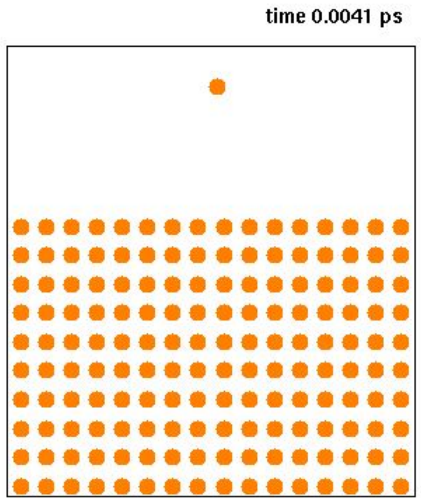
Molecular Dynamic Simulations

Jordan Stomps
CMSE 201
Honors Option

Molecular Dynamic Simulations

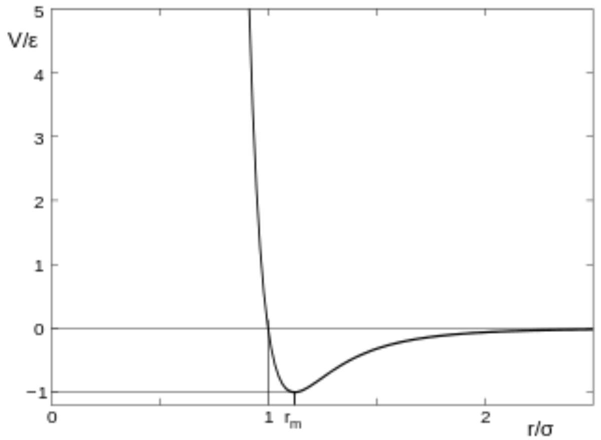
- Used to model systems of particles over time based on the physical laws desired to determine their behavior.
- Broad topic, can be used for Chemistry, Physics, Engineering, etc.



Lennard-Jones Potential

- Often seen in Chemistry
- Used to describe the attractive-repellant forces of two particles
- Good for describing particle interactions in a simplified system

$$U(r) = 4\varepsilon((\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6})$$



- If we want to calculate the motion of particles in a system, this potential energy can be devolved into a force...

$$\frac{F}{m} = -(\frac{dU}{dr}) = 4\varepsilon((\frac{6\sigma}{r})^{7} - (\frac{12\sigma}{r})^{13})$$

Reduced Units

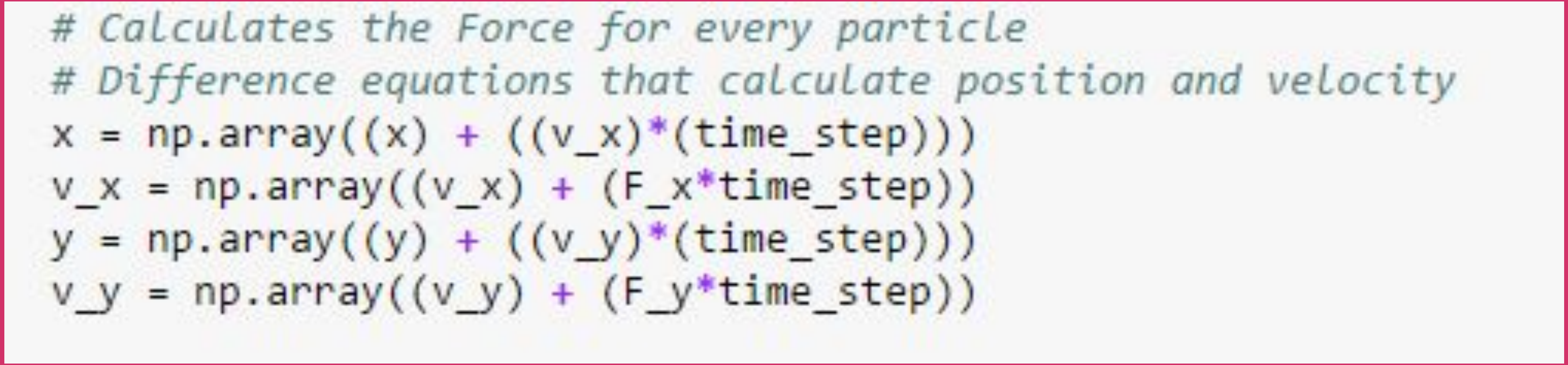
- Computers will have a hard time reading and calculating values of large magnitude
- By turning all values into a *reduced units* form, the computer can interpret the variables more clearly
- For this MDS, we will be using *Argon*

