Final Project Complete Draft: What is the origin of the arrow of time?

Zoe King
(Brown University PHYS1600: Computational Physics)
(Dated: April 30, 2024)

I. ABSTRACT

How can one account for the difference between the past and the future when an examination of the laws of physics concede only symmetry in time[1]? The characterization and understanding of the arrow of time, the inherent asymmetry in the temporal evolution of physical processes, remains a fundamental challenge in physics. While the underlying laws of physics are time-symmetric, real-world phenomena exhibit the clear directionality of time. This paper investigates the causes of time asymmetry through computational simulations of multiple molecular systems, considering factors such as particle characteristics, inter-particle interactions, system density, and environmental effects. By analyzing the behavior of molecular dynamics under varying conditions, including the incorporation of Langevin dynamics to model environmental influences, it explores the breakdown of time reversibility and its relation to the system's characteristics. The findings reveal the intricate interplay between initial conditions, system characteristics, and environmental factors in shaping the temporal asymmetry. Despite advancements, the underlying mechanisms driving time asymmetry remain complex and warrant further investigation. **This research aims to answer: what are the underlying sources of the directional bias in time for physical processes if the laws that govern them are impartial to it?** [2]

II. INTRODUCTION

The causes of the arrow of time are still under debate but likely constitute a multidisciplinary phenomenon with several potential justifications. One possibility under discussion is that the underlying microscopic laws are not perfectly time-symmetric, thereby causing this imbalance in time [3]. However, when tested, this violation of symmetry, while sound in its ideals, was found to be weak and observable less than one percent of the time [4]. Another possible cause of time asymmetry is that although the laws of physics define what is possible, it is the constraints of the problem—such as initial, boundary, or symmetry conditions—that are much more relevant in explaining the arrow of time [4].

This research aims to answer: what is the underlying source of the directional bias in time for physical processes if the laws that govern them are impartial to it? [2]

III. BACKGROUND AND THEORY

At the atomic scale, the world is a four-dimensional continuum extended in space and time. Yet, intuitively, we perceive the world as being extended in space while unfolding in time [4]. At some level, this distinction makes. We can remember the past, but, despite the fantasies of science fiction, we cannot see into or remember the future. We assign special significance to the present moment in time, viewing it as a wave continuously transforming and leaving behind a wake, otherwise known as the past [4]. This view of time moving linearly in one direction gave birth to the concept of "The Arrow of Time." However, according to the most fundamental laws of physics, this shouldn't be the case. The laws of physics allow a process to happen in either direction regardless of the direction of time; therefore, the distinction between the past and the future all but disappears [2]. Yet, the perception of the world as unfolding in time rather than being extended in it cannot be dismissed as simply a subjective view, as it has objective counterparts in biological, geographical, and astronomical processes. These processes, such as air leaking out of a tire and hot items losing their heat in room temperature environments, are, in practice, found to be irreversible [2]. There are also examples found to extend over more prolonged periods of time, such as the physiology underlying growth and development, and organic evolution creating an increasing variety of highly organized life forms. Both of these processes are found to only move in one direction [4].

As seen above, the processes that follow the arrow of time can be separated into two categories. There are those that generate order or information, transforming from a simpler state to a more complex one, such as evolution [4]. These are referred to by Layzer as the *'Historical' Arrow of Time*. The others are manifestations of the second law of

thermodynamics, which states that the natural process of things is to generate entropy. In other words, they destroy information and generate more disorder. Layzer referred to these as the *'Thermodynamic' Arrow of Time* [4]. Despite finding many natural examples at the macro-scale, both the historical and thermodynamic arrows of time are unable to be observed at the microscopic level.

Microscopic physics gives no special status to any moment in time and can only weakly distinguish between the direction of the past and that of the future [4]. The motion of a single particle generates neither information nor entropy, and the concept of "Order" is only a concept in a macroscopic view, and therefore is only a property of a system that is made up of many particles. The notion of order loses all meaning when it is applied to singular molecules in space. "In the physics of elementary particles, the world changes but does not evolve" [4].

IV. METHOD OF INVESTIGATION

The first step towards understanding the nature of the temporal asymmetry involved in the *Thermodynamic Arrow of Time* is the application of the laws of classical mechanics to individual molecules, from which a macroscopic system is composed [1]. In a sufficiently complex system, such as a physical reaction involving a large number of particles, a definite directionality of time flow can be observed. Conversely, when focusing on elementary interactions between fewer particles, the directionality of the flow is lost. This principle is contained in the second law of thermodynamics, which again states that ordered systems evolve in such a way as to increase their degree of disorder, or randomness. The evolution toward increasing randomness is what defines the direction of time [3]. Studying the microscopic and macroscopic origins of entropy and its relation to the arrow of time could help in understanding the underlying causes of the arrow of time.

In this project, the system under consideration is a simplified gas model consisting of varying numbers, N, of identical spherical particles. The system in which the particles move is isolated and adiabatic. The motion of the particles is treated according to the laws of classical non-relativistic mechanics [1].

A. Computational Algorithms

To study the breakdown of time reversibility in molecular systems, I utilized Molecular Dynamics Simulations as a starting point and modified them to simulate different molecular systems depending on the variable being tested. From there, I observed how the reversibility of time evolves as the system changes and determined at what point the reversibility of time breaks down.

The code is laid out in the following steps:

1) First, a function is defined to run the molecular dynamics simulation for a given system.

2) Then, a loop is implemented to iterate over different systems and run the simulation for each variation.

3) Relevant data, such as the evolution of energy, temperature, and other statistical quantities over time, is recorded.

4) Finally, the results are plotted to help determine at what point the reversibility of time starts to break down.

B. Equations

This problem combines principles from classical mechanics, statistical mechanics, and numerical methods to simulate the dynamics of molecular systems and study their behavior over time. The major equations used in order to solve this problem are:

1) **Newton's Laws of Motion:** The dynamics of the particles in the system are governed by Newton's laws, particularly the second law, which relates the force acting on a particle to its acceleration. Mathematically, these equations would be represented as follows.

First Law

Law of Inertia: An object at rest stays at rest and an object in motion stays in motion with the same speed and in the same direction unless acted upon by an unbalanced force.

$$\sum \mathbf{F} = 0 \tag{1}$$

Second Law

Law of Acceleration: The acceleration of an object is directly proportional to the net force acting on it and inversely proportional to its mass.

$$\mathbf{F} = m\mathbf{a} \tag{2}$$

Third Law

Action and Reaction: For every action, there is an equal and opposite reaction.

$$\mathbf{F}_{\text{on } A} = -\mathbf{F}_{\text{on } B} \tag{3}$$

2) **Entropy and the Second Law of Thermodynamics:** The code implicitly involves the second law of thermodynamics, which states that isolated systems tend to evolve towards states of higher entropy or disorder. This principle is reflected in the behavior of the system as it evolves over time.

3) **Verlet Integration:** The Verlet integration algorithm is used to numerically integrate the equations of motion. It updates the positions and velocities of the particles based on the forces acting on them.

4) **Lennard-Jones Potential:** The Lennard-Jones potential describes the pairwise interaction energy between particles in the system. It consists of a repulsive term proportional to r^-12 and an attractive term proportional to r^-6, where r is the distance between particles.

5) **Minimum Image Convention:** This principle is applied to handle periodic boundary conditions in the simulation. It ensures that particle interactions are correctly accounted for across periodic boundaries by considering the closest image of each particle.

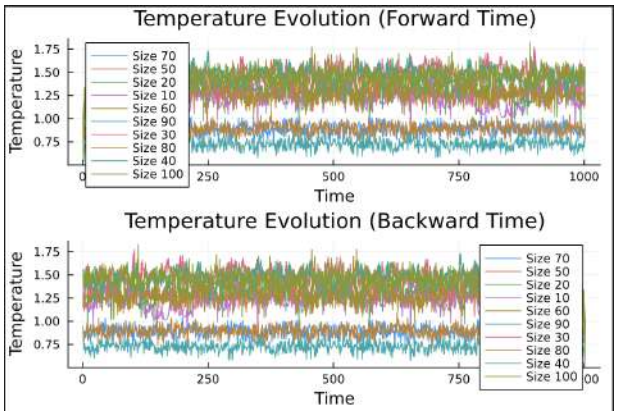
6) **Statistical Mechanics:** Concepts from statistical mechanics are used to compute thermodynamic quantities such as temperature, energy, and pressure. For example, the temperature of the system is related to the kinetic energy of the particles.

C. Control Simulations

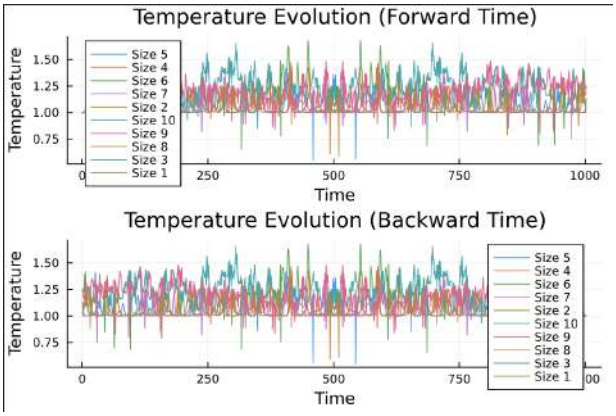
To confirm time reversibility in a simulation, particularly in the context of molecular dynamics or statistical mechanics, one would typically examine plots of physical quantities over time. The energy and temperature plots can provide valuable insights into the behavior of the system under time reversal. Here's how they can confirm time reversibility:

Temperature Behavior: Temperature is related to the average kinetic energy of the particles in the system. In a reversible system, the temperature should also remain constant over time. When the simulation is reversed in time, the temperature plot should exhibit the same values at corresponding time points as during the forward run. Any discrepancies in temperature behavior would suggest a violation of time reversibility.

Energy Conservation: Similar to the temperature, in a reversible system, the total energy of the system should remain constant over time. This means that when the simulation is run forward and then backward in time, the energy plot should exhibit the same values at corresponding time points. Any deviation from this behavior would indicate a lack of time reversibility.

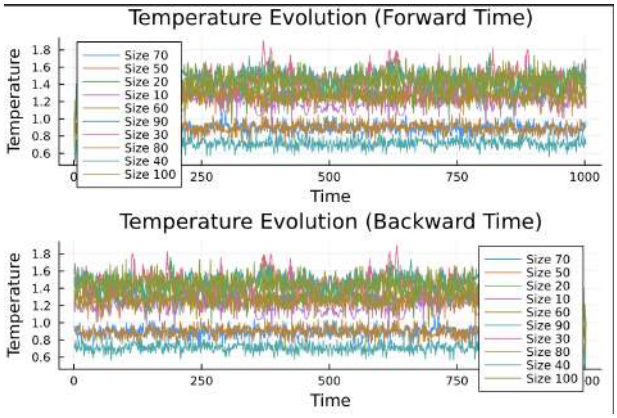


(a) Systems 10-100

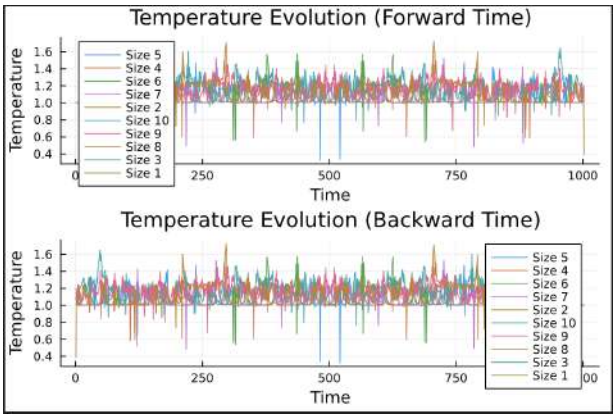


(b) Systems 1-10

FIG. 1: Temperature Evolution for Runtime = 50

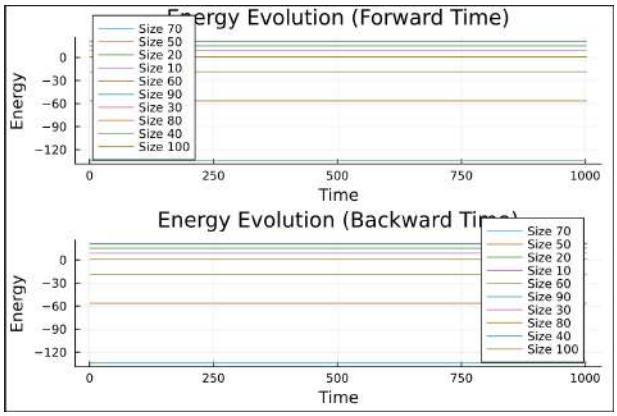


(a) Systems 10-100

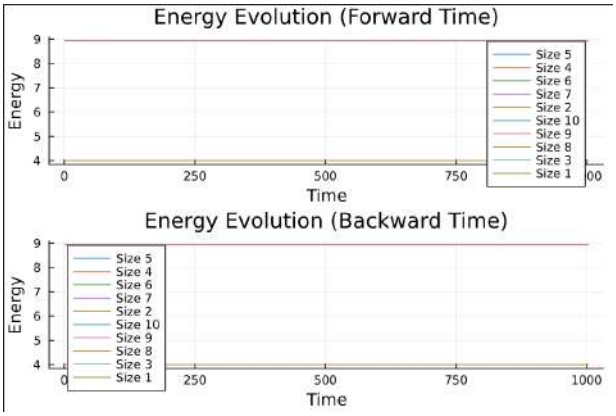


(b) Systems 1-10

FIG. 2: Temperature Evolution for Runtime = 5



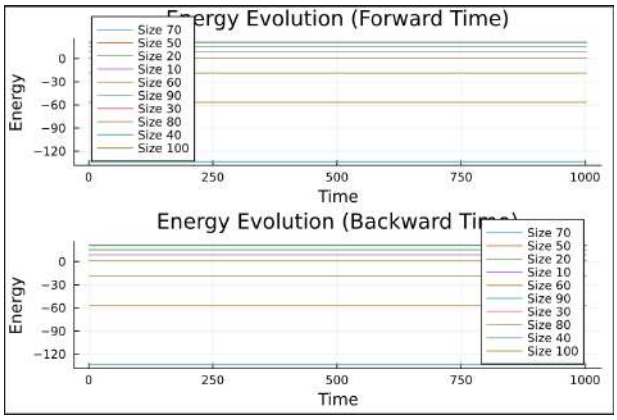
(a) Systems 10-100



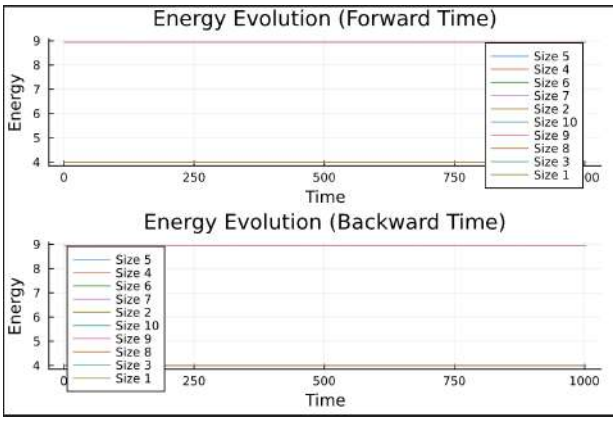
(b) System 1-10

FIG. 3: Energy Evolution for Runtime = 50

By comparing the energy and temperature plots obtained during the forward and backward runs of the simulation, one can assess whether the system behaves consistently under time reversal. Figures 1 and 2 show the temperature evolution for systems with varying particle numbers, both forward and backward in time. The close match between



(a) Systems 10-100



(b) System 1-10

FIG. 4: Energy Evolution for Runtime = 5

these figures provides evidence that the simulation algorithm is indeed time-reversible and accurately captures the dynamics of the system. Similarly, Figures 3 and 4 depict the energy of the systems, which remains unchanged, confirming the algorithm's time reversibility. Significant deviations in these plots could indicate issues with the simulation algorithm or numerical errors. These observations suggest that time reversibility is a possibility for all systems.

