E_avg / (N*T)

    return Kf_avg, If_avg, Pf_avg, E_avg, alpha


def zeros_and_wide_gaps_test( T, e_tests, tmax_frame):
    """
    Average zeros and wide gaps at temperature

    Prepares initial condition of temperature 'T'
    according to the Metropolis-Hastings Algorithm
    using 'heat_bath'
```

```
    Evololved for 'tmax_frame' frames,
    using 'next_frame'
    Measures numbers of zero-crossings and wide gaps every frame,
    using 'zeros_and_wide_gaps'
    Measures energy 'e_tests' times throughout,
    using 'energy'


    Returns
    -------
    E_avg :   average total energy
    z_avg :   average number of zero-crossings
    g_avg :   average number of wide gaps
    """

    # intial conditions of this temperature
    f_old, f = heat_bath(T)

    # reset counters
    z = 0
    g = 0
    E = 0

    # for each frame until tmax
    for j in range( tmax_frame ):

        # evolve to next frame
        f_old, f = next_frame(f_old, f)

        # on frame, count zeros and wide gaps
        z_new, g_new = zeros_and_wide_gaps( f )
        z += z_new
        g += g_new

        # rarely evaluate energy
        if j % (tmax_frame // e_tests) == 0:
            E += energy(f_old, f)[-1]

    # calculate average energy over all measurements
    E_avg = E / e_tests

    # calculate average average zeros and wide gaps over all frames
    z_avg = z / tmax_frame
    g_avg = g / tmax_frame

    return E_avg, z_avg, g_avg


def pairs_test( T, e_tests, tmax_frame):
    """
    Average pair number 'n' at temperature

    Prepares initial condition of temperature 'T'
    according to the Metropolis-Hastings Algorithm
    using 'heat_bath'
    Evololved for 'tmax_frame' frames,
    using 'next_frame'
    Measures numbers of pairs every frame,
    using 'pairs'
    Measures total energy 'e_tests' times throughout,
    using 'energy'

    Returns
    -------
    E_avg :   average total energy
    n_avg :   average number of pairs
    """
    # intial conditions of this temperature
    f_old, f = heat_bath(T)

    # reset counters
    n = 0
    E = 0

    # for each frame until tmax
    for j in range( tmax_frame ):
```

```
        # evolve to next frame
        f_old, f = next_frame(f_old, f)

        # on frame, count pairs
        n += pairs( f )


        # rarely evaluate energy
        if j % (tmax_frame // e_tests) == 0:
            E += energy(f_old, f)[-1]

    # calculate average energy over all measurements
    E_avg = E / e_tests

    # calculate mean pair number over all frames
    n_avg = n / tmax_frame

    return E_avg, n_avg

def Gamma_and_tau_test( T, e_tests = 1000, tmax_frame=10**5 ):
    """
    Creation rate, creation time over 'tmax_frame' at temperature 'T'

    Prepares initial condition of temperature 'T'
    according to the Metropolis-Hastings Algorithm
    using 'heat_bath'
    Evololved for 'tmax_frame' frames,
    using 'next_frame'
    Measures numbers of pairs every frame,
    using 'pairs'
    Calculates creation rate 'gamma' and creation time 'tau',
    using 'creation_rates'
    Measures total energy 'e_tests' times throughout,
    using 'energy'

    Returns
    -------
    E_avg :   average total energy
    Gamma :   creation rate
    tau   :   creation time
    """
    # initial conditions of this temperature
    f_old, f = heat_bath(T)

    # reset counters
    k_array = np.zeros(tmax_frame + buff_frame)
    E = 0

    # for each frame until tmax_frame
    for j in range( tmax_frame + buff_frame ):

        # evolve to next frame
        f_old, f = next_frame(f_old, f)

        # on frame, count pairs
        k_array[j] = pairs(f)

        # rarely evaluate energy
        if j % (tmax_frame // e_tests) == 0:
            E += energy(f_old, f)[-1]

    # calculate average energy over all measurements
    E_avg = E / e_tests

    # calculate the pair creation time
    Gamma, tau = creation_rates(k_array, tmax_frame)

    return E_avg, Gamma, tau
```

9 Appendix 2 - Plotting

9.1 Figure 2 - Fourier Plots

```
"""
Produces Figure 2
"""
from Discretisation import np, plt, L, N, dt, next_timestep, energy


def initial_fourier(scale):
    """
    Mimics the thermal spectrum of a thermalised state
    Uses a Fourier transform to create an approximately thermalised state
    Will serve as experimental initial conditions
    """
    # random phases
    phases_1 = 2*np.pi*np.random.rand(N)
    phases_2 = 2*np.pi*np.random.rand(N)

    # a range of amplitudes
    amplitudes_1 = 1 / np.concatenate(
                                    [100*np.ones(5),
                                    np.arange(6, (N//2)+1),
                                    np.arange((N//2), 0, -1)]
                                    )
    amplitudes_2 = 2 * np.pi / L

