Task: For the Argon-Argon interaction, estimate the parameters for the Mie(m, n=6) and the Morse potentials based on the experimental values given in the text. Compare with the Lennard-Jones potential.

This task first involved writing functions for the system energy for the different interaction potentials: Lennard-Jones, Mie and Morse. A function was then written to determine the equilibrium lattice constant, volume per atom, bulk modulus and cohesive energy for a given potential energy function for an fcc lattice, termed the 'parameter' function. A range of nearest neighbour distances for a given value of r_m, the position of the potential minimum, were calculated for the first 17 shells. The interaction potentials for the atoms in the system were calculated using the relevant potential function, summed and divided by 2 so that interactions weren't counted twice. The volume per atom was then calculated for the range of nearest neighbour distances. To ascertain the coordinates of 5 points closest to the minimum of the graph of energy vs volume, the energies were sorted in ascending order, and the nearest neighbour distances and volumes associated with these minimum energies were stored. Only 5 points around the minimum were chosen to get a better shape and fit with the polynomial. A quadratic polynomial was fit to these 5 datapoints and its second order derivative was determined. This polynomial was then plotted and the volume and lattice parameter at its minimum point were taken as the equilibrium values. The equilibrium bulk modulus was taken to be the equilibrium volume multiplied by the 2^{nd} order derivative of the energy with respect to volume which, as the fitting polynomial was a quadratic, was a simple scalar value. The equilibrium cohesive energy was the value of the minimum of the energy vs volume graph.

Initial values of r_m = 3.816 Å and ε = 0.0104 eV were assumed, the latter of which represents the absolute value of the minimum of potential. From these, the values of σ and α were calculated to be 3.400 Å and 1.665, respectively, which represent the distance at which the Lennard-Jones potential is zero and the parameter which governs the steepness of the repulsive wall in the Morse potential. The parameter m for the Mie potential was initially assumed to be 10. Putting these values along with the relevant potential functions into the parameters function gave the equilibrium output parameters for each potential using the initial values for the input parameters.

Next, the optimisation was performed for each potential, which was done using a random search method. The optimisation function Q was defined to quantify the deviation of the calculated equilibrium values for the lattice parameter, bulk modulus and cohesive energy from the experimental values, taken from Table 5.2 in Lesar. Functions were written for each potential which calculated Q for the initial equilibrium parameters, then iteratively modified one of the input parameters at random by up to 0.5 % of its initial value randomly up or down, then checked to see if the resulting equilibrium parameters resulted in an increase or decrease in Q. If the change resulted in a decrease, then the new parameter values were stored. This was done 1000 times with a maximum parameter change per iteration of 0.5 %, after which the maximum change was reduced to 0.05 % and 1000 more iterations performed, then 0.005 % and 1000 more iterations performed. The sequential reduction in the extent to which the parameters are varied results in more accurate estimates of the optimal values to be found. This optimisation was performed for each potential and the resultant input parameters were used to plot each potential as a function of lattice parameter and volume, the results of which are displayed in Figures 1 and 2. The initial and final optimised parameters for each potential are also listed in Table 1.

As can be seen, the Lennard-Jones potential has the steepest repulsive wall and so predicts stronger repulsive interactions between molecules at short distances. The Morse potential has the shallowest repulsive wall and so predicts the weakest repulsive interactions. All have a very similar shape around the minimum and at longer distances, so they all predict similarly strong attractive interactions at long distances. One optimised, the Morse and Mie potentials perform the best, showing no deviation from the experimental values for all output parameters. The Lennard-Jones potential performs the worst, with a small amount of deviation for both the equilibrium lattice parameter and the bulk modulus.

Table 1. Input and output parameters for all potentials.

