Molecular Dynamics Simulation: Lennard-Jones Cluster Collision

C.T.Hutchinson

December 16, 2016

Abstract

The use and design of a simple Molecular Dynamics program written in Python 3 is demonstrated to model the collision of two Lennard-Jones clusters. The system is evolved in time by integration of Newton's equations of motion via the Velocity Verlet algorithm. The two clusters are found to coalesce within 50 nanoseconds when the time-step is set to 1 picosecond, and initialised at distance 5 and incoming velocity 1e8.

Part I
Introduction

In this paper we model the collision of two Lennard-Jones clusters in dimensionless units through a Molecular Dynamics simulation code. Two clusters of particle size 70 and 77 [1] are used in the simulation, with one arbitrarily centred at the origin of the co-ordinate system. The other cluster is displaced some distance in the x-axis direction, with an initial velocity chosen by the user. A user assigned initial separation can be seen below in the 3D plot shown in Figure 1.

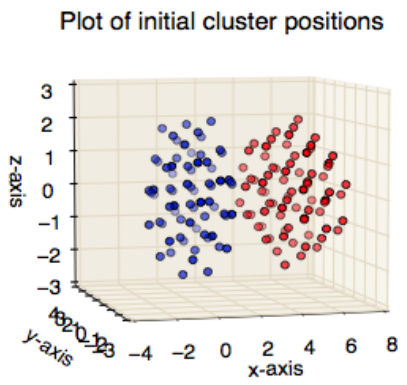

Figure 1: A plot of the initial positions of the two Lennard-Jones clusters.

The simulation is then allowed to proceed during a predetermined time interval of 50 nanoseconds with the force and energy at each particle calculated for each time step of 1 femtosecond [2], according to the Lennard-Jones potential:

$$U = 4\epsilon\Big[\Big(\frac{\sigma}{r}\Big)^{12} - \Big(\frac{\sigma}{r}\Big)^{6}\Big]$$

With ϵ the potential well depth, σ the distance at which the inter-particle potential is zero and r the distance between the two particles.

In reduced units:

$$\sigma, \epsilon = 1$$

And therefore the equation for the Lennard-Jones potential becomes:

$$U = 4\Big[\Big(\frac{1}{r}\Big)^{12} - \Big(\frac{1}{r}\Big)^{6}\Big]$$

Knowing the force is the derivative of the potential we can derive an expression for the component of force on each particle in each direction, due to the others. Suppose we wish to calculate the x-component of the force [3]:

$$F_x(r) = -\frac{\partial U(r)}{\partial x} = -\Big(\frac{x}{r}\Big)\Big(\frac{\partial U(r)}{\partial r}\Big)$$

Which yields:

$$F_x(r) = \frac{48x}{r^2}\Big(\frac{1}{r^{12}} - \frac{1}{2}\frac{1}{r^6}\Big)$$

The force for the y and z components are calculated likewise, with the total force being the sum of these components.

The core of the Molecular Dynamics simulation is in evolving the system to observe the dynamics and interactions between the clusters with time. In order to evolve the system we must integrate Newton's equations of motion, achieved in this simulation through the use of the Velocity Verlet algorithm [3]. The Verlet Velocity algorithm makes use of Taylor Expansions of position around (t+Δt) and (t-Δt) to calculate the new position and velocity of each particle at each time-step.
Position update:

$$r(t + \Delta t) \approx 2r(t) - t(t - \Delta t) + \frac{f(t)}{m}\Delta t^2$$

Velocity update:

$$v(t + \Delta t) = \frac{r(t + \Delta t) - r(t - \Delta t)}{2\Delta t}$$

With r the position at a chosen time, t the time, f the force at a given time t and Δt the predefined time-step.
In conjunction with our previous method of calculating the force according to the Lennard-Jones potential, we are now able to successfully evolve the system and observe the dynamics of the two clusters upon computation.

Furthermore, the instantaneous temperature at a time t can be calculated according to the equipartition theorem [3].

$$T(t) = \sum \frac{mv^2(t)}{N_f k_b}$$

Where k_b is the Boltzmann constant, *T(t)* the temperature at time t, m the mass of the particle, v the velocity of the particle at time t and N_f the number of degrees of freedom.

Part II
Methodology

The code is composed of two main sections. A section where the required functions for the simulation are defined, and a section where the program variables are input and functions called. The latter section is the core of the program and essentially where the simulation takes place.