V. RESULTS

A. System Size Reversibility(Ideal Molecules)

1. Varying Sized Systems over Long Time Periods

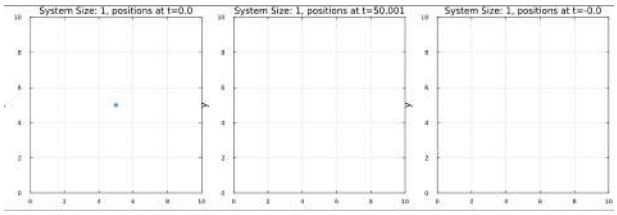
The initial approach to the problem involved testing a range of systems sized from 1 to 100 for a runtime of 50 units. The test attempts to show that smaller-sized systems, in theory, should show evidence of time reversal that would break down as the systems grew in size. However, that was not seen as none of the systems demonstrate the ability to be reversed in time (Figures 5a, 5b, 5c, 5d, 5e, 5f, 5g, 5h).

Despite the expectation that smaller-sized systems would exhibit evidence of time reversal, none of the tested systems demonstrated the ability to be reversed in time. This suggests that even in small systems, the inherent uncertainty in initial conditions may prevent perfect time reversal.

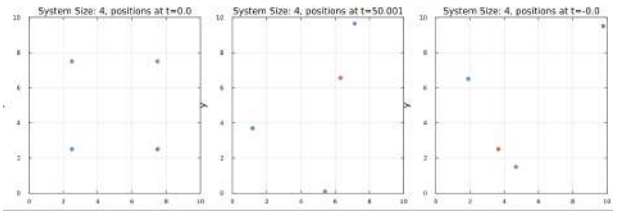
2. Varying Size of System over Shorter Time Periods

Since evidence of time reversal over large periods of time was not found, as shown in Figure 5, the amount of time the particles had to be reversed over was adjusted. Instead of a runtime of 50 units, the runtime was decreased to 5 units. It was expected, based on the literature, that time reversal would be visible in smaller-sized systems and that it would break down as the system size increased. However, this was not found to be the case. While smaller systems at the lower run time now showed time reversal (Figures 6c, 6b), the system made of only one particle was unable to be reversed (Figure 6a). Additionally, the assumption that in a sufficiently complex system, such as one involving a large number of particles, a definite directionality of time flow could be observed, and conversely, when focusing on elementary interactions between fewer particles, the directionality of the flow would be lost [3], also seems to not be the case. The systems with larger numbers of particles were also found to have lost the distinct significance of time (Figures 6d, 6e, 6f, 6g, 6h, 6i, 6j).

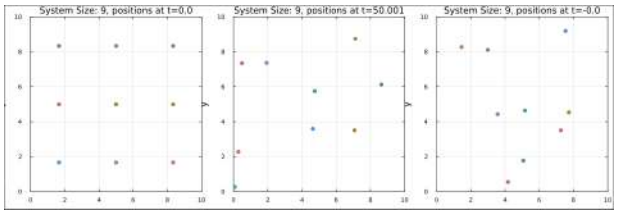
Contrary to expectations, the ability to reverse time did not deteriorate as system size increased. Even larger systems did not exhibit clear evidence of time reversal, indicating that factors beyond system size play a significant role in determining time asymmetry.



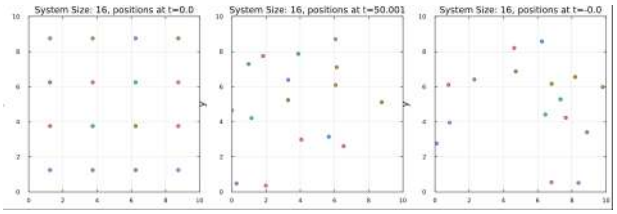
(a) System size 1



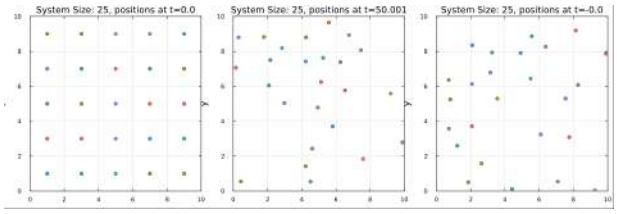
(b) System size 4



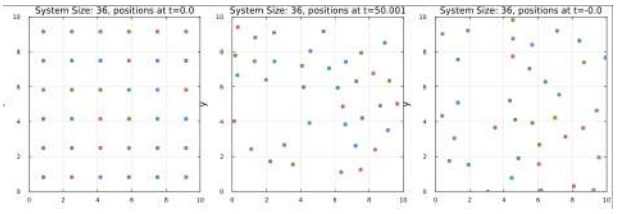
(c) System size 10



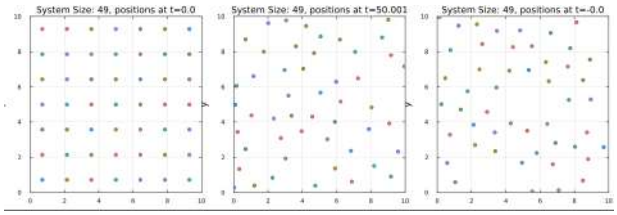
(d) System size 20



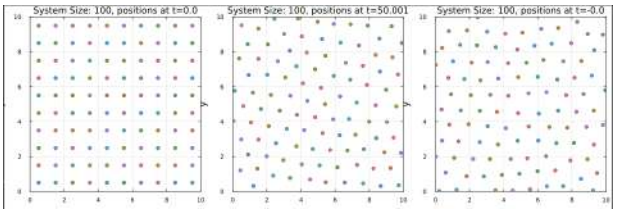
(e) System size 30



(f) System size 40



(g) System size 50



(h) System size 100

FIG. 5: Time Reversal: Runtime 50

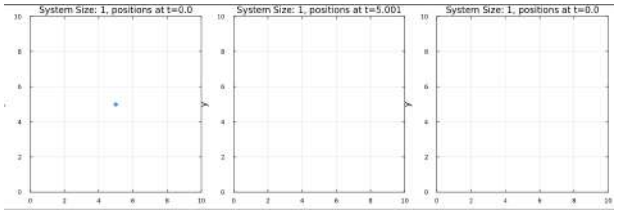
B. Atomic Interactions

So far, the project has been restricted to a perfectly ideal gas model with no inter-particle interaction. From this, system size does not seem to be the contributing factor to the reversibility of time, since in the ideal gas system, the particles travel in a straight line, making time reversal essentially exact. In this next phase, collisions between particles were taken into account, and the consequences of switching on their interactions were investigated.

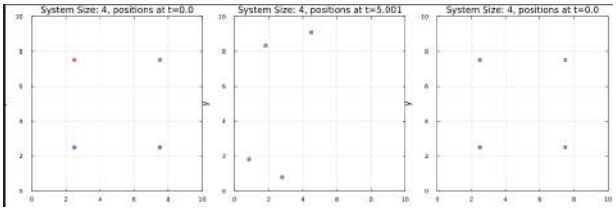
1. Varying Box Volume

Instead of changing the number of particles in the system, the relative density of the box is altered by running the simulation for varying box sizes. The particles are all held at a constant radius and mass for each volume trial, but for easy visualization, the particles in the graphs are scaled to show a more accurate depiction of the changes in density (Figure 7).

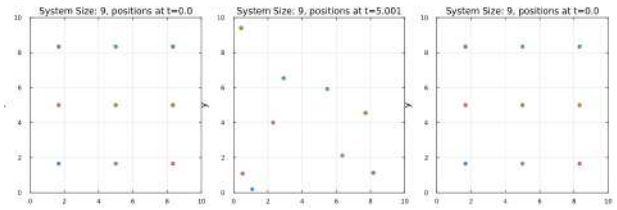
From the plots in Figure 7, it can be observed that particle interactions and density greatly affect a particle's ability to return to its starting position. In relatively high-density systems (Figures 7a & 7b), the particles don't perfectly return to their initial states but get fairly close. This could be due to the fact that, due to their closely packed density, they are unable to move that far in the first place. As the particles become more sparsely packed (Figures 7c & 7d), time reversibility worsens, leaving the final reversed state not even close to the initial particle arrangement. In the final density states (Figures 7e, 7f, & 7g), the particles behave similarly to the initial test where all the particles were able to return to their positions. This full time reversibility could be due to the fact that since the particles are so



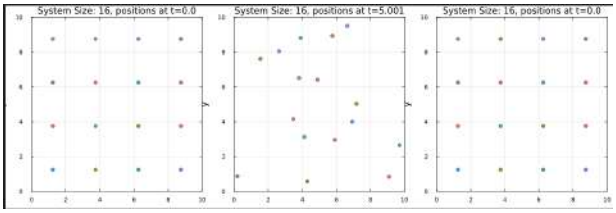
(a) System Size 1



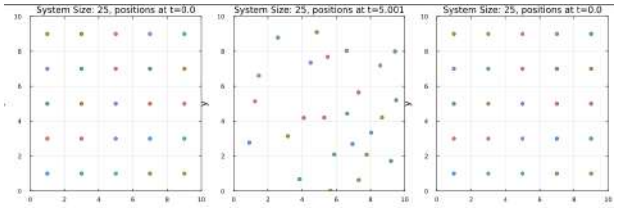
(b) System Size 4



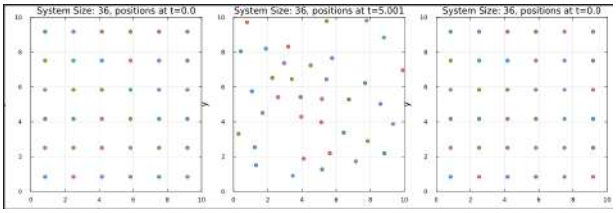
(c) System Size 10



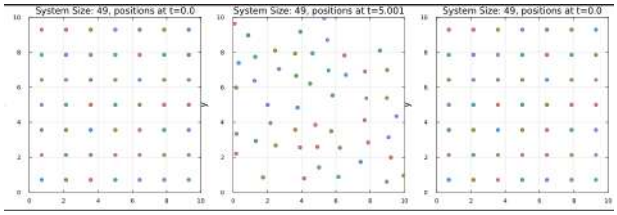
(d) System Size 20



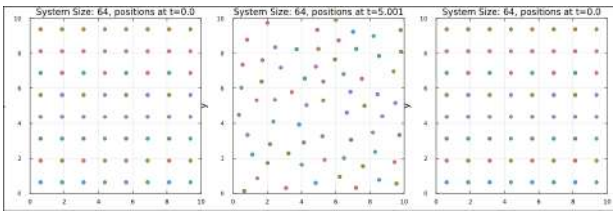
(e) System Size 30



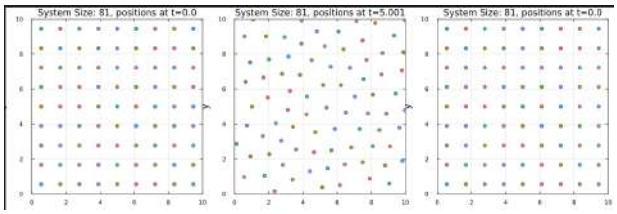
(f) System Size 40



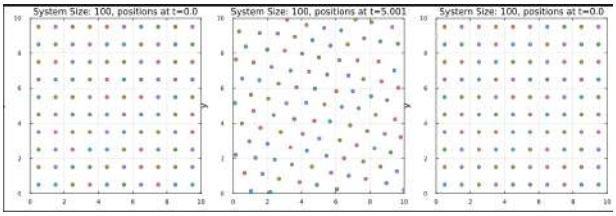
(g) System Size 50



(h) System Size 60



(i) System Size 80

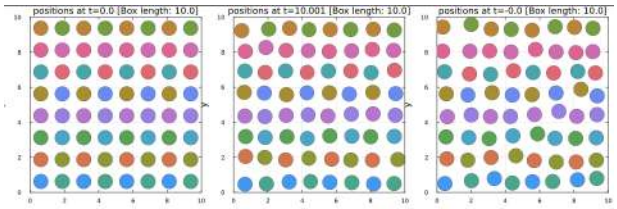


(j) System Size 100

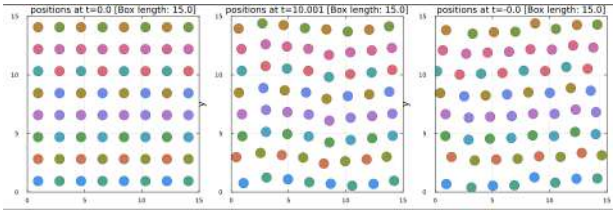
FIG. 6: Time Reversal: Runtime 5

spaced out, there are no interparticle collisions or interactions, meaning the particles are acting as though they are ideal particles and moving in a straight line, allowing for easy time reversibility.

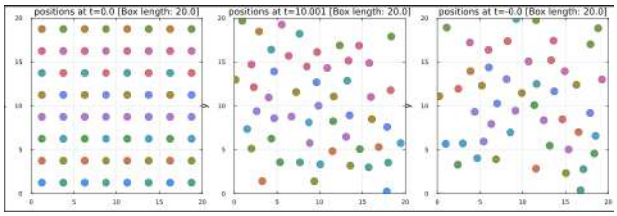
By varying the density, the sensitivity to initial conditions is examined, reflecting how initial uncertainty in the atoms' position and velocity evolves over time. This investigation aids in understanding how fine-grained entropy increases as a result, providing insight into the arrow of time. Specifically, the discussion about how particle interactions and density affect the ability of particles to return to their initial positions, along with the observations on time reversibility as density changes, directly addresses the investigation of how initial uncertainty evolves with time and contributes to the increase in entropy.



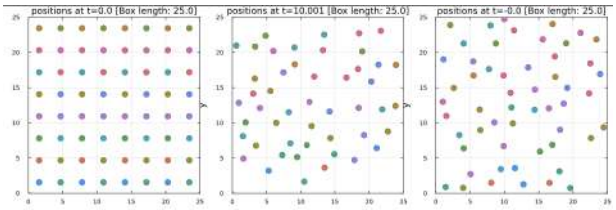
(a) Box Density: 0.64 particles per unit area



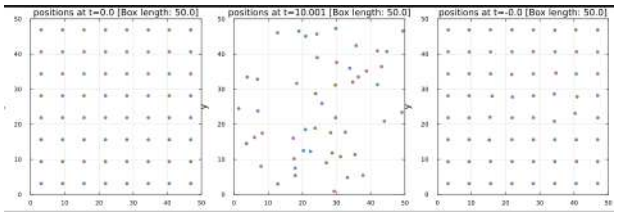
(b) Box Density: 0.2845 particles per unit area



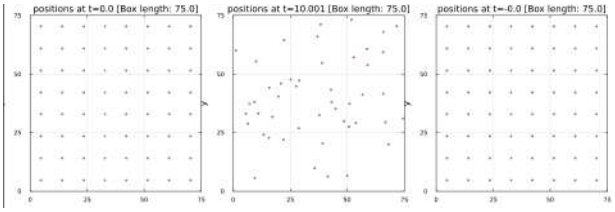
(c) Box Density: 0.16 particles per unit area



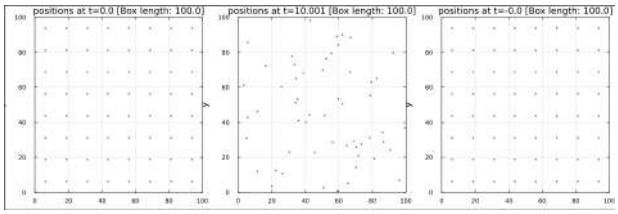
(d) Box Density: 0.1024 particles per unit area



(e) Box Density: 0.0256 particles per unit area



(f) Box Density: 0.0114 particles per unit area

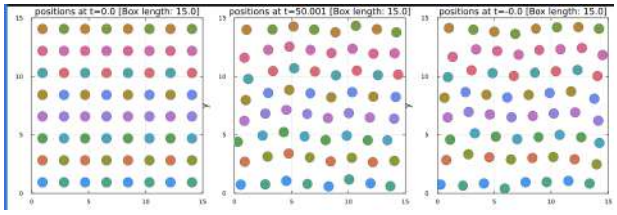


(g) Box Density: 0.0064 particles per unit area

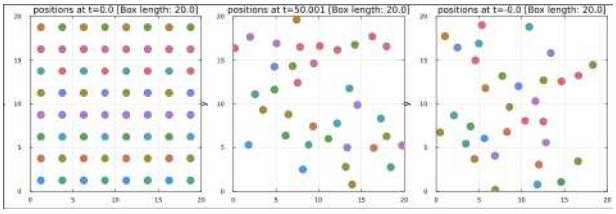
FIG. 7: Time Reversal: Runtime 10, Particle Mass 1, Particle Radius 1

2. *Varying Box Volume at Longer Time Intervals*

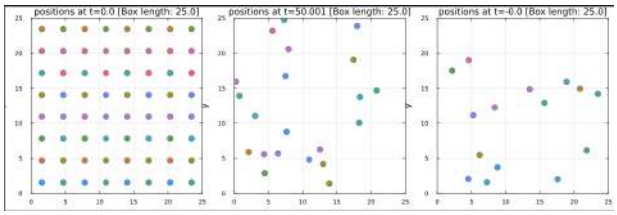
Performing the same trial but with a longer runtime produced the same results as seen in the Varying of Size test (Figures 5 & 6). With the increase in time from 10 to 50, there is a noticeable impact on the particles' ability to exhibit time reversal (Figure 8). This is especially prevalent when comparing the less dense systems that previously showed almost perfect time reversal (Figures 8d, 8e, & 8f).