    # fourier transforms of these
    z1 = np.fft.fft( amplitudes_1 * np.exp( 1j * phases_1 ) )
    z2 = np.fft.fft( amplitudes_2 * np.exp( 1j * phases_2 ) )

    # build initial conditions from these
    f_old = -1 + scale * np.real(z1)
    f =  f_old + scale * np.real(z2) * dt

    return f_old, f


scale = 0.8             # fourier 'scale' parameter
iter_max = 250          # short term evolution
tmax_dt = 10**6         # long term evolution
num_tests = 1000        # number of measurements to be made in long term


# Fourier Initial Conditions
f_old, f = initial_fourier( scale )

        # plot the initial conditions
E = int( energy(f_old, f)[-1] )
plt.figure()
plt.xlabel(r'$x$')
plt.ylabel(r'$f$')
axis = np.linspace(0,L,N)
plt.title('Fourier Initial Conditions with \'scale\' = ' \
              +str(scale)
              +'\n Current Energy '+r'$E=$'+str(E))
plt.plot(axis, np.ones(N), color='black', \
         linestyle = 'dashed', alpha = 0.7)
plt.plot(axis, np.zeros(N), color='black', \
         linestyle = 'dashed', alpha = 0.7)
plt.plot(axis, -np.ones(N), color='black', \
         linestyle = 'dashed', alpha = 0.7)
plt.plot(axis, f, color='black')


        # short term energy conservation

# initialise arrays
K_array = np.zeros( iter_max )
I_array = np.zeros( iter_max )
P_array = np.zeros( iter_max )
E_array = np.zeros( iter_max )

# For a number of iterations
for iter_num in range( iter_max ):
```

```
    # Once per iteration, measure energies and store
    K, I, P, E = energy(f_old, f)
    K_array[iter_num] = K
    I_array[iter_num] = I
    P_array[iter_num] = P
    E_array[iter_num] = E

    # Evolve by one time-step
    f_old, f = next_timestep(f_old, f)


        #   plot results
fig = plt.figure(figsize=(12, 6), dpi=80)

ax1 = fig.add_subplot(1, 2, 1)
fig.suptitle('Fourier Initial Conditions with \'scale\' = ' \
             +str(scale) )
ax1.set_xlabel('Timesteps')
ax1.set_ylabel('Energy')
ax1.plot(E_array, label = 'Total Energy', color = 'black')
ax1.plot(K_array, label = 'Kinetic Term')
ax1.plot(I_array, label = 'Interaction Term')
ax1.plot(P_array, label = 'Potential term')


        # long term energy conservation ( logarithmic )
time_array_dt = 10**np.linspace(
                    np.log10(iter_max), np.log10(tmax_dt), num_tests)

# initialise arrays
K_array = np.zeros( num_tests )
I_array = np.zeros( num_tests )
P_array = np.zeros( num_tests )
E_array = np.zeros( num_tests )
counter = iter_max

# For each specified time
for i, time_dt in enumerate( time_array_dt ):

    # progress bar
    # inside loop since each step takes longer than last
    print( str(counter) +' out of ' + str(tmax_dt),
          str(100*counter/tmax_dt)+'%')

    # evolve to next time
    while counter < time_dt:
        f_old, f = next_timestep( f_old, f )
        counter += 1

    # At time, measure energies and store
    K, I, P, E = energy(f_old, f)
    K_array[i] = K
    I_array[i] = I
    P_array[i] = P
    E_array[i] = E


        #   plot results
ax2 = fig.add_subplot(1, 2, 2, sharey = ax1)
ax2.set_xscale('log')
ax2.set_xlabel('Timesteps')
ax2.plot(time_array_dt, E_array, \
         label = 'Total Energy', color = 'black')
ax2.plot(time_array_dt, K_array, label = 'Kinetic Term')
ax2.plot(time_array_dt, I_array, label = 'Interaction Term')
ax2.plot(time_array_dt, P_array, label = 'Potential term')
ax2.legend(loc='center left')
```

9.2 Figure 3 - Heat Bath Algorithm over time

```
"""
Produces Figure 3

Attachs ground state to heat bath
Evolves for short period of time
Returns plot of Energy Distribution over Time

Evolves for long time period
Returns plot of Energy Distribution over log of time
"""
from Discretisation import np, plt, N, next_timestep, energy
from Initial_Conditions import heat_bath_iteration


T = 1.0                 # temperature of heat bath
sigma_factor = 0.05     # standard deviation factor
iter_max = 100          # iterations attached to heat bath
iter_max2 = 150         # timesteps of independent evolution
tmax_dt = 10**6         # timesteps to trace evolution over
num_tests = 1000        # number of energy evaluations in this time


