Modeling Molecular Dynamics using the Verlet Algorithm with a Lennard-Jones Potential

Travis Morton

October 17, 2014

1 16 Particles in a 10 x 10 box

This simulation started off with a system of 16 particles evenly spaced in a 10 x 10 box with periodic boundary conditions. It was then stepped through time steps of $dt = .01$ for a certain amount of time, dependent on what value was being determined.

1.1 Speed Distribution Calculation

The particles should have a speed distribution that follows a Maxwell-Boltzmann distribution for this system. This can be modeled with the following equation, which has been reduced from the original Maxwell-Boltzmann equation using the constants given for the system in the prompt.

$$f = \sqrt{\frac{2}{\pi}}\frac{s^2}{a^3}e^{\frac{-s^2}{2a^2}}$$

where

$$a = \sqrt{\frac{kT}{m}}$$

Since k and m are both 1 in this case, $a = \sqrt{\frac{\langle v^2 \rangle}{2}}$. Using this equation and the Maxwell-Boltzmann distribution above I was able to compare the speed distributions from 200 runs to the known distribution. Figure 1 shows the results.

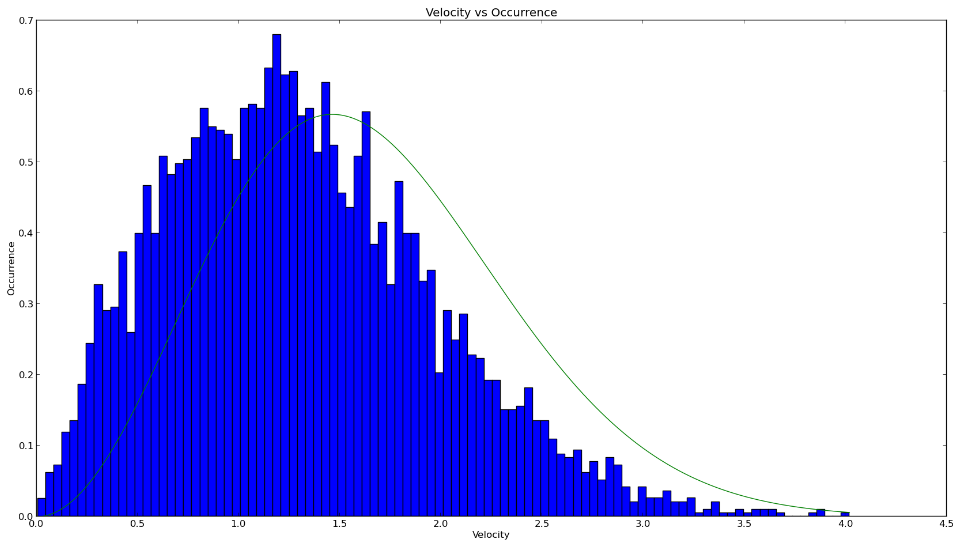

Figure 1: Speed distribution in 10 x 10 grid

The data I obtained from the simulation is skewed slightly from the the known distribution. This may be caused by the potential function being truncated at a certain value and thus not favoring the higher velocities. It also may have been due to a miscalculation of the average velocity in the function, but I'm not sure where that miscalculation is if that's so.

1.2 Diffusion Constant

I calculated the $\langle r^2 \rangle$ value after 15 seconds and plotted it vs the time to find the diffusion constant. $D_0 = \frac{slope}{4}$, so for this case $D_0 = .9035$. Figure 2 shows the plot of $\langle r^2 \rangle$ vs t with the linear fit overlayed on top.