$$T = time - unit = \sigma\sqrt{\frac{m}{\varepsilon}}$$

$$r* = \frac{r}{\sigma} \qquad \Delta t* = \frac{\Delta t}{T}$$

$$F* = 4((\frac{6}{r*})^7 - (\frac{12}{r*})^{13})$$

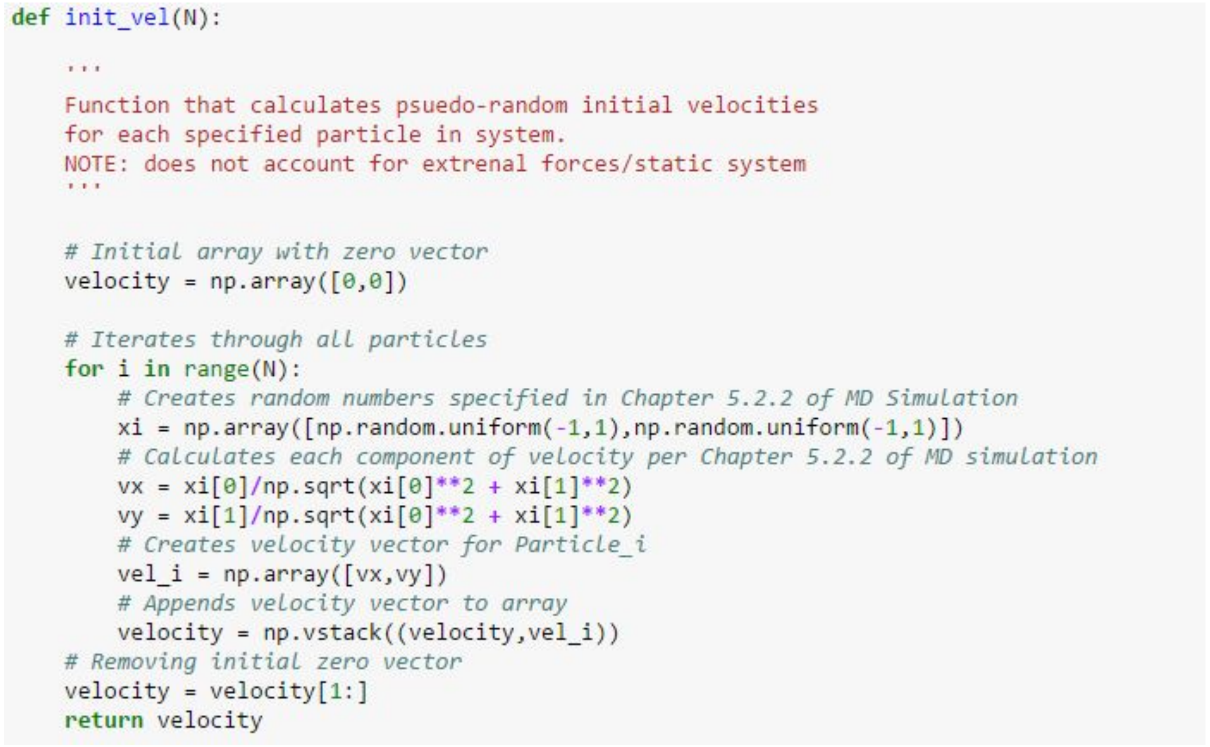
Difference Equations (ODE)



```
# Calculates the Force for every particle
# Difference equations that calculate position and velocity
x = np.array((x) + ((v_x)*(time_step)))
v_x = np.array((v_x) + (F_x*time_step))
y = np.array((y) + ((v_y)*(time_step)))
v_y = np.array((v_y) + (F_y*time_step))
```