# standard deviation
sigma = sigma_factor * np.sqrt(T)

# prepare the ground state
f_old = -np.ones(N)
f = -np.ones(N)

# initialise arrays
K_array = np.zeros( iter_max + iter_max2 )
I_array = np.zeros( iter_max + iter_max2 )
P_array = np.zeros( iter_max + iter_max2 )
E_array = np.zeros( iter_max + iter_max2 )

# For a number of iterations
# Evolve in contact with a Heat Bath
for iter_num in range(iter_max):
    f = heat_bath_iteration(f_old, f, T, sigma)

    # Once per iteration, measure energies, print and store
    K, I, P, E = energy(f_old, f)
    K_array[iter_num] = K
    I_array[iter_num] = I
    P_array[iter_num] = P
    E_array[iter_num] = E

    # evolve by a timestep
    f_old, f = next_timestep(f_old, f)

#   For a number of iterations
#   Evolve independent of Heat Bath
for iter_num in range( iter_max, iter_max + iter_max2 ):

    # Once per iteration, measure energies and store
    K, I, P, E = energy(f_old, f)
    K_array[iter_num] = K
    I_array[iter_num] = I
    P_array[iter_num] = P
    E_array[iter_num] = E

    # Evolve by one time-step
    f_old, f = next_timestep(f_old, f)


        # plot results
fig = plt.figure(figsize=(12, 6), dpi=80)
fig.suptitle(
    'Energy Distribution Acheived by Heat Bath Algorithm at Temperature '
              +r'$T=$'+str(T) )

ax1 = fig.add_subplot(1, 2, 1)
ax1.set_xlabel('Timesteps')
ax1.set_ylabel('Energy')
```

```
ax1.axvline( iter_max, linestyle='dashed', color = 'black')
ax1.plot(E_array, color = 'black')
ax1.plot(K_array)
ax1.plot(I_array)
ax1.plot(P_array)


        # long term energy conservation ( logarithmic )
time_array_dt = 10**np.linspace(
                    np.log10(iter_max), np.log10(tmax_dt), num_tests)

# initialise arrays
K_array = np.zeros( num_tests )
I_array = np.zeros( num_tests )
P_array = np.zeros( num_tests )
E_array = np.zeros( num_tests )
counter = iter_max

# For each specified time
for i, time_dt in enumerate( time_array_dt ):

    # evolve to next time
    while counter < time_dt:

        # progress bar
        # inside loop since each step takes longer than last
        print( str(counter) +' out of ' + str(tmax_dt),
              str(100*counter/tmax_dt)+'%')

        f_old, f = next_timestep( f_old, f )
        counter += 1

    # At time, measure energies and store
    K, I, P, E = energy(f_old, f)
    K_array[i] = K
    I_array[i] = I
    P_array[i] = P
    E_array[i] = E


        #   plot results
ax2 = fig.add_subplot(1, 2, 2, sharey = ax1)
ax2.set_xscale('log')
ax2.set_xlabel('Timesteps')
ax2.plot(time_array_dt, E_array, \
         label = 'Total Energy', color = 'black')
ax2.plot(time_array_dt, K_array, label = 'Kinetic Term')
ax2.plot(time_array_dt, I_array, label = 'Interaction Term')
ax2.plot(time_array_dt, P_array, label = 'Potential term')
ax2.legend(loc='center left')
```

9.3 Figure 4 - Heat Bath over Temperature

```
"""
Produces Figure 4
"""
from Discretisation import np, plt
from Test_Functions import heat_bath_T_test


num_tests = 100          # number of simulations to run
T_min = 10**-4           # minimum temperature
T_max = 10**4            # maximum temperature
iter_max = 100           # evolution time
sigma_factor = 0.05      # standard deviation factor


# initialise arrays
T_array = 10**np.linspace( np.log10(T_min), np.log10(T_max), num_tests )
Kf_avg_array = np.zeros( num_tests )
If_avg_array = np.zeros( num_tests )
Pf_avg_array = np.zeros( num_tests )
E_avg_array = np.zeros( num_tests )
alpha_array = np.zeros( num_tests )

# for each temperature
# calculate energy distribution acheived
for i,T in enumerate(T_array):

    # progress bar
    print( str(i+1) + ' out of ' + str(num_tests) )

    Kf_avg, If_avg, Pf_avg, E_avg, alpha = \
        heat_bath_T_test(T, iter_max, sigma_factor)
    Kf_avg_array[i] = Kf_avg
    If_avg_array[i] = If_avg
    Pf_avg_array[i] = Pf_avg
    E_avg_array[i] = E_avg
    alpha_array[i] = alpha