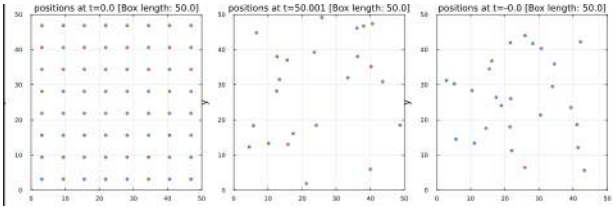
(a) Box Density: 0.2845 particles per unit area



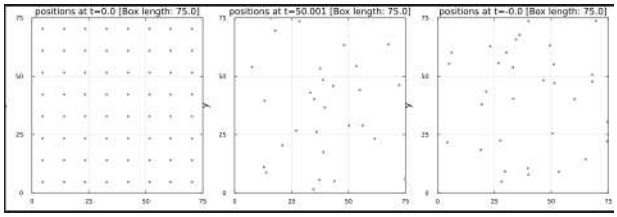
(b) Box Density: 0.16 particles per unit area



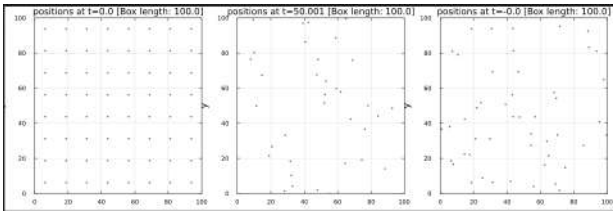
(c) Box Density: 0.1024 particles per unit area



(d) Box Density: 0.0256 particles per unit area



(e) Box Density: 0.0114 particles per unit area

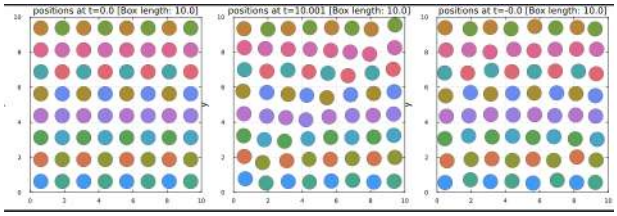


(f) Box Density: 0.0064 particles per unit area

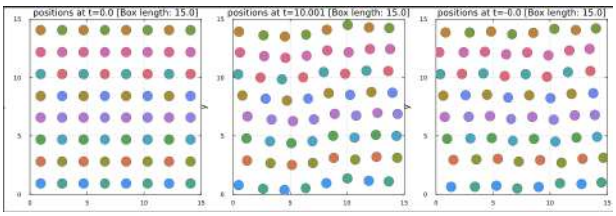
FIG. 8: Time Reversal: Runtime 50, Particle Mass 1, Particle Radius 1

3. Varying Box Volume with Heavier Particles

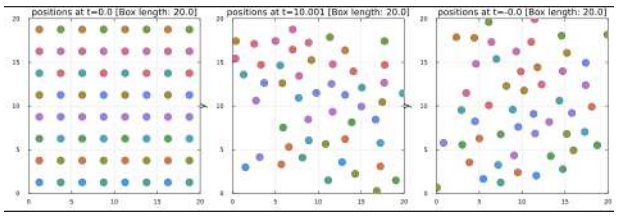
On the other hand, an increase in the mass of the particle seems to have minimal effects on the particles' ability to time reverse. Increasing the mass of the particles from 1 (Figure 7) to 10 (Figure 9) shows no significant change in particle behavior. This could be due to some improper calculations within the particle collisions that do not properly take into account the effects the change in mass should have on the particle, or it could simply be that mass is not a defining factor in the system.



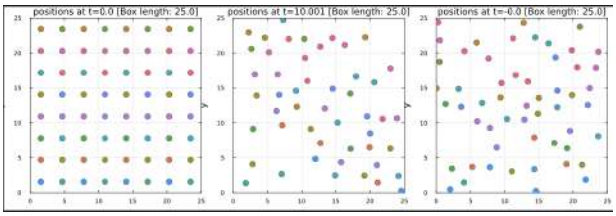
(a) Box Density: 6.4 particles per unit area



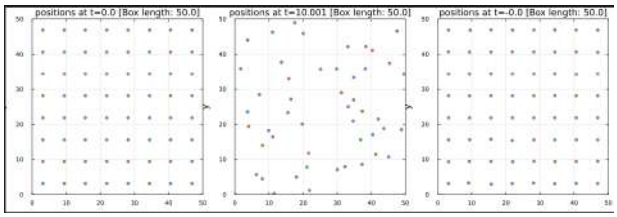
(b) Box Density: 2.8445 particles per unit area



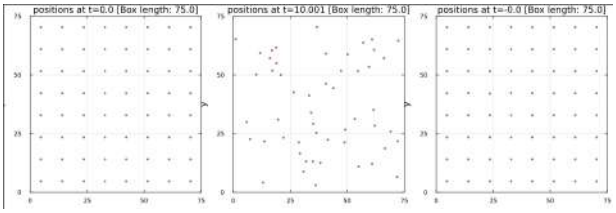
(c) Box Density: 1.6 particles per unit area



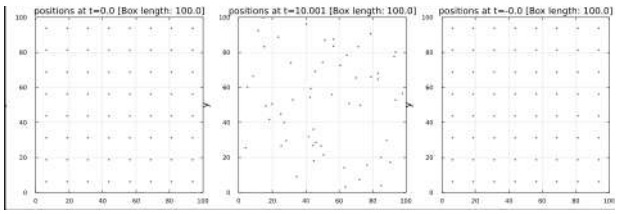
(d) Box Density: 1.024 particles per unit area



(e) Box Density: 0.256 particles per unit area



(f) Box Density: 0.1138 particles per unit area



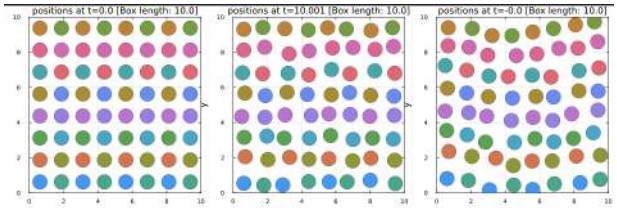
(g) Box Density: 0.064 particles per unit area

FIG. 9: Time Reversal: Runtime 10, Particle Mass 10, Particle Radius 1

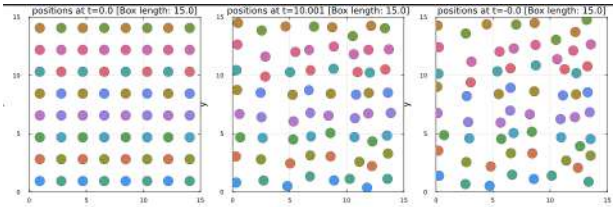
4. Varying Box Volume with Larger Particles

Reverting back to the lighter particles, an exploration was conducted to determine if the radius would have any effect on the time reversibility of the system. In the new system test, the radius of the particles was increased by a factor of 10 (Figure 10). It seems to be following the same patterns seen in section C1, and perhaps the particles are moving much less. The time reversibility for the small density areas is worse, which was unexpected since the lower radius particles would in theory have more room to move more chaotically, therefore making it harder to reverse.

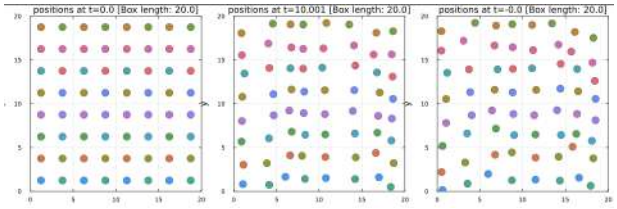
Introducing particle interactions and varying the density of the system had a significant impact on the ability of particles to return to their initial positions. Higher particle density resulted in better time reversibility, while lower density led to more pronounced deviations from the initial state. This highlights the sensitivity of the system's behavior to initial conditions and environmental factors. Altering particle mass and radius had minimal effects on



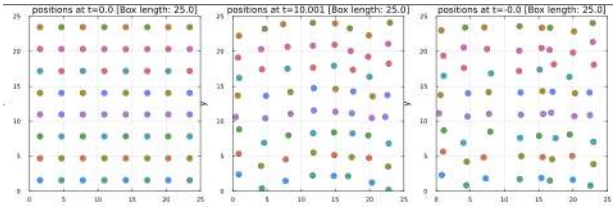
(a) Box Density: 0.64 particles per unit area



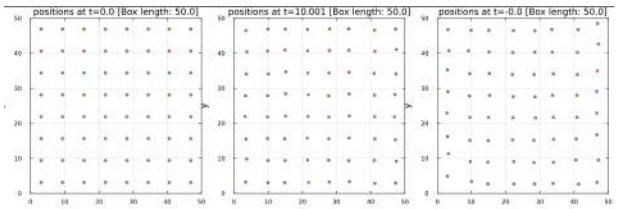
(b) Box Density: 0.2845 particles per unit area



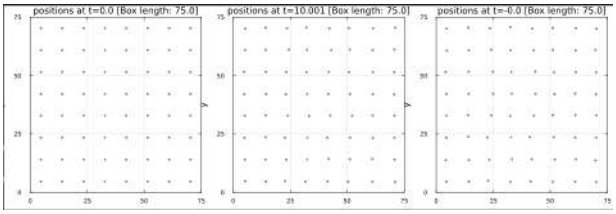
(c) Box Density: 0.16 particles per unit area



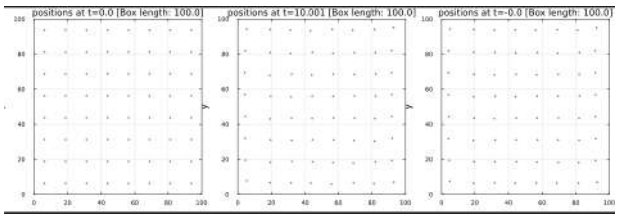
(d) Box Density: 0.1024 particles per unit area



(e) Box Density: 0.0256 particles per unit area



(f) Box Density: 0.0114 particles per unit area



(g) Box Density: 0.0064 particles per unit area

FIG. 10: Time Reversal: Runtime 10, Particle Mass 1, Particle Radius 10

the system's ability to reverse time. This suggests that factors other than particle characteristics may dominate the system's behavior and time asymmetry.

C. Environmental Effects on the System

Thus far, it has been observed that while system characteristics such as particle interactions and system density play a somewhat significant role in the behavior of molecular systems, they are not the sole determinants of time asymmetry. Uncertainty within the initial conditions also contributes significantly to the system's evolution over time. But where does the uncertainty in the initial conditions come from? Why is there an inherent unpredictability or imprecision in specifying the exact state of a system at a given point in time? The answer could stem from a variety of factors, including measurement limitations, microscopic fluctuations, and environmental disturbances.

In the next section, the aim is to extend the dynamics of the systems explored above to explicitly include their interactions with the environment. By incorporating environmental effects into the modeling framework, one could potentially gain a more comprehensive understanding of the system's behavior and its sensitivity to initial conditions.

In many cases, the entropic behavior of a system remains essentially unchanged by interactions with the outside world. For example, a clock in a sealed box would undoubtedly run down whether or not the motion of the walls of that box was taken into account [1]. Therefore, while environmental factors certainly contribute to the increase in entropy of the system, they are in no way necessary for the increase to occur. It is clear that an ice cube at room temperature will melt whether or not minute random influences from the galaxy are disturbing the molecules within [1].

1. Langevin Dynamics

In Langevin dynamics, stochastic forces are added to the equations of motion to mimic the effects of the environment on the system. This approach allows modeling of random fluctuations specifically due to interactions with the environment, such as thermal fluctuations or collisions with solvent molecules. Langevin dynamics is commonly used in classical molecular dynamics simulations to account for environmental influences and achieve equilibrium with the surroundings. By incorporating Langevin dynamics into the simulation, the system's behavior can be simulated in a more realistic environment [5].

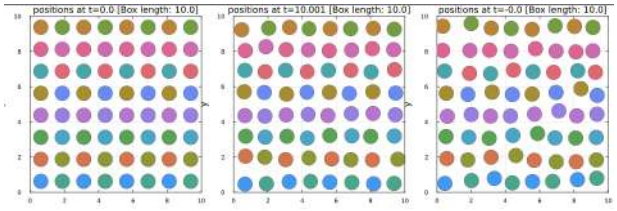
The equation for the random force in Langevin dynamics arises from the stochastic nature of the system, particularly from the random collisions between the particles and the surrounding molecules in the solvent or medium. This force is typically modeled as a stochastic process, representing the thermal fluctuations experienced by the particles.

The Langevin equation, which describes the dynamics of a particle in a dissipative and fluctuating environment, is given by:

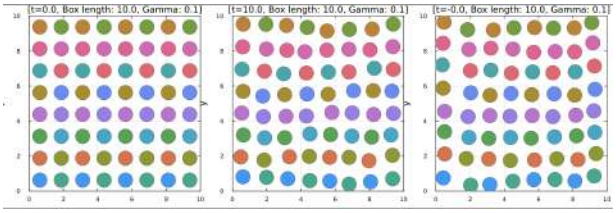
$$m\frac{d^2x}{dt^2} = -\gamma\frac{dx}{dt} + F_{\text{ext}} + \sqrt{2k_BT\gamma}\eta(t) \tag{4}$$

Where m is the mass of the particle, x is the position of the particle, t is time, γ is the friction coefficient, F_{ext} is an external force (if present), k_B is Boltzmann's constant, T is the temperature, and $\eta(t)$ is a stochastic variable representing the random force, often assumed to be Gaussian white noise with zero mean and unit variance.

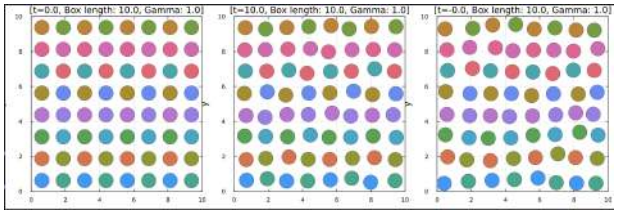
The term $\sqrt{2k_BT\gamma}\eta(t)$ represents the random force in Langevin dynamics. It arises from the fluctuation-dissipation theorem, which relates the friction coefficient to the strength of the random forces experienced by the particle due to its interactions with the surrounding medium at a given temperature. The focus will primarily be on this term when calculating the environmental effects on the initial conditions' random variation. Additionally, the simulation will be run for different values of the friction coefficient to observe how the increase or decrease in the random force affects the molecular dynamics.