The two core functions in the program are the Velocity Verlet algorithm and the force calculation, as discussed in the Introduction, named *verlet_vel* and *get_force* in the code respectively. Other "housekeeping" functions and additional features are also included in the first section of the code, with each being breifly described below in chronology with appearance in the source code.
The first function defined is the *get_clusters* function which simply reads the co-ordinates of the two clusters from a .dat file, and loads them as two matrices. These two matrices, along with an initial separation and velocity are passed to the next function: *init*, the initialisation function. Upon runtime the values of initial separation and velocity are input by the user. The initialisation function proceeds to stack these two matrices into one, creating the current position matrix: *c_pos*. The first cluster is set to be stationary, and the second cluster to be some user input distance away, travelling towards the first cluster at the given velocity. Additionally, the *init* function also creates the matrices used for storing the kinetic energy, potential energy and force values, initialising them at zero. These three values are stored in matrices *en_kin*, *en_pot* and *frc* respectively.
The current position, force and potential energy matrices then proceed into the force calculation, completed by the function: *get_force*. Here the program loops through each particle, being each row of the matrix, calculating the distance in the x, y and z components. These components are then combined to create a squared distance vector *r_sqr*, to be compared against the cutoff radius [3]. The cutoff radius is a result of truncation of the Lennard-Jones potential, which is required in order to limit the number of meaningful calculations we make. A truncation of interactions is suitable for a system concerned with short range interactions, one where particles within some cutoff distance *rcut* determine the potential energies. Simple truncation of the Lennard-Jones potential can result in substantial error in some cases, however given our simplified system this form of truncation is tolerable.

$$U^{trunc}(r) = \begin{cases} U^{LJ} & r \leq rcut \\ 0 & r > rcut \end{cases}$$

The value of *rcut* given in this simulation is 2.5, which corresponds to a value of ≈ 3.82Å[4] . This cutoff radius employed also gives rise to *ecut*, which is the value of potential energy at the cutoff radius [3]:

$$ecut = 4\left(\frac{1}{r_{cut}^{12}} - \frac{1}{r_{cut}^{6}}\right)$$

If the two particles compared during the force calculation are within the cutoff radius, the force for each component is calculated according to:

$$F_x(r) = \frac{48x}{r^2}\left(\frac{1}{r^{12}} - \frac{1}{2}\frac{1}{r^6}\right)$$

as derived in the Introduction, and the force in each component of each particle updated. The

force for one particle is updated as the positive value, and the other as the negative value, as:

$$F_{ij} = -F_{ji}$$

The potential energy of the particle is also updated, with each particle sharing the calculated value. The force and potential energy updates can be seen in the code as follows:

```
#force update in all cartesian components
frc[i,0]= frc[i,0]+ff*xr
frc[i,1]= frc[i,1]+ff*yr
frc[i,2]= frc[i,2]+ff*zr
frc[j,0]= frc[j,0]-ff*xr
frc[j,1]= frc[j,1]-ff*yr
frc[j,2]= frc[j,2]-ff*zr
#potential energy update
en_pot[i] = en_pot[i] + 2*(r6i*(r6i-1)-ecut)
en_pot[j] = en_pot[j] + 2*(r6i*(r6i-1)-ecut)
```

Code 1: Force and potential energy update.

Following this, the variables for position, velocity and force are passed to the Velocity Verlet algorithm: *verlet_vel*. In the code the variables of kinetic and potential energy are also passed to the function, however they are not required by the Velocity Verlet algorithm itself - they are passed to the function as a matter of convenience. Here the previous position of the particles is calculated using a simple time-step backwards, since:

$$\Delta x = vt$$

where Δx is the change in position, v the velocity and t the time-step. With all variables required by the algorithm now calculated, the cluster positions and velocities can be updated according to the Velocity Verlet algorithm:

$$r(t + \Delta t) \approx 2r(t) - t(t - \Delta t) + \frac{f(t)}{m}\Delta t^2$$

$$v(t + \Delta t) = \frac{r(t + \Delta t) - r(t - \Delta t)}{2\Delta t}$$

Implemented in Python through:

```
#previous position calculation
p_pos = c_pos - (c_vel*deltat)
#new position calculation
new_pos = 2*c_pos - p_pos + frc*(deltat**2)
#new velocity calculation
new_vel = (new_pos-p_pos)/(2*deltat)
```

Code 2: Vectorised implementation of Velocity Verlet algorithm.

The function then calculates the temperature through the equation:

$$T(t) = \sum \frac{mv^2(t)}{N_f k_b}$$

previously discussed in the Introduction, and the value of total energy per particle, *etot*, through the sum of the kinetic and potential energies:

```
#temperature calculation
temp= sumv_sqr/((3*Npart)-3)
#total energy per particle
etot = (sum_en_pot+0.5*sumv_sqr)/Npart
```

Code 3: Temperature and total energy per particle calculation.

The new values of cluster position, velocity, total energy per particle and sums of potential and kinetic energy are then returned by the function. A simple tree diagram can be seen in Figure 4, showing the control flow of the code at a basic level.

