Science-2 Project Part A

Lennard Jones Argon Atom- Normal Analysis

T.H. Arjun, CSD, 2019111012

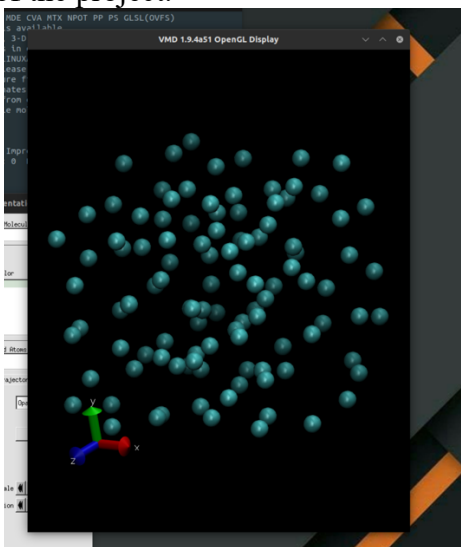
Question 1

Question 1 is to generate a random configuration for the initial structure. We are following the following constraints to generate the initial configuration.

- $N = 108$ (initial number of atoms)
- $L_x = L_y = L_z = 18A°$ (side of the cube), $\in = 0.238\ \frac{kcal}{mol}$ (Lennard Jones Energy Parameter) , $\sigma = 3.4\ A°$ (radius) $r_{ij} \geq 3.4\ A°$ (distance between any two pairs of atoms)

The module that deals with generation of the initial configuration is found in init.py file. The code is self-explanatory as it generates 108 random points such that no two points are at a distance less than 3.4 $A°$. The module saves the initial configuration in molecule.xyz file in xyz format where the first line is number of atoms, second line is comment line, and the next 108 lines follow the pattern < atom symbol x y z >. The data folder has the molecule.xyz file from the run submitted as part of the project.

Output of VMD:



Question 2

Question 2 is to calculate the famous Lennard Jones Potential of the Initial System.
The **Lennard-Jones potential** (also termed the **LJ potential** or **12-6 potential**) is an intermolecular pair potential. The used expression for the Lennard-Jones potential is

$$V_{LJ}(r) = 4 \in \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$$

This is interaction energy per pair. Total Energy will be sum over all such pairs. The calculation of this energy is done by get_energy() which takes in an geometry and is reported by the code in stdout. **The output for the submitted molecule is: 65717551.51082945.**

Question 3

In Question 3, it is asked to calculate the minimum energy configuration for the system from the initial random configuration following periodic boundary conditions and using the steepest descend algorithm for minimization.

Periodic Boundary Conditions

Periodic boundary condition (PBCs) are a set of boundary conditions which are often chosen for approximating a large (infinite) system by using a small part called a *unit cell*. The *minimum-image convention* is a common form of PBC particle bookkeeping in which each individual particle in the simulation interacts with the closest image of the remaining particles in the system. An object which has passed through one face of the simulation box should re-enter through the opposite face—or its image should do it. Evidently, a strategic decision must be made: Do we (A) "fold back" particles into the simulation box when they leave it, or do we (B) let them go on (but compute interactions with the nearest images)? The decision has no effect on the course of the simulation, but if the user is interested in mean displacements, diffusion lengths, etc., the second option is preferable. So I went with the second option. We define two functions in pbc.py which implements this. It calculates the distance to the nearest mirror image in the simulation. The code is an implementation of the C++ formula in Wikipedia in python3. The code is as follows:

```
def pbc_sep(p1, p2):
    arrow = sep(p1, p2)
    rem = np.mod(arrow, 18)  # The image in the first cube
    mic_separation_vector = np.mod(rem+18/2, 18)-18/2

    return np.array(mic_separation_vector)
```

So during the update of the coordinates of the atoms no check has to be done on the boundary conditions. While putting the solution in new_molecule.py I convert the out of bound coordinates to within the box as required.