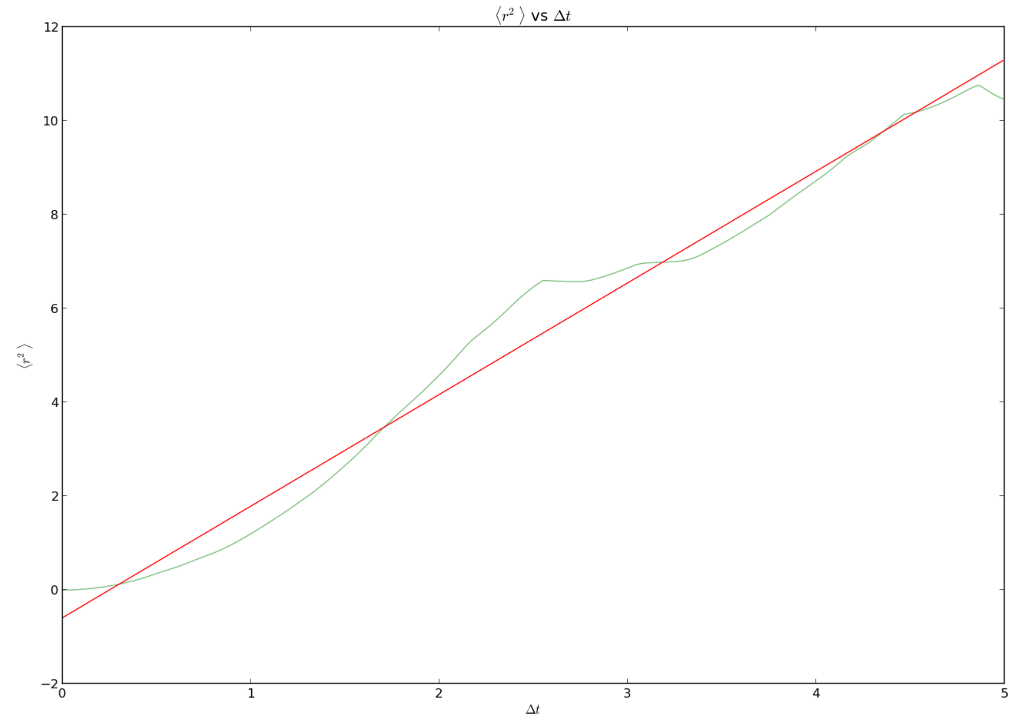

Figure 2: $\langle r^2 \rangle$ vs t in 10 x 10 grid for 15 seconds

1.3 Phase of System

Since the $\langle r^2 \rangle$ vs t graph does not plateau, this system is not a solid. This can also be seen by tracing the individual particle paths and seeing that they do not line up in any sort of lattice pattern no matter the time step. The system seem to not be diffuse enough to be a gas, so I think it is a liquid.

2 16 Particles in a 4 x 4 box

For this problem I set up a system of 16 particles in a 4 x 4 box with periodic boundary condition like in the first part. This one had smaller initial random flucuations in the positions and smaller initial random flucuations in the velocities. These adjustments made it less likely for particles to collide. Also, the time step was decreased to $dt = .001$ to allow for smoother convergence.

2.1 Temperature of the System

After running the simulation for 15 seconds, I calculated the average kinetic energy of each particle and then used that value to find the temperature in a 2D system. Using the units of this system I found $T = .25$. This seems reasonable because it is low and similar to the temperature of the other system, but in a smaller volume (with lower initial kinetic energy).

2.2 Equilibrium Structure

The equilibrium structure of the system is an offset lattice (shown in Figure 3). This is the lowest energy state since the particles get to this configuration and stay there, as seen in the $\langle r^2 \rangle$ vs t graph (Figure 4) as well as tracing the particles' positions between each time step.

2.3 Argument for 2D Solid

I would argue that this system models a 2D solid because once it reaches an equilibrium state the particles stay relatively stationary (but they do wiggle a bit). You can see this in the plateau of the $\langle r^2 \rangle$ vs t graph (Figure 4). Also, the lattice structure that is developed is similar to that of a solid.

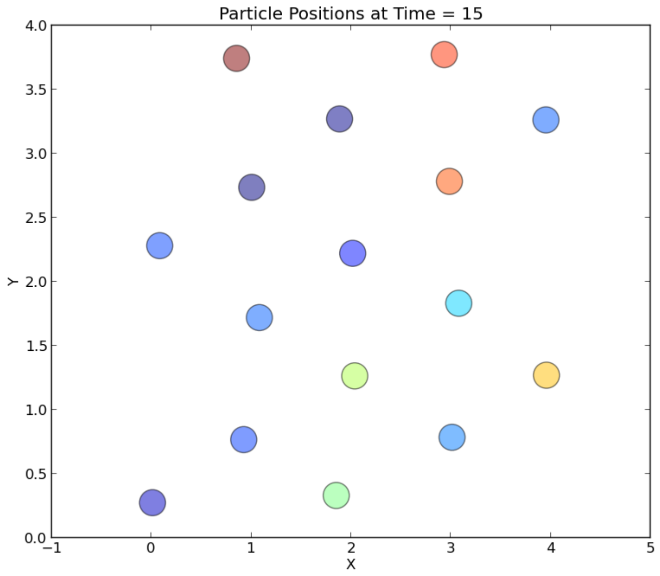

Figure 3: Positions of particles in 4x4 grid after 15 seconds

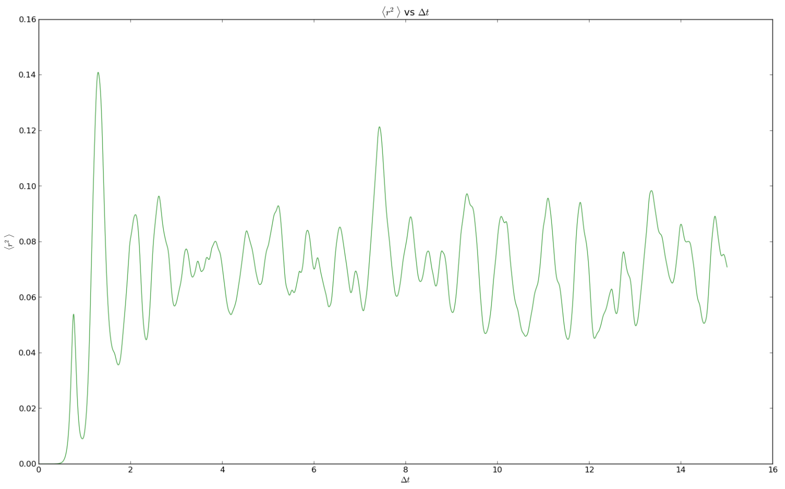

Figure 4: $\langle r^2 \rangle$ vs t in 4 x 4 grid for 15 seconds

3 Heating of 2D Solid

To heat the solid, I started with the converged lattice structure from part 2 and then slowly increased the random initial velocities of the particles until the $\langle r^2 \rangle$ vs t started to steadily increase. Once it stopped oscillating and began to increase, I calculated the temperature of the solid and determined the melting point was around that value.