Potential	Initial input parameters	Initial output parameters	Deviation of initial parameters from experimental values	Optimised input parameters	Optimised output parameters	Deviation of optimised parameters from experimental values
Lennard-Jones	rm = 3.816 ε = 0.0104	a = 3.71 B = 3.05 E = -0.088	Δa = 1.07% ΔB = 13.0% ΔE = 10.0%	rm = 3.868 ε = 0.0094	a = 3.76 B = 2.71 E = -0.080	Δa = 0.267% ΔB = 0.370% ΔE = 0%
Mie	rm = 3.816 ε = 0.0104 m = 10	a = 3.67 B = 3.07 E = -0.094	Δa = 2.13% ΔB = 13.7% ΔE = 17.5%	rm = 3.871 ε = 0.0092 m = 10.99	a = 3.75 B = 2.70 E = -0.080	Δa = 0% ΔB = 0% ΔE = 0%
Morse	rm = 3.816 ε = 0.0104 α = 3.400	a = 3.77 B = 2.98 E = -0.070	Δa = 0.533% ΔB = 10.4% ΔE = 12.5%	rm = 3.819 ε = 0.0115 α = 1.552	a = 3.75 B = 2.70 E = -0.080	Δa = 0% ΔB = 0% ΔE = 0%

For reference, experimental values are: a = 3.75 Å, B = 2.7 GPa and E = -0.08 eV/atom.

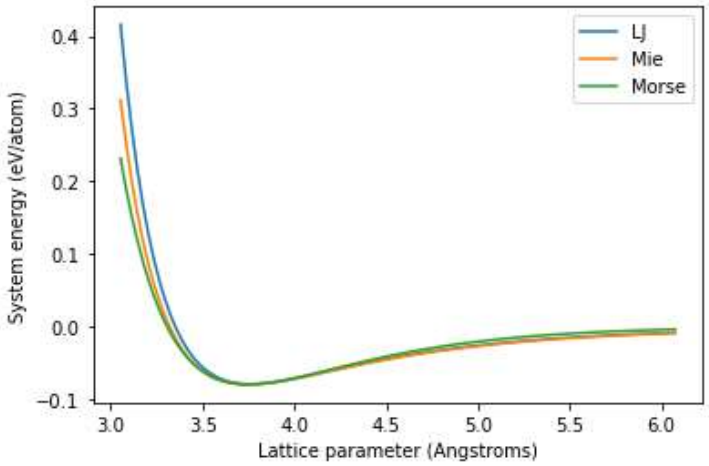

Figure 1. Energy vs lattice parameter for each potential.

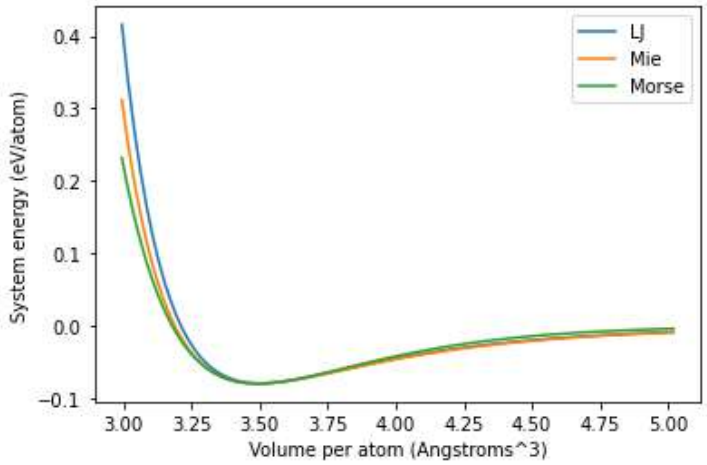

Figure 2. Energy vs volume per atom for each potential.

