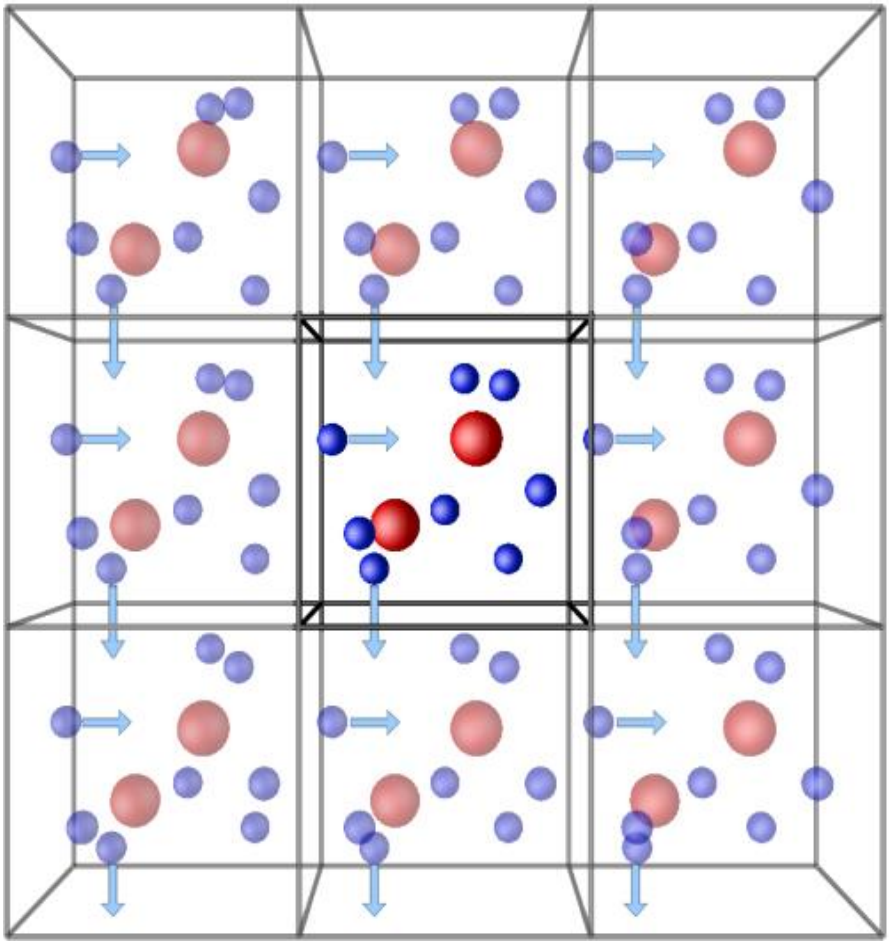
***Name**: Eleftherios Mainas*

***Department**: Chemistry*

***Status**: 1st year PhD Student*

***Course**: Theoretical Physics I: Classical Mechanics*

***Instructor**: Brad Marston*



Outline:

Abstract

In Molecular Dynamics simulations it is computationally impossible to calculate the interactions between 10^{23} particles (the order of Avogadro's number). Modern computers can simulate systems up to the order of 10^6 particles. In order for large systems to be simulated, a small part is selected (the 'unit cell') and periodic boundary conditions are employed. This means that when a particle reaches the edge of the unit cell it reappears from the other side with the same velocity. The problem is that when periodic boundary conditions are employed the total angular momentum of the unit cell is not conserved[1,3]. If one considers the unit cell to be an open system instead of a closed one, then a balance equation instead of conservation equation can be constructed for the angular momentum[2]. In this case, the rate of change of the angular momentum is the balance between the torque exerted from the particles outside the unit cell and the angular momentum flux through the boundaries. The result is that angular momentum oscillates around zero because of the interplay between the two aforementioned factors. In this study, the relationship between the fluctuations of the angular momentum and the physical characteristics of the system (System size, Temperature, Density) was explored. The distribution of the angular momentum was fitted using a Gaussian function and the variance was measured for different system sizes, temperatures and densities. Finally, a possible connection between the measured variances and finite size effects is discussed with the ambition of designing a tool that could potentially be used to predict the impact and the extent of finite size effects on the results of the simulation. I will try to answer the following question: Suppose we have a two dimensional tetragonal shaped unit cell with N interacting particles. How are the fluctuations of angular momentum related with the physical characteristics of the system and how can we exploit those to diagnose finite size system effects?

1. Introduction

It is computationally impossible to simulate systems that the number of particles approaches Avogadro's number and for this reason a smaller infinitely repeated cell is simulated. This is achieved by the use of periodic boundary conditions where particles that reach the surface of the box reappear from the opposite side. Although artificial surface effects are avoided, other spurious phenomena emerge since periodic boundaries impose some kind of symmetry that does not exist in the bulk of real fluids[3]. These phenomena are called finite size effects and sometimes they can lead to non realistic results. For example, when long range forces such as electrostatic forces are included, then it is possible that a particle will feel the force from itself and this is clearly a consequence of the periodic boundaries. The question is how one knows if these effects play an important role in the simulation and for this reason a new approach was followed; In Molecular Dynamics simulations with periodic boundary conditions, it is well known that angular momentum is not conserved[1,2,3] since the system evolves on the surface of a torus. Instead of remaining constant, the projection of the angular momentum to the plane of the simulation fluctuates around zero[2]. The main idea of this project is that the strength of these fluctuations as measured from the variance of a Gaussian distribution could reveal some interesting information about the finite size effects of the system.

2. *Background and Theory*

In the presence of periodic boundary conditions the system under investigation is not isolated and therefore Noerther's theorem does not apply and angular momentum is not conserved. Instead, angular momentum changes with time and a balance equation and not a conservation law can be formulated. This is based on two different factors: angular momentum flux and external torque exerted to the particles from their mirror images. Mathematically this can be shown by the following equation:

$$\frac{d\boldsymbol{L}}{dt} = \mathbf{T} + \mathbf{Q}\,, \quad \mathbf{L}(\mathrm{t}) = \sum \mathbf{r}_\mathrm{i} \times \mathrm{m}\mathbf{v}_\mathrm{i} \qquad (1)$$

The same notation as in reference 3 is used. The interplay between the torque and the angular momentum flux leads to the fluctuation of the angular momentum. If the particles interact with pair forces and we consider a time period [t'-h, t'+h] where a group of particles Λ_+ enter the simulation box and a group of particles Λ_- leave the simulation box, then equation (1) can be rewritten as:

$$\mathbf{L}(t' + h) - \mathbf{L}(t' - h) = \int_{t'-h}^{t'+h} (\mathbf{T} + \mathbf{Q})dt \qquad (2)$$

$$\mathbf{T}(t) = \sum_{i \in \Lambda(t), j \in \Lambda_{im}(t)} \mathbf{r}_i \times \mathbf{F}_{ij} \qquad (3)$$

Where Λ_{im} is the number of image particles at time t. Moreover the flux is given by:

$$\int_{t'-h}^{t'+h} \mathbf{Q}(t)dt = \sum_{i \in \Lambda_+} \mathbf{r}_i \times m\mathbf{v}_i - \sum_{i \in \Lambda_-} \mathbf{r}_i \times m\mathbf{v}_i \qquad (4)$$

Equations (2)-(4) provide the theoretical explanation of the fluctuating behavior of angular momentum.

3. Methods

3.1 Molecular Dynamics Methodology

The system under study consists of N particles that interact through a Lennard Jones potential. The initial configuration is a square lattice and the initial velocities are randomly assigned from a Gaussian distribution with the constraint that the temperature is kept constant. The system evolves in time by integrating the equations of motion using the Verlet integration scheme. Throughout the project the usual reduced units are used where the mass and the Lennard Jones parameters epsilon and sigma are equal to 1.

3.2 Installing Linux in a Windows Machine

It is computationally demanding to calculate the forces between the particles. Therefore, in order to simulate a system of 256 particles in a reasonable time period (e.g. 1 minute) a code acceleration scheme must be used. The performance of the code was enhanced by converting the computationally demanding for loops into C++ code and compiling them using a C++ compiler. The computer that was used throughout the project works with a Windows 10 system that does not have a C++ compiler. For this reason, the option “Windows Subsystem for Linux” was used and Ubuntu was installed together with the g++ compiler.

3.3 Using “weave” for the Acceleration of “for” Loops

In order to include C++ code inside the Python script, the tool “weave” was used [See Appendix]. Following this methodology, the “for” loops that were used for the

calculation of the interactions between the particles were compiled by the g++ compiler and a substantial acceleration was achieved. More specifically, the Python module "time" and the code was modified so that time was calculated for each run [See Appendix]. For 256 particles the performance enhancement was approximately 30-fold going from 30 minutes to 1 minute running time.