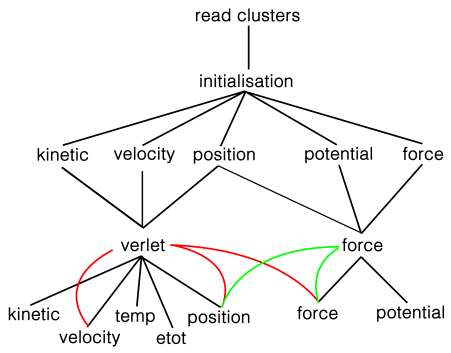

Figure 4: A simple tree graph showing the control flow of the code.

The red and green curved connections represent the recursive nature of the program, wherein the outputs of a function are given as inputs at the

next time-step, and hence the system evolved in time.
Additional, however inessential functions complete the first section of the code in: *refold* and *vel_rescale*. The *refold* function ensures that the particles remain within some defined bounds, and hence do not "escape" the concerned area of interaction. The *vel_rescale* function precludes a system "explosion", whereby the energy of the particles increases well above that desired. To remedy this problem the velocities are rescaled [5] [3] according to a rescaling factor of:

$$\lambda = \sqrt{\frac{T}{T_{inst}}}$$

where T is the desired system temperature and T_{inst} the instantaneous system temperature. These two functions were found to be of use above certain limits of cluster speed or simulation time only.
The second section of the code is concerned with the calling of the functions described above, and the output of final values to external files upon completion of the simulation.
At runtime the program first asks for input of the desired time-step size and number of steps for the simulation to be run, along with some recommended values. The *get_clusters* and *init* functions are then called, and the clusters loaded into the program. A new matrix for storing the outputs is also constructed for writing per 1000 iterations. The core of the simulation can now begin, where for each time-step the *get_force* and *verlet_vel* functions are called, and the output values printed to the storage matrix. This process can be represented through the pseudo-code:

```
while t < deltat*Nsteps:
   call get_force()
   call verlet_vel()
   #update positions and forces
   pos = new_pos
   vel = new_vel
   force = new_force
   potential = new_potential
   #write values to storage
   if iter_number%1000 ==0:
      write temp,etot,kinetic,
            potential,time
   #increment iteration number and time
   iter_number = iter_number + 1
   t = t + deltat
```

Code 5: Loop demonstrating calling of features

The remainder of the code deals with the technicalities of saving the final cluster positions to a file named *final_state.xyz*, and the stored values of temperate, total energy per particle, kinetic energy, potential energy and time to a file named *final_values.txt*. Furthermore, upon completion of the simulation the *matplotlib* package will display a 3D plot of the initial and final cluster positions for comparison, as such:

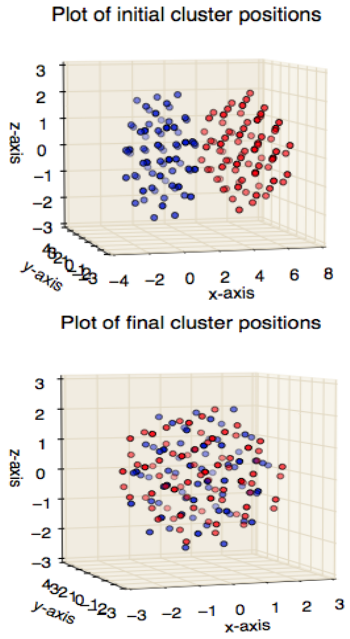

Figure 6: Plots of cluster position.

To begin the simulation one must simply run the command:

```
>>python LJCollison.py
```

Code 7: Running the simulation from command line.

in the directory of the file. The program simply requests input for the values of time-step, number of time steps, initial separation and initial velocity. The simulation, plotting and value writing is automated from hereon.

Part III
Collection of data

The following pages are output and graphs of results from a simulation with input parameters:

Parameter	Value
Time step	1e-12 s
No. time steps	50000
Initial separation	5
Initial velocity	1e8

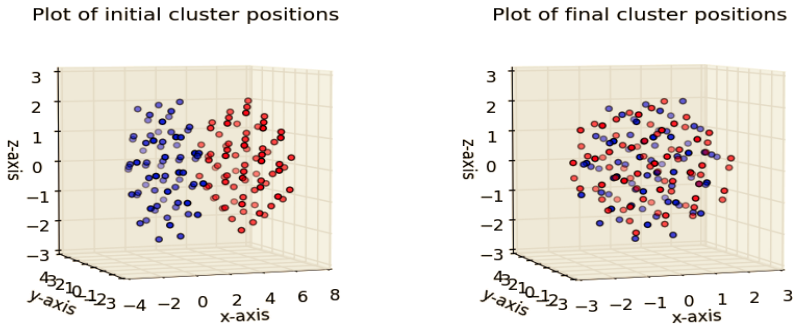

Figure 2: Plots of initial and final cluster position automatically generated by the program.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~      SIMULATION COMPLETE     ~~~~~~~~~~

With 1e-12 s time-steps, 50000 steps, an
initial separation of 5.0 and an initial
velocity of 1e8 , the simulation took: 4648 s.