        #   plot results
fig, (ax1, ax2) = plt.subplots(2, sharex=True)

ax1.set_ylabel('Energy '+r'$E$')
ax1.set_xscale('log')
ax1.set_yscale('log')
ax1.plot(T_array, E_avg_array, color='black')

ax2.set_xlabel('Temperature ' +r'$T$')
ax2.set_ylabel(r'$\alpha=\dfrac{E}{NT}$')
ax2.set_xscale('log')
ax2.scatter(T_array, alpha_array, color='black', marker ='+')

fig3, ax3 = plt.subplots()
ax3.set_xlabel('Temperature ' +r'$T$')
ax3.set_ylabel('Fraction of Total Energy')
ax3.set_ylim(0.0, 1.0)
ax3.set_xscale('log')
ax3.plot(T_array, Kf_avg_array, label = 'Kinetic Term')
ax3.plot(T_array, If_avg_array, label = 'Interaction Term')
ax3.plot(T_array, Pf_avg_array, label = 'Potential Term')
ax3.legend()
```

9.4 Figures 5 and 1 Frame Printer

```
    """
Produces Figures 1 and 5

Prepares a field configuration at temperature 'T'
Evolves for 'tmax' timesteps
Plots 'num_frames' field configurations over this evolution
"""

from Discretisation import np, plt, L, N, next_timestep, energy
from Initial_Conditions import heat_bath


T = 1.0              # temperature
tmax = 10**4         # max timesteps
num_frames = 10      # number of plots


space = tmax // num_frames  # timesteps between plots
axis = np.linspace(0,L,N)   # to plot field over
f_old, f = heat_bath(T)     # intitial conditions

for i in range(num_frames):

    # progress bar
    print( str(i+1) + ' out of ' + str(num_frames))

    # evolve over time
    for _ in range(space):
        f_old, f = next_timestep(f_old, f)
    # measure energy
    E = int( energy(f_old, f)[-1] )

    # plot
    plt.figure()
    plt.xlabel(r'$x$')
    plt.ylabel(r'$f$')
    plt.ylim(-1.5, 1.5)
    plt.plot(axis, np.ones(N), color='black', \
         linestyle = 'dashed', alpha = 0.7)
    plt.plot(axis, np.zeros(N), color='black', \
                      linestyle = 'dashed', alpha = 0.7)
    plt.plot(axis, -np.ones(N), color='black', \
         linestyle = 'dashed', alpha = 0.7)
    plt.plot(axis, f, color='black')
```

9.5 Figure 6 - Frozen Kink Plot

```
"""
Produces Figure 6

Attachs ground state to hot heat bath
Evolves for time
Attachs to cold heat bath

Returns plot of energy of time
Returns plot of 'frozen' kink
"""
from Discretisation import np, plt, L, N, energy, next_timestep
from Initial_Conditions import heat_bath_iteration


T1 = 1.0                # temperature of hot heat bath
iter_max = 100          # iterations attached to hot heat bath
iter_max2 = 300         # timesteps of evolution between heat baths
T2 = 0.01               # temperature of cold heat bath
iter_max3 = 100         # iterations attached to cold heat bath
sigma_factor = 0.05     # standard deviation factor


# standard deviation
sigma1 = sigma_factor * np.sqrt(T1)
sigma2 = sigma_factor * np.sqrt(T2)
```

```
# prepare the ground state
f_old = -np.ones(N)
f = -np.ones(N)

# initialise arrays
K_array = np.zeros( iter_max + iter_max2 + iter_max3 )
I_array = np.zeros( iter_max + iter_max2 + iter_max3 )
P_array = np.zeros( iter_max + iter_max2 + iter_max3 )
E_array = np.zeros( iter_max + iter_max2 + iter_max3 )

# For a number of iterations
# Evolve in contact with a hot Heat Bath
for iter_num in range(iter_max):
    f = heat_bath_iteration(f_old, f, T1, sigma1)

    # Once per iteration, measure energies, print and store
    K, I, P, E = energy(f_old, f)
    K_array[iter_num] = K
    I_array[iter_num] = I
    P_array[iter_num] = P
    E_array[iter_num] = E

    # evolve by a timestep
    f_old, f = next_timestep(f_old, f)

#   For a number of iterations
#   Evolve independent of Heat Bath
for iter_num in range( iter_max, iter_max + iter_max2 ):

    # Once per iteration, measure energies and store
    K, I, P, E = energy(f_old, f)
    K_array[iter_num] = K
    I_array[iter_num] = I
    P_array[iter_num] = P
    E_array[iter_num] = E

    # Evolve by one time-step
    f_old, f = next_timestep(f_old, f)


fig1, ax1 = plt.subplots()
E = int( energy(f_old, f)[-1] )
axis = np.linspace(0,L,N)
ax1.set_xlabel(r'$x$')
ax1.set_ylabel(r'$f$')
ax1.set_ylim(-1.5, 1.5)
ax1.plot(axis, np.ones(N), color='black', \
     linestyle = 'dashed', alpha = 0.7)
ax1.plot(axis, np.zeros(N), color='black', \
                 linestyle = 'dashed', alpha = 0.7)
ax1.plot(axis, -np.ones(N), color='black', \
     linestyle = 'dashed', alpha = 0.7)
ax1.plot(axis, f, color='black')