3.4 Angular Momentum and Angular Momentum Fluctuations calculation

The projection of the angular momentum to the plane of the simulation was measured by using the definition[4]:

$$L = \sum_{i=1}^{i=N} \left(x_i * v_{yi} - y_i * v_{xi} \right)$$

Where, x_i and y_i are the x and y coordinates of particle i and v_{yi} and v_{xi} are the velocity components. I calculated the angular momentum from different point (e.g. center of the box) but the variance is exactly the same. The resulting behavior was a fluctuating function around the value of zero (See figure 1) so in order to obtain a measure for the fluctuations a histogram was constructed using 100 bins and a proper normalization constant. Furthermore a Gaussian probability distribution function was fit (See figure 1, orange curve) and the values for the mean and the variance were calculated. Based on the central limit theorem, an infinitely long simulation would give a perfect Gaussian function as a result since none of the two directions are preferred for the angular momentum. This idea would break down if a magnetic field was present or if the system was rotating (presence of Coriolis force) and the resulting distribution would be asymmetric with respect to the mean. The variance of the resulting Gaussian distribution is considered to be a direct measure of the fluctuations of the projection of the angular momentum and was used as such throughout this study.

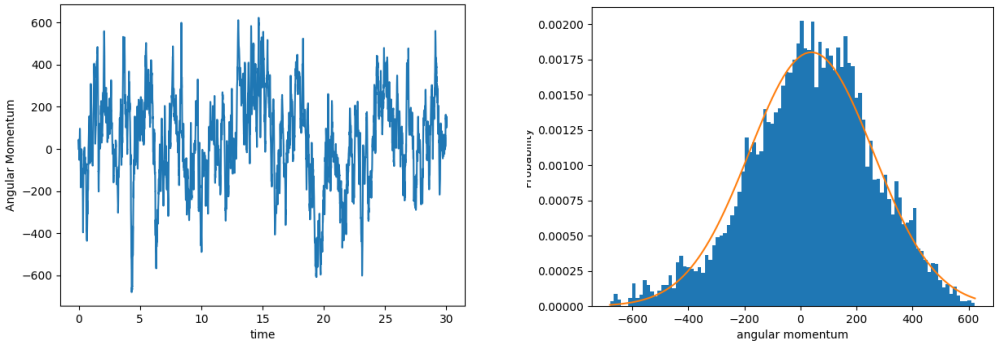


Figure 1. Example of the oscillatory behavior of angular momentum and the corresponding distribution on the right. The orange curve represents the Gaussian fit.

4. Results

4.1 Angular Momentum Fluctuations and System Size

Keeping the density constant (ρ = 1 particle/L^2), the fluctuations of the angular momentum were measured for different system sizes. Initially, the number of particles is 16 and the length of the side of the tetragonal unit cell is 4. Different simulations were set up by increasing the size of the side of the square by the number 2 every time and the number of particles was increased so that the density remains fixed. The results can be seen in table 1.

Simulation	***N***	***L^2***	***Variance1***	***Variance2***	***Error***
1	16	16	5.0620	4.5959	9.21%
2	36	36	10.541	10.7561	2.04%
3	64	64	21.718	24.0655	10.81%
4	100	100	40.660	35.8177	11.91%
5	144	144	51.152	63.5907	24.32%
6	196	196	74.629	82.854	11.02%
7	256	256	104.78	97.6628	6.79%
8	324	324	148.68	145.685	2.01%
9	400	400	160.13	149.148	6.86%
10	484	484	211.76	163.4284	22.82%

Table 1. For different system sizes the variance was measured. Two random seeds were used resulting to two different values for the variance.

The particle trajectories, for each one of the 10 simulations can be found in the Supplementary Material. The variance for each set of simulations seems to depend linearly on the size of the system (Area of the simulation box).

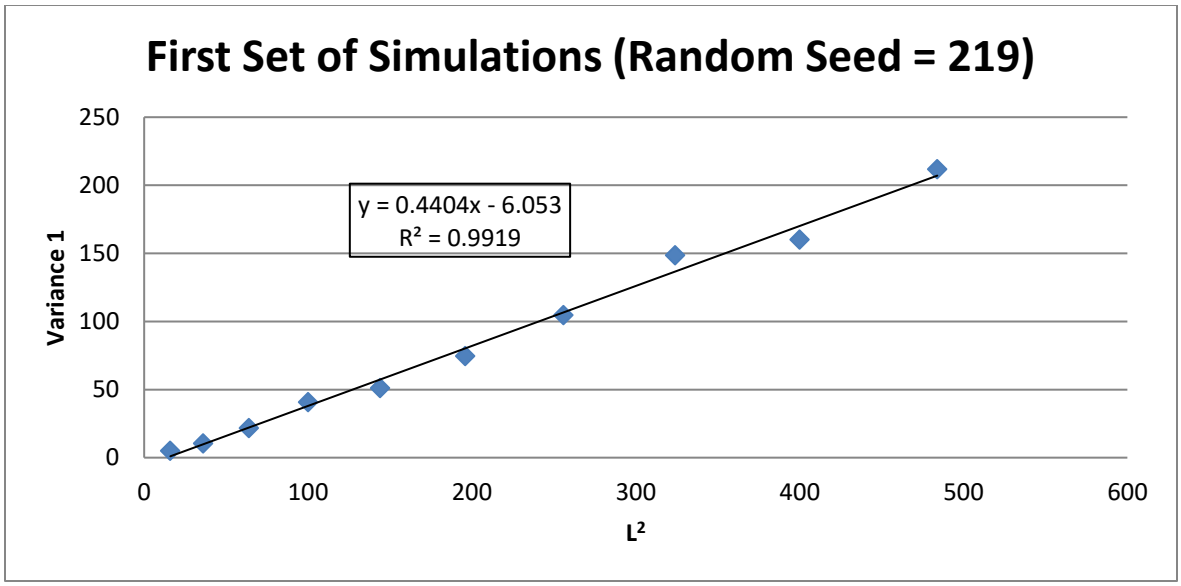


Figure 2. Linear relation between the variance and the size of the system for the first set of simulations.

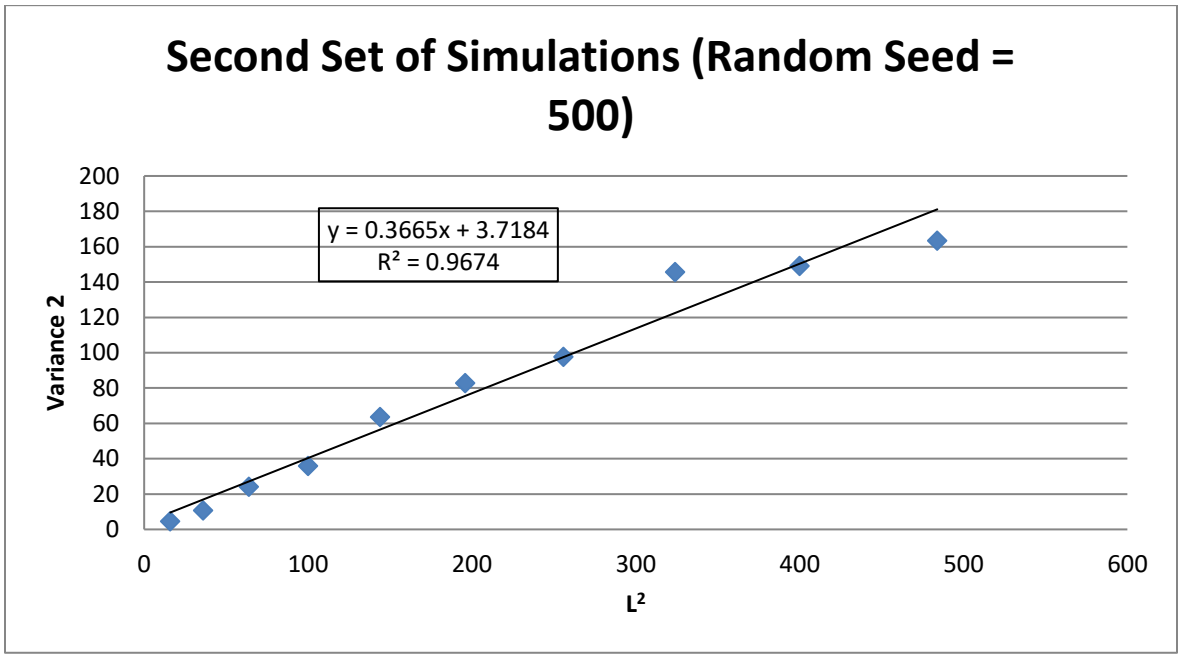


Figure 3. Linear relation between the variance and the size of the system for the second set of simulations

4.2 Angular Momentum Fluctuations and Temperature

Next, the angular momentum fluctuations of the system were measured varying the temperature and keeping the size and the density constant. For N=256 and L=16 values for the mean and the variance of the distribution were obtained changing the temperature from 0.001 to 10. Results are shown in table 2.

Simulation	***Initial Temperature***	***Variance 1***	***Variance 2***	***Error***
1	0.01	75.524	76.938	1.87%
2	0.05	110.70	92.710	16.3%
3	0.1	77.510	114.00	47.1%
4	0.5	86.181	92.550	7.39%
5	1	104.78	97.663	6.79%
6	2	145.12	143.00	1.46%
7	3	140.78	158.69	12.7%
8	4	157.48	180.92	14.9%
9	5	172.21	177.86	3.3%
10	6	171.97	194.12	12.9%
11	7	184.06	220.63	19.9%
12	8	218.16	220.83	1.22%
13	9	245.32	261.20	6.47%
14	10	221.35	251.35	13.5%

Table 2. For different initial temperatures the variance was measured. Two random seeds were used resulting to two different variances. The error is estimated between these two values.