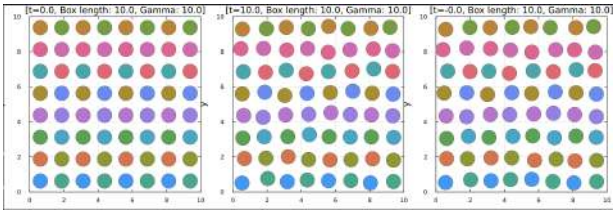
(a) No Langevin Dynamics



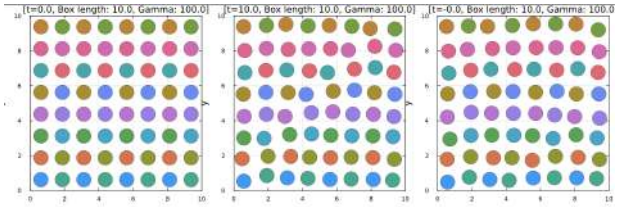
(b) Gamma: 0.1



(c) Gamma: 1

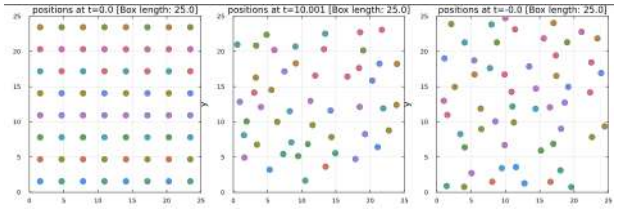


(d) Gamma 10

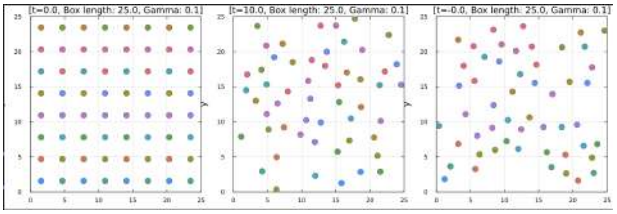


(e) Gamma 100

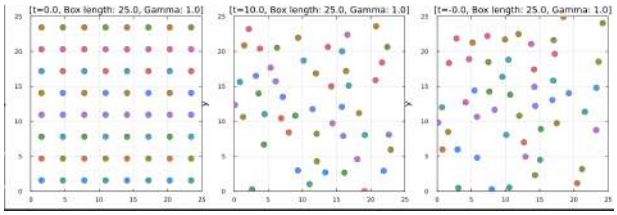
FIG. 11: Time Reversal: Runtime 10, Particle Mass 1, Particle Radius 1, Box Density: 0.64 particles per unit area



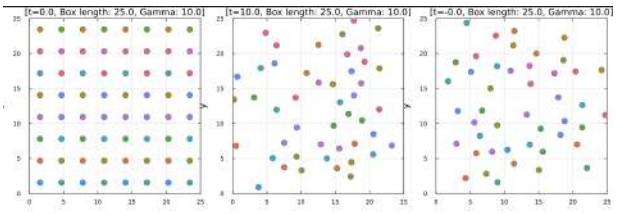
(a) No Langevin Dynamics



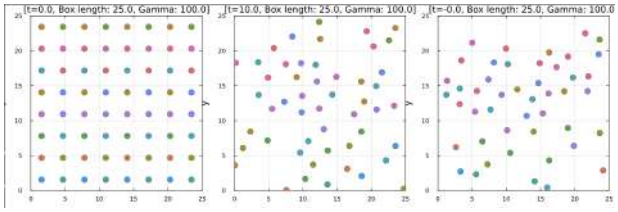
(b) Gamma: 0.1



(c) Gamma: 1

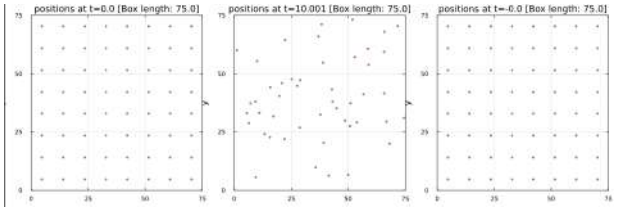


(d) Gamma 10

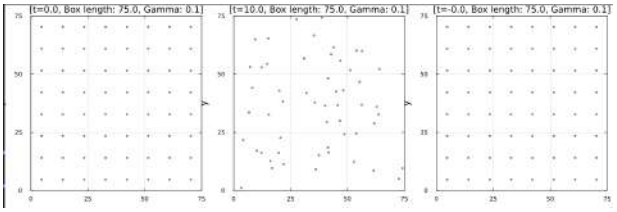


(e) Gamma 100

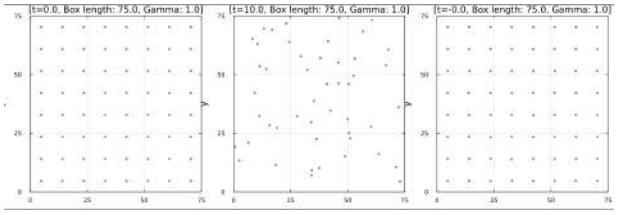
FIG. 12: Time Reversal: Runtime 10, Particle Mass 1, Particle Radius 1, Box Density: 0.1024 particles per unit area



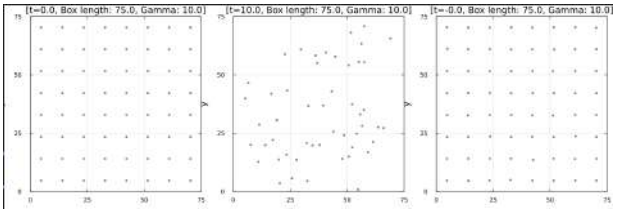
(a) No Langevin Dynamics



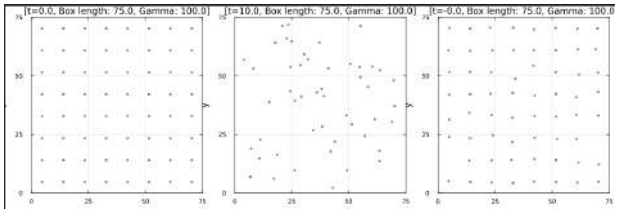
(b) Gamma: 0.1



(c) Gamma: 1



(d) Gamma 10



(e) Gamma 100

FIG. 13: Time Reversal: Runtime 10, Particle Mass 1, Particle Radius 1, Box Density: 0.0114 particles per unit area

The figures 11, 12, and 13 compare the varying friction coefficients for systems of the exact same size, volume, mass, radius, and run times. The comparable system with no Langevin dynamics can also be seen in figures 11a, 12a, and 13a. In all cases, the friction coefficient changed the motion of the particles but does not seem to significantly change the systems' adhesion to temporal asymmetry. Like before, the sparser cases showed the capability to completely time reverse (Figure 13) while the denser system (Figure 11) showed partial time symmetry, and finally, the middle system (Figure 12) displayed no evidence of time reversal.

VI. DISCUSSION

The simulations offer insights into molecular system dynamics, revealing complexities in time reversibility, environmental influences, and system characteristics' impacts. Contrary to expectations, the ideal gas model lacked clear time reversibility, underscoring the challenge of predicting molecular behavior over time. Particle interactions and density significantly influenced the system's ability to return to its initial state, emphasizing environmental factors' importance. Surprisingly, variations in particle characteristics minimally affected time asymmetry, suggesting broader influences dominate molecular dynamics. Incorporating Langevin dynamics elucidated environmental effects' role, demonstrating stochastic forces' contribution to system entropy. Overall, comprehending initial conditions, environmental factors, and system characteristics' interplay is crucial for unraveling molecular dynamics' complexities and the arrow of time.

In discussing whether including environmental interaction resolves or shifts the arrow of time problem, it's essential to consider broader implications. While integrating environmental interactions deepens molecular dynamics understanding, it may not entirely resolve time asymmetry. Instead, it shifts focus to the intricate system-surroundings interplay. Acknowledging the environment's role recognizes the arrow of time emerges from interaction networks rather than solely intrinsic system properties. While insightful, this introduces complexity layers without providing a definitive answer to time asymmetry's origin.

A. Other Work Done in This Area

The question of the arrow of time's origin isn't new, having intrigued scientists since Arthur Eddington's 1927 exposition. Various approaches have been pursued to pinpoint its driving factors. Parrondo et al. (2009) established a link between system entropy production and the distinguishability of processes from their time-reversed counterparts, quantifying this relationship across classical and quantum mechanics through relative entropy analysis [6]. Wei et al. (2018) employed a learning-based approach using videos to probe visual cues indicating time direction across three extensive video datasets [7].

Inspiration for this project stemmed from David Layzer's seminal work, which delved into phenomena such as particle diffusion from containers, phase space concepts, and the evolution of the universe [4]. Layzer also investigated the effects of entropy and random perturbations on systems [4].

Many experimental inquiries into the arrow of time involve quantum computing and mechanics. Lesovik et al. (2019) devised a quantum algorithm demonstrating backward time dynamics for an electron scattered on a two-level impurity using an IBM quantum computer [8].

The studies mentioned represent only a fraction of the extensive research that aims to tackle understanding, reversing, and even eliminating the arrow of time. Despite strides made, much remains veiled about its true origins.

VII. CONCLUSIONS

As stated by Davies, "The physics of time asymmetry has never been a single well-defined subject but rather a collection of consistency problems that arise in almost all branches of physics when confronted with a choice of boundary conditions compatible with the real world" [1]. This paper verifies that altering the initial and environmental conditions of a system have meaningful affects on the system's temporal symmetry. It is also seen that particles' random interactions with each other hold significance in the causation of the *Thermodynamics Arrow of Time*. However, they are unlikely the only element. More work needs to be done to find a complete explanation for time asymmetry.

1. Quantum Mechanics

Moving forward, future work could be done to assess a quantum mechanical system rather than the classical one used in this paper (I would, but as an engineer, quantum mechanics scares me).

In quantum mechanics, the uncertainty principle, formulated by Werner Heisenberg, states that there is a fundamental limit to the precision with which certain pairs of physical properties of a particle, such as position and momentum, can be simultaneously known. This principle implies that even in a system with perfect knowledge of its initial state, there will always be inherent uncertainty in the subsequent evolution of the system [9].

While quantum mechanics itself does not inherently generate entropy due to its unitary evolution, the interaction between quantum systems and their environment leads to the emergence of the irreversible increase in entropy in a process known as decoherence [10]. Decoherence occurs when a quantum system interacts with the external environment, causing the system's quantum coherence, its ability to exist simultaneously in multiple states, to weaken and the system to choose a single classical state [10]. This interaction with the environment introduces irreversibility into the system and leads to the emergence of entropy.

Therefore, while quantum mechanics provides the foundation for understanding initial phase space uncertainty, it is the interaction between quantum systems and their environment, resulting in decoherence and the emergence of classical behavior, that ultimately gives rise to the arrow of time and the increase in entropy [10][1].

---

[1] P. Davies, *The Physics of Times Asymmetry* (University of California Press Berkeley and Lose Angeles, 1974).

[2] H. Prince, *On the origins of the arrow of time: Why there is still a puzzle about the low-entropy past. Contemporary debates in philosophy of science* (2004) pp. 219–239.

[3] O. E. Overseth, Experiments in time reversal, Scientific American 221.4 , 88 (1969).

[4] D. Layzer, *The arrow of time*, Vol. 233(6) (Scientific America, 1975) pp. 56–69.

[5] N. Katz, Understanding the mathematical concepts behind langevin dynamics, https://towardsdatascience.com/langevin-dynamics-29bbb9407b47 (2021).

[6] C. V. d. B. Parrondo, Juan MR and R. Kawai, Entropy production and the arrow of time, New Journal of Physics 11.7 (2009).

[7] A. Z. W. T. F. Donglai Wei, Joseph J. Lim, Learning and using the arrow of time, Proceedings of the IEEE conference on computer vision and pattern recognition (2018).

[8] M. V. S. A. V. L. . V. M. V. G. B. Lesovik, I. A. Sadovskyy, Arrow of time and its reversal on the ibm quantum computer, Scientific Reports 9.1 (2019).

[9] C. Faculty., What is the uncertainty principle and why is it important?, Caltech Science Exchange.

[10] M. Vaughan, The concept of entropic time: A preliminary discussion, (2021).

Appendix A: Appendix

Disclaimer: the code did not format well into overleaf. If needed code can be provided separately.

1. A1: Code for Results Section System Size Reversibility and Control Simulations

using Statistics using Plots using StatsPlots

Define the ParticleSystem structure and functions from the Molecular Dynamics 2 code demos 0,1,3 can optionally make an animated gif

mutable struct ParticleSystem N::Int64 number of particles L::Float64 square box side length T::Float64 initial temperature t::Float64 system time dt::Float64 time step state::VectorFloat64 state space array steps::Int64 number of steps sampleInterval::Int64 interval for sampling data timeData::VectorFloat64 array of sampled time points energyData::VectorFloat64 array of sampled energy values tempData::VectorFloat64 array of sampled temperature values tempAccumulator::Float64 temperature accumulator squareTempAccumulator::Float64 $T^2 accumulator virial Accumulator :: Float64 virial accumulator x Data :: Vector Vector Float64 array of sampled position data v Da$ $Vector Vector Float64 array of sampled velocity data force Type :: String force end$

function ParticleSystem(N::Int64=64, L::Float64=10.0, T::Float64=1.0) t = 0.0 dt = 0.001 state = zeros(4N) state space array, [x1,y1,x2,y2,...,vx1,vy1,...] steps = 0 timeData = Float64[] energyData = Float64[] sampleInterval = 100 tempData = Float64[] tempAccumulator = 0.0 squareTempAccumulator = 0.0 virialAccumulator = 0.0 xData = VectorFloat64[] vData = VectorFloat64[] forceType = "lennardJones"

return ParticleSystem(N, L, T, t, dt, state, steps, sampleInterval, timeData, energyData, tempData, tempAccumulator, squareTempAccumulator, virialAccumulator, xData, vData, forceType) end

some useful "views" of the state array (read the performance tips chapter of the julia manual) @views positions(state) = state[1:Int64(length(state)/2)] @views velocities(state) = state[(Int64(length(state)/2)+1):end] @views xcomponent(vector) = vector[1:2:end] @views ycomponent(vector) = vector[2:2:end] @views particle(n, vector) = [vector[2n-1], vector[2n]]

INITIALIZATION

function $\text{set}_r andom_p ositions!(sys :: ParticleSystem) println("configuration : random") positions(sys.state). = rand(2 * sys.N). * sys.Lcool!(sys) end$

function $\text{set}_s quare_l attice_p ositions!(sys :: ParticleSystem) println("configuration : squarelattice")$

n = Int64(floor(sqrt(sys.N))) num lattice points per axis latticeSpacing = sys.L / n

if sys.N != $n^2 println("æ..your chosen N =$(sys.N) is not a square number") println(" ¯¿ resetting N to $(n^2).") sys.N = n^2 sys.state = zeros(4 * sys.N) end$

for i in 0:(n-1) for j in 0:(n-1) sys.state[2*(i*n+j)+1] = (i + 0.5) * latticeSpacing sys.state[2*(i*n+j)+2] = (j + 0.5) * latticeSpacing end end end

function $\text{add}_p osition_j itter!(sys :: ParticleSystem, jitter :: Float64 = 0.5) println("a weebit of random jitter to particle positions$

for i = 1:length(positions(sys.state)) sys.state[i] += rand() - jitter end end

function $\text{set}_r andom_v elocities!(sys :: ParticleSystem) println("configuration : random")$

velocities(sys.state) .= rand(2*sys.N) .- 0.5 $\text{zero}_t otal_m omentum!(sys) velocities(sys.state).* = sqrt(sys.T/temperature(sys)) e$

function $\text{zero}_t otal_m omentum!(sys :: ParticleSystem) xcomponent(velocities(sys.state)).- = mean(xcomponent(velocities(sy$ $mean(ycomponent(velocities(sys.state))) end$

