Overview

Consider a collection of particles where each particle exerts forces on all of the other particles according to the force equation:

$$f(r) = 24\left[\frac{2}{r^{13}} - \frac{1}{r^7}\right]$$

This forces comes from something called the Lennard-Jones potential and is appropriate for systems of inert particles (like Argon), and gets increasingly less accurate for other atom types. Here are some questions to ask yourself about this system:

1. What do you think will happen if we start the system at high temperature and slowly cool it to low temperature?
2. What distribution will the speeds of the particles follow? Does the shape of this distribution depend on the initial velocities?
3. Is the motion of a single particle diffusive, like we saw with random walks?

These are the types of questions you'll be thinking about this week and the simulations you'll build will produce fun movies of particles moving around and interacting with one another. Molecular dynamics simulations are heavily used in the field of materials simulation, which contribute to the discovery of new, high performing materials. Your simulations this week will involve 10-20 particles, whereas research-quality simulations typically involve simulations of thousands of particles. Thus you can begin to see the need for a high-performing supercomputer to do meaningful work.

The molecular dynamics simulation you will build this week will probably be the longest code you've written all semester. As always, your best approach is to build small chunks of code, testing the chunks thoroughly before moving on to the next chunk. To help you along in that process, I'll give you several important tasks:

1. Force vector calculation (periodic boundary conditions).
2. Initializing the locations of particles.

Force vector calculation (periodic boundary conditions)

As mentioned above, the equation for the force between particles is given by:

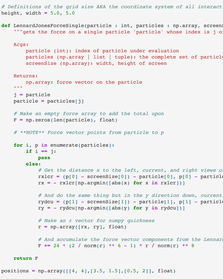
$$f(r) = 24\left[\frac{2}{r^{13}} - \frac{1}{r^7}\right]$$

One important task this week will be that of calculating the **net** force on a given particle due to all of the other particles. Furthermore, we will be employing periodic boundary conditions which means that you'll need to check to see if a periodic version of a particle is closer than the original. (read that last sentence again!) Below you will find a list of three particles that are confined to 5 x 5 box. (Plot included) Complete the following tasks:

1. Plot the force function above and comment on why it looks like a reasonable choice for interacting particles. Where is the force repulsive? Where is it attractive?
2. Now build a function that calculates the net force on a single particle. The function should take as arguments: i) the list of particle positions and ii) the particle for which the net force vector is sought. You may assume that the force between particles that are separated by more than 3σ is zero. Remember to enforce periodic boundary conditions; a neighboring particle may be closer than it appears.
3. Check your answer with the ones given in the table below.

Particle Location	Net Force
(4,4)	(0.01672618, -0.06456706)
(3.5,1.5)	(0.14096508, 0.06919638)
(0.5,2)	(-0.16669127, - 0.00462331)

In [ ]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from numpy.linalg import norm

# Define the Lennard-Jones force equation
def LenJon(r):
    return 24 * (2 / r ** 13 - 1 / r ** 7)
r = np.linspace(1.1, 2.0, 50)

# And plot it against r
plt.plot(r, F(r), 'b-')
plt.xlabel('r')
plt.ylabel('f(r)')
plt.title('Lennard-Jones Force vs. Distance')
plt.show()
```



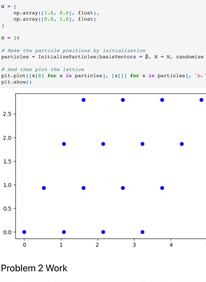
Part 1

The valley represents a point where the force on the particles will be attractive, while there are 2 equilibrium positions available for the system. The regions where the particle comes to close to another, the force they experience due to each others position will be repulsive. In the last case where separation distances become large, the force on another by one becomes negligibly small, $\lim_{r\to\infty} f(r) = 0$. As it is shown, the force goes to incredibly high values for distances around 1. Hence, we will control some of the randomization that occurs when it comes to generating random initial positions of the particles as to eliminate possibility of getting interaction forces of those magnitudes. Those would especially be problematic since the Lennard-Jones potential is only an approximation function for the true quantum mechanical events taking place. In other words, approximations to phenomena generally perform poorly in the extrema of reality.

Part 2

Next, I build the function which computes the force on a single particle due to all the others in the system WITH PERIODIC BOUNDARY CONDITIONS. This means the top of the grid is 'connected' to the bottom of the grid, and the right side of the grid is also 'connected' to the left side of the grid. This wrap around situation will result in a force interaction between particles using the MINIMUM distance between the two. Below is my attempt to translate this language into software.

In [ ]:
```
%matplotlib inline