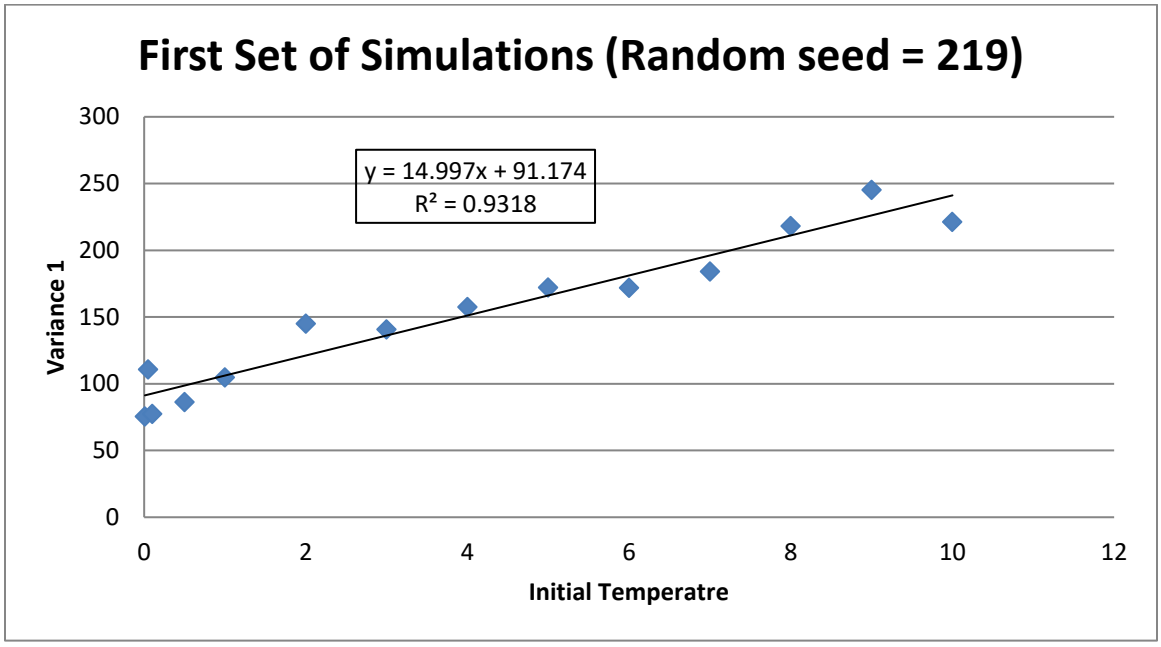


Figure 4. Variance as a function of the initial temperature for the first set of simulations

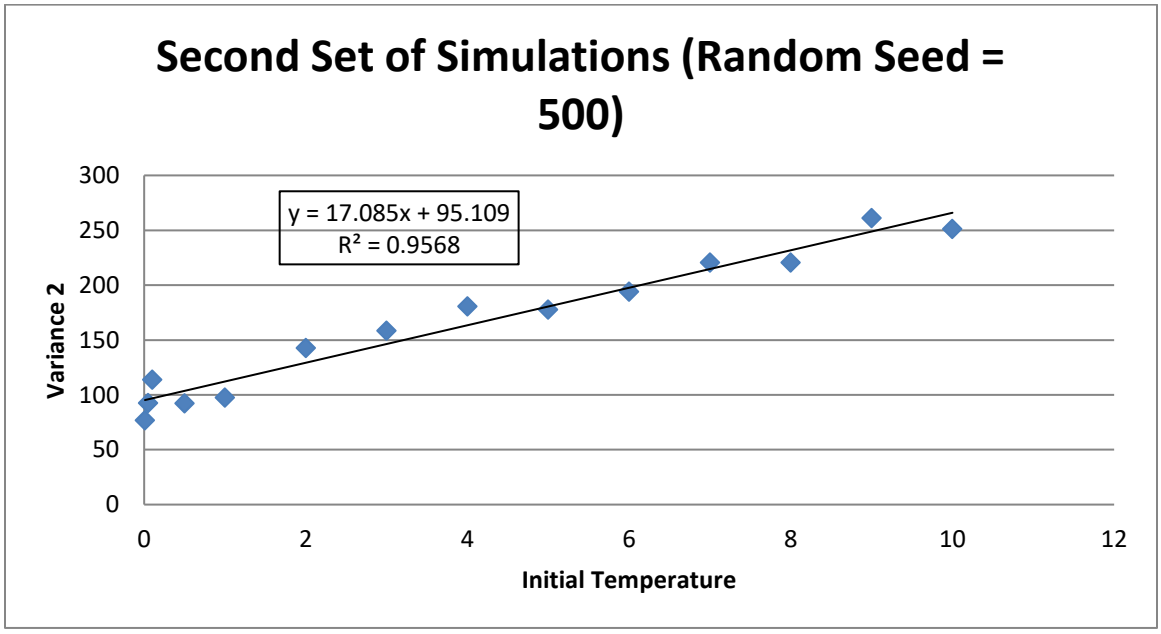


Figure 5. Variance as a function of the initial temperature for the second set of simulations

The particle trajectories, for the first set of simulations can be found in the Supplementary Material.

4.3 Angular Momentum Fluctuations and Density

For the next set of simulations the initial temperature was kept constant and equal to 1 and the number of particles was 256. To adjust the density of the system, different values for the size of the box were selected. The results can be seen in table 3.

Simulation	***L***	***Density***	***Variance 1***	***Variance 2***	***Error***
1	15.1	1.12276	116.088	112.796	2.84%
2	15.2	1.10803	116.039	113.030	2.59%
3	15.3	1.09360	115.687	126.300	9.17%
4	15.4	1.07944	129.749	101.838	21.5%
5	15.5	1.06556	104.100	124.054	19.2%
6	15.6	1.05194	135.644	115.783	14.6%
7	15.7	1.03858	100.077	122.417	22.3%
8	15.8	1.02548	132.248	111.094	16.0%
9	15.9	1.01262	116.563	98.7210	15.3%
10	16	1	104.782	97.6630	6.79%
11	16.1	0.987616	109.812	98.6994	10.1%
12	16.2	0.975461	86.953	94.8160	9.04%
13	16.3	0.963529	105.453	95.5480	9.39%
14	16.4	0.951814	111.258	97.2580	12.6%
15	16.5	0.940312	105.388	114.846	8.97%
16	16.6	0.929017	96.31	99.8890	3.71%
17	16.7	0.917925	106.774	89.3340	16.3%
18	16.8	0.907029	109.139	86.4280	20.8%
19	16.9	0.896327	84.23	110.829	31.6%

Table 3. For different system densities the mean and the variance were measured.

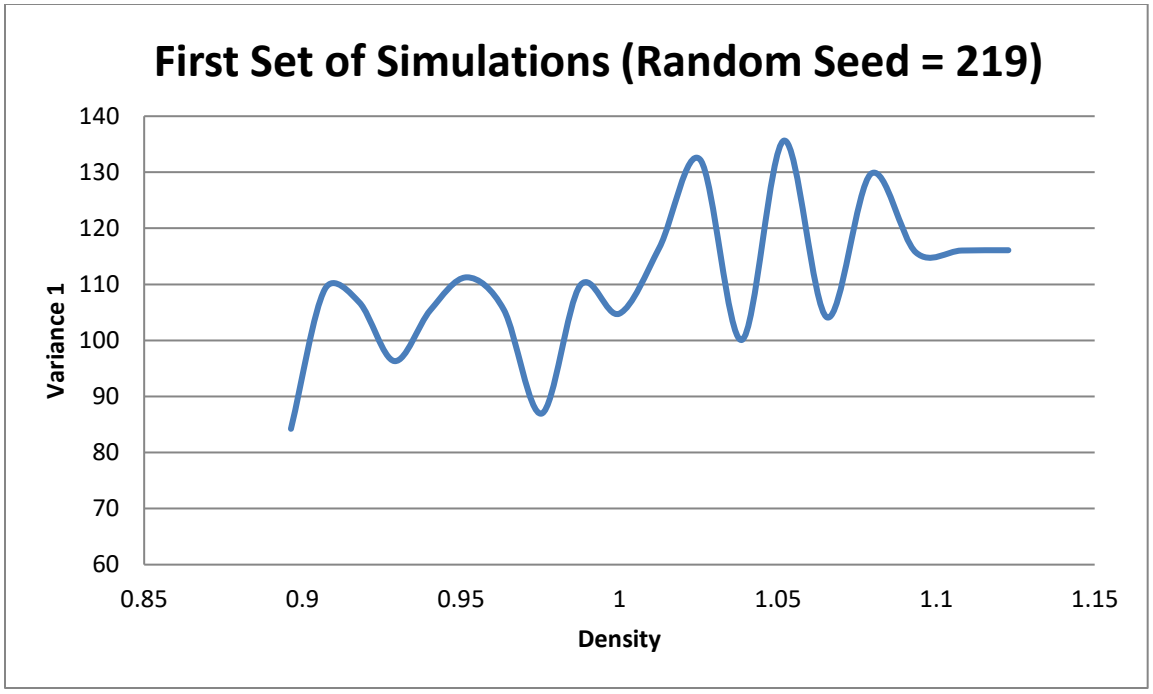


Figure 6. Fluctuating behavior of the variance as a function of the density of the system for the first set of simulations