Steepest Descend Algorithm to Calculate Minimum Potential Energy Configuration from initial Random Configuration

A local optimization method is one where we aim to find minima of a given function by beginning at some point w^0 and taking the steps $w^1, w^2 \ldots, w^k$ of the generic form.
The update rule of the algorithm is $w^k = w^{k-1} + \alpha d^k$ where d^k are *descent direction* vectors (which ideally are *descent directions* that lead us to lower and lower parts of a function) and α is called the *steplength* parameter. The negative gradient $-\nabla g(w)$ of a function g(w) computed at a particular point *always* defines a valid descent direction there.
So, $w^k = w^{k-1} - \alpha \nabla g(w^{k-1})$.
And it seems intuitive - at least at the outset - that because each and every direction is guaranteed to be one of descent here that (provided we set α appropriately, as we must do whenever using a local optimization method)taking such steps could lead us to a point near the a local minimum of any function.

Indeed this is precisely the *gradient descent algorithm*. It is it called *gradient descent* - in employing the (negative) gradient as our descent direction - we are repeatedly *descending* in the *(negative) gradient direction* at each step.

The gradient descent algorithm

---

input: function g, steplength α, maximum number of steps K, and initial point $\mathbf{w}^0$
`for` $k = 1 \ldots K$
$\quad \mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right)$
output: history of weights $\left\{\mathbf{w}^k\right\}_{k=0}^{K}$ and corresponding function evaluations $\left\{g\left(\mathbf{w}^k\right)\right\}_{k=0}^{K}$

How do we set the α parameter in general? I chose 0.001 but it can be anything and is a heuristic, also the termination condition is also a heuristic which I chose to be 100 iterations for the submitted output.

A generic Python implementation of the gradient descent algorithm

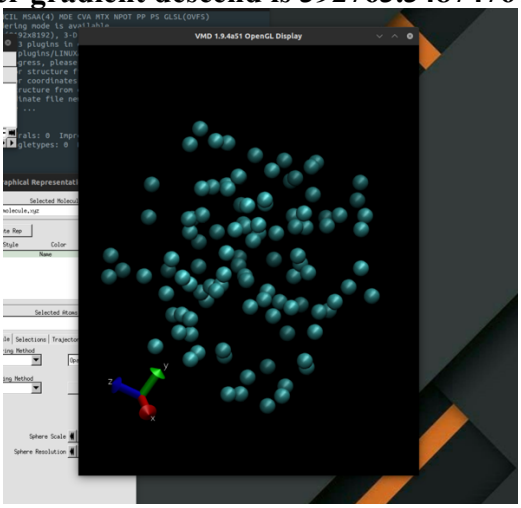
In the file grad.py I implemented gradient descent as described above. It involves just a few requisite initializations, the computation of the gradient function via an Automatic Differentiator library called autograd which gives the gradient of the provided function at the given point, and the very simple for loop. The output is a history of the weights and corresponding cost function values at each step of the gradient descent algorithm. In the main.py file. new_molecule.xyz is the new_molecule in xyz format. I had to move some code to grad.py from other files since autograd uses its own implementation of NumPy and I cannot use normal NumPy here. This has added to some redundancy in the code. Below is the python code for gradient descend.

```python
def gradient_descent(alpha, max_its, w):
    # compute gradient module using autograd
    gradient = grad(get_energy)

    # run the gradient descent loop
    weight_history = [w]           # container for weight history
    # container for corresponding cost function history
    cost_history = [get_energy(w)]
    for k in range(max_its):
        # evaluate the gradient, store current weights and cost function value
        #print(k, w)
        print("It:", k)
        grad_eval = gradient(w)

        # take gradient descent step
        w = w - alpha*grad_eval
        print(get_energy(w))
        # record weight and cost
        weight_history.append(w)
        cost_history.append(get_energy(w))
    return weight_history, cost_history
```

New Energy after gradient descend is 392763.3487476705

Output of VMD: 