# For a number of iterations
# Evolve in contact with a cold Heat Bath
for iter_num in range( iter_max + iter_max2, \
                       iter_max + iter_max2 + iter_max3 ):
    f = heat_bath_iteration(f_old, f, T2, sigma2)

    # Once per iteration, measure energies, print and store
    K, I, P, E = energy(f_old, f)
    K_array[iter_num] = K
    I_array[iter_num] = I
    P_array[iter_num] = P
    E_array[iter_num] = E

    # evolve by a timestep
    f_old, f = next_timestep(f_old, f)

        # plot results
fig2, ax2 = plt.subplots()
ax2.set_xlabel('Timesteps')
ax2.set_ylabel('Energy')
ax2.axvline( iter_max, linestyle='dashed', color = 'black')
```

```
ax2.axvline( iter_max + iter_max2, linestyle='dashed', color = 'black')
ax2.plot(E_array, label = 'Total Energy', color = 'black')
ax2.plot(K_array, label = 'Kinetic Term')
ax2.plot(I_array, label = 'Interaction Term')
ax2.plot(P_array, label = 'Potential term')
ax2.legend()


        # plot field
fig3, ax3 = plt.subplots()
E = int( energy(f_old, f)[-1] )
axis = np.linspace(0,L,N)
ax3.set_xlabel(r'$x$')
ax3.set_ylabel(r'$f$')
ax3.set_ylim(-1.5, 1.5)
ax3.plot(axis, np.ones(N), color='black', \
     linestyle = 'dashed', alpha = 0.7)
ax3.plot(axis, np.zeros(N), color='black', \
                 linestyle = 'dashed', alpha = 0.7)
ax3.plot(axis, -np.ones(N), color='black', \
     linestyle = 'dashed', alpha = 0.7)
ax3.plot(axis, f, color='black')
```

9.6 Figures 7 and 8 - Zeros and Wide Gaps

```
 """
Produces Figures 7 and 8

Energy Dependence of Average Numbers of Zero-Crossings and Wide Gaps
"""
from Discretisation import np, plt, N, frame_space
from Initial_Conditions import heat_bath
from Test_Functions import zeros_and_wide_gaps_test


num_tests = 25          # number of tests
T_min = 10**-1          # minimum temperature
T_max = 10**3           # maximum Temperature
e_tests = 1000          # nuber of energy evaluations
tmax_frame = 10**5      # number of frames for evolution to be traced over


# initialise arrays
T_array = 10**np.linspace( np.log10(T_min), np.log10(T_max), num_tests )
E_array = np.zeros( num_tests )
zeros_array = np.zeros( num_tests )
gaps_array = np.zeros( num_tests )


# for each temperature
for i, T in enumerate(T_array):

    # progress bar
    print(str(i+1) + ' out of '+str(num_tests))

    # intial conditions of this temperature
    f_old, f = heat_bath(T)

    # calculate average energy, number of zeros and gaps
    E, z, g = zeros_and_wide_gaps_test(T, e_tests, tmax_frame)
    E_array[i] = E
    zeros_array[i] = z
    gaps_array[i] = g

    print('Energy: ' + str(int(E)) +' zeros: '+str(z) +' wide gaps: '+str(g))

        # plot results
power = np.log10( tmax_frame * frame_space)
fig1, ax1 = plt.subplots()
ax1.set_xlabel('Energy  '+r'$E$')
ax1.set_ylabel('Average Number of Zero Crossings')
ax1.set_xscale('log')
ax1.scatter( E_array, zeros_array, color ='black', marker ='+')

fig2, ax2 = plt.subplots()
ax2.set_xlabel('Energy  '+r'$E$')
ax2.set_ylabel('Average Number of Big Gaps')
ax2.set_xscale('log')
ax2.scatter( E_array, gaps_array, color ='black', marker ='+')

fig3, ax3 = plt.subplots()
ax3.set_xlabel('Temperature  '+r'$T$')
ax3.set_ylabel(r'$\alpha=\dfrac{E}{NT}$')
ax3.set_xscale('log')
ax3.scatter( T_array, E_array / (N*T_array), color ='black', marker ='+' )
```

9.7 Figure 9 - Pathology

```
 """
Produces Figure 9
Pathological Field Configuration (artificially)
"""

import numpy as np
import matplotlib.pyplot as plt
L = 100
N = 512
axis = np.linspace(0,L,N)
array=np.sin(2*np.pi*axis/L)


v = np.array( [0.2, 0.5, 0.8, 1.0, 1.2, 1.2, 1.0, 0.8, 0.5, 0.2] )
u = np.zeros(90)

array4 = np.concatenate( (u, -v,  u, -v, np.zeros(112), v, u, v, u ) )

f = np.cbrt(array) + array4 + np.random.random(N)*0.1

fig1, ax1 = plt.subplots()
axis = np.linspace(0,L,N)
ax1.set_xlabel(r'$x$')
ax1.set_ylabel(r'$f$')
ax1.set_ylim(-1.5, 1.5)
ax1.plot(axis, np.ones(N), color='black', \
     linestyle = 'dashed', alpha = 0.7)
ax1.plot(axis, np.zeros(N), color='black', \
                 linestyle = 'dashed', alpha = 0.7)
ax1.plot(axis, -np.ones(N), color='black', \
     linestyle = 'dashed', alpha = 0.7)
ax1.plot(axis, f, color='black')
```

9.8 Figure 10 - Pairs Cumulative Count