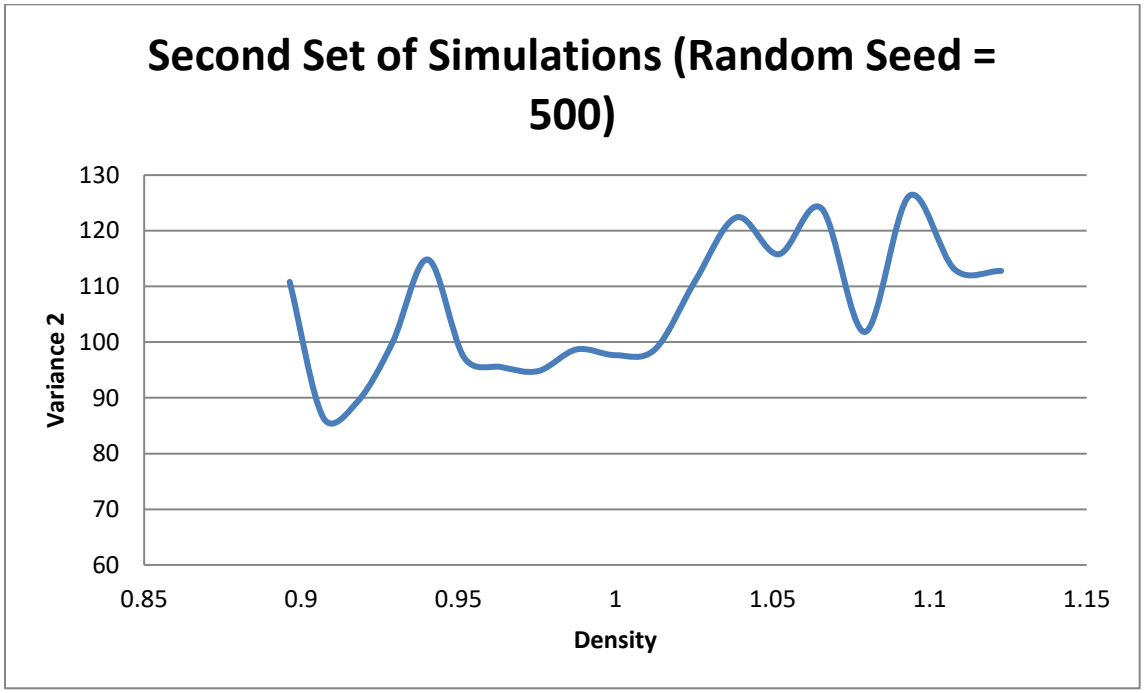


Figure 7. Fluctuating behavior of the variance as a function of the density for the second set of simulations

The particle trajectories, for each one of the 9 simulations can be found in the Supplementary Material.

5. Discussion

5.1 System Size and Variance

The aim of this study was to investigate possible connections between the non conservative behavior of angular momentum and the physical characteristics of a two dimensional Lennard Jones fluid in molecular dynamics simulations. The oscillatory behavior of the projection of the angular momentum to the plane of the simulation was reproduced successfully as expected by intuition and matches the resulting behavior obtained in reference 2. From the first part of the results it can be seen that as the system gets larger the fluctuations get larger but there is no obvious way that this is related to the trajectories of the particles. For example, when going from Simulation 5 to Simulation 6 there is a clear difference, since in the latter one there is a group of particles on the top right that have diffused away from their initial positions giving this blurry spot. Although, the system is technically the same in some of the simulations this diffusive behavior was observed and in these simulations the variance increment is larger. This can be observed when going from Simulation 5 → 6, 6 →7, 7→8 and 9→10. These increments in the variance of the distribution reflect the higher diffusivity of certain groups of molecules and it is not observed for smaller systems. There is a possibility that this is caused by the assignment of a very high initial velocity but there is no reason why this would not happen in the smaller systems (Simulations 1 to 5). Since variations in the angular momentum are sensitive to the change of the diffusivity of particles then it would be reasonable to think a connection with phase transitions. As the system approaches a more “mobile” state (e.g. from liquid to vapor or from solid to liquid) fluctuations in the angular momentum become larger. For Simulation 11 to 16 there exists some hesitation when it comes to drawing conclusions because the distributions seem to not follow the Gaussian behavior and larger simulation times are needed.

5.2 Temperature and Variance

When it comes to the connection between variance and temperature it is clear that as the system has a higher average temperature the particles move more rapidly and this is reflected to larger angular momentum fluctuations.

5.3 Density and Variance

Surprisingly, the fluctuations seemed not to depend on the density of the system. Intuition suggested that as the system gets less dense and these diffuse spots to appear as in the first set of simulations, the variance would be larger. Apparently, variance changes only a little and it’s even more surprising that for very dilute systems where particles can practically move almost everywhere in the simulation cell, the fluctuations in the angular momentum are not as large as expected.

6. Conclusions

From this work a number of conclusions can be drawn about the relation between angular momentum fluctuations and the characteristics of the system. The ultimate goal of this study is to relate the variance of the angular momentum distribution to finite size effects. Apart from the aforementioned connection to phase transitions other effects could be quantified. For example, the diffusion coefficient of a particle in a periodic fluid needs to be corrected[3] due to the hydrodynamic flow fields of all the periodic images of the particle that decay as 1/r. This effect is similar mathematically to the Coulomb case where the range is very large compared to Lennard Jones interactions and spurious correlations may rise from interactions between a charged particle and its periodic images. Moreover, when a particle moves in a fluid its momentum is transferred to the rest of the fluid as sound and over damped shear waves through particle collisions. After these waves travel distance that is comparable to the size of the box, the particle will interact with itself giving rise to artificial correlations. It is not clear how these effects can be related to the variances measured and probably more specific simulations should be made. The next step would be to compare the results with different diffusion coefficients and see if there are connections. Moreover a particle could be tagged and carefully track its trajectory for a very long time to see if the hydrodynamics interactions with itself are related with the variations to its angular momentum. Finally I would like to suggest a few ways how this project can be improved. Firstly, more simulations with different seeds must be run so that the error is minimized. This can also be done by increasing the time of the simulation, letting this way the distribution of the angular momentum to approach the Gaussian limit. The next step would be to investigate further the mathematical relation between the variance and the physical characteristics of the system. From a first simple approach it can be deduced that the variance changes linearly with the size of the system and the initial temperature but seems to fluctuate around a constant value as a function of the density of the system.

7. References

1) Hoover Wm. G. *Lecture Notes in Physics: Molecular Dynamic.* Berlin: Springer-Verlag, 1986

2) Kuzkin, V.A. "On Angular Momentum Balance for Particle Systems with Periodic Boundary Conditions." *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik* 95, no. 11 (2015): 1290-95.

3) Frenkel, D. "Simulations: The Dark Side." *The European Physical Journal Plus* 128, no. 1 (January 25 2013): 10.

4) Fetter, Alexander L., and John Dirk Walecka. *Theoretical Mechanics of Particles and Continua.* New York: McGraw-Hill, 1980.