# Definitions of the grid size AKA the coordinate system of all interactions
height, width = 5.0, 5.0

def LennardJonesForcedSingle(particle : int, particles : np.array, screendim : np.array):
    '''gets the force on a single particle 'particle' whose index is j of the 'particles' (array | list | tuple)

    Args:
        particle (int): index of particle under evaluation
        particles (np.array | list | tuple): the complete set of particle positions in the system, including the one of interest
        screendim (np.array): width, height of screen

    Returns:
    --- np.array: force vector on the particle
    '''
    j = particle
    particle = particles[j]

    # Make an empty force array to add the total upon
    F = np.zeros(len(particle), float)

    # **HOTT** Force vector points from particle to p

    for i, p in enumerate(particles):
        if i == j:
            pass
        else:
            # Get the distance x to the left, current, and right views of the grid by moving p a screenwidth over at a time
            rxlor = (p[0] - screendim[0]) - particle[0], p[0] - particle[0], (p[0] + screendim[0]) - particle[0]
            rx = rxlor[np.argmin([abs(x) for x in rxlor])]

            # And do the same thing but in the y direction down, current, and up
            rydou = (p[1] - screendim[1]) - particle[1], p[1] - particle[1], (p[1] + screendim[1]) - particle[1]
            ry = rydou[np.argmin([abs(y) for y in rydou])]

            # Make an r vector for sumpy quickness
            r = np.array([rx, ry], float)

            # And accumulate the force vector components from the Lennard-Jones Force function
            F += 24 * (2 / norm(r) ** 6 - 1) * r / norm(r) ** 8

    return F

positions = np.array([[4, 4],[3.5, 1.5],[0.5, 2]], float)

plt.plot([x[0] for x in positions],[x[1] for x in positions], 'r.', ms = 12)
plt.xlim(0, 5)
plt.ylim(0, 5)
plt.show()

screendim = np.array([width, height], float)

# Generate the table displayed at the top of the page
f0 = LennardJonesForcedSingle(particle = 0, particles = positions, screendim = screendim)
f1 = LennardJonesForcedSingle(particle = 1, particles = positions, screendim = screendim)
f2 = LennardJonesForcedSingle(particle = 2, particles = positions, screendim = screendim)

print('     x          y')
print(f'0) {f0[0]:.4f}    {f0[1]:.4f}')
print(f'1) {f1[0]:.4f}    {f1[1]:.4f}')
print(f'2) {f2[0]:.4f}    {f2[1]:.4f}')
```