3.1 Melting Temperature of 2D solid

Using a binary search of initial starting velocity magnitudes, I changed the temperature (velocity) of the initial conditions, but used the same converged lattice structure each time. I ran the simulation for 5 seconds each time and when the $\langle r^2 \rangle$ vs t graph started to rise (Figure 5) I calculated the temperature and determined the melting point to be greater than about $T = 0.3$. There was some variance in this, but for the most part, each trial resulted in a melted system above $T = 0.3$ and a solid system below $T = 0.3$.

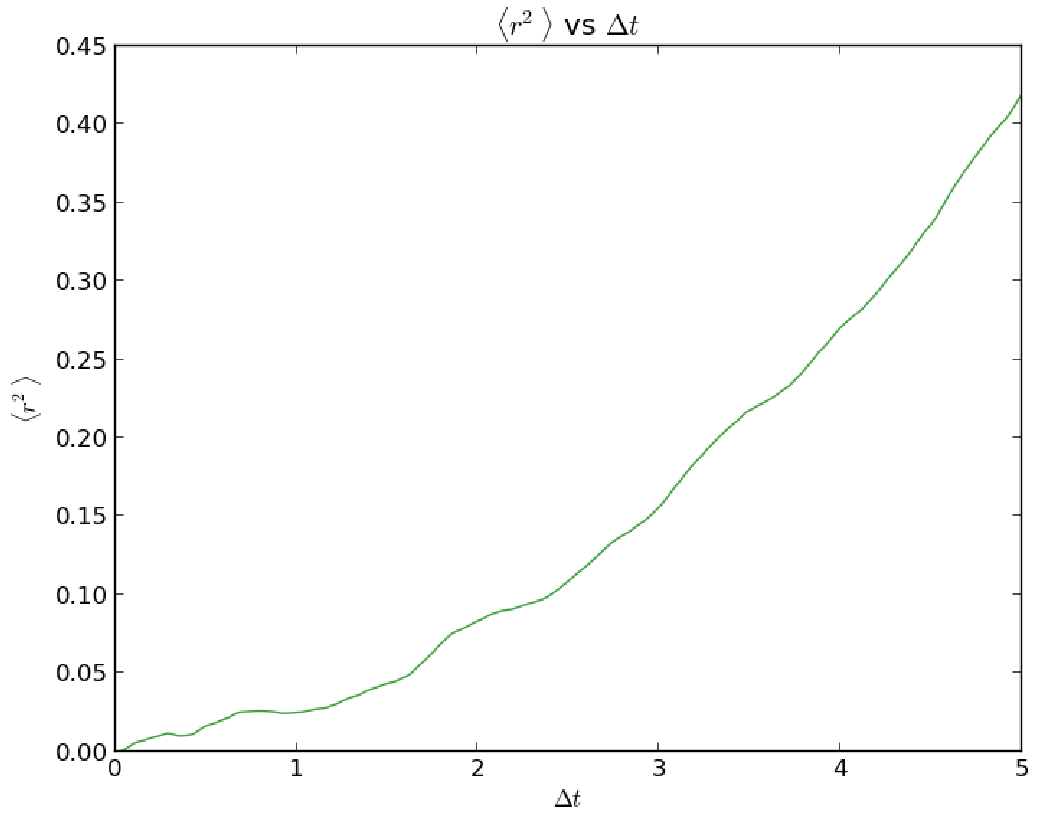

Figure 5: Melting of a 2D solid as seen by $\langle r^2 \rangle$ vs t

A Problem 1 Code

```
#!/usr/bin/python

import os
import math
import numpy
import random
import matplotlib.pyplot as plt
import matplotlib.animation as animation


# Initialize variables
dt = .01  # .01 for problem 1 and .005 for problem 2 and 3
initialDr = .01  # .01 for problem 1 and 0 for problem 2 and 3
boxWidth = 10.0  # 4.0 for problem 2 and 3, 10.0 for problem 1
boxHeight = 10.0  # 4.0 for problem 2 and 3, 10.0 for problem 1
particleNumber = 16  # should be a square number
timeLength = 2.0  # 4.0 for r2 calculation, can be less for temp calculation
vRange = 3.0
initialDv = 1  # .1 for problem 1 and .0001 for problem 2 and 3
r2 = []  # average distance squared list, in order of time steps
s = []  # list of speeds for each particle

# initialize position, velocity,
# and acceleration arrays
r = numpy.zeros((particleNumber, 2))
v = numpy.zeros((particleNumber, 2))
a = numpy.zeros((particleNumber, 2))

# filling r array with equally spaced particles
w = numpy.ndenumerate(r)
count = 0
for x, y in w:
    xStep = boxWidth / float(math.sqrt(particleNumber))
    yStep = boxHeight / float(math.sqrt(particleNumber))
    if x[1] == 0:
        r[x[0]][x[1]] = count / int(math.sqrt(particleNumber)) * \
                        xStep + (random.random() - .5) * initialDr
        count += 1
    else:
        r[x[0]][x[1]] = x[0] % math.sqrt(particleNumber) * \
                        yStep + (random.random() - .5) * initialDr

# Saving initial r array for use later in calculating
# total distance travelled, state, etc.
rInitial = numpy.copy(r)

# Fills velocity array with random velocities homogeneously
# distributed between -3/2 and 3/2
z = numpy.ndenumerate(v)
for x, y in z:
    v[x[0]][x[1]] = (random.random() * 3.0 - 1.5) * initialDv


# functions for finding force/acceleration
def forcex(dx, dy):
    return -24 * dx * ((dx ** 2 + dy ** 2) ** 3 - 2) / (dx ** 2 + dy ** 2) ** 7
```

```
def forcey(dx, dy):
    return -24.0 * dy * ((dx ** 2 + dy ** 2) ** 3 - 2.0) / (dx ** 2 + dy ** 2) ** 7


# multiple runs for getting multiple values of s array for the same time step
def run():
    global dt
    global initialDr
    global boxWidth
    global boxHeight
    global particleNumber
    global timeLength
    global vRange
    global initialDv
    global r2
    global s
    r2 = []
    # iterate through arrays with time step
    for t in range(int(timeLength / dt)):
        rPrev = numpy.copy(r)
        # Calculating the r array using velocities and accelerations
        # found in the previous time step. Makes use of a periodic boundary
        # condition
        for i in range(particleNumber):
            r[i][0] = rPrev[i][0] + dt * v[i][0] + dt ** 2 / 2 * a[i][0]
            r[i][1] = rPrev[i][1] + dt * v[i][1] + dt ** 2 / 2 * a[i][1]
            if r[i][0] > boxWidth:
                r[i][0] -= boxWidth
            elif r[i][0] < 0:
                r[i][0] += boxWidth
            elif r[i][1] > boxHeight:
                r[i][1] -= boxHeight
            elif r[i][1] < 0:
                r[i][1] += boxHeight

        # Adding pseudo-particles to each boundary of box to allow for
        # periodic forces
        rPseudo = numpy.copy(r)
        for particle in range(particleNumber):
            # left edge
            if vRange >= rPseudo[particle, 0] >= 0:
                rPseudo = numpy.append(rPseudo, [[r[particle, 0] + boxWidth, r[particle, 1]]], axis=0)
            # right edge
            if boxWidth >= rPseudo[particle, 0] >= boxWidth - vRange:
                rPseudo = numpy.append(rPseudo, [[r[particle, 0] - boxWidth, r[particle, 1]]], axis=0)
            # bottom edge
            if vRange >= rPseudo[particle, 1] >= 0:
                rPseudo = numpy.append(rPseudo, [[r[particle, 0], r[particle, 1] + boxHeight]], axis=0)
            # top edge
            if boxHeight >= rPseudo[particle, 1] >= boxHeight - vRange:
                rPseudo = numpy.append(rPseudo, [[r[particle, 0], r[particle, 1] - boxHeight]], axis=0)
            # bottom left corner
            if (vRange >= rPseudo[particle, 0] >= 0) and (vRange >= rPseudo[particle, 1] >= 0):
                rPseudo = numpy.append(rPseudo, [[r[particle, 0] + boxWidth, r[particle, 1] +
                boxHeight]], axis=0)
            # bottom right corner
            if (boxWidth >= rPseudo[particle, 0] >= boxWidth - vRange) and
            (vRange >= rPseudo[particle, 1] >= 0):
                rPseudo = numpy.append(rPseudo, [[r[particle, 0] - boxWidth, r[particle, 1] +
                boxHeight]], axis=0)
            # top left corner
            if (vRange >= rPseudo[particle, 0] >= 0) and
            (boxHeight >= rPseudo[particle, 1] >= boxHeight - vRange):
                rPseudo = numpy.append(rPseudo, [[r[particle, 0] + boxWidth, r[particle, 1] -
                boxHeight]], axis=0)
            # top right corner
            if (boxWidth >= rPseudo[particle, 0] >= boxWidth - vRange) and
            (boxHeight >= rPseudo[particle, 1] >= boxHeight - vRange):
                rPseudo = numpy.append(rPseudo, [[r[particle, 0] - boxWidth, r[particle, 1] -
                boxHeight]], axis=0)

        aPrev = numpy.copy(a)
        # Calculating force/acceleration matrix using potential function
        # and the pseudo particles created above for the boundary conditions
        for i in range(particleNumber):
            a[i][0] = 0
            a[i][1] = 0
            x = rPseudo[i][0]
            y = rPseudo[i][1]
            for j in range(rPseudo.shape[0]):
                # might need to switch particle i and j around...
                dx = x - rPseudo[j][0]
                dy = y - rPseudo[j][1]
                if math.sqrt(dx ** 2 + dy ** 2) < vRange and dx != 0 and dy != 0:
                    a[i][0] += forcex(dx, dy)
                    a[i][1] += forcey(dx, dy)

        vPrev = numpy.copy(v)
        # Calculating the velocities of each particle using the
        # current acceleration, previous acceleration, and previous velocity
        for i in range(particleNumber):
            v[i][0] = vPrev[i][0] + dt / 2 * (aPrev[i][0] + a[i][0])
            v[i][1] = vPrev[i][1] + dt / 2 * (aPrev[i][1] + a[i][1])


        r2Placeholder = 0
        # Calculating distance (squared) each particle has travelled so far
        # and returning the average at this time step
        for i in range(particleNumber):
            dim = numpy.array([boxWidth, boxHeight])
            diff = numpy.abs(rInitial[i] - r[i])
            diff = numpy.where(diff > 0.5 * dim, dim - diff, diff)
            r2Placeholder += (diff[0] ** 2 + diff[1] ** 2) / particleNumber
        r2.append(r2Placeholder)

        # Calculating speed of each particle
        if t == timeLength / dt - 1:
            for i in range(particleNumber):
                s.append(math.sqrt(v[i][0] ** 2 + v[i][1] ** 2))

for i in range(300):
    run()
    print i

T = sum([i ** 2 for i in s]) / 2 / len(s)

def fv(v, T):
    return math.sqrt(2 / math.pi) * v ** 2 / math.pow(T, 3.0 / 2.0) * math.pow(math.e, - v ** 2 / (2 * T))
```

```
print "T = " + str(T)

v = numpy.arange(0., max(s), .01)
o = numpy.vectorize(fv)

r2 = numpy.array(r2)
t = numpy.arange(0., timeLength, dt)

fit = numpy.polyfit(t, r2, 1)
output = numpy.poly1d(fit)

print "fit: " + str(fit)

f = plt.figure(1)
af = f.add_subplot(111)
plt.title(r'$\langle r^2 \rangle$ vs $\Delta t$')
plt.xlabel(r'$\Delta t$')
plt.ylabel(r'$\langle r^2 \rangle$')
af.plot(t, r2, alpha=.5, c="green")  # plots average distance travelled vs time
af.plot(t, output(t), c="red")
f.canvas.draw()

g = plt.figure(2)
ag = g.add_subplot(111)
plt.title(r'Velocity vs Occurrence')
plt.xlabel(r'Velocity')
plt.ylabel(r'Occurrence')
ag.hist(s, 100, align='mid', normed=1)  # plots histogram of speeds
ag.plot(v, o(v, T))  # plots maxwell-boltzmann distribution
g.canvas.draw()

h = plt.figure(3)
ah = h.add_subplot(111)
x = numpy.split(r, 2, axis=1)[0]  # x array of positions
y = numpy.split(r, 2, axis=1)[1]  # y array of positions
ah.scatter(x, y)  # plots x and y coordinates
h.canvas.draw()

plt.show()
raw_input()
```