FORCES / POTENTIALS

function force(sys::ParticleSystem) if sys.forceType == "lennardJones" force, virial = $\text{lennard}_j ones_f orce(sys) else if sys.forceT$ $"powerLaw" force, virial = power_l aw_f orce(sys) end$

sys.virialAccumulator += virial

return force end

the minimum image approximation (periodic boundary conditions) function $\text{minimum}_i mage(xij :: Float64, L ::$ $Float64) if xij > (L/2) xij- = L else if xij < -(L/2) xij+ = L end return xij end$

function $\text{lennard}_j ones_f orce(sys :: ParticleSystem) x = xcomponent(positions(sys.state)) y = ycomponent(positions(sys.state$ $0.0) force = zeros(2 * sys.N)$

Threads.@threads for i = 1:(sys.N-1) for j = (i+1):sys.N dx = $\text{minimum}_i mage(x[i]-x[j], sys.L) dy = minimum_i mage(y[i]-$ $y[j], sys.L)$

r2inv = 1.0 / ($dx^2 + dy^2$) $f = 48.0 * r2inv^7 - 24.0 * r2inv^4 fx = dx * f fy = dy * f$

force[2*i-1] += fx force[2*i] += fy force[2*j-1] -= fx force[2*j] -= fy

virial += fx * dx + fy * dy end end

return force, 0.5 * virial end

function lennard$_jones_potential(sys :: ParticleSystem)x = xcomponent(positions(sys.state))y = ycomponent(positions(sys.s$
0.0

Threads.@threads for i in 1:(sys.N-1) for j in (i+1):sys.N dx = minimum$_image(x[i]-x[j], sys.L)dy = minimum_image(y[i]-$
$y[j], sys.L)$

r2inv = 1.0 / (dx^2 + $dy^2)U+ = r2inv^6 - r2inv^3endendreturn4.0 * Uend$

function power$_law_force(sys :: ParticleSystem)end$

function power$_law_potential(sys :: ParticleSystem)end$

TIME EVOLUTION

function keep$_particles_in_box!(sys :: ParticleSystem)foriin1 : (2*sys.N)ifpositions(sys.state)[i] > sys.Lpositions(sys.state)[i$
$sys.Lelseifpositions(sys.state)[i] < 0.0positions(sys.state)[i]+ = sys.Lendend$

another way of doing this: use the ternary operator for i in 1:(2 * sys.N) positions(sys.state)[i] ¡ 0.0 ? positions(sys.state)[i] positions(sys.state)[i] end end

function verlet$_step!(sys :: ParticleSystem)computeaccelerationatcurrenttimeacceleration = force(sys)$

compute positions at t + dt positions(sys.state) .+= velocities(sys.state) .* sys.dt .+ 0.5 .* acceleration .* (sys.dt)^2

enforce boundary conditions (basically check if any particles left the box and put them back) see function implementation for deets keep$_particles_in_box!(sys)$

compute velocities at t + dt velocities(sys.state) .+= 0.5 * sys.dt .* (acceleration + force(sys)) end

function evolve!(sys::ParticleSystem, runtime::Float64=10.0) numsteps = Int64(abs(runtime/sys.dt) + 1)

print$_evolution_message(runtime, numsteps)$

@time for step in 1:numsteps verlet$_step!(sys)zero_total_momentum!(sys)$

if (step push!(sys.timeData, sys.t) push!(sys.energyData, energy(sys)) push!(sys.xData, positions(sys.state)) push!(sys.vData, velocities(sys.state))

T = temperature(sys) push!(sys.tempData, T) sys.tempAccumulator += T sys.squareTempAccumulator += T^2end

sys.t += sys.dt sys.steps += 1 end println("done.") end

function reverse$_time!(sys)sys.dt* = -1println("reversed!dt =(sys.dt)") end$

function cool!(sys::ParticleSystem, cooltime::Float64=1.0) numsteps = Int64(cooltime/sys.dt) for step in 1:numsteps verlet$_step!(sys)velocities(sys.state).* = (1.0 - sys.dt)endreset_statistics!(sys)end$

MEASUREMENTS function kinetic$_energy(sys :: ParticleSystem)return0.5*sum(velocities(sys.state).*velocities(sys.state)$

function potential$_energy(sys :: ParticleSystem)returnlennard_jones_potential(sys)end$

function temperature(sys::ParticleSystem) return kinetic$_energy(sys)/sys.Nend$

function energy(sys::ParticleSystem) return potential$_energy(sys) + kinetic_energy(sys)end$

STATISTICS function reset$_statistics!(sys :: ParticleSystem)sys.steps = 0sys.tempAccumulator = 0.0sys.squareTempAccu$
$0.0sys.virialAccumulator = 0.0sys.xData = []sys.vData = []end$

function mean$_temperature(sys :: ParticleSystem)returnsys.tempAccumulator/sys.stepsend$

function mean$_square_temperature(sys :: ParticleSystem)returnsys.squareTempAccumulator/sys.stepsend$

function mean$_pressure(sys :: ParticleSystem)factorofhalfbecauseforceiscalculatedtwiceeachstepmeanVirial =$
$0.5 * sys.virialAccumulator/sys.stepsreturn1.0 + 0.5 * meanVirial/(sys.N * mean_temperature(sys))end$

function heat$_capacity(sys :: ParticleSystem)meanTemperature = mean_temperature(sys)meanSquareTemperature =$
$mean_square_temperature(sys)2 = meanSquareTemperature-meanTemperature^2denom = 1.0-2*sys.N/meanTemperature^2r$

function mean$_energy(sys :: ParticleSystem)returnmean(sys.energyData)end$

function std$_energy(sys :: ParticleSystem)returnstd(sys.energyData)end$

MATH / ADDITIONAL FUNCTIONS

function dot(v1::VectorFloat64, v2::VectorFloat64) return sum(v1 .* v2) end

GRAPHS

function initialize$_plot()plot(size = (800, 800), titlefontsize = 12, guidefontsize = 12,)end$

function plot$_positions_t(sys :: ParticleSystem, t :: Int64)initialize_plot()forn = 1 : sys.Nscatter!([sys.xData[t][2n-$
$1]], [sys.xData[t][2n]], markersize = 4.0, markercolor = n, markerstrokewidth = 0.4, grid = true, framestyle =:$
$box, legend = false,)endend$

function animate(sys::ParticleSystem, interval::Int64=1) println("gif...")

scatter!() animation = @animate for t in 1:length(sys.xData) scatter() for n = 1:sys.N scatter!([sys.xData[t][2n-1]], [sys.xData[t][2n]], markersize = 4.0, markercolor = n, markerstrokewidth = 0.4, grid = true, framestyle = :box, legend = false,) end xlims!(0, sys.L) ylims!(0, sys.L) xlabel!("x") ylabel!("y") end every interval

gif(animation, "./animation.gif") println("done.") end

function plot$_positions(sys :: ParticleSystem)initialize_plot()forn = 1 : sys.Nscatter!([xcomponent(positions(sys.state))[n]],$
$4.0, markercolor = n, markerstrokewidth = 0.4, grid = true, framestyle =: box, legend = false,)endxlims!(0, sys.L)ylims!(0, s$
positions at t=$(round(sys.t, digits = 4))")$

end

function plot$_t$rajectories(sys :: ParticleSystem, particles :: VectorInt64 = [1])initialize$_p$lot()forn = 1 : sys.Nscatter!([xcomponent(positions(sys.state))[n]], [ycomponent(positions(sys.state))[n]], markersize = 4.0, markercolor = n, markerstrokewidth = 0.4, grid = true, framestyle =: box, legend = false,)end

for n in collect(particles) xdata = [sys.xData[i][2n-1] for i in 1:length(sys.xData)] ydata = [sys.xData[i][2n] for i in 1:length(sys.xData)]

plot trajectory line for nth particle scatter!(xdata, ydata, color = n, markerstrokewidth = 0, markerstrokecolor = n, markersize = 0.7, markeralpha = 0.5, label = false, widen = false,)

plot initial position for nth particle scatter!([sys.xData[1][2n-1]], [sys.xData[1][2n]], markersize = 4.0, markercolor = n, markerstrokewidth = 0.4, markeralpha = 0.3, label = "pcl. n@t = t", widen = false,)

plot final position for nth particle scatter!([sys.xData[end][2n-1]], [sys.xData[end][2n]], markersize = 4.0, markercolor = n, markerstrokewidth = 0.4, markeralpha = 1.0, label = "pcl n@t = t", widen = false,)endtitle!("positionstrajectoriesat digits=2))") plot!() end

function plot$_t$emperature(sys :: ParticleSystem)initialize$_p$lot()plot!(sys.timeData, sys.tempData, widen = true,)ylims!(mea std(sys.tempData)∗20, mean(sys.tempData)+std(sys.tempData)∗20,)xlabel!("t")ylabel!("T(t)")title!("temperaturevstime") end

function plot$_e$nergy(sys :: ParticleSystem, ylimit :: Float64 = 1.0)initialize$_p$lot()plot!(sys.timeData, sys.energyData, widen true,)ylims!(ylimit ∗ (mean(sys.energyData) − 1), ylimit ∗ (mean(sys.energyData) + 1)mean(sys.energyData) − std(sys.energyData)∗10, mean(sys.energyData)+std(sys.energyData)∗10,)xlabel!("t")ylabel!("E(t)")title!("energyvstime")en

function plot$_s$peed$_d$istribution(sys :: ParticleSystem, numSamples :: Int64 = 5)initialize$_p$lot()

numDataPoints = Int64(length(sys.vData)) interval = Int64(floor(numDataPoints / numSamples))

samples = collect(1:interval:numDataPoints) for s in samples speed = sqrt.(xcomponent(sys.vData[s]).^{2}.∗ycomponent(sys.vDat pdf, label = "t =(round(sys.timeData[s], digits=2))",) end xlabel!("speed") title!("speed distribution") end

CONSOLE PRINT DATA

function print$_h$ello()println("dynamics!")println("numberofthreads : ", Threads.nthreads())end

function print$_b$onjour()println("")end

function print$_s$ystem$_p$arameters(sys :: ParticleSystem)println("parameters : ")println(" =(sys.N) (number of particles)") println("= (sys.L)(sidelengthofsquarebox)")println(" =(sys.dt) (time step)") end

function print$_s$ystem$_d$ata(sys :: ParticleSystem)println("dataattimet =(round(sys.t, digits=4))")

if sys.steps == 0 println(": (temperature(sys))")println(" :(energy(sys))") else println("evolved: (sys.steps)")println(" :(temp println(": (energy(sys))")println("energy :(mean$_e$nergy(sys))")println("energy :(std$_e$nergy(sys))")println("capacity :(heat$_c$a

function print$_e$volution$_m$essage(runtime, numsteps)println("...")end

DEMOS DEMO 0: APPROACH TO EQUILIBRIUM function demo$_0$(; gif = 0)println("0 : APPROACHTOEQUILIBRIU −−")

sys = ParticleSystem(64, 120.0, 1.0) print$_s$ystem$_p$arameters(sys)

set$_s$quare$_l$attice$_p$ositions!(sys)set$_r$andom$_v$elocities!(sys)print$_s$ystem$_d$ata(sys)p1 = plot$_p$ositions(sys)

evolve!(sys, 20.0) print$_s$ystem$_d$ata(sys)

p2 = plot$_t$rajectories(sys, collect(1 : 64))p3 = plot$_e$nergy(sys)p4 = plot$_t$emperature(sys)

make gif if gif == 1 animate(sys, 1) end

plot(p1, p2, p3, p4, layout = grid(2,2, heights=[0.7,0.3]), size = (1280,720)) end

DEMO 1: TIME REVERSAL TEST function demo$_1$(; gif = 0)println("1 : TIMEREVERSALTEST")println("− −−")

sys = ParticleSystem(64, 120.0, 1.0) print$_s$ystem$_p$arameters(sys)

set$_s$quare$_l$attice$_p$ositions!(sys)set$_r$andom$_v$elocities!(sys)print$_s$ystem$_d$ata(sys)p1 = plot$_p$ositions(sys)

evolve!(sys, 50.0) p2 = plot$_t$rajectories(sys, collect(1 : 64))p2 = plot$_p$ositions(sys)

reverse$_t$ime!(sys)evolve!(sys, 50.0)print$_s$ystem$_d$ata(sys)p3 = plot$_t$rajectories(sys, collect(1 : 64))p3 = plot$_p$ositions(sys)

make gif if gif == 1 animate(sys, 4) end

plot(p1, p2, p3, layout = (1,3), size = (1200,400)) end

DEMO 2: SPEED DISTRIBUTION function demo$_2$()println("2 : SPEEDDISTRIBUTION")println(" −−− −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−")

sys = ParticleSystem[]

array for speed distribution plots ps = Plots.PlotPlots.GRBackend[]

array for trajectory plots pt = Plots.PlotPlots.GRBackend[]

initialize three systems with different initial conditions but same KE and PE, evolve, and save plots for i = 1:3 push!(sys, ParticleSystem(64, 120.0, 1.0))

println("i")print$_s$ystem$_p$arameters(sys[i])

set$_s$quare$_l$attice$_p$ositions!(sys[i])add$_p$osition$_j$itter!(sys[i])set$_r$andom$_v$elocities!(sys[i])print$_s$ystem$_d$ata(sys[i])

evolve!(sys[i], 40.0) print$_s$ystem$_d$ata(sys[i])push!(ps, plot$_s$peed$_d$istribution(sys[i], 5))push!(pt, plot$_t$rajectories(sys[i], collect(1 64)))end

plot speed distribution and trajectory plots plot(ps[1], ps[2], ps[3], pt[1], pt[2], pt[3], layout = (2,3), size = (1920,1080)) end

DEMO 3: MELTING TRANSITION function $demo_3(; gif = 0) println("3 : MELTINGTRANSITION") println("- -")$

initialize system of particles on square lattice with zero velocity sys = ParticleSystem(100, 10.0, 5.0) $set_square{}_lattice_positions!(s$ $plot_positions(sys)$

evolve the system and watch them "crystallize" into a triangular lattice formation evolve!(sys, 20.0) $print_system{}_data(sys) p2 = plot_trajectories(sys, collect(1 : 100))$

now, increase the temperature of the system by giving the particles some velocity. evolve the system and plot the trajectories. $set_random{}_velocities!(sys) evolve!(sys, 60.0) print_system{}_data(sys) p3 = plot_trajectories(sys, collect(1 : 100))$

some more plots p4 = $plot_energy(sys, 0.0) p5 = plot_temperature(sys) p6 = plot_speed{}_distribution(sys, 20)$

make gif if gif == 1 animate(sys, 1) end

plot(p1, p2, p3, p4, p5, p6, layout = (2,3), size = (1280,720)) end

DEMO 11: Create a function like demo one but runs for a bunch of different sizes function $demo_11(system_size :: Int; gif = 0) println("11 : TIMEREVERSALTESTforSystemSize system_size") println(" -")$

sys = ParticleSystem($system_size) print_system{}_parameters(sys)$

$set_square{}_lattice_positions!(sys) set_random{}_velocities!(sys) print_system{}_data(sys) p1 = plot_positions(sys)$

evolve!(sys, 50.0) p2 = $plot_trajectories(sys, collect(1 : 64)) p2 = plot_positions(sys)$