The final positions of the particles can be found in
the file final_state.xyz

The values, logged at every 1000 iterations can be
found in the file final_values.txt in the format:
----------------------------------------
| Temp | E_tot  | E_kin | E_pot | Time  |
|------|--------|-------|-------|-------|
|      |        |       |       |       |

The matplotlib package should now be displaying a
plot of the initial and final cluster positions.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Code 8: Output from program upon completion of simulation

$Temperature$	$TotalEnergy$	$KineticEnergy$	$PotentialEnergy$	$Time(s)$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	-576.255	0
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	-426.301	$1 \cdot 10^{-9}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	449.776	$2 \cdot 10^{-9}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$6{,}787.484$	$3 \cdot 10^{-9}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$66{,}709.585$	$4 \cdot 10^{-9}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$8.391 \cdot 10^{5}$	$5 \cdot 10^{-9}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$1.35 \cdot 10^{7}$	$6 \cdot 10^{-9}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$1.756 \cdot 10^{8}$	$7 \cdot 10^{-9}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$8.448 \cdot 10^{8}$	$8 \cdot 10^{-9}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$2.067 \cdot 10^{10}$	$9 \cdot 10^{-9}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$4.171 \cdot 10^{10}$	$1 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$1.323 \cdot 10^{9}$	$1.1 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$2.746 \cdot 10^{8}$	$1.2 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$6.525 \cdot 10^{8}$	$1.3 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$1.188 \cdot 10^{9}$	$1.4 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$2.298 \cdot 10^{8}$	$1.5 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$3.802 \cdot 10^{9}$	$1.6 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$3.699 \cdot 10^{10}$	$1.7 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$7.247 \cdot 10^{9}$	$1.8 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$2.53 \cdot 10^{9}$	$1.9 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$2.677 \cdot 10^{9}$	$2 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$3.568 \cdot 10^{9}$	$2.1 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$3.544 \cdot 10^{8}$	$2.2 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$3.854 \cdot 10^{7}$	$2.3 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$3.232 \cdot 10^{7}$	$2.4 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$3.778 \cdot 10^{7}$	$2.5 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$5.442 \cdot 10^{7}$	$2.6 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$2.21 \cdot 10^{8}$	$2.7 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$3.475 \cdot 10^{8}$	$2.8 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$9.674 \cdot 10^{8}$	$2.9 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$4.689 \cdot 10^{8}$	$3 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$1.284 \cdot 10^{8}$	$3.1 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$1.475 \cdot 10^{8}$	$3.2 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$9.396 \cdot 10^{7}$	$3.3 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$1.094 \cdot 10^{8}$	$3.4 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$6.665 \cdot 10^{9}$	$3.5 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$2.021 \cdot 10^{11}$	$3.6 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$2.97 \cdot 10^{10}$	$3.7 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$7.74 \cdot 10^{10}$	$3.8 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$4.765 \cdot 10^{11}$	$3.9 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$4.574 \cdot 10^{12}$	$4 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$6.154 \cdot 10^{13}$	$4.1 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$3.649 \cdot 10^{12}$	$4.2 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$4.218 \cdot 10^{9}$	$4.3 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$3.419 \cdot 10^{10}$	$4.4 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$9.679 \cdot 10^{10}$	$4.5 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$1.955 \cdot 10^{11}$	$4.6 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$5.474 \cdot 10^{12}$	$4.7 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$1.849 \cdot 10^{11}$	$4.8 \cdot 10^{-8}$
$1.758 \cdot 10^{15}$	$2.619 \cdot 10^{15}$	$3.85 \cdot 10^{17}$	$6.048 \cdot 10^{9}$	$4.9 \cdot 10^{-8}$

Table 9: Output values from simulation.

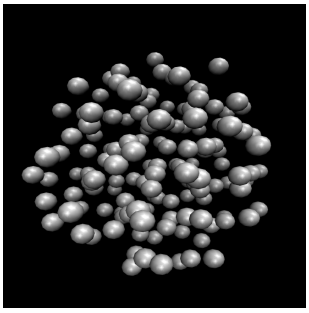

Figure 10: VMD output from file *final_state.xyz*.