B Problem 2 Code

```
#!/usr/bin/python

import os
import math
import numpy
import random
import matplotlib.pyplot as plt
import hickle as hkl
import matplotlib.animation as animation

# Initialize variables
dt = .001  # .01 for problem 1 and .005 for problem 2 and 3
initialDr = 0.0  # .01 for problem 1 and 0 for problem 2 and 3
boxWidth = 4.0  # 4.0 for problem 2 and 3, 10.0 for problem 1
boxHeight = 4.0  # 4.0 for problem 2 and 3, 10.0 for problem 1
particleNumber = 16  # should be a square number
```

```
timeLength = 15.0  # 15.0 is length of time required for part two
vRange = 3.0
initialDv = .0001  # 1 for problem 1 and .0001 for problem 2 and 3
r2 = []  # average distance squared list, in order of time steps
s = []  # list of speeds for each particle

# initialize position, velocity,
# and acceleration arrays
r = numpy.zeros((particleNumber, 2))
v = numpy.zeros((particleNumber, 2))
a = numpy.zeros((particleNumber, 2))

# filling r array with equally spaced particles
w = numpy.ndenumerate(r)
count = 0
for x, y in w:
    xStep = boxWidth / float(math.sqrt(particleNumber))
    yStep = boxHeight / float(math.sqrt(particleNumber))
    if x[1] == 0:
        r[x[0]][x[1]] = count / int(math.sqrt(particleNumber)) * \
                        xStep + (random.random() - .5) * initialDr
        count += 1
    else:
        r[x[0]][x[1]] = x[0] % math.sqrt(particleNumber) * \
                        yStep + (random.random() - .5) * initialDr

# Saving initial r array for use later in calculating
# total distance travelled, state, etc.
rInitial = numpy.copy(r)

# Fills velocity array with random velocities homogeneously distributed
z = numpy.ndenumerate(v)
for x, y in z:
    v[x[0]][x[1]] = (random.random() - .5) * initialDv


# functions for finding force/acceleration
def forcex(dx, dy):
    return -24.0 * dx * ((dx ** 2 + dy ** 2) ** 3 - 2) / (dx ** 2 + dy ** 2) ** 7


def forcey(dx, dy):
    return -24.0 * dy * ((dx ** 2 + dy ** 2) ** 3 - 2.0) / (dx ** 2 + dy ** 2) ** 7


# multiple runs for getting multiple values of s array for the same time step
def run():
    global dt
    global initialDr
    global boxWidth
    global boxHeight
    global particleNumber
    global timeLength
    global vRange
    global initialDv
    global r2
    global s
    # iterate through arrays with time step
    for t in range(int(timeLength / dt)):
        rPrev = numpy.copy(r)
```

```
# Calculating the r array using velocities and accelerations
# found in the previous time step. Makes use of a periodic boundary
# condition
for i in range(particleNumber):
    r[i][0] = rPrev[i][0] + dt * v[i][0] + dt ** 2 / 2 * a[i][0]
    r[i][1] = rPrev[i][1] + dt * v[i][1] + dt ** 2 / 2 * a[i][1]
    if r[i][0] > boxWidth:
        r[i][0] -= boxWidth
    elif r[i][0] < 0:
        r[i][0] += boxWidth
    elif r[i][1] > boxHeight:
        r[i][1] -= boxHeight
    elif r[i][1] < 0:
        r[i][1] += boxHeight

# Adding pseudo-particles to each boundary of box to allow for
# periodic forces
rPseudo = numpy.copy(r)
for particle in range(particleNumber):
    # left edge
    if vRange >= rPseudo[particle, 0] >= 0:
        rPseudo = numpy.append(rPseudo, [[r[particle, 0] + boxWidth, r[particle, 1]]], axis=0)
    # right edge
    if boxWidth >= rPseudo[particle, 0] >= boxWidth - vRange:
        rPseudo = numpy.append(rPseudo, [[r[particle, 0] - boxWidth, r[particle, 1]]], axis=0)
    # bottom edge
    if vRange >= rPseudo[particle, 1] >= 0:
        rPseudo = numpy.append(rPseudo, [[r[particle, 0], r[particle, 1] + boxHeight]], axis=0)
    # top edge
    if boxHeight >= rPseudo[particle, 1] >= boxHeight - vRange:
        rPseudo = numpy.append(rPseudo, [[r[particle, 0], r[particle, 1] - boxHeight]], axis=0)
    # bottom left corner
    if (vRange >= rPseudo[particle, 0] >= 0) and (vRange >= rPseudo[particle, 1] >= 0):