$reverse_time!(sys) evolve!(sys, 50.0) print_system{}_data(sys) p3 = plot_trajectories(sys, collect(1 : 64)) p3 = plot_positions(sys)$

make gif if gif == 1 animate(sys, 4) end

plot(p1, p2, p3, layout = (1,3), size = (1200,400)) end

DEMO 12: TIME REVERSAL TEST DEMO 12: TIME REVERSAL TEST function $demo_12(sys :: ParticleSystem; runtime : Float64, gif = 0) println("12 : TIMEREVERSALTESTforSystemSize(sys.N)")$ println("———————————-") $print_system{}_parameters(sys) set_square{}_lattice_positions!(sys) set_random{}_velocities!(sys) print_system{}_data(sys) p1 = plot_positions(sys)$

evolve!(sys, runtime) p2 = $plot_positions(sys) reverse_time!(sys) evolve!(sys, runtime) print_system{}_data(sys) p3 = plot_positions(sys) Displayplots display(p1) display(p2) display(p3) display(plot(p1, p2, p3, layout = (1, 3), size = (1200, 400),)) end$

Function to run molecular dynamics simulation for a given system size function $run_simulation(system_size :: Int) Initializetheparticlesystem sys = ParticleSystem(system_size) Setinitialconditions(e.g., positions, velocities) set_square{}_latt$

Function to analyze results and plot relevant data function $analyze_results(results :: DictInt, TupleVectorFloat64, VectorFloat$ $plot() for(size, data) inresults energies, temperatures = data plot!(plt, 1 : length(energies), energies, label = "Size size")$ end xlabel!(plt, "Time") ylabel!(plt, "Energy") title!(plt, "Energy Evolution for Different System Sizes") display(plt) end

Function to run time reversal tests for different system sizes function $study_time{}_reversibility(min_size :: Int, max_size :: Int, step :: Int) sizes = min_size : step : max_size results = DictInt, TupleVectorFloat64, VectorFloat64()$

for size in sizes println("Running time reversal test for system size $size...") sys = ParticleSystem(size)$

Set initial conditions $set_square{}_lattice_positions!(sys) set_random{}_velocities!(sys)$

Evolve forward evolve!(sys, 50.0)

Reverse time $reverse_time!(sys)$

Evolve backward evolve!(sys, 50.0)

Collect data energies = sys.energyData temperatures = sys.tempData

results[size] = (energies, temperatures) end return results end

Function to analyze time reversal test results and plot relevant data function $analyze_time{}_reversibility{}_results(results :: DictInt, TupleVectorFloat64, VectorFloat64) Createsubplotsforenergyandtemperature plt_energy{}_forward = plot(legend = false, xlabel = "Time", ylabel = "Energy", title = "EnergyEvolution(ForwardTime)") plt_temperature{}_forward = plot(legend = false, xlabel = "Time", ylabel = "Temperature", title = "TemperatureEvolution(ForwardTime)") plt_energy{}_bac$ $plot(legend = false, xlabel = "Time", ylabel = "Energy", title = "EnergyEvolution(BackwardTime)") plt_temperature{}_backwar$ $plot(legend = false, xlabel = "Time", ylabel = "Temperature", title = "TemperatureEvolution(BackwardTime)")$

for (size, data) in results energies, temperatures = data

Forward time evolution plot!($plt_energy{}_forward, 1 : length(energies), energies, label = "Size size")$ plot!($plt_temperature{}_forwa$ $length(temperatures), temperatures, label = "Size size")$

Backward time evolution plot!($plt_energy{}_backward, 1 : length(energies), reverse(energies), label = "Size size")$ plot!($plt_temperature{}_backward, 1 : length(temperatures), reverse(temperatures), label = "Size size")$

end

Add legends to the plots $plt_energy{}_forward = plot!(plt_energy{}_forward, legend = true) plt_temperature{}_forward = plot!(plt_temperature{}_forward, legend = true) plt_energy{}_backward = plot!(plt_energy{}_backward, legend = true) plt_temperature{}_backw$

$plot!(plt_temperature_backward, legend = true) display(plot(plt_energy_forward, plt_energy_backward, layout = (2,1))) display(plot(p$
$(2,1)))$

end

Main code function main() $min_size = 1 max_size = 10 step = 1 Runtime reversal tests results = study_time_reversibility(min_size,$

Run $demo_12 for different system sizes for size in min_size : step : max_size println("Running demo_12 for system size size...")$
sys = ParticleSystem(size) $demo_12(sys, runtime = 5.0, gif = 0) end$

end

Run the main function main()

2. A2: Code for Results Section System Volume Reversibility

using Statistics using StatsPlots

mutable struct ParticleSystem N::Int64 number of particles L::Float64 square box side length T::Float64 initial temperature t::Float64 system time dt::Float64 time step state::VectorFloat64 state space array steps::Int64 number of steps sampleInterval::Int64 interval for sampling data timeData::VectorFloat64 array of sampled time points energyData::VectorFloat64 array of sampled energy values tempData::VectorFloat64 array of sampled temperature values tempAccumulator::Float64 temperature accumulator squareTempAccumulator::Float64 $T^2 accumulator virial Accumulator :: Float64 virial accumulator x Data :: Vector Vector Float64 array of sampled position data v Da$
$Vector Vector Float64 array of sampled velocity data force Type :: String force end$

function ParticleSystem(N::Int64=64, L::Float64=10.0, T::Float64=1.0) t = 0.0 dt = 0.001 state = zeros(4N) state space array, [x1,y1,x2,y2,...,vx1,vy1,...] steps = 0 timeData = Float64[] energyData = Float64[] sampleInterval = 100 tempData = Float64[] tempAccumulator = 0.0 squareTempAccumulator = 0.0 virialAccumulator = 0.0 xData = VectorFloat64[] vData = VectorFloat64[] forceType = "lennardJones"

return ParticleSystem(N, L, T, t, dt, state, steps, sampleInterval, timeData, energyData, tempData, tempAccumulator, squareTempAccumulator, virialAccumulator, xData, vData, forceType) end

some useful "views" of the state array (read the performance tips chapter of the julia manual) @views positions(state) = state[1:Int64(length(state)/2)] @views velocities(state) = state[(Int64(length(state)/2)+1):end] @views xcomponent(vector) = vector[1:2:end] @views ycomponent(vector) = vector[2:2:end] @views particle(n, vector) = [vector[2n-1], vector[2n]]

INITIALIZATION

function $set_random_positions!(sys :: ParticleSystem) println("configuration : random") positions(sys.state). = rand(2 * sys.N). * sys.Lcool!(sys) end$

function $set_square_lattice_positions!(sys :: ParticleSystem) println("configuration : square lattice")$

n = Int64(floor(sqrt(sys.N))) num lattice points per axis latticeSpacing = sys.L / n

if sys.N != $n^2 println("¯\_(ツ)_/¯ .your chosen N =$(sys.N) is not a square number") println("¯_(ツ)_/¯ resetting N to $(n^2).") sys.N = n^2 sys.state = zeros(4 * sys.N) end$

for i in 0:(n-1) for j in 0:(n-1) sys.state[2*(i*n+j)+1] = (i + 0.5) * latticeSpacing sys.state[2*(i*n+j)+2] = (j + 0.5) * latticeSpacing end end end

function $set_triangular_lattice_positions!(sys :: ParticleSystem) end$

function $add_position_jitter!(sys :: ParticleSystem, jitter :: Float64 = 0.5) println("a wee bit of random jitter to particle positions$

for i = 1:length(positions(sys.state)) sys.state[i] += rand() - jitter end end

function $set_random_velocities!(sys :: ParticleSystem) println("configuration : random")$

velocities(sys.state) .= rand(2*sys.N) .- 0.5 $zero_total_momentum!(sys) velocities(sys.state).* = sqrt(sys.T/temperature(sys)) e$

function $zero_total_momentum!(sys :: ParticleSystem) xcomponent(velocities(sys.state)).- = mean(xcomponent(velocities(sy$
$mean(ycomponent(velocities(sys.state))) end$

FORCES / POTENTIALS

function force(sys::ParticleSystem) if sys.forceType == "lennardJones" force, virial = $lennard_jones_force(sys) else if sys.forceT$
$"powerLaw" force, virial = power_law_force(sys) end$

sys.virialAccumulator += virial

return force end

the minimum image approximation (periodic boundary conditions) function $minimum_image(xij :: Float64, L :: Float64) if xij > (L/2) xij- = L else if xij < -(L/2) xij+ = L end return xij end$

function $lennard_jones_force(sys :: ParticleSystem) x = xcomponent(positions(sys.state)) y = ycomponent(positions(sys.state$
$0.0) force = zeros(2 * sys.N)$

Threads.@threads for i = 1:(sys.N-1) for j = (i+1):sys.N dx = $minimum_image(x[i]-x[j], sys.L) dy = minimum_image(y[i]-$
$y[j], sys.L)$

r2inv = 1.0 / $(dx^2 + dy^2) f = 48.0 * r2inv^7 - 24.0 * r2inv^4 fx = dx * f fy = dy * f$

force[2*i-1] += fx force[2*i] += fy force[2*j-1] -= fx force[2*j] -= fy
virial += fx * dx + fy * dy end end
return force, 0.5 * virial end
function lennard$_j ones_p otential(sys :: ParticleSystem)x = xcomponent(positions(sys.state))y = ycomponent(positions(sys.s$
0.0
Threads.@threads for i in 1:(sys.N-1) for j in (i+1):sys.N dx = minimum$_i mage(x[i]-x[j], sys.L)dy = minimum_i mage(y[i]-$
$y[j], sys.L)$
r2inv = 1.0 / (dx$^2 + dy^2)U+ = r2inv^6 - r2inv^3 endendreturn 4.0 * Uend$
function power$_l aw_f orce(sys :: ParticleSystem)end$
function power$_l aw_p otential(sys :: ParticleSystem)end$
TIME EVOLUTION
function keep$_p articles_i n_b ox!(sys :: ParticleSystem)foriin1 : sys.Nifpositions(sys.state)[2*i-1] > sys.Lpositions(sys.state)$
$i - 1]- = sys.Lelseifpositions(sys.state)[2 * i - 1] < 0.0positions(sys.state)[2 * i - 1]+ = sys.Lend$
if positions(sys.state)[2*i] ¿ sys.L positions(sys.state)[2*i] -= sys.L elseif positions(sys.state)[2*i] ¡ 0.0 positions(sys.state)[2*i]
+= sys.L end end end
function verlet$_s tep!(sys :: ParticleSystem, particle_r adius :: Float64, mass)computeaccelerationatcurrenttimeacceleration =$
$force(sys)$
compute positions at t + dt positions(sys.state) .+= velocities(sys.state) .* sys.dt .+ 0.5 .* acceleration .* (sys.dt)^2
handle collisions with the walls handle$_c ollisions!(sys, particle_r adius, mass)Passingmasshere$
compute velocities at t + dt velocities(sys.state) .+= 0.5 * sys.dt .* (acceleration + force(sys)) end
function handle$_c ollisions!(sys :: ParticleSystem, particle_r adius :: Float64, mass :: Float64)x = xcomponent(positions(sys.st$
$ycomponent(positions(sys.state))vx = xcomponent(velocities(sys.state))vy = ycomponent(velocities(sys.state))L =$
$sys.L$
for i in 1:sys.N for j in i+1:sys.N dx = minimum$_i mage(x[i] - x[j], L)dy = minimum_i mage(y[i] - y[j], L)r =$
$sqrt(dx^2 + dy^2)ifr < 2 * particle_r adiusCalculatenormalandtangentialcomponentsofvelocitiesnx = dx/rny =$
$dy/rv1n = vx[i] * nx + vy[i] * nyv1t = -vx[i] * ny + vy[i] * nxv2n = vx[j] * nx + vy[j] * nyv2t = -vx[j] * ny + vy[j] * nx$
Update normal components after collision v1n$_a fter = (v1n * (mass - mass) + 2 * mass * v2n)/(mass +$
$mass)v2n_a fter = (v2n * (mass - mass) + 2 * mass * v1n)/(mass + mass)$
Convert normal and tangential velocities back to x and y components vx[i] = v1n$_a fter * nx - v1t * nyvy[i] =$
$v1n_a fter * ny + v1t * nxvx[j] = v2n_a fter * nx - v2t * nyvy[j] = v2n_a fter * ny + v2t * nxendendendend$
function evolve!(sys::ParticleSystem, particle$_r adius :: Float64, mass :: Float64, runtime :: Float64)numsteps =$
$Int64(abs(runtime/sys.dt) + 1)$
print$_e volution_m essage(runtime, numsteps)$
@time for step in 1:numsteps verlet$_s tep!(sys, particle_r adius, mass)zero_t otal_m omentum!(sys)$
if (step push!(sys.timeData, sys.t) push!(sys.energyData, energy(sys)) push!(sys.xData, positions(sys.state)) push!(sys.vData,
velocities(sys.state))
T = temperature(sys) push!(sys.tempData, T) sys.tempAccumulator += T sys.squareTempAccumulator += T$^2 end$
sys.t += sys.dt sys.steps += 1 end println("done.") end
function reverse$_t ime!(sys)sys.dt* = -1println("reversed!dt =(sys.dt)") end$
function cool!(sys::ParticleSystem, cooltime::Float64=1.0) numsteps = Int64(cooltime/sys.dt) for step in 1:num-
steps verlet$_s tep!(sys)velocities(sys.state).* = (1.0 - sys.dt)endreset_s tatistics!(sys)end$
MEASUREMENTS
function kinetic$_e nergy(sys :: ParticleSystem)return 0.5 * sum(velocities(sys.state). * velocities(sys.state))end$
function potential$_e nergy(sys :: ParticleSystem)returnlennard_j ones_p otential(sys)end$
function temperature(sys::ParticleSystem) return kinetic$_e nergy(sys)/sys.Nend$
function energy(sys::ParticleSystem) return potential$_e nergy(sys) + kinetic_e nergy(sys)end$
STATISTICS
function reset$_s tatistics!(sys :: ParticleSystem)sys.steps = 0sys.tempAccumulator = 0.0sys.squareTempAccumulator =$
$0.0sys.virialAccumulator = 0.0sys.xData = []sys.vData = []end$
function mean$_t emperature(sys :: ParticleSystem)returnsys.tempAccumulator/sys.stepsend$
function mean$_s quare_t emperature(sys :: ParticleSystem)returnsys.squareTempAccumulator/sys.stepsend$
function mean$_p ressure(sys :: ParticleSystem)factorofhalfbecauseforceiscalculatedtwiceeachstepmeanVirial =$
$0.5 * sys.virialAccumulator/sys.stepsreturn 1.0 + 0.5 * meanVirial/(sys.N * mean_t emperature(sys))end$
function heat$_c apacity(sys :: ParticleSystem)meanTemperature = mean_t emperature(sys)meanSquareTemperature =$
$mean_s quare_t emperature(sys)2 = meanSquareTemperature-meanTemperature^2denom = 1.0-2*sys.N/meanTemperature^2r$
function mean$_e nergy(sys :: ParticleSystem)returnmean(sys.energyData)end$
function std$_e nergy(sys :: ParticleSystem)returnstd(sys.energyData)end$
MATH / ADDITIONAL FUNCTIONS

function dot(v1::VectorFloat64, v2::VectorFloat64) return sum(v1 .* v2) end

GRAPHS

function initialize$_plot()plot(size = (800, 800), titlefontsize = 12, guidefontsize = 12,)end$

function plot$_positions_t(sys :: ParticleSystem, t :: Int64)initialize_plot()forn = 1 : sys.Nscatter!([sys.xData[t][2n-1]], [sys.xData[t][2n]], markersize = 4.0, markercolor = n, markerstrokewidth = 0.4, grid = true, framestyle =: box, legend = false,)endend$

function animate(sys::ParticleSystem, interval::Int64=1) println("gif...")