8. Appendix-Code used throughout the project (Brad Marston's code taken from canvas):

```
#!/usr/bin/env python

import numpy
import time
from numpy import *
from pylab import *  # plotting library
from scipy.stats import norm
import matplotlib.mlab as mlab

import weave
from weave import converters

class MolecularDynamics:

    """Class that describes the molecular dynamics of a gas of atoms in units where m = epsilon = sigma = kb = 1"""

    dt = 0.001 # time increment
    sampleInterval = 100

    def __init__(self, N=4, L=10.0, initialTemperature=0.0, initialAngularMomentum=0.0):

        numpy.random.seed(219) # random number generator used for initial velocities (and sometimes positions) seed was 219

        self.N = N  # number of particles
        self.L = L  # length of square side
        self.initialTemperature = initialTemperature
        self.initialAngularMomentum = initialAngularMomentum

        self.t = 0.0 # initial time
        self.tArray = array([self.t]) # array of time steps that is added to during integration
        self.steps = 0

        self.EnergyArray = array([]) # list of energy, sampled every sampleInterval time steps
        self.sampleTimeArray = array([])
        self.angularMomentumArray = array([])

        # accumulate statistics during time evolution
        self.temperatureArray = array([self.initialTemperature])
        self.temperatureAccumulator = 0.0
        self.angularMomentumArray = array([self.initialAngularMomentum])
        self.angularMomentumAccumulator = 0.0
        self.squareTemperatureAccumulator = 0.0
        self.virialAccumulator = 0.0

        self.x = zeros(2*N) # NumPy array of N (x, y) positions
        self.v = zeros(2*N) # array of N (vx, vy) velocities

        self.xArray = array([]) # particle positions that is added to during integration
        self.vArray = array([]) # particle velocities
```

```
    self.forceType = "weavelennardJones"


def minimumImage(self, x): # minimum image approximation (Gould Listing 8.2)

    L = self.L
    halfL = 0.5 * L

    return (x + halfL) % L - halfL


def force(self):

    if (self.forceType == "weavelennardJones"):
        #f, virial = self.lennardJonesForce()
        f, virial = self.weaveLennardJonesForce()

    if (self.forceType == "weavepowerLaw"):
        #f, virial = self.powerLawForce()
        f, virial = self.weavePowerLawForce()

    self.virialAccumulator += virial

    return f


def lennardJonesForce(self): # Gould Eq. 8.3 (NumPy vector form which is faster)

    N = self.N
    virial = 0.0
    tiny = 1.0e-40 # prevents division by zero in calculation of self-force
    L = self.L
    halfL = 0.5 * L

    x = self.x[arange(0, 2*N, 2)]
    y = self.x[arange(1, 2*N, 2)]
    f = zeros(2*N)

    minimumImage = self.minimumImage

    for i in range(N):  # The O(N**2) calculation that slows everything down

        dx = minimumImage(x[i] - x)
        dy = minimumImage(y[i] - y)

        r2inv = 1.0/(dx**2 + dy**2 + tiny)
        c = 48.0 * r2inv**7 - 24.0 * r2inv**4
        fx = dx * c
        fy = dy * c

        fx[i] = 0.0 # no self force
        fy[i] = 0.0
        f[2*i] = fx.sum()
        f[2*i+1] = fy.sum()
```

```
            virial += dot(fx, dx) + dot(fy, dy)

        return f, 0.5*virial


    def weaveLennardJonesForce(self): # Gould Eq. 8.3

        N = self.N
        L = self.L
        halfL = 0.5 * L

        x = self.x[arange(0, 2*N, 2)]
        y = self.x[arange(1, 2*N, 2)]

        f = zeros(2*N)
        virial = zeros(1)

        code = """
               double dx, dy, r2inv, r6inv, r8inv, c, fx, fy;

               for (int i = 0; i < N; i++) {
                  for (int j = i+1; j < N; j++) {

                     dx = x(i) - x(j);
                     if (dx > halfL) dx = dx - L;
                     if (dx < -halfL) dx = dx + L;

                     dy = y(i) - y(j);
                     if (dy > halfL) dy = dy - L;
                     if (dy < -halfL) dy = dy + L;

                     r2inv = 1.0 / (dx*dx + dy*dy);
                     r6inv = r2inv*r2inv*r2inv;
                     r8inv = r2inv*r6inv;
                     c = 48.0 * r8inv*r6inv - 24.0 * r8inv;
                     fx = dx * c;
                     fy = dy * c;

                     f(2*i) += fx;
                     f(2*i+1) += fy;
                     f(2*j) -= fx; // Newton's 3rd law
                     f(2*j+1) -= fy;

                     virial(0) += fx*dx + fy*dy; // for virial accumulator (calculation of pressure)

                  }
               }
               """

        weave.inline(code, ['N', 'x', 'y', 'L', 'halfL', 'f', 'virial'], type_converters=converters.blitz, compiler='gcc')

        return f, virial[0]



    def powerLawForce(self):
```

```
    N = self.N
    virial = 0.0
    tiny = 1.0e-40 # prevents division by zero in calculation of self-force
    halfL = 0.5 * self.L

    x = self.x[arange(0, 2*N, 2)]
    y = self.x[arange(1, 2*N, 2)]
    f = zeros(2*N)
    minimumImage = self.minimumImage
    for i in range(N):  # The O(N**2) calculation that slows everything down

        dx = minimumImage(x[i] - x)
        dy = minimumImage(y[i] - y)

        r2 = dx**2 + dy**2 + tiny
        r6inv = pow(r2, -3)
        fx = dx * r6inv
        fy = dy * r6inv

        fx[i] = 0.0 # no self force
        fy[i] = 0.0
        f[2*i] = fx.sum()
        f[2*i+1] = fy.sum()

        virial += dot(fx, dx) + dot(fy, dy)

    return f, 0.5 * virial


def weavePowerLawForce(self): # Gould Eq. 8.3

    N = self.N
    L = self.L
    halfL = 0.5 * L

    x = self.x[arange(0, 2*N, 2)]
    y = self.x[arange(1, 2*N, 2)]

    f = zeros(2*N)
    virial = zeros(1)

    code = """
        double dx, dy, r2inv, r6inv, r8inv, c, fx, fy;

        for (int i = 0; i < N; i++) {
           for (int j = i+1; j < N; j++) {

              dx = x(i) - x(j);
              if (dx > halfL) dx = dx - L;
              if (dx < -halfL) dx = dx + L;

              dy = y(i) - y(j);
              if (dy > halfL) dy = dy - L;
              if (dy < -halfL) dy = dy + L;
```

```
                r2inv = 1.0 / (dx*dx + dy*dy);
                r6inv = r2inv*r2inv*r2inv;
                fx = dx * r6inv;
                fy = dy * r6inv;

                f(2*i) += fx;
                f(2*i+1) += fy;
                f(2*j) -= fx; // Newton's 3rd law
                f(2*j+1) -= fy;

                virial(0) += fx*dx + fy*dy; // for virial accumulator (calculation of pressure)

              }
            }
            """

    weave.inline(code, ['N', 'x', 'y', 'L', 'halfL', 'f', 'virial'], type_converters=converters.blitz, compiler='gcc')

    return f, virial[0]


# TIME EVOLUTION METHODS

  def verletStep(self): # Gould Eqs. 8.4a and 8.4b

    a = self.force()
    self.x += self.v * self.dt + 0.5 * self.dt**2 * a
    self.x = self.x % self.L    # periodic boundary conditions
    self.v += 0.5 * self.dt * (a + self.force())


  def evolve(self, time=10.0):

    steps = int(abs(time/self.dt))
    for i in range(steps):

      self.verletStep()
      self.zeroTotalMomentum()

      self.t += self.dt
      self.tArray = append(self.tArray, self.t)

      if (i % self.sampleInterval == 0): # only calculate energy every sampleInterval steps to reduce load
        self.EnergyArray = append(self.EnergyArray, self.energy())
        self.sampleTimeArray = append(self.sampleTimeArray, self.t)
        self.xArray = append(self.xArray, self.x)
        self.vArray = append(self.vArray, self.v)

      T = self.temperature()
      self.steps += 1
      self.temperatureArray = append(self.temperatureArray, T)
      self.temperatureAccumulator += T
      self.squareTemperatureAccumulator += T*T

      L = self.angularMomentum()
      self.steps += 1
```

```
            self.angularMomentumArray = append(self.angularMomentumArray, L)
            self.angularMomentumAccumulator += L


    def zeroTotalMomentum(self):

        vx = self.v[arange(0, 2*self.N, 2)]
        vy = self.v[arange(1, 2*self.N, 2)]

        vx -= vx.mean() # zero mean momentum
        vy -= vy.mean()

        self.v[arange(0, 2*self.N, 2)] = vx
        self.v[arange(1, 2*self.N, 2)] = vy


    def reverseTime(self):

        self.dt = -self.dt


    def cool(self, time=1.0):

        steps = int(time/self.dt)
        for i in range(steps):
            self.verletStep()
            self.v *= (1.0 - self.dt) # friction slows down atoms

        self.resetStatistics()



# INITIAL CONDITION METHODS

    def randomPositions(self):

        self.x = self.L * numpy.random.random(2*self.N)

        self.forceType = "weavepowerLaw"
        self.cool(time=1.0)
        self.forceType = "weavelennardJones"


    def triangularLatticePositions(self):

        self.rectangularLatticePositions()
        #self.randomPositions()
        self.v += numpy.random.random(2*self.N) - 0.5 # jiggle to break symmetry

        self.forceType = "weavepowerLaw"
        self.cool(time=10.0)
        self.forceType = "weavelennardJones"


    def rectangularLatticePositions(self): # assume that N is a square integer (4, 16, 64, ...)
```

```
        nx = int(sqrt(self.N))
        ny = nx
        dx = self.L / nx
        dy = self.L / ny

        for i in range(nx):
            x = (i + 0.5) * dx
            for j in range(ny):
                y = (j + 0.5) * dy
                self.x[2*(i*ny+j)] = x
                self.x[2*(i*ny+j)+1] = y


    def randomVelocities(self):

        self.v = numpy.random.random(2*self.N) - 0.5

        self.zeroTotalMomentum()

        T = self.temperature()
        self.v *= sqrt(self.initialTemperature/T)



# MEASUREMENT METHODS

    def kineticEnergy(self):

        return 0.5 * (self.v * self.v).sum()


    def potentialEnergy(self):

        return self.weaveLennardJonesPotentialEnergy()
        #return self.lennardJonesPotentialEnergy()


    def lennardJonesPotentialEnergy(self): # Gould Eqs. 8.1 and 8.2

        tiny = 1.0e-40 # prevents division by zero in calculation of self-force
        halfL = 0.5 * self.L
        N = self.N

        x = self.x[arange(0, 2*N, 2)]
        y = self.x[arange(1, 2*N, 2)]
        U = 0.0
        minimumImage = self.minimumImage
        for i in range(N):  # The O(N**2) calculation that slows everything down

            dx = minimumImage(x[i] - x)
            dy = minimumImage(y[i] - y)

            r2inv = 1.0/(dx**2 + dy**2 + tiny)
            dU = r2inv**6 - r2inv**3
            dU[i] = 0.0 # no self-interaction
            U += dU.sum()
```

```
    return 2.0 * U


def weaveLennardJonesPotentialEnergy(self): # Gould Eqs. 8.1 and 8.2

    L = self.L
    halfL = 0.5 * L
    N = self.N

    x = self.x[arange(0, 2*N, 2)]
    y = self.x[arange(1, 2*N, 2)]
    U = zeros(1)

    code = """
        double dx, dy, r2inv, r6inv;

        for (int i = 0; i < N; i++) {
            for (int j = i+1; j < N; j++) {

                dx = x(i) - x(j);
                if (dx > halfL) dx = dx - L;
                if (dx < -halfL) dx = dx + L;

                dy = y(i) - y(j);
                if (dy > halfL) dy = dy - L;
                if (dy < -halfL) dy = dy + L;

                r2inv = 1.0 / (dx*dx + dy*dy);
                r6inv = r2inv*r2inv*r2inv;

                U(0) += r6inv*r6inv - r6inv;

            }
        }
        """

    weave.inline(code, ['N', 'x', 'y', 'L', 'halfL', 'U'], type_converters=converters.blitz, compiler='gcc')

    return 4.0 * U[0]


def energy(self):

    return self.potentialEnergy() + self.kineticEnergy()

def temperature(self): # Gould Eq. 8.6

    return self.kineticEnergy() / self.N

def angularMomentum(self): # Calculate z component of angular momentum using definition

    vx = self.v[arange(0, 2*self.N, 2)]
    vy = self.v[arange(1, 2*self.N, 2)]

    x = self.x[arange(0, 2*self.N, 2)]
```

```
        y = self.x[arange(1, 2*self.N, 2)]

        return ((x-self.L/2)*vy-(y-self.L/2)*vx).sum()


    # STATISTICS METHODS

    def resetStatistics(self):

        self.steps = 0
        self.temperatureAccumulator = 0.0
        self.angularMomentumAccumulator = 0.0
        self.squareTemperatureAccumulator = 0.0
        self.virialAccumulator = 0.0
        self.xArray = array([])
        self.vArray = array([])


    def meanTemperature(self):

        return self.temperatureAccumulator / self.steps


    def meanSquareTemperature(self):

        return self.squareTemperatureAccumulator / self.steps


    def meanPressure(self): # Gould Eq. 8.9

        meanVirial = 0.5 * self.virialAccumulator / self.steps # divide by 2 because force is calculated twice
per step
        return 1.0 + 0.5 * meanVirial / (self.N * self.meanTemperature())


    def heatCapacity(self): # Gould Eq. 8.12

        meanTemperature = self.meanTemperature()
        meanSquareTemperature = self.meanSquareTemperature()
        sigma2 = meanSquareTemperature - meanTemperature**2
        denom = 1.0 - sigma2 * self.N / meanTemperature**2
        return self.N / denom

    def meanEnergy(self):

        return self.EnergyArray.mean()

    def stdEnergy(self):

        return self.EnergyArray.std()


    # PLOTTING METHODS

    def plotPositions(self):
```

```
    figure(1)
    scatter(self.x[arange(0, 2*self.N, 2)], self.x[arange(1, 2*self.N, 2)], s=5.0, marker='o', alpha=1.0)
    xlabel("x")
    ylabel("y")


def plotTrajectories(self, number=1):

    figure(2)
    xlabel("x")
    ylabel("y")
    N = self.N
    size = len(self.xArray)/(2*N)
    r = reshape(self.xArray, [size, 2*N])
    for i in range(number):
        x = r[:, 2*i]
        y = r[:, 2*i+1]
        plot(x, y, ".")


def plotTemperature(self):

    figure(3)
    plot(self.tArray, self.temperatureArray)
    xlabel("time")
    ylabel("temperature")


def plotEnergy(self):

    figure(4)
    plot(self.sampleTimeArray, self.EnergyArray)
    xlabel("time")
    ylabel("Energy")


def velocityHistogram(self):

    figure(5)
    hist(self.vArray, bins=100, normed=1)
    xlabel("velocity in x- or y-directions")
    ylabel("probability")

def plotAngularMomentum(self):

    figure(6)
    plot(self.tArray, self.angularMomentumArray)
    xlabel("time")
    ylabel("Angular Momentum")

def angularMomentumHistogram(self):

    figure(7)
    hist(self.angularMomentumArray, bins=100, normed=1)
    xlabel('angular momentum')
```

```
        ylabel('Probability')

        mu = mean(self.angularMomentumArray)
        variance = var(self.angularMomentumArray)
        sigma = sqrt(variance)
        print("Mean of the Gaussian Distribution is : ",mu)
        print("Sigma of the Gaussian Distribution is : ",sigma)

        t = linspace(min(self.angularMomentumArray), max(self.angularMomentumArray), 100)
        plot(t, mlab.normpdf(t, mu, sigma))

    def showPlots(self):
        show()


# RESULTS METHODS

    def results(self):
        print("\n\nRESULTS\n")
        print("time = ", md.t, " total energy = ", md.energy(), " and temperature = ", md.temperature())
        if (self.steps > 0):
            print("Mean energy = ", md.meanEnergy(), " and standard deviation = ", md.stdEnergy())
            print("Cv = ", md.heatCapacity(), " and PV/NkT = ", md.meanPressure())


start = time.time()
#md = MolecularDynamics(N=16, L=4, initialTemperature=1.0, initialAngularMomentum=0.0) # instantiate
object
#md = MolecularDynamics(N=36, L=6, initialTemperature=1.0, initialAngularMomentum=0.0) # instantiate
object
#md = MolecularDynamics(N=64, L=8, initialTemperature=1.0, initialAngularMomentum=0.0) # instantiate
object
#md = MolecularDynamics(N=100, L=10, initialTemperature=1.0, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=144, L=12, initialTemperature=1.0, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=196, L=14, initialTemperature=1.0, initialAngularMomentum=0.0) #
instantiate object

#md = MolecularDynamics(N=256, L=30, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=20, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=19, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=18, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=17, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=15.1, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=15.2, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=15.3, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
```