        rPseudo = numpy.append(rPseudo, [[r[particle, 0] + boxWidth, r[particle, 1] +
        boxHeight]], axis=0)
    # bottom right corner
    if (boxWidth >= rPseudo[particle, 0] >= boxWidth - vRange) and
    (vRange >= rPseudo[particle, 1] >= 0):
        rPseudo = numpy.append(rPseudo, [[r[particle, 0] - boxWidth, r[particle, 1] +
        boxHeight]], axis=0)
    # top left corner
    if (vRange >= rPseudo[particle, 0] >= 0) and
    (boxHeight >= rPseudo[particle, 1] >= boxHeight - vRange):
        rPseudo = numpy.append(rPseudo, [[r[particle, 0] + boxWidth, r[particle, 1] -
        boxHeight]], axis=0)
    # top right corner
    if (boxWidth >= rPseudo[particle, 0] >= boxWidth - vRange) and
    (boxHeight >= rPseudo[particle, 1] >= boxHeight - vRange):
        rPseudo = numpy.append(rPseudo, [[r[particle, 0] - boxWidth, r[particle, 1] -
        boxHeight]], axis=0)

aPrev = numpy.copy(a)
# Calculating force/acceleration matrix using potential function
# and the pseudo particles created above for the boundary conditions
for i in range(particleNumber):
    a[i][0] = 0
    a[i][1] = 0
    x = rPseudo[i][0]
    y = rPseudo[i][1]
    for j in range(rPseudo.shape[0]):
```

```
                # might need to switch particle i and j around...
                dx = x - rPseudo[j][0]
                dy = y - rPseudo[j][1]
                if math.sqrt(dx ** 2 + dy ** 2) < vRange and dx != 0 and dy != 0:
                    a[i][0] += forcex(dx, dy)
                    a[i][1] += forcey(dx, dy)

        vPrev = numpy.copy(v)
        # Calculating the velocities of each particle using the
        # current acceleration, previous acceleration, and previous velocity
        for i in range(particleNumber):
            v[i][0] = vPrev[i][0] + dt / 2 * (aPrev[i][0] + a[i][0])
            v[i][1] = vPrev[i][1] + dt / 2 * (aPrev[i][1] + a[i][1])


        r2Placeholder = 0
        # Calculating distance (squared) each particle has travelled so far
        # and returning the average at this time step
        for i in range(particleNumber):
            dim = numpy.array([boxWidth, boxHeight])
            diff = numpy.abs(rInitial[i] - r[i])
            diff = numpy.where(diff > 0.5 * dim, dim - diff, diff)
            r2Placeholder += (diff[0] ** 2 + diff[1] ** 2) / particleNumber
        r2.append(r2Placeholder)

        # Calculating speed of each particle
        if t == timeLength / dt - 1:
            for i in range(particleNumber):
                s.append(math.sqrt(v[i][0] ** 2 + v[i][1] ** 2))

for i in range(1):  # 300 looks nice
    run()
    print i

T = sum([i ** 2 for i in s]) / 2 / len(s)

def fv(v, T):
    return math.sqrt(2 / math.pi) * v ** 2 / math.pow(T, 3.0 / 2.0) * math.pow(math.e, - v ** 2 / (2 * T))


print T

#hkl.dump(r, 'coords.hkl')

v = numpy.arange(0., max(s), .01)
o = numpy.vectorize(fv)

r2 = numpy.array(r2)
t = numpy.arange(0., timeLength, dt)

f = plt.figure(1)
af = f.add_subplot(111)
plt.title(r'$\langle r^2 \rangle$ vs $\Delta t$')
plt.xlabel(r'$\Delta t$')
plt.ylabel(r'$\langle r^2 \rangle$')
af.plot(t, r2, alpha=.75, c="green")  # plots average distance travelled vs time
f.canvas.draw()

g = plt.figure(2)
ag = g.add_subplot(111)
```

```
ag.hist(s, 100, align='mid', normed=1)  # plots histogram of speeds
ag.plot(v, o(v, T))  # plots maxwell-boltzmann distribution
g.canvas.draw()

h = plt.figure(3)
ah = h.add_subplot(111)
plt.title(r'Particle Positions at Time = 15')
plt.xlabel(r'X')
plt.ylabel(r'Y')
colors = numpy.random.rand(particleNumber)
x = numpy.split(r, 2, axis=1)[0]  # x array of positions
y = numpy.split(r, 2, axis=1)[1]  # y array of positions
ah.scatter(x, y, s=150 * numpy.pi, c=colors, alpha=0.5)  # plots x and y coordinates
h.canvas.draw()

plt.show()
raw_input()
```