scaling$_factor = 10/sys.L$

scatter!() animation = @animate for t in 1:length(sys.xData) scatter() for n = 1:sys.N scatter!([sys.xData[t][2n-1]], [sys.xData[t][2n]], markersize = 4.0 * scaling$_factor, markercolor = n, markerstrokewidth = 0.4, grid = true, framestyle =: box, legend = false,)endxlims!(0, sys.L)ylims!(0, sys.L)xlabel!("x")ylabel!("y")endeveryinterval$

gif(animation, "./animation.gif") println("done.") end

function plot$_positions(sys :: ParticleSystem)scaling_factor = 40/sys.L$

initialize$_plot()forn = 1 : sys.Nscatter!([xcomponent(positions(sys.state))[n]], [ycomponent(positions(sys.state))[n]], marker$ 4.0$*scaling_factor, markercolor = n, markerstrokewidth = 0.4, grid = true, framestyle =: box, legend = false,)endxlims!(0, sy$ digits=4)) [Box length: $(sys.L)]")end$

function plot$_trajectories(sys :: ParticleSystem, particles :: VectorInt64 = [1])initialize_plot()forn = 1 : sys.Nscatter!([xcomponent(positions(sys.state))[n]], [ycomponent(positions(sys.state))[n]], markersize = 4.0, markercolor = n, markerstrokewidth = 0.4, grid = true, framestyle =: box, legend = false,)end$

for n in collect(particles) xdata = [sys.xData[i][2n-1] for i in 1:length(sys.xData)] ydata = [sys.xData[i][2n] for i in 1:length(sys.xData)]

plot trajectory line for nth particle scatter!(xdata, ydata, color = n, markerstrokewidth = 0, markerstrokecolor = n, markersize = 0.7, markeralpha = 0.5, label = false, widen = false,)

plot initial position for nth particle scatter!([sys.xData[1][2n-1]], [sys.xData[1][2n]], markersize = 4.0, markercolor = n, markerstrokewidth = 0.4, markeralpha = 0.3, label = "pcl. $n@t = t", widen = false,)$

plot final position for nth particle scatter!([sys.xData[end][2n-1]], [sys.xData[end][2n]], markersize = 4.0, markercolor = n, markerstrokewidth = 0.4, markeralpha = 1.0, label = "pcl $n@t = t", widen = false,)endtitle!("positionstrajectoriesat$ digits=2))") plot!() end

function plot$_temperature(sys :: ParticleSystem)initialize_plot()plot!(sys.timeData, sys.tempData, widen = true,)ylims!(mea$ $std(sys.tempData)*20, mean(sys.tempData)+std(sys.tempData)*20,)xlabel!("t")ylabel!("T(t)")title!("temperaturevstime")$ end

function plot$_energy(sys :: ParticleSystem, ylimit :: Float64 = 1.0)initialize_plot()plot!(sys.timeData, sys.energyData, widen$ $true,)ylims!(ylimit * (mean(sys.energyData) - 1), ylimit * (mean(sys.energyData) + 1)mean(sys.energyData) -$ $std(sys.energyData)*10, mean(sys.energyData)+std(sys.energyData)*10,)xlabel!("t")ylabel!("E(t)")title!("energyvstime")e$

function plot$_speed_distribution(sys :: ParticleSystem, numSamples :: Int64 = 5)initialize_plot()$

numDataPoints = Int64(length(sys.vData)) interval = Int64(floor(numDataPoints / numSamples))

samples = collect(1:interval:numDataPoints) for s in samples speed = sqrt.(xcomponent(sys.vData[s]).$^2 .*ycomponent(sys.vDat$ $pdf, label = "t =$(round(sys.timeData[s], digits=2))",) end xlabel!("speed") title!("speed distribution") end

CONSOLE PRINT DATA

function print$_system_parameters(sys :: ParticleSystem)println("parameters : ")println("$ =(sys.N) (number of particles)") println("$= (sys.L)(sidelengthofsquarebox)")println("$ =(sys.dt) (time step)") end

function print$_system_data(sys :: ParticleSystem)println("dataattimet =$(round(sys.t, digits=4))")

if sys.steps == 0 println(": $(temperature(sys))")println("$:(energy(sys))") else println("evolved: $(sys.steps)")println("$:(temp println(": $(energy(sys))")println("energy :$(mean$_energy(sys))")println("energy :$(std$_energy(sys))")println("capacity :$(heat$_ca$

function print$_evolution_message(runtime, numsteps)println("...")end$

DEMOS

DEMO 0: APPROACH TO EQUILIBRIUM function demo$_0(; gif = 0)println("0 : APPROACHTOEQUILIBRIUM")print$ $- -")$

sys = ParticleSystem(64, 120.0, 1.0) print$_system_parameters(sys)$

set$_square_lattice_positions!(sys)set_random_velocities!(sys)print_system_data(sys)p1 = plot_positions(sys)$

evolve!(sys, 20.0) print$_system_data(sys)$

p2 = plot$_trajectories(sys, collect(1 : 64))p3 = plot_energy(sys)p4 = plot_temperature(sys)$

make gif if gif == 1 animate(sys, 1) end

plot(p1, p2, p3, p4, layout = grid(2,2, heights=[0.7,0.3]), size = (1280,720)) end

DEMO 1: TIME REVERSAL TEST function demo$_1(particle_radius :: Float64, mass :: Float64; gif :: Int64 = 0, runtime, sys)println("1 : TIMEREVERSALTEST")println("- -")$

sys = ParticleSystem(64, 120.0, 1.0) print$_system_parameters(sys)$

set$_{s}quare_{l}attice_{p}ositions!(sys)set_{r}andom_{v}elocities!(sys)print_{s}ystem_{d}ata(sys)p1 = plot_{p}ositions(sys)$
evolve!(sys, particle$_{r}adius, mass, runtime)p2 = plot_{p}ositions(sys)$
reverse$_{t}ime!(sys)evolve!(sys, particle_{r}adius, mass, runtime)print_{s}ystem_{d}ata(sys)p3 = plot_{p}ositions(sys)$
if gif == 1 animate(sys, 4) end
display(plot(p1, p2, p3, layout=(1,3), size=(1200,400))) end
DEMO 2: SPEED DISTRIBUTION function demo$_{2}()println("2 : SPEEDDISTRIBUTION")println(" ---$
$------------------------------------")$
sys = ParticleSystem[]
array for speed distribution plots ps = Plots.PlotPlots.GRBackend[]
array for trajectory plots pt = Plots.PlotPlots.GRBackend[]
initialize three systems with different initial conditions but same KE and PE, evolve, and save plots for i = 1:3 push!(sys, ParticleSystem(64, 120.0, 1.0))
println("$i")print_{s}ystem_{p}arameters(sys[i])$
set$_{s}quare_{l}attice_{p}ositions!(sys[i])add_{p}osition_{j}itter!(sys[i])set_{r}andom_{v}elocities!(sys[i])print_{s}ystem_{d}ata(sys[i])$
evolve!(sys[i], 40.0) print$_{s}ystem_{d}ata(sys[i])push!(ps, plot_{s}peed_{d}istribution(sys[i], 5))push!(pt, plot_{t}rajectories(sys[i], collect(1$
$64)))end$
plot speed distribution and trajectory plots plot(ps[1], ps[2], ps[3], pt[1], pt[2], pt[3], layout = (2,3), size = (1920,1080)) end
DEMO 3: MELTING TRANSITION function demo$_{3}(; gif = 0)println("3 : MELTINGTRANSITION")println("-$
$-------------------------------------")$
initialize system of particles on square lattice with zero velocity sys = ParticleSystem(100, 10.0, 5.0) set$_{s}quare_{l}attice_{p}ositions!(s$
$plot_{p}ositions(sys)$
evolve the system and watch them "crystallize" into a triangular lattice formation evolve!(sys, 20.0) print$_{s}ystem_{d}ata(sys)p2 =$
$plot_{t}rajectories(sys, collect(1 : 100))$
now, increase the temperature of the system by giving the particles some velocity. evolve the system and plot the trajectories. set$_{r}andom_{v}elocities!(sys)evolve!(sys, 60.0)print_{s}ystem_{d}ata(sys)p3 = plot_{t}rajectories(sys, collect(1 :$
$100))$
some more plots p4 = plot$_{e}nergy(sys, 0.0)p5 = plot_{t}emperature(sys)p6 = plot_{s}peed_{d}istribution(sys, 20)$
make gif if gif == 1 animate(sys, 1) end
plot(p1, p2, p3, p4, p5, p6, layout = (2,3), size = (1280,720)) end
function main() Define particle radius, mass, and runtime particle$_{r}adius = 10.0mass = 1.0Assumingallparticleshavethesame$
10.0
num$_{p}articles = 64definenumberofparticlesintheboxtemp_{i}nital = 1.0defineinitialtemperature$
box$_{l}engths = [10.0, 15.0, 20.0, 25.0, 50.0, 75.0, 100.0]Arrayofdifferentboxlengths$
Iterate over each box length for box$_{l}engthinbox_{l}engthsprintln("RUNNINGDEMO_{1}FORBOXLENGTH :box_{l}ength")$
Create a ParticleSystem object with the specified box length sys = ParticleSystem(num$_{p}articles, box_{l}ength, temp_{i}nital)$
Calculate box density box$_{v}olume = box_{l}ength^{2}Assumingit'sasquareboxbox_{d}ensity = num_{p}articles*mass/box_{v}olume$
Run demo$_{1}withnewinputrequirementsforparticlecollisionsdemo_{1}(particle_{r}adius, mass, gif = 0, runtime = runtime, sys =$
$sys)$
Print box density println("Box Density: $box_{d}ensityparticlesperunitarea")endend$
main()

3. A3: Code for Results Section Langevin Dynamics

using Statistics using StatsPlots
mutable struct ParticleSystem N::Int64 number of particles L::Float64 square box side length T::Float64 initial temperature t::Float64 system time dt::Float64 time step state::VectorFloat64 state space array steps::Int64 number of steps sampleInterval::Int64 interval for sampling data timeData::VectorFloat64 array of sampled time points energyData::VectorFloat64 array of sampled energy values tempData::VectorFloat64 array of sampled temperature values tempAccumulator::Float64 temperature accumulator squareTempAccumulator::Float64 T$^{2}accumulatorvirialAccumulator :: Float64virialaccumulatorxData :: VectorVectorFloat64arrayofsampledpositiondatavDa$
$VectorVectorFloat64arrayofsampledvelocitydataforceType :: Stringforceend$
function ParticleSystem(N::Int64=64, L::Float64=10.0, T::Float64=1.0) t = 0.0 dt = 0.001 state = zeros(4N) state space array, [x1,y1,x2,y2,...,vx1,vy1,...] steps = 0 timeData = Float64[] energyData = Float64[] sampleInterval = 100 tempData = Float64[] tempAccumulator = 0.0 squareTempAccumulator = 0.0 virialAccumulator = 0.0 xData = VectorFloat64[] vData = VectorFloat64[] forceType = "lennardJones"

return ParticleSystem(N, L, T, t, dt, state, steps, sampleInterval, timeData, energyData, tempData, tempAccumulator, squareTempAccumulator, virialAccumulator, xData, vData, forceType) end

some useful "views" of the state array (read the performance tips chapter of the julia manual) @views positions(state) = state[1:Int64(length(state)/2)] @views velocities(state) = state[(Int64(length(state)/2)+1):end] @views xcomponent(vector) = vector[1:2:end] @views ycomponent(vector) = vector[2:2:end] @views particle(n, vector) = [vector[2n-1], vector[2n]]

INITIALIZATION

function set$_{r}andom_{p}ositions$!$(sys$ :: $ParticleSystem)println("configuration$: $random")positions(sys.state)$. = $rand(2 * sys.N)$. * $sys.Lcool$!$(sys)end$

function set$_{s}quare_{l}attice_{p}ositions$!$(sys$ :: $ParticleSystem)println("configuration$: $squarelattice")$

n = Int64(floor(sqrt(sys.N))) num lattice points per axis latticeSpacing = sys.L / n

if sys.N != n$^2 println($"æ..yourchosen$N$ =(sys.N) is not a square number") println("¯¿ resetting N to (n^2).")$sys.N = n^2 sys.state = zeros(4 * sys.N)end$

for i in 0:(n-1) for j in 0:(n-1) sys.state[2*(i*n+j)+1] = (i + 0.5) * latticeSpacing sys.state[2*(i*n+j)+2] = (j + 0.5) * latticeSpacing end end end

function set$_{t}riangular_{l}attice_{p}ositions$!$(sys$ :: $ParticleSystem)end$

function add$_{p}osition_{j}itter$!$(sys$ :: $ParticleSystem, jitter$ :: $Float64 = 0.5)println("aweebitofrandomjittertoparticlepositions$

for i = 1:length(positions(sys.state)) sys.state[i] += rand() - jitter end end

function set$_{r}andom_{v}elocities$!$(sys$ :: $ParticleSystem)println("configuration$: $random")$

velocities(sys.state) .= rand(2*sys.N) .- 0.5 zero$_{t}otal_{m}omentum$!$(sys)velocities(sys.state).* = sqrt(sys.T/temperature(sys))e$

function zero$_{t}otal_{m}omentum$!$(sys$ :: $ParticleSystem)xcomponent(velocities(sys.state)).- = mean(xcomponent(velocities(sy$ $mean(ycomponent(velocities(sys.state)))end$

FORCES / POTENTIALS

function force(sys::ParticleSystem) if sys.forceType == "lennardJones" force, virial = lennard$_{j}ones_{f}orce(sys)elseifsys.forceT$ $"powerLaw"force, virial = power_{l}aw_{f}orce(sys)end$

sys.virialAccumulator += virial

return force end

the minimum image approximation (periodic boundary conditions) function minimum$_{i}mage(xij$ :: $Float64, L$ :: $Float64)ifxij > (L/2)xij- = Lelseifxij < -(L/2)xij+ = Lendreturnxijend$

function lennard$_{j}ones_{f}orce(sys$ :: $ParticleSystem)x = xcomponent(positions(sys.state))y = ycomponent(positions(sys.state$ $0.0force = zeros(2 * sys.N)$

Threads.@threads for i = 1:(sys.N-1) for j = (i+1):sys.N dx = minimum$_{i}mage(x[i]-x[j], sys.L)dy = minimum_{i}mage(y[i]-$ $y[j], sys.L)$

r2inv = 1.0 / (dx^2 + $dy^2)f = 48.0 * r2inv^7 - 24.0 * r2inv^4 fx = dx * ffy = dy * f$

force[2*i-1] += fx force[2*i] += fy force[2*j-1] -= fx force[2*j] -= fy

virial += fx * dx + fy * dy end end

return force, 0.5 * virial end

function lennard$_{j}ones_{p}otential(sys$ :: $ParticleSystem)x = xcomponent(positions(sys.state))y = ycomponent(positions(sys.s$ 0.0

Threads.@threads for i in 1:(sys.N-1) for j in (i+1):sys.N dx = minimum$_{i}mage(x[i]-x[j], sys.L)dy = minimum_{i}mage(y[i]-$ $y[j], sys.L)$

r2inv = 1.0 / (dx^2 + $dy^2)U+ = r2inv^6 - r2inv^3 endendreturn 4.0 * Uend$

function power$_{l}aw_{f}orce(sys$ :: $ParticleSystem)end$

function power$_{l}aw_{p}otential(sys$ :: $ParticleSystem)end$

TIME EVOLUTION

function langevin$_{r}andom_{f}orce(sys$ :: $ParticleSystem, gamma$ :: $Float64)println("gamma$:gamma, sys.dt: $(sys.dt), sys.T$:(sy intermediate$_{v}alue = 2*abs(sys.dt)*gamma*abs(sys.T)println("Intermediatevalue$:intermediate$_{v}alue")rand_{f}orce = sqrt(2 * abs(sys.dt) * gamma * abs(sys.T)) * randn(2 * sys.N)returnrand_{f}orce$

adjust the friction coefficient gamma to control the strength of the damping effect and the magnitude of the stochastic forces end

function keep$_{p}articles_{i}n_{b}ox$!$(sys$ :: $ParticleSystem)foriin1 : sys.Nifpositions(sys.state)[2*i-1] > sys.Lpositions(sys.state)$ $i - 1]- = sys.Lelseifpositions(sys.state)[2 * i - 1] < 0.0positions(sys.state)[2 * i - 1]+ = sys.Lend$

if positions(sys.state)[2*i] ¿ sys.L positions(sys.state)[2*i] -= sys.L elseif positions(sys.state)[2*i] ¡ 0.0 positions(sys.state)[2*i] += sys.L end end end

function verlet$_{s}tep$!$(sys$ :: $ParticleSystem, particle_{r}adius$ :: $Float64, mass$ :: $Float64, gamma$ :: $Float64)Computeacceleration$ $force(sys)$