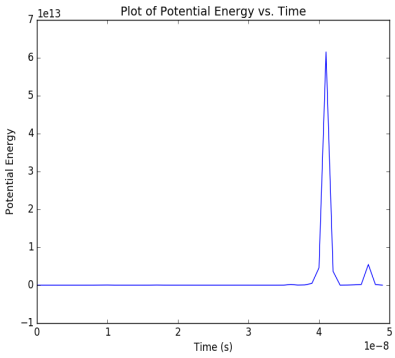

Figure 11: Plot of Potential Energy vs Time.

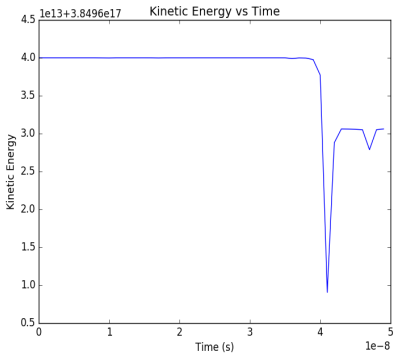

Figure 12: Plot of Kinetic vs Time.

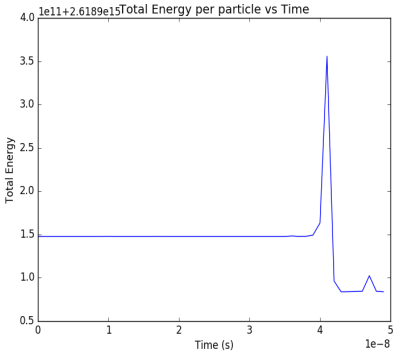

Figure 13: Plot of Total Energy per particle vs Time.

As can be seen above, the two plots of energy follow a similar pattern. The initial peak around 40 nanoseconds represents the incoming cluster passing the cut-off radius, and hence a spike in the total energy per particle due to the introduction of the Lennard-Jones potential. This is the start of the coalescence process. The reduction in total energy per particle is caused by the drop in potential following this, and a corresponding increase in kinetic energy. There is once again a peak in potential energy, however the kinetic energy never returns to its initial value, resulting in an overall drop in total energy per particle.

Part IV
Conclusion

A simple molecular dynamics program modelling the collision of two Lennard-Jones clusters was created and demonstrated using Python 3. The values of *time-step size*, *time-step number*, *initial seperation* and *initial velocity* are given to the program as user input, with the program initialising one cluster at the centre of the co-ordinate axes and the other travelling towards it. The Lennard-Jones potential was used to calculate the potential energy and force on the particles. The force calculation was then used in the Velocity Verlet algorithm to solve Newton's Equations of motion and evolve the system in time, and the equipartition theorem used to calculate the temperature of the system. The calculated vales of *temperature*, *total energy*, *kinetic energy* and *potential energy* are then printed to an output file, along with the final positions of the particles upon simulation completion. Initialisation values of:

Parameter	Value
Time step	1e-12 s
No. time steps	50000
Initial separation	5
Initial velocity	1e8

were found to successfully coalesce the two clusters within 50 nanoseconds.

References

[1] Doye, Jonathan. *The Cambridge Cluster Database* `http://doye.chem.ox.ac.uk/jon/structures/LJ/tables.150.html` (accessed December 12, 2016)

[2] Franzen, Stefan. *Introduction to Molecular Dynamics.* `http://www4.ncsu.edu/ franzen/public_html/CH795N/lecture/IV/IV.html` North Carolina State University (accessed December 12, 2016)

[3] Frenkel, Daan; Smit, Berend. *Understanding Molecular Simulation: From Algorithms to Applications p.69* Amsterdam: Elsevier Science, 2001. (accessed December 12, 2016)

[4] Ercolessi, Furio. *Example code: example_MD_lj.f90* SISSA, Trieste, May 1995

[5] Shell, M.S. *Advanced Molecular Dynamics Techniques.* `https://engineering.ucsb.edu/shell/che210d/Advanced_molecular_dynamics.pdf` University of California, Santa Barbara. (accessed December 12, 2016)

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
#~~~~~~~~~~~~~~~~~~~Lennard-Jones Cluster Collision Simulation~~~~~~~~~~~~~~~~~#
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#

#A simple molecular dynamics program simulating the collision of two Lennard-Jones
#clusters, written in Python 3.5.2. The program computes the Kinetic Energy, Force
#and Total Energy as function of time for the Lennard-Jones system. The structure of the
    cluster
#after collison is printed to an output file of the .xyz format. For the integration
#of Newton's equations of motion the verlet algorithm is used, and the
#Lennard-Jones potential is truncated using 'Simple Truncation' for simplicity.