```
# -*- coding: utf-8 -*-
"""
Created on Thu Mar 18 11:09:24 2021

@author: tom_m
"""
import numpy as np

def neighbours():

    nshells = 17
    shell_at = np.empty(nshells)
    shell_r = np.empty(nshells)

    shell_at[0] = 12
    shell_r[0] = 1
    shell_at[1] = 6
    shell_r[1] = np.sqrt(2)
    shell_at[2] = 24
    shell_r[2] = np.sqrt(3)
    shell_at[3] = 12
    shell_r[3] = 2
    shell_at[4] = 24
    shell_r[4] = np.sqrt(5)
    shell_at[5] = 8
    shell_r[5] = np.sqrt(6)
    shell_at[6] = 48
    shell_r[6] = np.sqrt(7)
    shell_at[7] = 6
    shell_r[7] = np.sqrt(8)
    shell_at[8] = 24
    shell_r[8] = 3
    shell_at[9] = 12
    shell_r[9] = 3
    shell_at[10] = 24
    shell_r[10] = np.sqrt(10)
    shell_at[11] = 24
    shell_r[11] = np.sqrt(11)
    shell_at[12] = 24
    shell_r[12] = np.sqrt(12)
    shell_at[13] = 48
    shell_r[13] = np.sqrt(13)
    shell_at[14] = 24
    shell_r[14] = np.sqrt(13)
    shell_at[15] = 48
    shell_r[15] = np.sqrt(15)
    shell_at[16] = 24
    shell_r[16] = 4

    return shell_at, shell_r
```

```
# -*- coding: utf-8 -*-
"""
Created on Thu Mar 18 16:13:04 2021

@author: tom_m
"""
import numpy as np

def phi_LJ(dist, eps, sig):
    """
    Returns the lennard Jones (12-6) pair potential interaction
    energy as a function of the distance (dist) between the atoms
    eps, sigma are set to one
    """
    phi = 4*eps * ((sig/dist)**12-(sig/dist)**6)
    return phi

def LJ(dist, r_m, eps, one):
    """
    Lennard-Jones potential

    dist : distance
    r_m : position of potential minimum
    """
    phi = eps * ((r_m/dist)**12 - 2*(r_m/dist)**6) * one
    return phi

# def Mie_n6(dist, m, sig, eps):
#     """
#     Mie potential

#     dist : distance
#     m : parameter 1
#     n : parameter 2
#     sig : distance at which potential is zero
#     eps : depth of potential well
#     """

#     phi = (eps/(m-6)) * ((m**m)/(6**6))**(1/(m-6)) * ((sig/dist)**m -
#                                                       (sig/dist)**6)

def Mie_n6(dist, rm, m, eps):
    """
    Mie potential

    dist : distance
    rm : position of potential minimum
    m : parameter 1
    eps : depth of potential well
    """
    A = ((m/6)**(m/(6-m)))*(rm/dist)**m
    B = ((m/6)**(6/(6-m)))*(rm/dist)**6

    phi = (eps/(m-6)) * ((m**m)/(6**6))**(1/(m-6)) * (A - B)


    return phi
```

```
# def Morse(dist, r_m, a, eps):
#     """
#     dist : distance
#     r_m : position of potential minimum
#     a : alpha, governs steepness of repulsive wall
#     eps : depth of potential well
#     """

#     phi = eps * (np.exp(-2*a*(r-r_m)) - 2 * np.exp(-a*(r-r_m)))
#     return phi


def Morse(dist, rm, a, eps):
    """
    dist : distance
    r_m : position of potential minimum
    a : alpha, governs steepness of repulsive wall
    eps : depth of potential well
    """

    phi = eps * (np.exp(-2*a*(dist-rm)) - 2 * np.exp(-a*(dist-rm)))
    return phi
```

```
# -*- coding: utf-8 -*-
"""
Created on Wed Mar 17 13:55:14 2021

@author: tom_m
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from random import random
import random as rand

from fcc_neighbours import neighbours
from potentials import LJ, Mie_n6, Morse


def quadratic(x, a, b, c):
    y = a*x**2 + b*x + c
    return y


r0_exp = 3.75 # experimental nearest neighbour distance, which in an fcc
# is equivalent to the lattice parameter, in Angstroms
E0_exp = -0.08 # cohesive energy, in eV/atom