```
     x          y
0) 0.0167    -0.0646
1) 0.1500     0.0925
2) -0.1667    -0.0046
```

Problem 3 – Comments on Results

As shown, the table that I was able to reproduce gives the exact same results as the table at the top of the page.

Initializing particle locations

Before you can simulate the motion of your particles, you must initialize their positions and velocities. This can be done in one of two ways:

1. Place the particles on a periodic lattice.
2. Place the particles randomly in the box.

In addition, when initializing your particles, it will be important that particles are not too close to each other. (From your plot from above, what will happen when the separation between two particles is less than about 1?)

1. Write a function that initializes the locations of 16 particles in a 4 x 4 box by putting them on the sites of a triangular lattice. Use the following lattice vectors:

$$\vec{a}_1 = 1.07457(1, 0)$$

$$\vec{a}_2 = 1.07457(0.5, \frac{\sqrt{3}}{2})$$

   Plot the simulation cell to visually verify that particles aren't too close. Note: Make your function general so that it works for any lattice vectors, not just a triangular lattice.

2. Write a function that initializes the locations of particles in a 10 x 10 box to pseudo-random positions inside the box, following the steps below to do so.

   A. Place the particles on a square lattice.

$$\vec{a}_1 = (2, 0)$$

$$\vec{a}_2 = (0, 2)$$

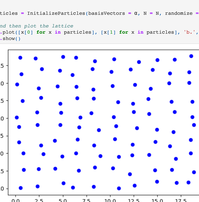
   B. Add random displacements to each particle. Randomize both the direction of the displacement as well as the magnitude, but the magnitude should be no larger than $\frac{1}{2}$

   C. Plot the simulation cell to visually verify that particles aren't too close.

Problem 1

Below, I have a function which accomplishes tasks 1 and 2, along with each subtask of 2. The next portion has the second plot containing random positioning of the other particles on the lattice. As shown below, the lattice shows a very neat structure on a well-defined basis coordinate system.

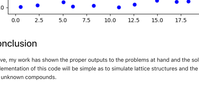
In [ ]:
```
%matplotlib inline
import random

# Function to generate a set of N particles given a standardized basis for a specified lattice structure (the basis vectors which form that lattice)
def InitializeParticles(basisVectors : list, N : int, randomize : bool = False):
    '''returns an initialized set of particles on a lattice geometry defined by the basis vectors passed in

    Args:
        basisVectors (list): list or pair of np.array's containing the basis vector set for a 2D lattice
        N (int): number of particles in the lattice

    Returns:
    --- particles: a list of np.array's
    '''
    # Create an empty list to contain the particle positions
    particles = [_ for _ in range(N)]

    base = int(N ** 0.5)

    for i in range(N):
        # The coefficients of the basis vectors are integral, which enforces lattice points to follow
        # the geometric shape given by the basis vectors, as well as a designable spacing between points
        # if they are to be used when not randomized
        particles[i] = np.copy(basisVectors[0] * (i % base) + basisVectors[1] * (i // base))

        if randomize:
            # Since the max magnitude given by a 2 component random.random function is sqrt(2), i divide by sqrt(2)
            # to normalize and multiply by 3 to get max displacement of 3 / 2 for pseudo-random positioning on the lattice
            particles[i] += 3 / 2 * np.random.random(size = 2) // np.sqrt(2)

    return particles

# Problem 1 -> Triangular lattice basis
β = [
    1.07457 * np.array([1.0, 0.0], float),
    1.07457 * np.array([0.5, 3 ** 0.5 / 2], float)
]

α = [
    np.array([1.0, 0.0], float),
    np.array([0.0, 1.0], float)
]

N = 16

# Make the particle positions by initialization
particles = InitializeParticles(basisVectors = β, N = N, randomize = 0)

# And then plot the lattice
plt.plot([x[0] for x in particles], [x[1] for x in particles], 'b.', ms = 12)
plt.show()
```



Problem 2 Work

Below, I have initialized more particles once again but this time, with the new basis geometry for the lattice. It is clear that the positioning is functional with randomization as well. This results in a very beautifully chaotic display of particles in the lattice most generally represented by its basis vectors.

In [ ]:
```
%matplotlib inline
# Plotting the 10 x 10 box with basis vectors a1 = [2, 0] and a2 = [0, 2] and then randomize their positions after placement on the lattice

N = 100
α = [
    np.array([2.0, 0.0], float),
    np.array([0.0, 2.0], float)
]

particles = InitializeParticles(basisVectors = α, N = N, randomize = True)

# And then plot the lattice
plt.plot([x[0] for x in particles], [x[1] for x in particles], 'b.', ms = 12)
plt.show()
```



Conclusion

Above, my work has shown the proper outputs to the problems at hand and the solutions are all functional. To apply this into a larger program will be straightforward and at the very most, time consuming. Implementation of this code will be simple as to simulate lattice structures and the time evolution of the energy levels and forces acting on the system. Thus, simulating materials and geometries of both known and unknown compounds.