```
"""
Produces Figure 10

Plots cumulative count of kink number over time
"""
from Discretisation import np, plt, next_frame, energy, frame_space
from Initial_Conditions import heat_bath
from Kinks_and_Creations import pairs

T = 1.0
target = 10**4

f_old, f = heat_bath(T)
E = int( energy(f_old, f)[-1] )
cum_sum_pairs = np.array( [pairs(f)] )
time_array_dt = np.array( [0] )

while cum_sum_pairs[-1] < target:

    # evolve to next frame
    f_old, f = next_frame(f_old, f)
    time_array_dt = np.append( time_array_dt, \
                              time_array_dt[-1] + frame_space)
    cum_sum_pairs = np.append( cum_sum_pairs, \
                              cum_sum_pairs[-1] + pairs(f))
    print( str(cum_sum_pairs[-1]) +' out of ' +str(target))

plt.figure()
plt.title('Field prepared at Temperature ' +r'$T=$'+str(T) \
             +'\n Initial Energy '+r'$E=$'+str(E))
plt.xlabel('Timesteps')
plt.ylabel('Cumulative Sum of Pair Detections')
plt.plot(time_array_dt, cum_sum_pairs, drawstyle='steps-post' )
```

9.9 Figure 11 - Pair Number Energy Dependence - long

```
"""
Produces Figure 11

Energy Dependence of the Average Number of Pairs, long range
"""
from Discretisation import np, plt, N, frame_space
from Initial_Conditions import heat_bath
from Test_Functions import pairs_test


num_tests = 25          # number of tests
T_min = 10**-1          # minimum temperature
T_max = 10**3           # maximum Temperature
e_tests = 1000          # nuber of energy evaluations
tmax_frame = 10**5      # number of frames for evolution to be traced over


# initialise arrays

T_array = 10**np.linspace( np.log10(T_min), np.log10(T_max), num_tests )
E_array = np.zeros( num_tests )
pairs_array = np.zeros( num_tests )


# for each temperature
for i, T in enumerate(T_array):

    # progress bar
    print(str(i+1) + ' out of '+str(num_tests))

    # intial conditions of this temperature
    f_old, f = heat_bath(T)

    # calculate average energy, number of zeros and gaps
    E, n = pairs_test(T, e_tests, tmax_frame)
    E_array[i] = E
    pairs_array[i] = n

        # plot results
power = np.log10( tmax_frame * frame_space)
fig1, ax1 = plt.subplots()
ax1.set_xlabel('Energy  '+r'$E$')
ax1.set_ylabel('Average Number of Pairs'+r'$\langle n \rangle$')
ax1.set_xscale('log')
ax1.scatter( E_array, pairs_array, color ='black', marker ='+')


fig3, ax3 = plt.subplots()
ax3.set_xlabel('Temperature  '+r'$T$')
ax3.set_ylabel(r'$\alpha=\dfrac{E}{NT}$')
ax3.set_xscale('log')
ax3.scatter( T_array, E_array / (N*T_array), color ='black', marker ='+' )
```

9.10 Figure 12 - Pair Number Energy Dependence - short

```
"""
Produces Figure 12

Energy Dependence of the Average Number of Pairs, short range
"""
from Discretisation import np, plt, N, frame_space, next_frame, energy
from Initial_Conditions import initial_fourier, heat_bath_iteration, heat_bath
from Test_Functions import pairs_test


num_tests = 25          # number of tests
T_min = 0.7             # minimum temperature
T_max = 1.5             # maximum temperature
e_tests = 1000          # number of energy evaluations
tmax_frame = 10**5      # number of frames for evolution to be traced over


# initialise arrays
T_array = np.linspace( T_min, T_max, num_tests )
E_array = np.zeros( num_tests )
pairs_array = np.zeros( num_tests )


# for each temperature
for i, T in enumerate(T_array):

    # progress bar
    print(str(i+1) + ' out of '+str(num_tests))

    # intial conditions of this temperature
    f_old, f = heat_bath(T)

    # calculate average energy, number of zeros and gaps
    E, n = pairs_test(T, e_tests, tmax_frame)
    E_array[i] = E
    pairs_array[i] = n