Initializing Velocity

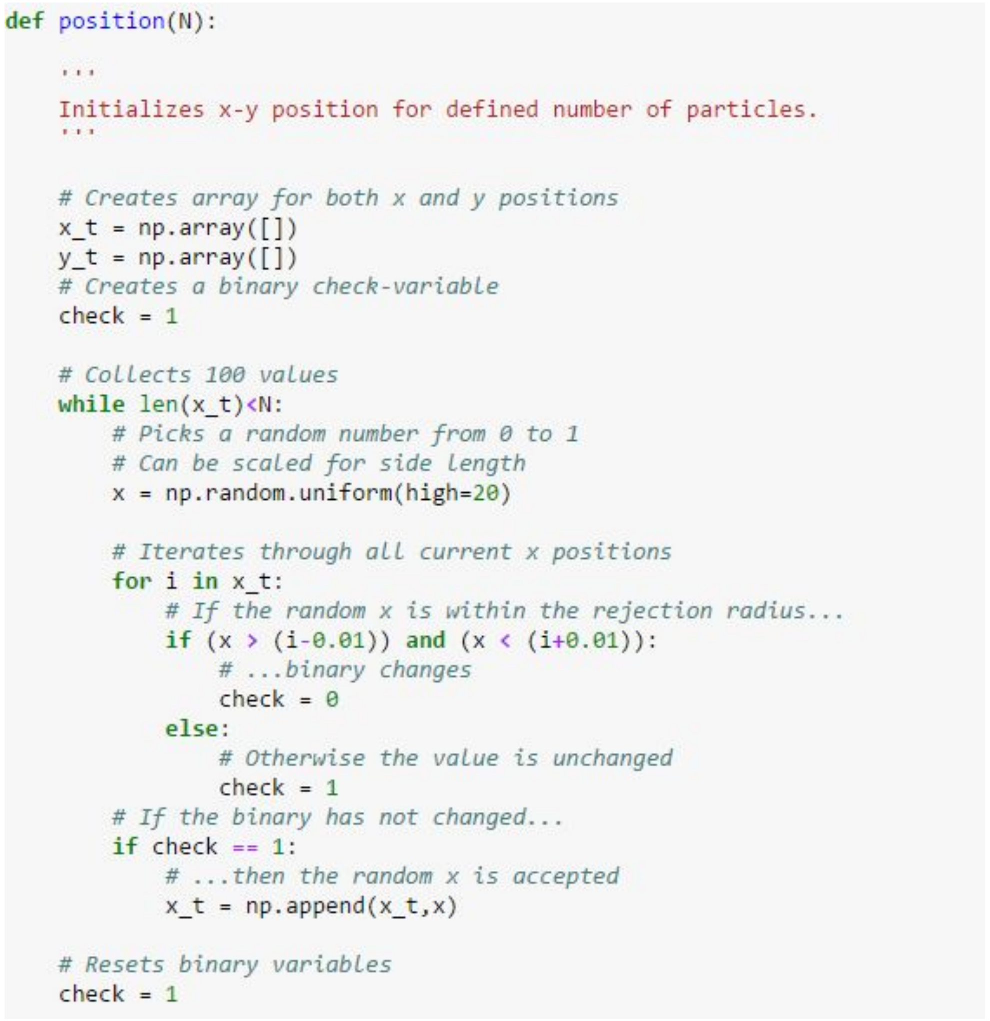
- The values for each random number are normalized so that even if the directions/values of velocity are random, they are devolve to a velocity of 1



```python
def init_vel(N):
    '''
    Function that calculates psuedo-random initial velocities
    for each specified particle in system.
    NOTE: does not account for extrenal forces/static system
    '''

    # Initial array with zero vector
    velocity = np.array([0,0])

    # Iterates through all particles
    for i in range(N):
        # Creates random numbers specified in Chapter 5.2.2 of MD Simulation
        xi = np.array([np.random.uniform(-1,1),np.random.uniform(-1,1)])
        # Calculates each component of velocity per Chapter 5.2.2 of MD simulation
        vx = xi[0]/np.sqrt(xi[0]**2 + xi[1]**2)
        vy = xi[1]/np.sqrt(xi[0]**2 + xi[1]**2)
        # Creates velocity vector for Particle_i
        vel_i = np.array([vx,vy])
        # Appends velocity vector to array
        velocity = np.vstack((velocity,vel_i))
    # Removing initial zero vector
    velocity = velocity[1:]
    return velocity
```

Initializing Position

- To avoid particles from overlapping in the initialization of the program, introduce a *rejection radius*

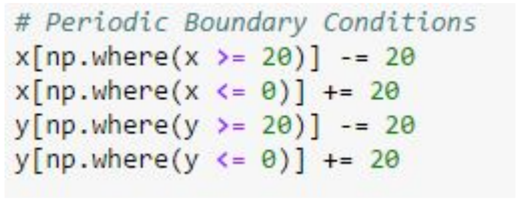


```
def position(N):
    '''
    Initializes x-y position for defined number of particles.
    '''

    # Creates array for both x and y positions
    x_t = np.array([])
    y_t = np.array([])
    # Creates a binary check-variable
    check = 1

    # Collects 100 values
    while len(x_t)<N:
        # Picks a random number from 0 to 1
        # Can be scaled for side length
        x = np.random.uniform(high=20)

        # Iterates through all current x positions
        for i in x_t:
            # If the random x is within the rejection radius...
            if (x > (i-0.01)) and (x < (i+0.01)):
                # ...binary changes
                check = 0
            else:
                # Otherwise the value is unchanged
                check = 1
        # If the binary has not changed...
        if check == 1:
            # ...then the random x is accepted
            x_t = np.append(x_t,x)

    # Resets binary variables
    check = 1
```