C Problem 3 Code

```
#!/usr/bin/python

import os
import math
import numpy
import random
import matplotlib.pyplot as plt
import hickle as hkl
import matplotlib.animation as animation

# Initialize variables
initialDr = 0.0  # .01 for problem 1 and 0 for problem 2 and 3
boxWidth = 4.0  # 4.0 for problem 2 and 3, 10.0 for problem 1
boxHeight = 4.0  # 4.0 for problem 2 and 3, 10.0 for problem 1
particleNumber = 16  # should be a square number
timeLength = 5.0 # 15.0 is length of time required for part two
vRange = 3.0
initialDv = 1  # 1 for problem 1 and .0001 for problem 2 and .75 to melt in 3
dt = .005  # / (initialDv/.0001)  # .01 for problem 1 and .005 for problem 2 and 3
r2 = []  # average distance squared list, in order of time steps
s = []  # list of speeds for each particle

# initialize position, velocity,
# and acceleration arrays
r = hkl.load("coords.hkl", safe=True)
v = numpy.zeros((particleNumber, 2))
a = numpy.zeros((particleNumber, 2))

# Saving initial r array for use later in calculating
# total distance travelled, state, etc.
rInitial = numpy.copy(r)

# Fills velocity array with random velocities homogeneously
# distributed between -3/2 and 3/2
```

```
z = numpy.ndenumerate(v)
for x, y in z:
    v[x[0]][x[1]] = (random.random() - .5) * initialDv


# functions for finding force/acceleration
def forcex(dx, dy):
    return -24.0 * dx * ((dx ** 2 + dy ** 2) ** 3 - 2) / (dx ** 2 + dy ** 2) ** 7


def forcey(dx, dy):
    return -24.0 * dy * ((dx ** 2 + dy ** 2) ** 3 - 2.0) / (dx ** 2 + dy ** 2) ** 7


# multiple runs for getting multiple values of s array for the same time step
def run():
    global dt
    global initialDr
    global boxWidth
    global boxHeight
    global particleNumber
    global timeLength
    global vRange
    global initialDv
    global r2
    global s
    s = []
    # iterate through arrays with time step
    for t in range(int(timeLength / dt)):
        print float(t)/(timeLength / dt)
        rPrev = numpy.copy(r)
        # Calculating the r array using velocities and accelerations
        # found in the previous time step. Makes use of a periodic boundary
        # condition
        for i in range(particleNumber):
            r[i][0] = rPrev[i][0] + dt * v[i][0] + dt ** 2 / 2 * a[i][0]
            r[i][1] = rPrev[i][1] + dt * v[i][1] + dt ** 2 / 2 * a[i][1]
            if r[i][0] > boxWidth:
                r[i][0] -= boxWidth
            elif r[i][0] < 0:
                r[i][0] += boxWidth
            elif r[i][1] > boxHeight:
                r[i][1] -= boxHeight
            elif r[i][1] < 0:
                r[i][1] += boxHeight

        # Adding pseudo-particles to each boundary of box to allow for
        # periodic forces
        rPseudo = numpy.copy(r)
        for particle in range(particleNumber):
            # left edge
            if vRange >= rPseudo[particle, 0] >= 0:
                rPseudo = numpy.append(rPseudo, [[r[particle, 0] + boxWidth, r[particle, 1]]], axis=0)
            # right edge
            if boxWidth >= rPseudo[particle, 0] >= boxWidth - vRange:
                rPseudo = numpy.append(rPseudo, [[r[particle, 0] - boxWidth, r[particle, 1]]], axis=0)
            # bottom edge
            if vRange >= rPseudo[particle, 1] >= 0:
                rPseudo = numpy.append(rPseudo, [[r[particle, 0], r[particle, 1] + boxHeight]], axis=0)
            # top edge
            if boxHeight >= rPseudo[particle, 1] >= boxHeight - vRange:
                rPseudo = numpy.append(rPseudo, [[r[particle, 0], r[particle, 1] - boxHeight]], axis=0)
            # bottom left corner
            if (vRange >= rPseudo[particle, 0] >= 0) and (vRange >= rPseudo[particle, 1] >= 0):
                rPseudo = numpy.append(rPseudo, [[r[particle, 0] + boxWidth, r[particle, 1] +
                boxHeight]], axis=0)