        # plot results

power = int(np.log10( tmax_frame * frame_space))
fig1, ax1 = plt.subplots()
ax1.set_title('Over 10^' +str(power)+' timesteps')
ax1.set_xlabel('Energy  '+r'$E$')
ax1.set_ylabel('Average Number of Pairs '+r'$\langle n \rangle$')
ax1.scatter( E_array, pairs_array, color ='black', marker ='+')

fig2, ax2 = plt.subplots()
ax2.set_xlabel('Temperature  '+r'$T$')
ax2.set_ylabel(r'$\alpha=\dfrac{E}{NT}$')
ax2.scatter( T_array, E_array / (N*T_array), color ='black', marker ='+' )
```

9.11 Figures 13 and 14 - Smoothing

```
"""
Produces Figures 13 and 14

Plots pair number over time, unsmoothed and smoothed
"""
from Discretisation import np, plt, frame_space, next_frame, energy
from Initial_Conditions import heat_bath
from Kinks_and_Creations import pairs, smooth,buff_frame

T = 1.0
tmax_frame = 200

time_array_dt = np.arange(0, (buff_frame+tmax_frame)*frame_space, frame_space)
num_frames = len(time_array_dt)
pairs_array = np.zeros( num_frames)
f_old, f = heat_bath(T)
E = 0

for i in range(num_frames):
    f_old, f = next_frame(f_old, f)

    pairs_array[i] = pairs(f)
    E += energy(f_old, f)[-1]

print( 'Average Energy: '+str( E / num_frames ) )
fig1, ax1 = plt.subplots()
ax1.set_xlim(0, tmax_frame*frame_space)
ax1.set_ylim(-0.1, 2.1)
ax1.set_xlabel('Timesteps')
ax1.set_ylabel('Pair Number '+r'$n$')
ax1.plot(time_array_dt, pairs_array, color='black', drawstyle='steps-post')

smoothed_array = smooth(pairs_array, tmax_frame)
fig2, ax2 = plt.subplots()
ax2.set_xlim(0, tmax_frame*frame_space)
ax2.set_ylim(-0.1, 2.1)
ax2.set_xlabel('Timesteps')
ax2.set_ylabel('Pair Number '+r'$n$')
ax2.plot(time_array_dt[:tmax_frame], smoothed_array,
         color='black', drawstyle='steps-post')
```

9.12 Figure 15 - Creation Time τ_0

```
"""
Produces Figure 15

Energy Dependence of creation time
"""
from Discretisation import np, N, plt
from Test_Functions import Gamma_and_tau_test


num_tests = 25          # number of tests
T_min = 0.5             # minimum temperature
T_max = 1.0             # maximum Temperature
e_tests = 1000          # nuber of energy evaluations
tmax_frame = 10**5      # number of frames for evolution to be traced over

T_array = np.linspace( T_min, T_max, num_tests )


#   initialise arrays
E_array = np.zeros( num_tests )
tau_array = np.zeros( num_tests )

for i, T in enumerate(T_array):

    #   progress bar
    print(str(i+1) + ' out of '+str(num_tests))

    E, gamma, tau = Gamma_and_tau_test(T, e_tests, tmax_frame)
    E_array[i] = E
    tau_array[i] = tau

    print( 'Energy: ' + str(int(E)) +' Creation Time: '+str(tau) )

        #   plot results
fig1, ax1 = plt.subplots()
ax1.set_xlabel('Energy  '+r'$E$')
ax1.set_ylabel('Creation Time ' + r'$\tau_0$')
ax1.scatter( E_array, tau_array, color='black', marker ='+')


fig2, ax2 = plt.subplots()
ax2.set_xlabel('Temperature  '+r'$T$')
ax2.set_ylabel(r'$\alpha=\dfrac{E}{NT}$')
ax2.scatter( T_array, E_array / (N*T_array), color='black', marker ='+' )
```

9.13 Figure 16 -Creation Rate Γ

```
"""
Produces Figure 16

Energy Dependence of creation rate gamma
"""
from Discretisation import np, plt, N
from Test_Functions import Gamma_and_tau_test


num_tests = 25          # number of tests
e_tests = 1000          # number of energy evaluations
tmax_frame = 10**5      # number of frames for evolution to be traced over


#   initialise arrays
inverse_T_array = np.linspace( 0, 2, num_tests)
G_array = np.zeros( num_tests )
inverse_E_array = np.zeros( num_tests )

for i, inverse_T in enumerate(inverse_T_array):

    #   progress bar
    print(str(i+1) + ' out of '+str(num_tests))

    E, Gamma = Gamma_and_tau_test( 1/inverse_T, e_tests, tmax_frame )[:-1]

    inverse_E_array[i] = 1/E
    G_array[i] = Gamma


#   plot results
fig1, ax1 = plt.subplots()
ax1.set_xlabel('Inverse Energy  '+r'$1/E$')
ax1.set_ylabel('Creation Rate ' + r'$\Gamma$')
ax1.set_yscale('log')
ax1.scatter( inverse_E_array, G_array, color='black', marker ='+')


fig2, ax2 = plt.subplots()
ax2.set_xlabel('Temperature  '+r'$T$')
ax2.set_ylabel(r'$\alpha=\dfrac{E}{NT}$')
ax2.scatter( inverse_T_array, inverse_T_array / (N*inverse_E_array) ,
            color='black', marker ='+' )
```

10 Appendix 3 - Miscellaneous

10.1 Animation Function