Question 4

In question number 4 we are required to get the Hessian Matrix, Eigen values and Eigen Vectors. The Hessian Matrix *H* is defined as:

$$\mathbf{H}_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \frac{\partial^2 f}{\partial x_2\,\partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n\,\partial x_1} & \frac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix},$$

or, by stating an equation for the coefficients using indices i and j,

$$(\mathbf{H}_f)_{i,j} = \frac{\partial^2 f}{\partial x_i\,\partial x_j}.$$

Here we should not that the LJ Potential is discrete and is not differentiable. So we use the method of finite differences for calculating the partial derivatives. Let us consider the approximation of finite differences in 1 variable. It is given by:

$$\nabla f(x)_i = \frac{\partial f}{\partial x_i}(x) \doteq \frac{f(x + h_i e_i) - f(x)}{h_i},$$

where e_i is the *i*th standard basis vector. The vector $x+h_i e_i$ results from perturbing only the *i*th component of *x*. It is important to notice that estimating the gradient using finite differences costs *n* times as much as evaluating the function (2*n* times as much if central differences are used). This is a significant expense that may be unbearable if function evaluations are expensive.

The Hessian can be viewed as the Jacobian of the gradient. The result H is unlikely to be exactly symmetric, so it is important to replace the H obtained by finite differences by

$$\frac{1}{2}\left(H + H^T\right)$$

which will be symmetric for sure. The approximation formula that can be derived from Taylor series is given by:

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(x) = \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i) - f(x + h_j e_j) + f(x)}{h_i h_j} + O(\max\{|h_i|, |h_j|\})$$

We can use this approximation in this model and compute the Hessian which is a 3N x 3N matrix. The formula applied to our specific use case is:

Let N be the number of atoms and let (x_A, y_A, z_A) be the Cartesian coordinates of the A^{th} atom.

$$(\mathbf{H})_{AB} = \frac{\partial^2 E}{\partial X_A \partial X_B} \qquad (X_{3A-2}, X_{3A-1}, X_{3A-0}) = (x_A, y_A, z_A) \qquad \text{for } A \in \{1, \ldots, N\} \tag{1}$$

$$\frac{\partial^2 E}{\partial X_A^2} = \frac{E(X_A + h) + E(X_A - h) - 2E(X_A)}{h^2} \qquad \text{for } X_A = X_B \tag{2}$$

$$\frac{\partial^2 E}{\partial X_A \partial X_B} = \frac{1}{2h^2}\left(E(X_A + h, X_B + h) + E(X_A - h, X_B - h) - E(X_A + h, X_B) - E(X_A - h, X_B)\right.$$
$$\left. - E(X_A, X_B + h) - E(X_A, X_B - h) + 2E(X_A, X_B)\right) \qquad \text{for } X_A \neq X_B \tag{3}$$

The module that does this calculation is found in Hessian.py which has the Hessian Class which loads in new_molecule.xyz and constructs a Hessien matrix based upon these equations. It makes use of get_energy() and Molecule Class from molecule.py to do the same. It is worth noting that main drawback in using finite differences to approximate the Hessian is the expense, and since the code is written in python and we need to calculate the energy with respect to all pairs the code takes a bit of time. Initially it was taking about 1hr to construct the hessian matrix. So I decided to use the Pool() from python multiprocessing to do parallel computing of for loops. By this optimisation, I was able to get the code constructing the hassien matrix in 15 mins. It is very easy to get the Eigen vectors and Eigen Values from the Hassien Matrix using NumPy Library. It is just a function call. The hassien matrix is saved in hassien.py, eigen values in eigen_values.dat, eigen vectors in eigen_vectors.dat.