            # bottom right corner
            if (boxWidth >= rPseudo[particle, 0] >= boxWidth - vRange) and
            (vRange >= rPseudo[particle, 1] >= 0):
                rPseudo = numpy.append(rPseudo, [[r[particle, 0] - boxWidth, r[particle, 1] +
                boxHeight]], axis=0)
            # top left corner
            if (vRange >= rPseudo[particle, 0] >= 0) and
            (boxHeight >= rPseudo[particle, 1] >= boxHeight - vRange):
                rPseudo = numpy.append(rPseudo, [[r[particle, 0] + boxWidth, r[particle, 1] -
                boxHeight]], axis=0)
            # top right corner
            if (boxWidth >= rPseudo[particle, 0] >= boxWidth - vRange) and
            (boxHeight >= rPseudo[particle, 1] >= boxHeight - vRange):
                rPseudo = numpy.append(rPseudo, [[r[particle, 0] - boxWidth, r[particle, 1] -
                boxHeight]], axis=0)

        aPrev = numpy.copy(a)
        # Calculating force/acceleration matrix using potential function
        # and the pseudo particles created above for the boundary conditions
        for i in range(particleNumber):
            a[i][0] = 0
            a[i][1] = 0
            x = rPseudo[i][0]
            y = rPseudo[i][1]
            for j in range(rPseudo.shape[0]):
                # might need to switch particle i and j around...
                dx = x - rPseudo[j][0]
                dy = y - rPseudo[j][1]
                if math.sqrt(dx ** 2 + dy ** 2) < vRange and dx != 0 and dy != 0:
                    a[i][0] += forcex(dx, dy)
                    a[i][1] += forcey(dx, dy)

        vPrev = numpy.copy(v)
        # Calculating the velocities of each particle using the
        # current acceleration, previous acceleration, and previous velocity
        for i in range(particleNumber):
            v[i][0] = vPrev[i][0] + dt / 2 * (aPrev[i][0] + a[i][0])
            v[i][1] = vPrev[i][1] + dt / 2 * (aPrev[i][1] + a[i][1])


        r2Placeholder = 0
        # Calculating distance (squared) each particle has travelled so far
        # and returning the average at this time step
        for i in range(particleNumber):
            dim = numpy.array([boxWidth, boxHeight])
            diff = numpy.abs(rInitial[i] - r[i])
            diff = numpy.where(diff > 0.5 * dim, dim - diff, diff)
            r2Placeholder += (diff[0] ** 2 + diff[1] ** 2) / particleNumber
        r2.append(r2Placeholder)

        # Calculating speed of each particle
        if t == timeLength / dt - 1:
            for i in range(particleNumber):
                s.append(math.sqrt(v[i][0] ** 2 + v[i][1] ** 2))
```

```
run()

T = sum([i ** 2 for i in s]) / 2 / len(s)

def fv(v, T):
    return math.sqrt(2 / math.pi) * v ** 2 / math.pow(T, 3.0 / 2.0) * math.pow(math.e, - v ** 2 / (2 * T))


print "T = " + str(T)

#hkl.dump(r, 'coords.hkl')

v = numpy.arange(0., max(s), .01)
o = numpy.vectorize(fv)

r2 = numpy.array(r2)
t = numpy.arange(0., timeLength, dt)

f = plt.figure(1)
af = f.add_subplot(111)
plt.title(r'$\langle r^2 \rangle$ vs $\Delta t$')
plt.xlabel(r'$\Delta t$')
plt.ylabel(r'$\langle r^2 \rangle$')
af.plot(t, r2, alpha=.75, c="green")  # plots average distance travelled vs time
f.canvas.draw()

g = plt.figure(2)
ag = g.add_subplot(111)
ag.hist(s, 100, align='mid', normed=1)  # plots histogram of speeds
ag.plot(v, o(v, T))  # plots maxwell-boltzmann distribution
g.canvas.draw()

h = plt.figure(3)
ah = h.add_subplot(111)
plt.title(r'Particle Positions at Time = 15')
plt.xlabel(r'X')
plt.ylabel(r'Y')
colors = numpy.random.rand(particleNumber)
x = numpy.split(r, 2, axis=1)[0]  # x array of positions
y = numpy.split(r, 2, axis=1)[1]  # y array of positions
ah.scatter(x, y, s=150 * numpy.pi, c=colors, alpha=0.5)  # plots x and y coordinates
h.canvas.draw()

plt.show()
raw_input()
```