#Callum Hutchinson, King's College London, December 2016.
#
#Contact:
#callum.hutchinson@kcl.ac.uk
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#

#CODE STARTS HERE#

#import some relevant python libraries to aid our calculations and plotting
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import time
import os

# A function which loads cluster co-ordinates into a matrix from two external files
def get_clusters(filename1,filename2):
   cluster1 = pd.read_csv(filename1,header=None,delim_whitespace=True).as_matrix()
   cluster2 = pd.read_csv(filename2,header=None,delim_whitespace=True).as_matrix()
   return cluster1,cluster2

#function which initialises the positions, velocities, forces and energies
#of the two clusters
def init(cluster1,cluster2,init_dist,init_vel):
   row1,col1 = cluster1.shape
   row2,col2 = cluster2.shape
   Npart = row1+row2

   #initialises the clusters with cluster 1 centered at the origin by default
   #cluster two is set at some initial distance along the x-axis.
   #then combines each cluster into one matrix.
   c1_pos =cluster1
   cluster2[:,0] =cluster2[:,0] + init_dist
   c2_pos = cluster2
   c_pos = np.vstack((c1_pos,c2_pos))

   #initialises cluster 1 at the origin to be stationary, and cluster two to
```

```
    #be moving towards it at some initial velocity along the x-axis.
    vel1 = np.zeros([row1,col1])
    vel2 = np.zeros([row2,col2])
    vel2[:,0] = vel2[:,0] - init_vel
    c_vel = np.vstack((vel1,vel2))

    #initialises the force at zero
    frc = np.zeros([Npart,col1])

    #initialises the potential energy of the system to zero
    en_pot = np.zeros([Npart,1])
    en_pot = en_pot.reshape(Npart,1)

    #initialises the kinetic energy
    en_kin = abs(c_vel).sum(axis=1)
    en_kin = ((en_kin.reshape(Npart,1)**2)*0.5)

    return c_pos,c_vel,frc,en_pot,en_kin,Npart,init_dist


def get_force(c_pos,frc,en_pot,Npart):
    #forces and potential energies initialsied to zero again at
    #start of each get_force run.
    en_pot = np.zeros([Npart,1])
    en_pot = en_pot.reshape(Npart,1)
    frc = np.zeros([Npart,3])

    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
    #~~~~~~~~~~~~~~~~~~~~~~VARIABLE BLOCK~~~~~~~~~~~~~~~~~~~~~~~~#

    #These codes become awfully cluttered after a while, and
    #although Python doesn't use the convention of variable
    #declaration blocks like FORTRAN, it is convenient to use
    #them in times like these for the purposes of
    #organisation/sanity.

    #sigma value ~1.122 sigma AA (reduced units)
    sigma = 1.0

    #epsilon value 0.010394 eV (reduced units)
    eps = 1.0

    #cut-off radius, cut-off distance = 2.5*sigma=3.82AA
    rcut = 2.5

    #cut-off radius squared
    rc2 = rcut**2

    #cut-off potential energy (Frenkel-Smit)
    ecut = 4*(1/(rcut**12)-1/(rcut**6))
```

```
    #~~~~~~~~~~~~~~~~~~END OF VARIABLE BLOCK~~~~~~~~~~~~~~~~#
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#

    #calculating the distance between each pair in the system
    for i in range(Npart-1):
       for j in range(i+1,Npart):
          xr = c_pos[i,0] - c_pos[j,0]
          yr = c_pos[i,1] - c_pos[j,1]
          zr = c_pos[i,2] - c_pos[j,2]
          r_sqr = xr**2 + yr**2 + zr**2

          #if within cutoff limit, calculate force using LJ potential (force is deriv. of
              pot)
          if r_sqr < rc2:
             r2i = 1/r_sqr
             r6i = r2i**3
             #calculate the lennard jones potential Fx=dU/dx=dU/dr*dr/dx=dU/dr*(x/r)
             ff = 48*r2i*(r6i**2-0.5*r6i) #this is the dU/dx term, times by the distance to
                 calc force
             #update the forces, negative for one because Fij = - Fji
             frc[i,0]= frc[i,0]+ff*xr
             frc[i,1]= frc[i,1]+ff*yr
             frc[i,2]= frc[i,2]+ff*zr
             frc[j,0]= frc[j,0]-ff*xr
             frc[j,1]= frc[j,1]-ff*yr
             frc[j,2]= frc[j,2]-ff*zr
             #update the potential energy (shared between the two)
             en_pot[i] = en_pot[i] + 2*(r6i*(r6i-1)-ecut) #difference between potential
                 energy and cutoff
             en_pot[j] = en_pot[j] + 2*(r6i*(r6i-1)-ecut)
    return frc,en_pot


#a function which uses the velocity verlet algorithm to integrate Newton's
#equations of motion
def verlet_vel(frc,c_pos,c_vel,en_pot,en_kin):
    #initialise sums of velocites to be zero
    sumv_sqr=0