Question 5

Note that, if the mass-weighted Hessian matrix,

$$(\tilde{\mathbf{H}}_0)_{AB} = \left(\frac{\partial^2 E_e}{\partial \tilde{X}_A \partial \tilde{X}_B}\right)_0$$

were diagonal, we would have a sum of harmonic oscillator Hamiltonians. This dream of a simple Hamiltonian can be realized by choosing a new set of coordinates {q1, . . . , q3N }, called normal coordinates, in terms of which the Hessian is diagonal. Since the mass-weighted Hessian is real and symmetric, it can be diagonalized by a real orthogonal matrix.

$$\tilde{\mathbf{H}}_0 = \tilde{\mathbf{Q}}\mathbf{K}\tilde{\mathbf{Q}}^T \qquad \mathbf{K} = \begin{pmatrix} k_1 & 0 & \ldots & 0 \\ 0 & k_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & k_{3N} \end{pmatrix} \qquad \tilde{\mathbf{Q}}\tilde{\mathbf{Q}}^T = \tilde{\mathbf{Q}}^T\tilde{\mathbf{Q}} = \mathbf{1}$$

where the columns of $\tilde{\mathbf{Q}}$ are eigenvectors $\tilde{\boldsymbol{q}}_1, \ldots, \tilde{\boldsymbol{q}}_{3N}$ of $\tilde{\mathbf{H}}_0$. Back-transforming to un-mass-weighted Cartesian space gives the *normal modes* of the system.

$$\boldsymbol{q}_1 = \mathbf{M}^{-1/2}\tilde{\boldsymbol{q}}_1 \qquad \boldsymbol{q}_2 = \mathbf{M}^{-1/2}\tilde{\boldsymbol{q}}_2 \qquad \cdots \qquad \boldsymbol{q}_{3N} = \mathbf{M}^{-1/2}\tilde{\boldsymbol{q}}_{3N} \qquad \text{where } \mathbf{M} \equiv \begin{pmatrix} m_1 & 0 & 0 & 0 & \cdots \\ 0 & m_1 & 0 & 0 & \cdots \\ 0 & 0 & m_1 & 0 & \cdots \\ 0 & 0 & 0 & m_2 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Given the Hassien Matrix Calculated in previous step, the eigen values and eigen vectors we can calculate the normal mode frequencies as such.

Let N be the number of atoms and let m_A and (x_A, y_A, z_A) be the mass and Cartesian coordinates of the A^{th} atom.

$$(\mathbf{H})_{AB} = \frac{\partial^2 E}{\partial X_A \partial X_B} \qquad (X_{3A-2}, X_{3A-1}, X_{3A-0}) = (x_A, y_A, z_A) \qquad \text{for } A \in \{1, \ldots, N\} \tag{1}$$

$$(\tilde{\mathbf{H}})_{AB} = \frac{(\mathbf{H})_{AB}}{\sqrt{M_A M_B}} \qquad (M_{3A-2}, M_{3A-1}, M_{3A-0}) = (m_A, m_A, m_A) \qquad \text{for } A \in \{1, \ldots, N\} \tag{2}$$

$$\tilde{\mathbf{H}}\,\tilde{\boldsymbol{q}}_A = k_A \tilde{\boldsymbol{q}}_A \qquad \text{for } A \in \{1, \ldots, N\} \tag{3}$$

$$(\boldsymbol{q}_A)_B = \frac{(\tilde{\boldsymbol{q}}_A)_B}{\sqrt{M_B}} \qquad \text{for } A, B \in \{1, \ldots, N\} \tag{4}$$

$$k_A \left[\frac{E_h \text{rad}^2}{a_0^2 u}\right] = k_A \left(\frac{E_h[\text{J}]}{a_0[\text{m}]^2 u[kg]}\right) \left[\frac{\text{rad}^2}{\text{s}^2}\right] \qquad \nu_A\,[\text{Hz}] = \frac{1}{2\pi}\left[\frac{\text{cyc}}{\text{rad}}\right] \cdot \sqrt{k_A \left[\frac{\text{rad}^2}{\text{s}^2}\right]} \qquad \tilde{\nu}_A\left[\text{cm}^{-1}\right] = \frac{\nu_A\,[\text{Hz}]}{c\,[\text{cm/s}]} \tag{5}$$

This calculations are done in frequencies.py. And the normal modes are written to modes.xyz file in the following format, where each corresponds to a normal mode.