Boundary Conditions

<u>Period Boundary Conditions:</u>

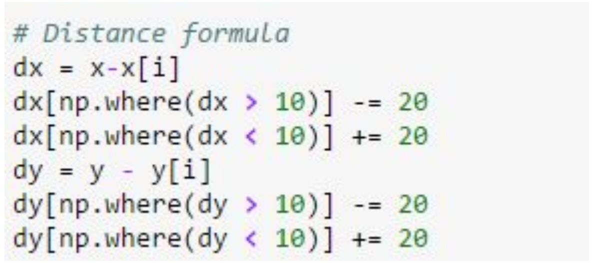
- Used to confine the particles to a specific volume (maintain density) without changing velocity values



```
# Periodic Boundary Conditions
x[np.where(x >= 20)] -= 20
x[np.where(x <= 0)] += 20
y[np.where(y >= 20)] -= 20
y[np.where(y <= 0)] += 20
```

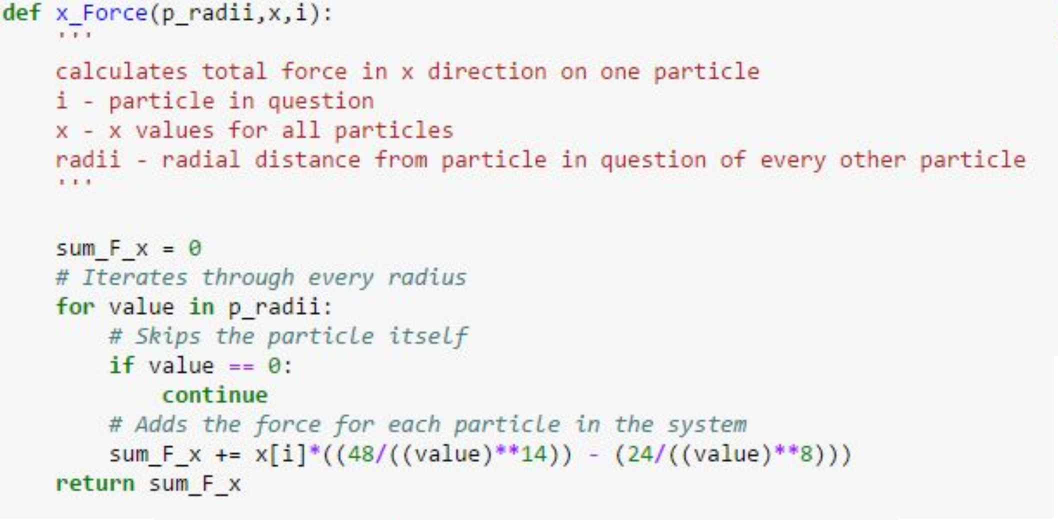
<u>Minimum Image Convention:</u>

- Used to model the larger “bulk” of the system
- The evaluations below only affect the calculation of radius → force → velocity

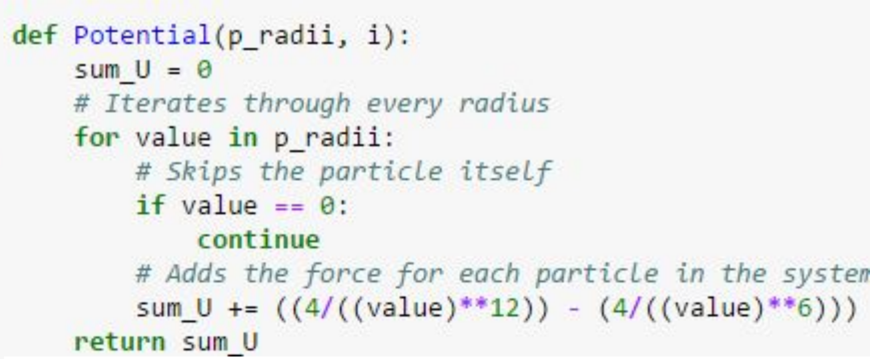


```
# Distance formula
dx = x-x[i]
dx[np.where(dx > 10)] -= 20
dx[np.where(dx < 10)] += 20
dy = y - y[i]
dy[np.where(dy > 10)] -= 20
dy[np.where(dy < 10)] += 20
```