B0_exp = 2.7 # bulk modulus, in GPa

print('Experimental a =', r0_exp)
print('Experimental B =', B0_exp)
print('Experimental E =', E0_exp)

def parameters(rm, E_fun, arg1, arg2):
    """
    rm : position of potential minimum
    E_fun : potential energy function
    arg1 : input 1 for energy function
    arg2 : input 2 for energy function

    returns the equilibrium volume per atom, lattice parameter, bulk
modulus
    and cohesive energy
    """

    cnt = 100 # number of datapoints
    n_shells = 17 # number of shells included

    # number of atoms per shell, relative shell distance in units of
    # nearest neighbour distance
    shell_at, shell_r = neighbours()

    # vectors to store the nearest neighbour distances and the energies
    # corresponding to them
    E = np.zeros(cnt)
    r = np.zeros(cnt)

    for i in range(cnt):
        # vector of the nearest neighbour distances from rm
        r[i] = rm * (i+cnt)*4/(5*cnt)
```

```
        for j in range(n_shells):
            # atom distance from rm is the relative shell distance
            # multiplied by the nearest neighbour distance from rm
            dist = r[i] * shell_r[j]

            # potential energy for an atom at that distance, can use
energy
            # functino of choice
            pot = E_fun(dist, rm, arg1, arg2)

            # Energy for each nearest neighbour distance is the potential
for
            # each atom in the shells divided by 2, so interactions
aren't
            # counted twice
            E[i] += pot * shell_at[j]/2

    # volume per atom
    v = (r * np.sqrt(2)) ** (3/4)

    # sorts the energies in ascending order
    ascend_E = np.sort(E)

    # number of datapoints around the minimum for which a parabola will
be fit
    minima = 5
    # empty array to store the energies, NN distances and volumes per
atom
    # for the datapoints around the minimum
    min_E = []
    min_r = []
    min_v = []

    # below loop runs through the datapoints in the energies vector
    # 'minima' number of times, the minimum energies and their
corresponding
    # NN distances and volumes per atom are stored in the above vectors
to
    # give the coordinates to fit the parabola
    for k in range(minima):
        for l in range(len(E)):
            if E[l] == ascend_E[k]:

                min_E.append(E[l])
                min_r.append(r[l])
                min_v.append(v[l])


    # returns the coefficients of the 2nd order polynomial fitted to the
    # 'minima' number of datapoints around the minimum, plus their
covariance
    poly, poly_cov = curve_fit(quadratic, min_v, min_E)

    # returns the coefficients of the 2nd order derivative of the
polynomial
    d2E_dv2 = np.polyder(poly, 2)
```

```
    # abscissa and ordinates of polynomial around minimum
    x = np.linspace(min(min_v), max(min_v), 100)
    y = quadratic(x, *poly)

    # equilibrium volume per atom
    eq_v = x[np.argmin(y)]

    # equilibrium lattice constant
    eq_a = (eq_v**(4/3)) / np.sqrt(2)

    # equilibrium bulk modulus
    eq_B = eq_v * d2E_dv2

    # equilibrium cohesive energy
    eq_E = min(y)

    return eq_v, eq_a, eq_B, eq_E


rm = 3.816 # initial value for position of potential minimum
sig = rm / (2**(1/6)) # corresponding distance at which potential is 0
for LJ
eps = 0.0104 # absolute value of potential well
alp = np.log(2) / (rm - sig) # alpha, governs steepness of repulsive wall
# for Morse potential

print('Initial rm =', rm)