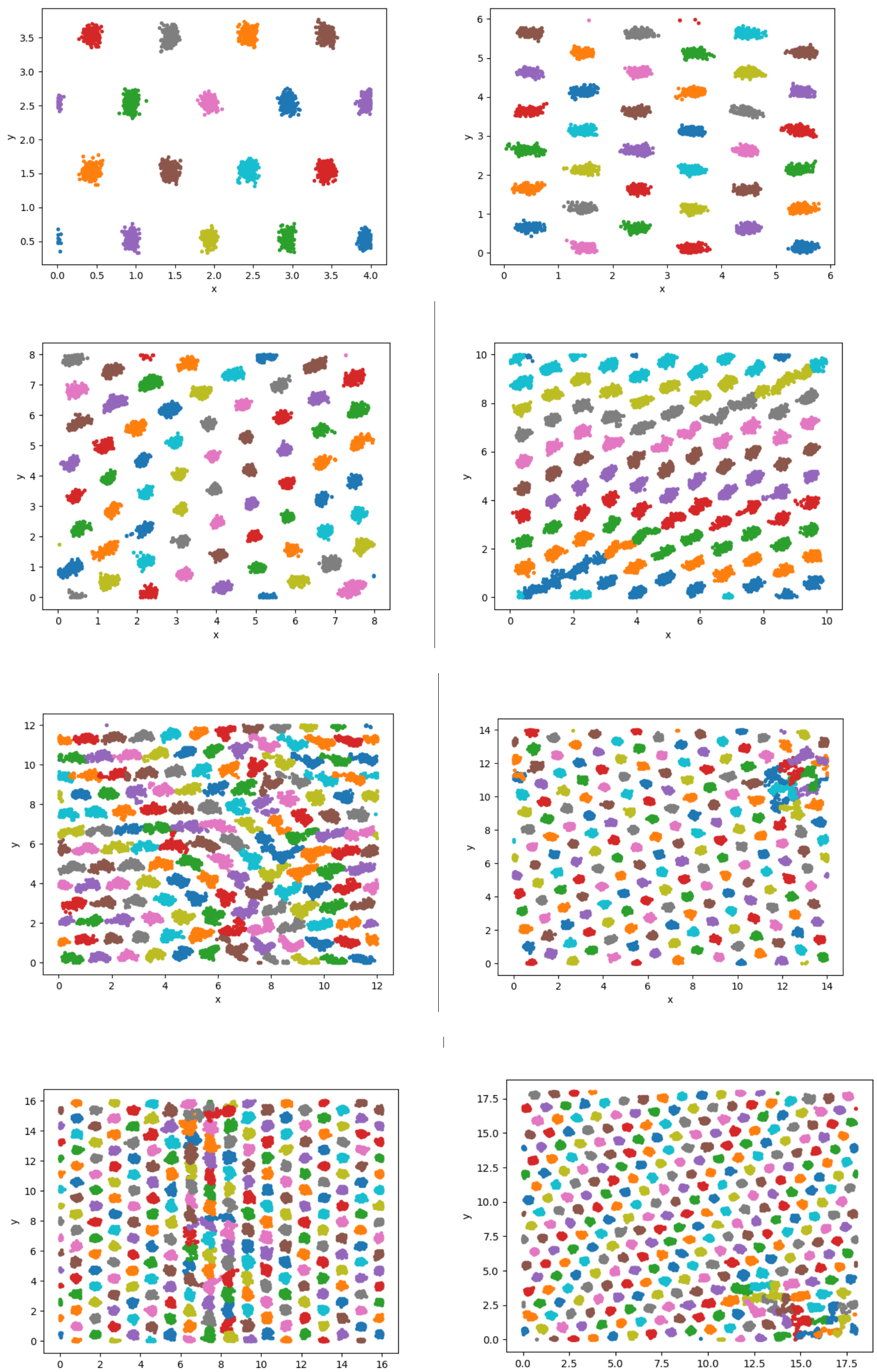
```
#md = MolecularDynamics(N=256, L=15.4, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=15.5, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=15.6, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=15.7, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=15.8, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=15.9, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object

#md = MolecularDynamics(N=256, L=16.1, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16.2, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16.3, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16.4, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16.5, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16.6, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16.7, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16.8, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object
md = MolecularDynamics(N=256, L=16.9, initialTemperature=1, initialAngularMomentum=0.0) #
instantiate object


#md = MolecularDynamics(N=256, L=16, initialTemperature=0.05, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16, initialTemperature=0.1, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16, initialTemperature=0.5, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16, initialTemperature=1.0, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16, initialTemperature=2.0, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16, initialTemperature=3.0, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16, initialTemperature=4.0, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16, initialTemperature=5.0, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16, initialTemperature=6.0, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16, initialTemperature=7.0, initialAngularMomentum=0.0) #
instantiate object
#md = MolecularDynamics(N=256, L=16, initialTemperature=8.0, initialAngularMomentum=0.0) #
instantiate object
```