```
"""
Animation

Not used in project directly,
we found it useful to see how the field behaved
"""
from Discretisation import np, plt, L, N, next_timestep, energy
from Initial_Conditions import heat_bath
from Kinks_and_Creations import pairs
from matplotlib.animation import FuncAnimation


T = float( input("Enter Temperature: " ) )
ani_frame_space = int( input("Enter Frame Spacing: ") )

def animate(i):
    global f_old, f, TimeCounter
    if not pause:

        for _ in range(ani_frame_space):
            f_old, f = next_timestep(f_old, f)
            TimeCounter+=1
        E = int(energy(f_old, f)[-1])
        p = pairs(f)


    line.set_ydata(f)
    line2.set_ydata(np.sign(f))
    title.set_text("\n Time Elapsed = "+str(TimeCounter) +' timesteps'
                       +"\n Energy = "+str(E)
                       +"\n Pairs = "+str(p)
                       )
    return line,


# Pause function
def toggle_pause(event):
    global pause
    pause = not pause
    if pause:
        ani.event_source.stop()
    else:
        ani.event_source.start()


# Initialize
TimeCounter=0
f_old, f = heat_bath(T)
fig, ax = plt.subplots()
ax.set_ylim(-1.5, 1.5)
x = np.linspace(0, L, N)
title = ax.text(0.5,0.85,"",bbox={'facecolor':'w', 'alpha':0.5, 'pad':5},
                transform=ax.transAxes, ha="center")
line, = ax.plot(x, f, color='black')
line3, = ax.plot(np.zeros(L), alpha=0.5, \
                 linestyle='dashed', color = 'black')
line4, = ax.plot(1.0*np.ones(L), alpha=0.5, \
                 linestyle='dashed', color = 'black')
line5, = ax.plot(-1.0*np.ones(L), alpha=0.5, \
                 linestyle='dashed', color = 'black')
pause = False
plt.connect('key_press_event', toggle_pause)

ani = FuncAnimation(fig, animate, frames=None, interval=100,
                    cache_frame_data=False)
plt.show()
```

10.2 Speed Comparison

```
"""
Produces Data for Table in section 2.1

Comparison of Speeds of 3 implementations of the finite difference method.
"""
from Discretisation import np, C_1,C_2, C_3, N, next_timestep
from Initial_Conditions import heat_bath
import time


T = 1.0
tmax = 10**5


def Explicit_Loop( f_old , f ):
    """
    Given the previous and current field configurations, 'f_old' and 'f',
    updates the field configurations to the next timestep
    according to equation (13)

    In particular uses the Explicit Loop

    Returns
    -------
    f :     current field configuration
    f_new : next field configuration
    """
    f_new = - f_old + C_3 * f ** 3
    for i in range ( N ) :
        f_new [ i - 1 ] += C_1 * f [ i - 1 ] + C_2 * ( f [ i - 2 ] + f [ i ] )
    return f , f_new


M = np . zeros ( N * N ) . reshape (( N , N ) )
for i in range( N ):
    M [i , i - 1 ] = C_2
    M [i , i ] = C_1
    M [ i -1 , i ] = C_2

def Matrix( f_old , f ):
    """
    Given the previous and current field configurations, 'f_old' and 'f',
    updates the field configurations to the next timestep
    according to equation (13)

    In particular uses the Matrix Method equation (14)

    Returns
    -------
    f :     current field configuration
    f_new : next field configuration
    """
    return f , - f_old + np . matmul (M , f ) + C_3 * f * f * f


# test of explicit loop
f_old, f = heat_bath(T)
t0 = time.time()
for _ in range(tmax):
    f_old, f = Explicit_Loop( f_old , f )
t1 = time.time()
t = t1-t0

print('Explicit Loop ' +
      'Time Taken: '+str(t) +
      ' Updates per second: '+ str(tmax/t) )



# test of Matrix Method
f_old, f = heat_bath(T)
t0 = time.time()
for _ in range(tmax):
    f_old, f = Matrix( f_old , f )
```

```
t1 = time.time()
t = t1-t0
print('Matrix ' +
      'Time Taken: '+str(t) +
      ' Updates per second: '+ str(tmax/t) )


# test of Rolling Array

f_old, f = heat_bath(T)
t0 = time.time()
for _ in range(tmax):
    f_old, f = next_timestep( f_old , f )
t1 = time.time()
t = t1-t0
print('Rolling Array ' +
      'Time Taken: '+str(t) +
      ' Updates per second: '+ str(tmax/t) )
```