print('Initial eps =', eps)
print('Initial alpha =', alp)


# equilibrium v, a, B and E for LJ using initial values of rm and eps
LJ_v, LJ_a, LJ_B, LJ_E = parameters(rm, LJ, eps, 1)

print('Initial LJ v =', LJ_v)
print('Initial LJ a =', LJ_a)
print('Initial LJ B =', LJ_B)
print('Initial LJ E =', LJ_E)


# equilibrium v, a, B and E for Mie using initial values of rm, eps and m
Mie_v, Mie_a, Mie_B, Mie_E = parameters(rm, Mie_n6, 10, eps)

print('Initial Mie v =', Mie_v)
print('Initial Mie a =', Mie_a)
print('Initial Mie B =', Mie_B)
print('Initial Mie E =', Mie_E)


# equilibrium v, a, B and E for Morse using initial values of rm, ep and
alpha
Mo_v, Mo_a, Mo_B, Mo_E = parameters(rm, Morse, alp, eps)

print('Initial Morse v =', Mo_v)
print('Initial Morse a =', Mo_a)
print('Initial Morse B =', Mo_B)
print('Initial Morse E =', Mo_E)
```

```
def optimise_LJ(rm0, eps0, r_exp, B_exp, E_exp):
    """

    Parameters
    ----------
    rm0 : initial value for rm
    eps0 : initial value for epsilon
    r_exp : experimental value for r (or a)
    B_exp : experimental value for B
    E_exp : experimental value for E

    Function optimises the parameters rm and epsilon for LJ potential
around
    given experimental values for a, B and E
    """

    # calculates initial values for atomic volume, NN distance, bulk
modulus
    # and system energy
    v0, a0, B0, E0 = parameters(rm0, LJ, eps0, 1)

    def Q_calc(a, B, E):
        # optimisation function
        Q = (a/r_exp - 1)**2 + (B/B_exp - 1)**2 + (E/E_exp - 1)**2
        return Q

    # number of optimisation trials per run
    n_opt = 1000

    # vector to store values of all parameters and Q after each iteration
    Q = np.empty(n_opt)
    r_m = np.empty(n_opt)
    ep = np.empty(n_opt)
    a = np.empty(n_opt)
    B = np.empty(n_opt)
    E = np.empty(n_opt)

    # sets initial values for all parameters and Q
    Q[0] = Q_calc(a0, B0, E0)
    r_m[0] = rm0
    ep[0] = eps0
    a[0] = a0
    B[0] = B0
    E[0] = E0

    Q_start = Q[0]

    # vector of maximum proportions by which variables can change,
decreases
    # sequentially such that the change is smaller per run to narrow in
    # optimal value more accurately
    changes = np.array([0.01, 0.001, 0.0001])

    for h in range(2):
        # sets 3 runs, n_opt=1000 iterations per run
        for i in range(1, n_opt):
```

```
            # current parameter values equal the previous values
            r_m[i] = r_m[i-1]
            ep[i] = ep[i-1]
            a[i] = a[i-1]
            B[i] = B[i-1]
            E[i] = E[i-1]

            # vector of parameters to be changed at current iteration
only
            var = np.array([r_m[i], ep[i]])

            # random integer between 0 and 1
            ran = rand.randint(0,1)

            # random paramter selected and the maximum amount by which it
can
            # change is determined
            maxchange = var[ran] * changes[h]

            # value changes in random direction
            change = (random()-0.5) * maxchange

            # new value is current value plus change in random direction
            var[ran] += change

            # temporary value of parameters
            r_m_c = var[0]
            ep_c = var[1]

            # equilibrium paramters calculated from the temporary values
            v_c, a_c, B_c, E_c = parameters(r_m_c, LJ, ep_c, 1)

            # new value of optimisation function calculated