    #calculates previous position for use in verlet, using simple time step
    #backwards.
    p_pos = c_pos - (c_vel*deltat)

    #vectorised implementation of position update
    new_pos = 2*c_pos - p_pos + frc*(deltat**2)
    #vectorised implementation of velocity update
    new_vel = (new_pos-p_pos)/(2*deltat)

    #sum of velocities/velocities squared
    sumv_sqr=sum(sum(new_vel**2))
    #temperature according to equiparition
    temp= sumv_sqr/((3*Npart)-3)
```

```
    #total energy per particle
    sum_en_pot = sum(en_pot)
    etot = (sum_en_pot+0.5*sumv_sqr)/Npart
    sum_en_kin = 0.5*sumv_sqr
    return new_pos, new_vel, temp,etot,sum_en_kin,sum_en_pot

#It is possible to encounter an "explosion" with certain simulation
#times/speeds.To remedy this a position refolding function can be
#introduced.
def refold(c_pos,init_dist,Npart,n_refolds):
    init_dist_sqr = init_dist**2
    for i in range(Npart-1):
        for j in range(i+1,Npart):
            xr = c_pos[i,0] - c_pos[j,0]
            yr = c_pos[i,1] - c_pos[j,1]
            zr = c_pos[i,2] - c_pos[j,2]
            r_sqr = xr**2 + yr**2 + zr**2
            #refold the positions (x-axis) depending on their displacement
            if r_sqr > init_dist_sqr and c_pos[i,0] > 0:
                c_pos[i,0] = c_pos[i,0] - 0.5*init_dist
                n_refolds = n_refolds+1
            if r_sqr > init_dist_sqr and c_pos[i,0] < 0:
                c_pos[i,0] = c_pos[i,0] + 0.5*init_dist
                n_refolds = n_refolds+1
            if r_sqr > init_dist_sqr and c_pos[j,0] > 0:
                c_pos[j,0] = c_pos[j,0] - 0.5*init_dist
                n_refolds=n_refolds+1
            if r_sqr > init_dist_sqr and c_pos[i,0] < 0:
                c_pos[j,0] = c_pos[j,0] + 0.5*init_dist
                n_refolds=n_refolds+1
    return n_refolds

#A velocity rescaling was also introduced, only required at certain
#simulation lengths/speeds.
def vel_rescale(ConstT,sum_en_kin,c_vel):
    #verify below equations, (Bussi, Donadio paper)
    scaling_factor = np.sqrt((3*ConstT/2)/sum_en_kin)
    k =1.38e-23
    rescaled=0
    if sum_en_kin > (3*k*ConstT)/2:
        c_vel = c_vel * scaling_factor
    return c_vel

#progress bar
def update_progress(workdone):
     print("\rProgress: [{0:50s}] {1:.1f}%".format('#' * int(workdone * 50), workdone*100),
         end="", flush=True)

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~SIMULATION BEGINS HERE~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
#print title
```

```
os.system('cls' if os.name == 'nt' else 'clear')
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
print("~~~~~~~   MOLECULAR DYNAMICS SIMULATION   ~~~~~~~")
print("~~~~~~~  LENNARD-JONES CLUSTER COLLISION  ~~~~~~~")
print("")
print("Created by: Callum Hutchinson.")
print("King's College London, 2016.")
print("")
print("Please enter requested values below.")
print("")
#Before we get started lets set some global simulation control variables.
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
#~~~~~~~~~~GLOBAL VARIABLES~~~~~~~~~#
print("Picosecond = 1e-12 s")
print("Femtosecond = 1e-15 s")
print("")
print('Enter time step:')
deltat = float(input()) #ps is 1e-12
print("")
print("Min. time steps =1000")
print("")
print('Enter number of time steps:')
Nsteps = int(input())
print("")
while Nsteps < 1000:
   print('Error: Min. timesteps = 1000')
   print('Enter number of timesteps:')
   Nsteps = int(input())

 #total of 5 picoseconds(5000)

ConstT = 20 #kelvin

#~~~~~~END OF GLOBAL VARIABLES~~~~~#
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#


#time initialised to zero
t=0

#number of refolds initialised at 0
n_refolds=0

#open the cluster co-ordinates from the external files
cluster1 ,cluster2 = get_clusters('cluster_size_70.dat','cluster_size_77.dat')

#request values for initial seperation and initial velocity
print(">5 for clusters fully apart")
print("")
print("Initial separation:")
init_sep = float(input())
print("")
```

```
print("(1e9 for merge within 5ns)")
print("")
print("Initial velocity:")
init_vel=float(input())
print("")
#initialise the cluster positions, velocities, forces...
c_pos,c_vel,frc,en_pot,en_kin,Npart,init_dist = init(cluster1,cluster2,init_sep,init_vel)
initial_pos = c_pos