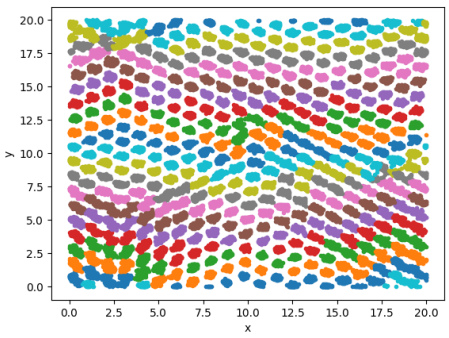
```
#md = MolecularDynamics(N=256, L=16, initialTemperature=9.0, initialAngularMomentum=0.0) # instantiate object
#md = MolecularDynamics(N=256, L=16, initialTemperature=10.0, initialAngularMomentum=0.0) # instantiate object

#md = MolecularDynamics(N=324, L=18, initialTemperature=1.0, initialAngularMomentum=0.0) # instantiate object
#md = MolecularDynamics(N=400, L=20, initialTemperature=1.0, initialAngularMomentum=0.0) # instantiate object
#md = MolecularDynamics(N=484, L=22, initialTemperature=1.0, initialAngularMomentum=0.0) # instantiate object
#md = MolecularDynamics(N=576, L=24, initialTemperature=1.0, initialAngularMomentum=0.0) # instantiate object
#md = MolecularDynamics(N=676, L=26, initialTemperature=1.0, initialAngularMomentum=0.0) # instantiate object
#md = MolecularDynamics(N=784, L=28, initialTemperature=1.0, initialAngularMomentum=0.0) # instantiate object
#md = MolecularDynamics(N=900, L=30, initialTemperature=1.0, initialAngularMomentum=0.0) # instantiate object
#md = MolecularDynamics(N=1024, L=32, initialTemperature=1.0, initialAngularMomentum=0.0) # instantiate object
#md = MolecularDynamics(N=1156, L=34, initialTemperature=1.0, initialAngularMomentum=0.0) # instantiate object


# EQUILIBRATION AND STATISTICS
md.triangularLatticePositions()
md.randomVelocities()
md.plotPositions()
md.results()
md.evolve(time=10.0) # initial time evolution
md.resetStatistics() # remove transient behavior
md.evolve(time=20.0) # accumulate statistics
md.results()
md.angularMomentum()

end = time.time()
print('Time is %.2f'%(end-start))

md.plotEnergy()
md.plotTrajectories(md.N)
md.plotTemperature()
md.velocityHistogram()
md.plotAngularMomentum()
md.angularMomentumHistogram()
md.showPlots()
```

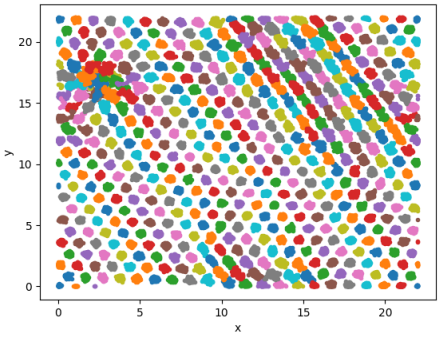
9. *Supplementary Material*

9.1 Angular Momentum and System Size Figures

Particle trajectories with increasing system size (From simulation 1 to 16):

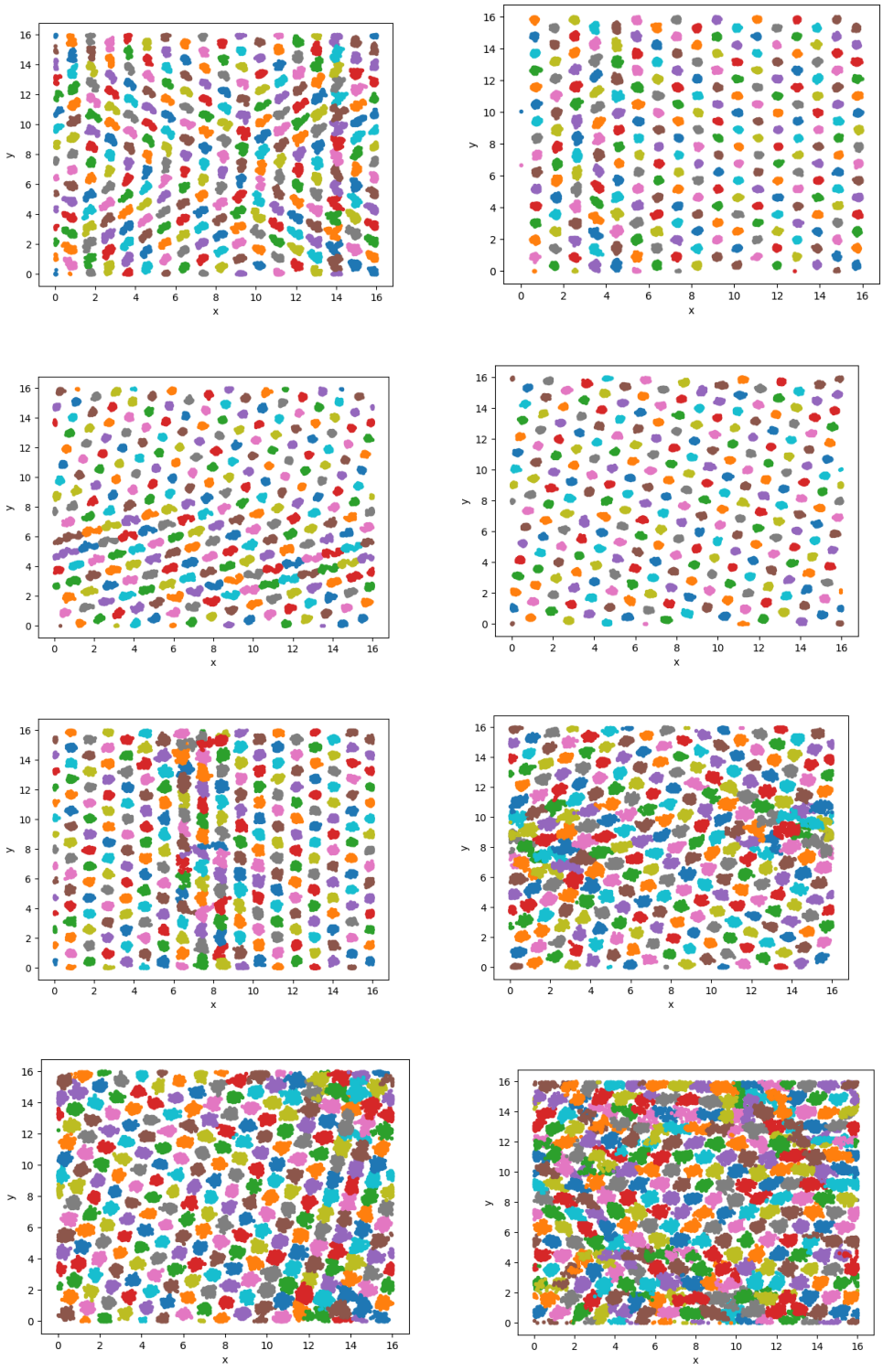


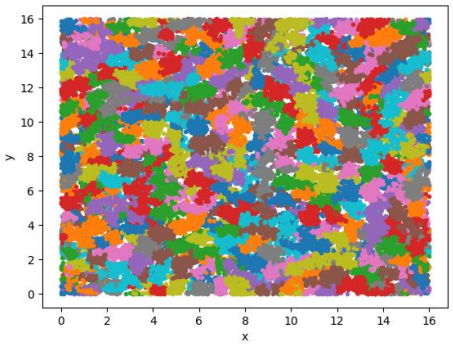

20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
y
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
x