            Q[i] = Q_calc(a_c, B_c, E_c)

            # sets the condition that if Q has reduced, the new parameter
            # values are accepted
            if Q[i] < Q[i-1]:

                r_m[i] = r_m_c
                ep[i] = ep_c
                a[i] = a_c
                B[i] = B_c
                E[i] = E_c

                Q_start = Q[i]

        # sets starting values of all parameters and Q to optimised
values
        # for the new run
        Q[0] = Q_start
        r_m[0] = r_m[-1]
        ep[0] = ep[-1]
        a[0] = a[-1]
        B[0] = B[-1]
        E[0] = E[-1]

    return r_m[-1], ep[-1], a[-1], B[-1], E[-1]
```

```
LJ_rm, LJ_ep, LJ_a2, LJ_B2, LJ_E2 = optimise_LJ(rm, eps, r0_exp, B0_exp,
                                                E0_exp)

print('Improved LJ rm =', LJ_rm)
print('Improved LJ eps =', LJ_ep)
print('Improved LJ a =', LJ_a2)
print('Improved LJ B =', LJ_B2)
print('Improved LJ E =', LJ_E2)

def optimise_Mie(rm0, eps0, m0, r_exp, B_exp, E_exp):
    """

    Parameters
    ----------
    rm0 : initial value for rm
    eps0 : initial value for epsilon
    m0 : initial value for m
    r_exp : experimental value for r (or a)
    B_exp : experimental value for B
    E_exp : experimental value for E

    Same optimisation function but now for Mie potential

    """

    v0, a0, B0, E0 = parameters(rm0, Mie_n6, m0, eps0)

    def Q_calc(a, B, E):

        Q = (a/r_exp - 1)**2 + (B/B_exp - 1)**2 + (E/E_exp - 1)**2
        return Q


    n_opt = 1000

    Q = np.empty(n_opt)
    r_m = np.empty(n_opt)
    ep = np.empty(n_opt)
    m = np.empty(n_opt)
    a = np.empty(n_opt)
    B = np.empty(n_opt)
    E = np.empty(n_opt)


    Q[0] = Q_calc(a0, B0, E0)
    r_m[0] = rm0
    ep[0] = eps0
    m[0] = m0
    a[0] = a0
    B[0] = B0
    E[0] = E0

    Q_start = Q[0]

    changes = np.array([0.01, 0.001, 0.0001])
```

```
    for h in range(2):

        for i in range(1, n_opt):

            r_m[i] = r_m[i-1]
            ep[i] = ep[i-1]
            m[i] = m[i-1]
            a[i] = a[i-1]
            B[i] = B[i-1]
            E[i] = E[i-1]

            var = np.array([r_m[i], ep[i], m[i]])

            ran = rand.randint(0,2)

            maxchange = var[ran] * changes[h]

            change = (random()-0.5) * maxchange

            var[ran] += change

            r_m_c = var[0]
            ep_c = var[1]
            m_c = var[2]

            v_c, a_c, B_c, E_c = parameters(r_m_c, Mie_n6, m_c, ep_c)

            Q[i] = Q_calc(a_c, B_c, E_c)

            if Q[i] < Q[i-1]:

                r_m[i] = r_m_c
                ep[i] = ep_c
                m[i] = m_c
                a[i] = a_c
                B[i] = B_c
                E[i] = E_c

                Q_start = Q[i]

        Q[0] = Q_start
        r_m[0] = r_m[-1]
        ep[0] = ep[-1]
        m[0] = m[-1]
        a[0] = a[-1]
        B[0] = B[-1]
        E[0] = E[-1]

    return r_m[-1], ep[-1], m[-1], a[-1], B[-1], E[-1]


Mie_rm, Mie_ep, Mie_m, Mie_a2, Mie_B2, Mie_E2 = optimise_Mie(rm, eps, 10,
                                                             r0_exp,
B0_exp,
                                                             E0_exp)

print('Improved Mie rm =', Mie_rm)
print('Improved Mie eps =', Mie_ep)
```

```
print('Improved Mie m =', Mie_m)
print('Improved Mie a =', Mie_a2)
print('Improved Mie B =', Mie_B2)
print('Improved Mie E =', Mie_E2)


def optimise_Morse(rm0, eps0, alp0, r_exp, B_exp, E_exp):
    """