Add Langevin random forces rand$_{f}orce = langevin_{r}andom_{f}orce(sys, gamma)acceleration+ = rand_{f}orce/mass$

Compute positions at t + dt positions(sys.state) .+= velocities(sys.state) .* sys.dt .+ 0.5 .* acceleration .* (sys.dt)^2

Handle collisions with the walls handle$_c$ollisions!($sys, particle_radius, mass$)

Compute velocities at t + dt velocities(sys.state) .+= 0.5 * sys.dt .* (acceleration + force(sys)) end

function handle$_c$ollisions!(sys :: $ParticleSystem, particle_radius$ :: $Float64, mass$ :: $Float64$)$x = xcomponent(positions(sys.st$
$ycomponent(positions(sys.state))vx = xcomponent(velocities(sys.state))vy = ycomponent(velocities(sys.state))L =$
$sys.L$

for i in 1:sys.N for j in i+1:sys.N dx = minimum$_i mage(x[i] - x[j], L)dy = minimum_i mage(y[i] - y[j], L)r =$
$sqrt(dx^2 + dy^2)if r < 2 * particle_radiusCalculatenormalandtangentialcomponentsofvelocitiesnx = dx/rny =$
$dy/rv1n = vx[i] * nx + vy[i] * nyv1t = -vx[i] * ny + vy[i] * nxv2n = vx[j] * nx + vy[j] * nyv2t = -vx[j] * ny + vy[j] * nx$

Update normal components after collision v1n$_a fter = (v1n * (mass - mass) + 2 * mass * v2n)/(mass +$
$mass)v2n_a fter = (v2n * (mass - mass) + 2 * mass * v1n)/(mass + mass)$

Convert normal and tangential velocities back to x and y components vx[i] = v1n$_a fter * nx - v1t * nyvy[i] =$
$v1n_a fter * ny + v1t * nxvx[j] = v2n_a fter * nx - v2t * nyvy[j] = v2n_a fter * ny + v2t * nxendendendend$

function evolve!(sys::ParticleSystem, particle$_r adius$ :: $Float64, mass$ :: $Float64, runtime$ :: $Float64, gamma)numsteps =$
$Int64(abs(runtime/sys.dt) + 1)$

print$_e volution_m essage(runtime, numsteps)$

@time for step in 1:numsteps verlet$_s tep!(sys, particle_radius, mass, gamma)zero_t otal_m omentum!(sys)$

if (step push!(sys.timeData, sys.t) push!(sys.energyData, energy(sys)) push!(sys.xData, positions(sys.state)) push!(sys.vData, velocities(sys.state))

T = temperature(sys) push!(sys.tempData, T) sys.tempAccumulator += T sys.squareTempAccumulator += T$^2 end$

sys.t += sys.dt sys.steps += 1 end println("done.") end

function reverse$_t ime!(sys)sys.dt* = -1println("reversed!dt =$(sys.dt)") end

function cool!(sys::ParticleSystem, cooltime::Float64=1.0) numsteps = Int64(cooltime/sys.dt) for step in 1:numsteps verlet$_s tep!(sys)velocities(sys.state).* = (1.0 - sys.dt)endreset_s tatistics!(sys)end$

MEASUREMENTS

function kinetic$_e nergy(sys$ :: $ParticleSystem)return0.5 * sum(velocities(sys.state). * velocities(sys.state))end$

function potential$_e nergy(sys$ :: $ParticleSystem)returnlennard_j ones_p otential(sys)end$

function temperature(sys::ParticleSystem) return kinetic$_e nergy(sys)/sys.Nend$

function energy(sys::ParticleSystem) return potential$_e nergy(sys) + kinetic_e nergy(sys)end$

STATISTICS

function reset$_s tatistics!(sys$ :: $ParticleSystem)sys.steps = 0sys.tempAccumulator = 0.0sys.squareTempAccumulator =$
$0.0sys.virialAccumulator = 0.0sys.xData = []sys.vData = []end$

function mean$_t emperature(sys$ :: $ParticleSystem)returnsys.tempAccumulator/sys.stepsend$

function mean$_s quare_t emperature(sys$ :: $ParticleSystem)returnsys.squareTempAccumulator/sys.stepsend$

function mean$_p ressure(sys$ :: $ParticleSystem)factorofhalfbecauseforceiscalculatedtwiceeachstepmeanVirial =$
$0.5 * sys.virialAccumulator/sys.stepsreturn1.0 + 0.5 * meanVirial/(sys.N * mean_t emperature(sys))end$

function heat$_c apacity(sys$ :: $ParticleSystem)meanTemperature = mean_t emperature(sys)meanSquareTemperature =$
$mean_s quare_t emperature(sys)2 = meanSquareTemperature-meanTemperature^2denom = 1.0-2*sys.N/meanTemperature^2r$

function mean$_e nergy(sys$ :: $ParticleSystem)returnmean(sys.energyData)end$

function std$_e nergy(sys$ :: $ParticleSystem)returnstd(sys.energyData)end$

MATH / ADDITIONAL FUNCTIONS

function dot(v1::VectorFloat64, v2::VectorFloat64) return sum(v1 .* v2) end

GRAPHS

function initialize$_p lot()plot(size = (800, 800), titlefontsize = 12, guidefontsize = 12,)end$

function plot$_p ositions_t(sys$ :: $ParticleSystem, t$ :: $Int64)initialize_p lot()forn = 1 : sys.Nscatter!([sys.xData[t][2n-$
$1]], [sys.xData[t][2n]], markersize = 4.0, markercolor = n, markerstrokewidth = 0.4, grid = true, framestyle =:$
$box, legend = false,)endend$

function animate(sys::ParticleSystem, interval::Int64=1) println("gif...")

scaling$_f actor = 10/sys.L$

scatter!() animation = @animate for t in 1:length(sys.xData) scatter() for n = 1:sys.N scatter!([sys.xData[t][2n-$
1]], [sys.xData[t][2n]], markersize = 4.0 * scaling$_f actor, markercolor = n, markerstrokewidth = 0.4, grid =$
$true, framestyle =: box, legend = false,)endxlims!(0, sys.L)ylims!(0, sys.L)xlabel!("x")ylabel!("y")endeveryinterval$

gif(animation, "./animation.gif") println("done.") end

function plot$_p ositions(sys$ :: $ParticleSystem, gamma)scaling_f actor = 40/sys.L$

initialize$_p lot()forn = 1 : sys.Nscatter!([xcomponent(positions(sys.state))[n]], [ycomponent(positions(sys.state))[n]], marker$
$4.0*scaling_f actor, markercolor = n, markerstrokewidth = 0.4, grid = true, framestyle =: box, legend = false,)endxlims!(0, s$
digits=2)), Box length: $(sys.L), Gamma$:gamma]") end

function plot$_t rajectories(sys :: ParticleSystem, particles :: VectorInt64 = [1]) initialize_p lot() forn = 1 : sys.N scatter!([xcomponent(positions(sys.state))[n]], [ycomponent(positions(sys.state))[n]], markersize = 4.0, markercolor = n, markerstrokewidth = 0.4, grid = true, framestyle =: box, legend = false,) end$

for n in collect(particles) xdata = [sys.xData[i][2n-1] for i in 1:length(sys.xData)] ydata = [sys.xData[i][2n] for i in 1:length(sys.xData)]

plot trajectory line for nth particle scatter!(xdata, ydata, color = n, markerstrokewidth = 0, markerstrokecolor = n, markersize = 0.7, markeralpha = 0.5, label = false, widen = false,)

plot initial position for nth particle scatter!([sys.xData[1][2n-1]], [sys.xData[1][2n]], markersize = 4.0, markercolor = n, markerstrokewidth = 0.4, markeralpha = 0.3, label = "pcl. $n@t = t", widen = false,)$

plot final position for nth particle scatter!([sys.xData[end][2n-1]], [sys.xData[end][2n]], markersize = 4.0, markercolor = n, markerstrokewidth = 0.4, markeralpha = 1.0, label = "pcl $n@t = t", widen = false,) end title!("positionstrajectoriesat$
digits=2))") plot!() end

function plot$_t emperature(sys :: ParticleSystem) initialize_p lot() plot!(sys.timeData, sys.tempData, widen = true,) ylims!(mea$
$std(sys.tempData)*20, mean(sys.tempData)+std(sys.tempData)*20,) xlabel!("t") ylabel!("T(t)") title!("temperaturevstime")$
end

function plot$_e nergy(sys :: ParticleSystem, ylimit :: Float64 = 1.0) initialize_p lot() plot!(sys.timeData, sys.energyData, widen$
$true,) ylims!(ylimit * (mean(sys.energyData) - 1), ylimit * (mean(sys.energyData) + 1) mean(sys.energyData) -$
$std(sys.energyData)*10, mean(sys.energyData)+std(sys.energyData)*10,) xlabel!("t") ylabel!("E(t)") title!("energyvstime") e$

function plot$_s peed_d istribution(sys :: ParticleSystem, numSamples :: Int64 = 5) initialize_p lot()$

numDataPoints = Int64(length(sys.vData)) interval = Int64(floor(numDataPoints / numSamples))

samples = collect(1:interval:numDataPoints) for s in samples speed = sqrt.(xcomponent(sys.vData[s]).$^2.* ycomponent(sys.vDat$
$pdf, label = "t =$(round(sys.timeData[s], digits=2))",) end xlabel!("speed") title!("speed distribution") end

CONSOLE PRINT DATA

function print$_s ystem_p arameters(sys :: ParticleSystem) println("parameters : ") println("$ =(sys.N) (number of particles)") println("= $(sys.L)(sidelengthofsquarebox)") println("$ =(sys.dt) (time step)") end

function print$_s ystem_d ata(sys :: ParticleSystem) println("dataattimet =$(round(sys.t, digits=4))")

if sys.steps == 0 println(": $(temperature(sys))") println("$:(energy(sys))") else println("evolved: $(sys.steps)") println("$:(temp
println(": $(energy(sys))") println("energy :$(mean$_e nergy(sys))") println("energy :$(std$_e nergy(sys))") println("capacity :$(heat$_c a$

function print$_e volution_m essage(runtime, numsteps) println("...") end$

DEMOS

DEMO 0: APPROACH TO EQUILIBRIUM function demo$_0(; gif = 0) println("0 : APPROACHTOEQUILIBRIUM") print$
$- -")$

sys = ParticleSystem(64, 120.0, 1.0) print$_s ystem_p arameters(sys)$

set$_s quare_l attice_p ositions!(sys) set_r andom_v elocities!(sys) print_s ystem_d ata(sys) p1 = plot_p ositions(sys)$

evolve!(sys, 20.0) print$_s ystem_d ata(sys)$

p2 = plot$_t rajectories(sys, collect(1 : 64)) p3 = plot_e nergy(sys) p4 = plot_t emperature(sys)$

make gif if gif == 1 animate(sys, 1) end

plot(p1, p2, p3, p4, layout = grid(2,2, heights=[0.7,0.3]), size = (1280,720)) end

DEMO 1: TIME REVERSAL TEST function demo$_1(particle_r adius :: Float64, mass :: Float64; gif :: Int64 = 0, runtime, sys, gamma) println("1 : TIMEREVERSALTEST") println(" -")$

sys = ParticleSystem(64, 120.0, 1.0) print$_s ystem_p arameters(sys)$

set$_s quare_l attice_p ositions!(sys) set_r andom_v elocities!(sys) print_s ystem_d ata(sys) p1 = plot_p ositions(sys, gamma)$

evolve!(sys, particle$_r adius, mass, runtime, gamma) p2 = plot_p ositions(sys, gamma)$

reverse$_t ime!(sys) evolve!(sys, particle_r adius, mass, runtime, gamma) print_s ystem_d ata(sys) p3 = plot_p ositions(sys, gamma)$

if gif == 1 animate(sys, 4) end

display(plot(p1, p2, p3, layout=(1,3), size=(1200,400))) end

DEMO 2: SPEED DISTRIBUTION function demo$_2() println("2 : SPEEDDISTRIBUTION") println(" -")$

sys = ParticleSystem[]

array for speed distribution plots ps = Plots.PlotPlots.GRBackend[]

array for trajectory plots pt = Plots.PlotPlots.GRBackend[]

initialize three systems with different initial conditions but same KE and PE, evolve, and save plots for i = 1:3 push!(sys, ParticleSystem(64, 120.0, 1.0))

println("$i") print_s ystem_p arameters(sys[i])$

set$_s quare_l attice_p ositions!(sys[i]) add_p osition_j itter!(sys[i]) set_r andom_v elocities!(sys[i]) print_s ystem_d ata(sys[i])$

evolve!(sys[i], 40.0) print$_s ystem_d ata(sys[i]) push!(ps, plot_s peed_d istribution(sys[i], 5)) push!(pt, plot_t rajectories(sys[i], collect(1$
$64))) end$

plot speed distribution and trajectory plots plot(ps[1], ps[2], ps[3], pt[1], pt[2], pt[3], layout = (2,3), size = (1920,1080)) end

DEMO 3: MELTING TRANSITION function $\text{demo}_3(; gif = 0) println("3 : MELTINGTRANSITION") println("-$ $----------")$

initialize system of particles on square lattice with zero velocity sys = ParticleSystem(100, 10.0, 5.0) $\text{set}_square_lattice_positions!(s$ $plot_positions(sys)$

evolve the system and watch them "crystallize" into a triangular lattice formation evolve!(sys, 20.0) $\text{print}_system_data(sys) p2 = plot_trajectories(sys, collect(1:100))$

now, increase the temperature of the system by giving the particles some velocity. evolve the system and plot the trajectories. $\text{set}_random_velocities!(sys) evolve!(sys, 60.0) print_system_data(sys) p3 = plot_trajectories(sys, collect(1:100))$

some more plots p4 = $\text{plot}_energy(sys, 0.0) p5 = plot_temperature(sys) p6 = plot_speed_distribution(sys, 20)$

make gif if gif == 1 animate(sys, 1) end

plot(p1, p2, p3, p4, p5, p6, layout = (2,3), size = (1280,720)) end

function main() Define particle radius, mass, and runtime $\text{particle}_radius = 1.0 mass = 1.0 Assumingallparticleshavethesamem$ $10.0 gamma = [0.1, 1.0, 10.0, 100.0] Ajustthefrictioncoefficienttocontrolthestrengthofthedampingeffectandthemagnitudeofth$

$\text{num}_particles = 64 definenumberofparticlesintheboxtemp_inital = 1.0 defineinitialtemperature(CANTBENEGANDSTIL$

$\text{box}_lengths = [10.0, 15.0, 20.0, 25.0, 50.0, 75.0, 100.0] Arrayofdifferentboxlengths$

Iterate over each box length for $\text{box}_lengthinbox_lengthsforgammaingammaprintln("RUNNINGDEMO_1FORBOXLENGT$

Create a ParticleSystem object with the specified box length sys = $\text{ParticleSystem}(\text{num}_particles, box_length, temp_inital)$

Calculate box density $\text{box}_volume = box_length^2 Assumingit'sasquareboxbox_density = num_particles*mass/box_volume$

Run $\text{demo}_1 withnewinputrequirementsforparticlecollisionsdemo_1(particle_radius, mass, gif = 0, runtime = runtime, sys = sys, gamma = gamma)$

Print box density println("Box Density: $box_densityparticlesperunitarea$") $endendend$

main()