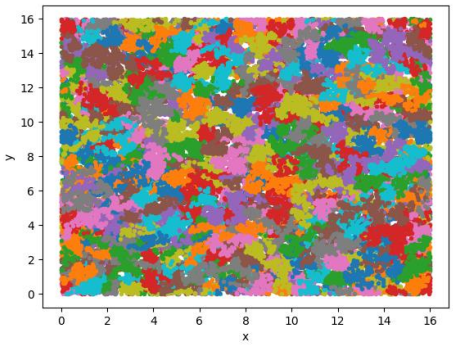

20
15
10
5
0
y
0
5
10
15
20
x

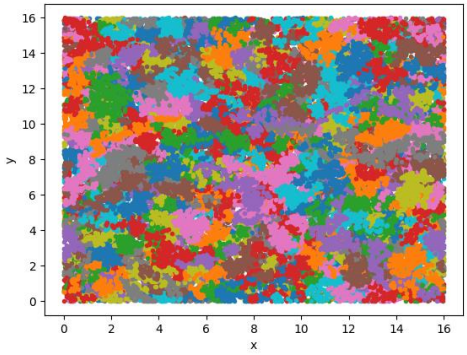
9.2 Angular Momentum and Temperature Figures

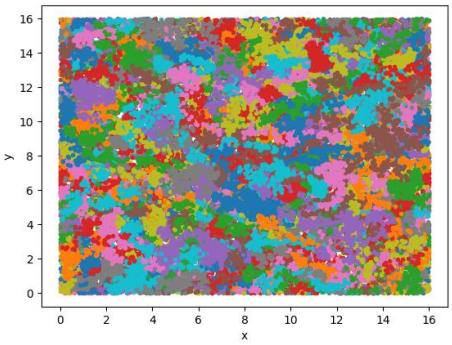
Particle trajectories of different simulations with increasing initial temperature (N=256 and ρ=1):

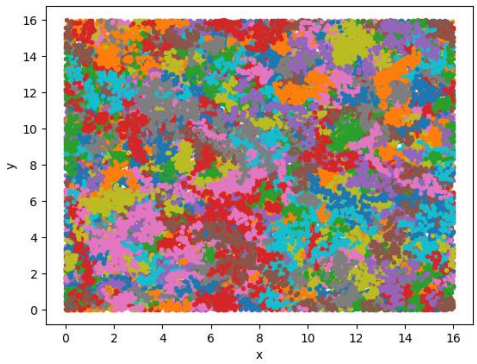


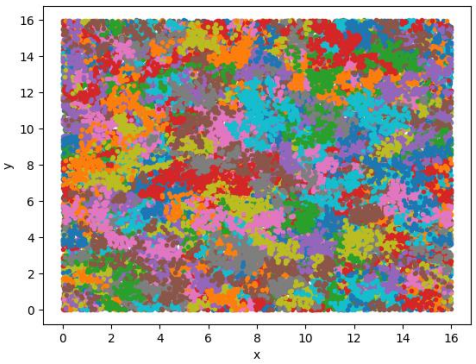

y
x
0 2 4 6 8 10 12 14 16


y
x
0 2 4 6 8 10 12 14 16


y
x
0 2 4 6 8 10 12 14 16

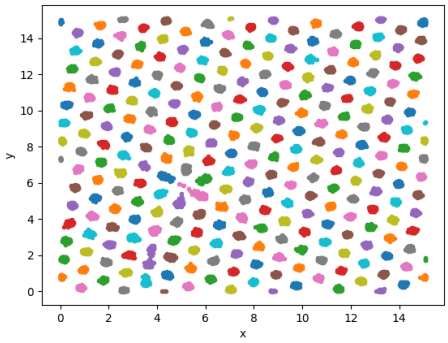

y
x
0 2 4 6 8 10 12 14 16

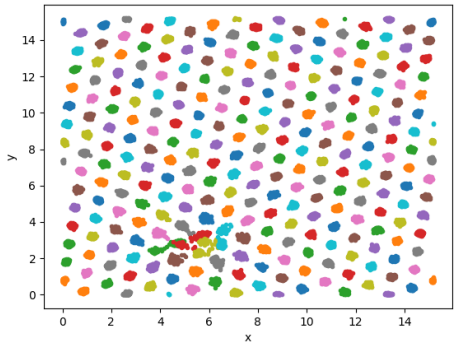

y
x
0 2 4 6 8 10 12 14 16

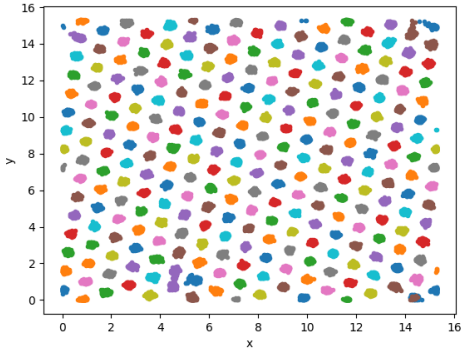

y
x
0 2 4 6 8 10 12 14 16

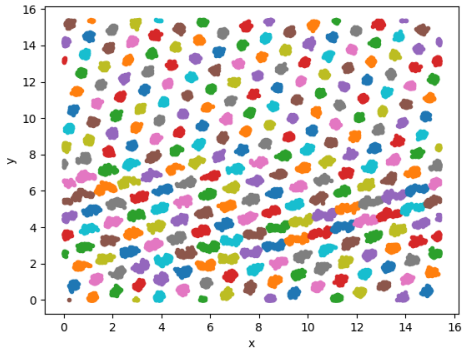
9.3 Angular Momentum and Density Figures

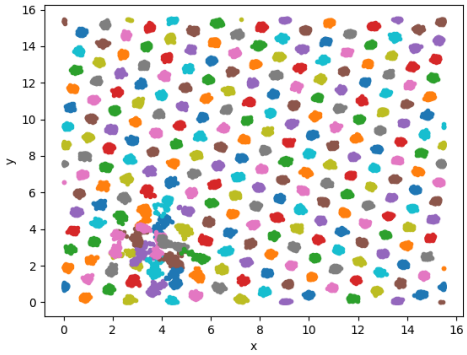
Particle trajectories from different simulations with decreasing density (N=256 and initial temperature=1 for all simulations):

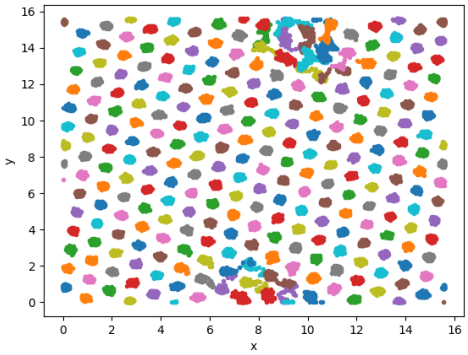


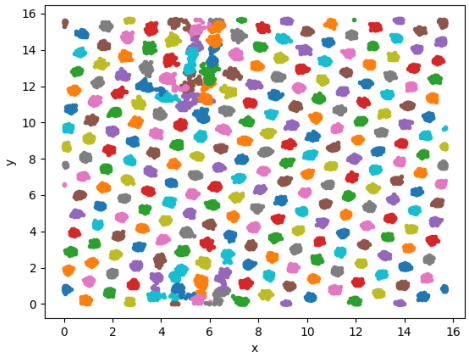


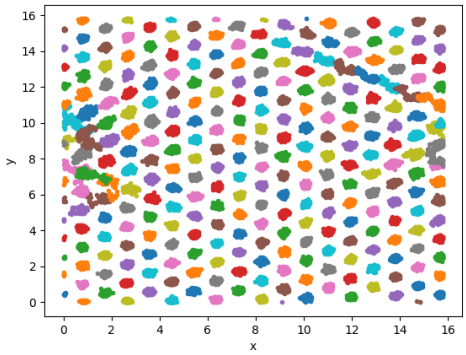


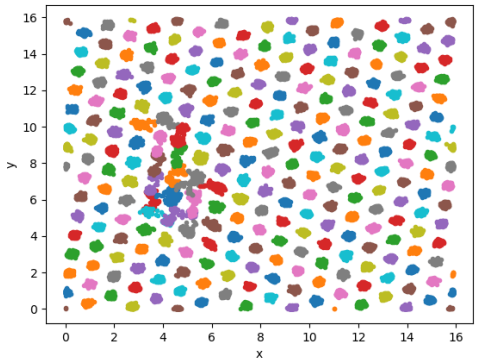







y
x
0
2
4
6
8
10
12
14
16


y
x
0
2
4
6
8
10
12
14
16


y
x
0
2
4
6
8
10
12
14
16