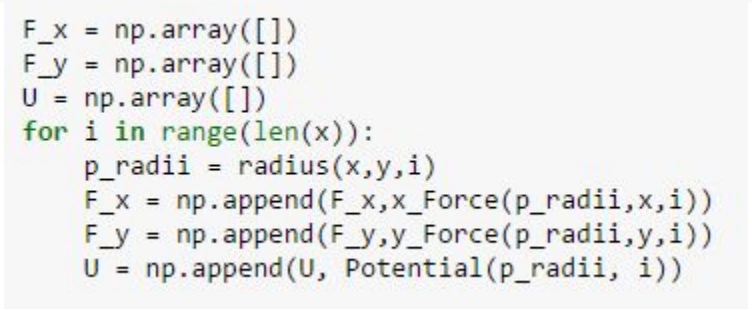
Calculating Force and Potential



```python
def x_Force(p_radii,x,i):
    '''
    calculates total force in x direction on one particle
    i - particle in question
    x - x values for all particles
    radii - radial distance from particle in question of every other particle
    '''

    sum_F_x = 0
    # Iterates through every radius
    for value in p_radii:
        # Skips the particle itself
        if value == 0:
            continue
        # Adds the force for each particle in the system
        sum_F_x += x[i]*((48/((value)**14)) - (24/((value)**8)))
    return sum_F_x
```