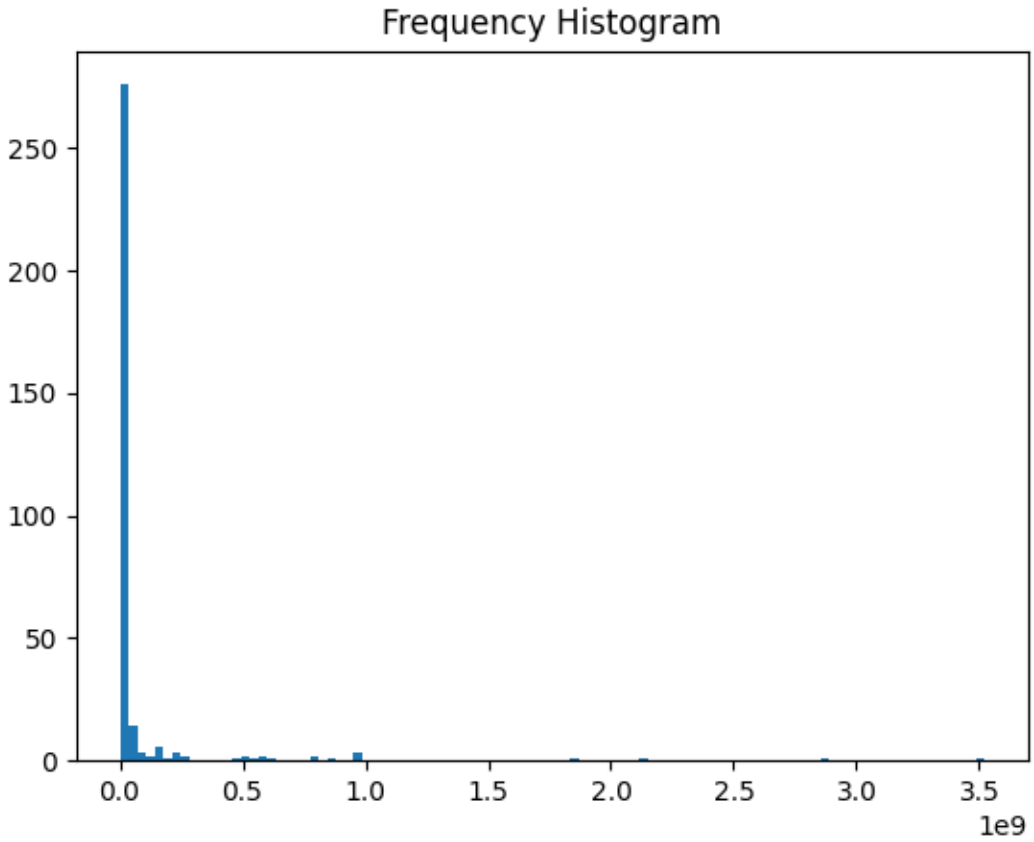
```
N
COMMENT LINE 1
A1 x1 y1 z1   dx11 dy11 dz11
A2 x2 y2 z2   dx12 dy12 dz12
...
AN xN yN zN   dx1N dy1N dz1N

N
COMMENT LINE 2
A1 x1 y1 z1   dx21 dy21 dz21
A2 x2 y2 z2   dx22 dy22 dz22
...
AN xN yN zN   dx2N dy2N dz2N

...

N
COMMENT LINE N
A1 x1 y1 z1   dxN1 dyN1 dzN1
A2 x2 y2 z2   dxN2 dyN2 dzN2
...
AN xN yN zN   dxNN dyNN dzNN
```

Plotting the Histogram for Frequencies



Some modes are very high in number while others are very low.