#3d plot of initial positions
fig = plt.figure()
ax1 = fig.add_subplot(121,projection='3d')
ax1.set_title('Plot of initial cluster positions')
ax1.set_xlabel('x-axis')
ax1.set_ylabel('y-axis')
ax1.set_zlabel('z-axis')
ax1.scatter(c_pos[:70,0],c_pos[:70,1],c_pos[:70,2],c='b')
ax1.scatter(c_pos[70:,0],c_pos[70:,1],c_pos[70:,2],c='r')


#create a matrix/table to keep track of values,
# ------------------------------------------------
#| Temp | E_tot  | E_kin | E_pot | Time     |
#|------|--------|-------|-------|----------|
#|      |        |       |       |          |

values = np.zeros([int(Nsteps/1000)+1,5])
iter_number = 0

#######################################
#####    CORE OF SIMULATION  ######
#######################################

#actually evolve the system and call our functions

#start a timer to monitor our simulation times
start = time.time()
while t < deltat*Nsteps:
   update_progress(iter_number/Nsteps)

   new_frc,new_en_pot = get_force(c_pos,frc,en_pot,Npart)
   new_pos,new_vel,temp,etot,sum_en_kin,sum_en_pot =
       verlet_vel(new_frc,c_pos,c_vel,new_en_pot,en_kin)
   #n_refolds = refold(c_pos,init_dist,Npart,n_refolds)
   #c_vel = vel_rescale(ConstT,sum_en_kin,c_vel)

   #update values and introduce a fail-safe for if
   #there is an error and the cluster isnt moving
   if np.array_equal(c_pos,new_pos):
      print("Fatal Error: No movement!")
      quit()
   else:
```

```
      c_pos = new_pos
      c_vel = new_vel
      frc=new_frc
      en_pot=new_en_pot

   #add values to matrix for every 1000 iterations
   #program is terribly slow if you print these
   #values for every single timestep
   if iter_number%1000 == 0:
      it_no = int(iter_number/1000)
      values[it_no,0] = temp
      values[it_no,1] = etot
      values[it_no,2] = sum_en_kin
      values[it_no,3] = sum_en_pot
      values[it_no,4] = t

   #increment iteration number and time
   iter_number = iter_number + 1
   t = t+deltat
   final_pos=c_pos

#end timer
end=time.time()
elapsed = end-start

#print some useful statistics
if np.array_equal(final_pos, initial_pos):
   print("Clusters haven't moved!")

#print(n_refolds)

#print time difference (down here for formating on screen)
os.system('cls' if os.name == 'nt' else 'clear')
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
print("~~~~~~~~~~     SIMULATION COMPLETE     ~~~~~~~~~~")
print("")
print("")
print("With ",deltat,"s time-steps, ",Nsteps," steps, an")
print("initial separation of ",init_sep," and an initial")
print("velocity of ",init_vel,", the simulation took: ",round(elapsed),"s.")
print("")
print("The final positions of the particles can be found in")
print("the file final_state.xyz")
print("")
print("The values, logged at every 1000 iterations can be")
print("found in the file final_values.txt in the format:")
print("")
print(" _____________________________________________")
print("| Temp | E_tot  | E_kin | E_pot | Time     |")
print("|------|--------|-------|-------|----------|")
print("|      |        |       |       |          |")
print("")
```

```
print("The matplotlib package should now be displaying a")
print("plot of the initial and final cluster positions.")
print("")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
#3d plot of final positions
ax2 = fig.add_subplot(122,projection='3d')
ax2.set_title('Plot of final cluster positions')
ax2.set_xlabel('x-axis')
ax2.set_ylabel('y-axis')
ax2.set_zlabel('z-axis')
ax2.scatter(c_pos[:70,0],c_pos[:70,1],c_pos[:70,2],c='b')
ax2.scatter(c_pos[70:,0],c_pos[70:,1],c_pos[70:,2],c='r')
plt.show()



#saves values to txt file
np.savetxt('final_values.txt',values,delimiter=' ')

#write the final positions to an XYZ file (messy but works)
np.savetxt('final_state.xyz',c_pos,delimiter=' ')
data = pd.read_csv('final_state.xyz',header=None,delim_whitespace=True)
data.insert(0,'element','H')
data.to_csv('final_state.xyz',sep=' ',header=None,index=False)
def line_prepender(position,filename, line):
    with open(filename, 'r+') as f:
        content = f.read()
        f.seek(int(position), 0)
        f.write(line.rstrip('\r\n') + '\n' + content)
        f.close()
line_prepender(1,'final_state.xyz','Values for the final positions of the clusters')
line_prepender(0,'final_state.xyz',str(Npart))
```