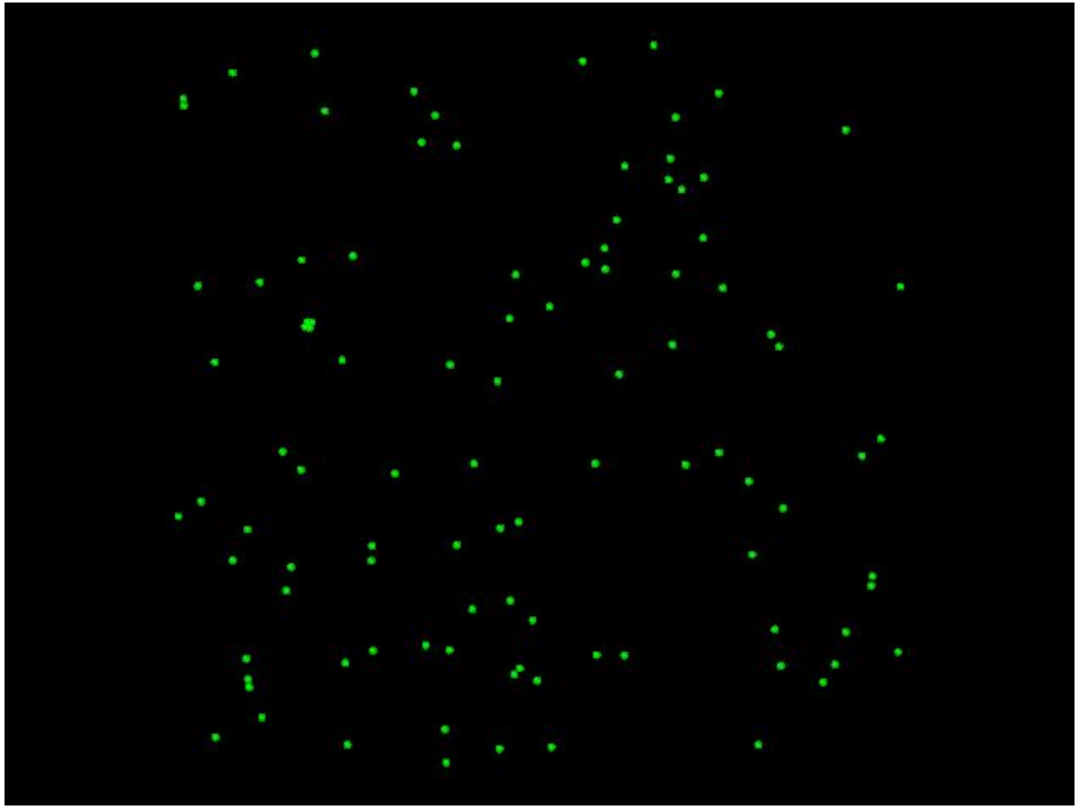


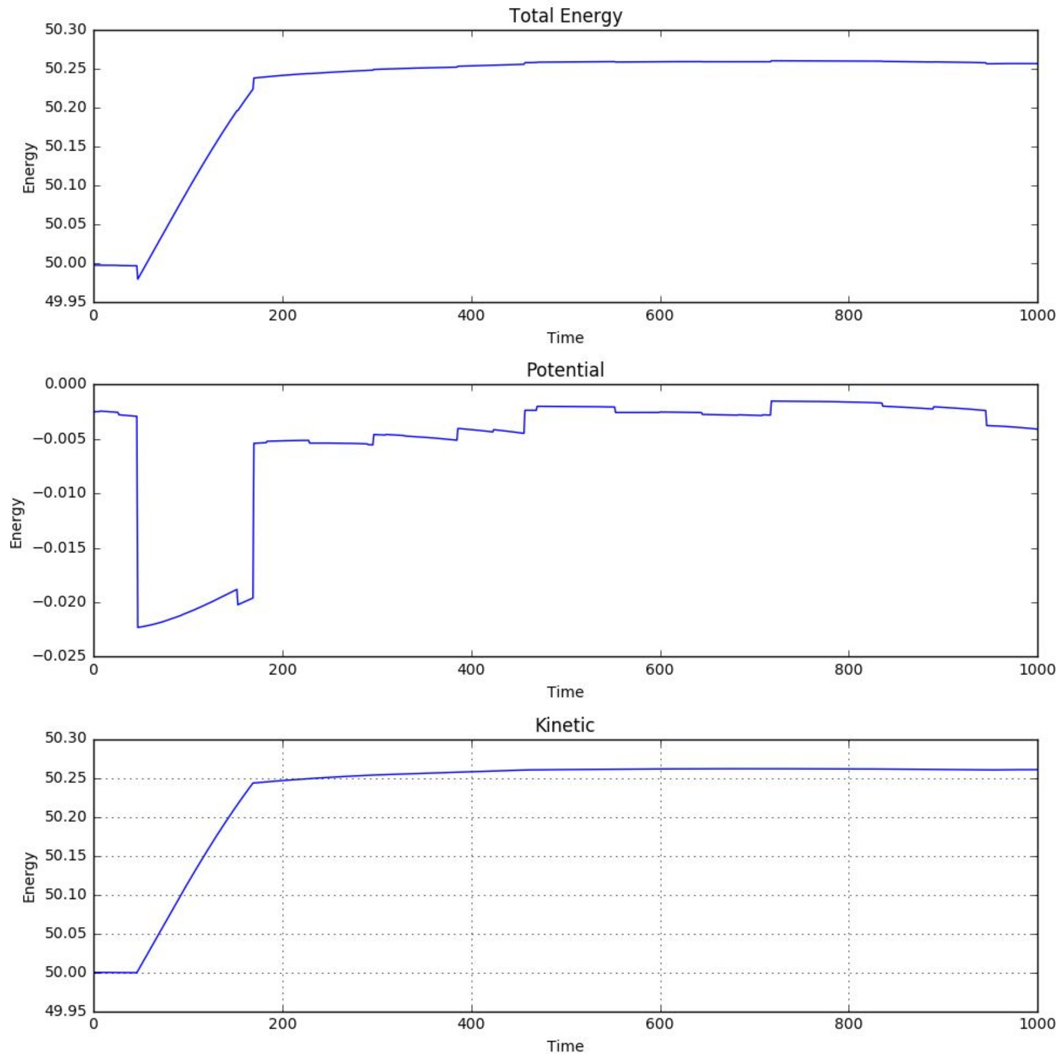
```python
def Potential(p_radii, i):
    sum_U = 0
    # Iterates through every radius
    for value in p_radii:
        # Skips the particle itself
        if value == 0:
            continue
        # Adds the force for each particle in the system
        sum_U += ((4/((value)**12)) - (4/((value)**6)))
    return sum_U
```



```python
F_x = np.array([])
F_y = np.array([])
U = np.array([])
for i in range(len(x)):
    p_radii = radius(x,y,i)
    F_x = np.append(F_x,x_Force(p_radii,x,i))
    F_y = np.append(F_y,y_Force(p_radii,y,i))
    U = np.append(U, Potential(p_radii, i))
```

Ovito - program used for animation





Graphs of Energy over Time

So what?

- The hope would be that as particles get closer to an *equilibrium*, they create orbitals of particles where the probable distance of one particle being from another particle is high
- This program can be extended for many different systems:
    - Open system
    - Different Temperature → Energy
    - Different type of particle