    Parameters
    ----------
    rm0 : initial value for rm
    eps0 : initial value for epsilon
    alp0 : initial value for alpha
    r_exp : experimental value for r (or a)
    B_exp : experimental value for B
    E_exp : experimental value for E

    Same optimisation function but now for Morse potential

    """

    v0, a0, B0, E0 = parameters(rm0, Morse, alp0, eps0)

    def Q_calc(a, B, E):

        Q = (a/r_exp - 1)**2 + (B/B_exp - 1)**2 + (E/E_exp - 1)**2
        return Q

    n_opt = 1000

    Q = np.empty(n_opt)
    r_m = np.empty(n_opt)
    ep = np.empty(n_opt)
    al = np.empty(n_opt)
    a = np.empty(n_opt)
    B = np.empty(n_opt)
    E = np.empty(n_opt)


    Q[0] = Q_calc(a0, B0, E0)
    r_m[0] = rm0
    ep[0] = eps0
    al[0] = alp0
    a[0] = a0
    B[0] = B0
    E[0] = E0

    Q_start = Q[0]

    changes = np.array([0.01, 0.001, 0.0001])

    for h in range(2):

        for i in range(1, n_opt):

            r_m[i] = r_m[i-1]
            ep[i] = ep[i-1]
```

```
            al[i] = al[i-1]
            a[i] = a[i-1]
            B[i] = B[i-1]
            E[i] = E[i-1]

            var = np.array([r_m[i], ep[i], al[i]])

            ran = rand.randint(0,2)

            maxchange = var[ran] * changes[h]

            change = (random()-0.5) * maxchange

            var[ran] += change

            r_m_c = var[0]
            ep_c = var[1]
            al_c = var[2]

            v_c, a_c, B_c, E_c = parameters(r_m_c, Morse, al_c, ep_c)

            Q[i] = Q_calc(a_c, B_c, E_c)

            if Q[i] < Q[i-1]:

                r_m[i] = r_m_c
                ep[i] = ep_c
                al[i] = al_c
                a[i] = a_c
                B[i] = B_c
                E[i] = E_c

                Q_start = Q[i]

        Q[0] = Q_start
        r_m[0] = r_m[-1]
        ep[0] = ep[-1]
        al[0] = al[-1]
        a[0] = a[-1]
        B[0] = B[-1]
        E[0] = E[-1]

    return r_m[-1], ep[-1], al[-1], a[-1], B[-1], E[-1]


Mo_rm, Mo_ep, Mo_al, Mo_a2, Mo_B2, Mo_E2 = optimise_Morse(rm, eps, alp,
                                                          r0_exp, B0_exp,
                                                          E0_exp)

print('Improved Morse rm =', Mo_rm)
print('Improved Morse eps =', Mo_ep)
print('Improved Morsee alpha =', Mo_al)
print('Improved Morse a =', Mo_a2)
print('Improved Morse B =', Mo_B2)
print('Improved Morse E =', Mo_E2)


"""
```

```
With equilibrium parameters for each potential calculated and optimised,
the below code will plot them for comparison
"""

cnt = 100 # number of datapoints
n_shells = 17 # number of shells included

# number of atoms per shell, relative shell distance in units of
# nearest neighbour distance
shell_at, shell_r = neighbours()

# vectors to store the nearest neighbour distances and the energies
# corresponding to them for each potential
E_LJ = np.zeros(cnt)
E_Mie = np.zeros(cnt)
E_Mo = np.zeros(cnt)

r = np.zeros(cnt)

for i in range(cnt):
    # vector of all the nearest neighbour distances from rm
    r[i] = rm * (i+cnt)*4/(5*cnt)

    for j in range(n_shells):
        # atom distance from rm is the relative shell distance
        # multiplied by the nearest neighbour distance from rm
        dist = r[i] * shell_r[j]

        # potential energies for an atom at that distance
        pot_LJ = LJ(dist, LJ_rm, LJ_ep, 1)
        pot_Mie = Mie_n6(dist, Mie_rm, Mie_m, Mie_ep)
        pot_Mo = Morse(dist, Mo_rm, Mo_al, Mo_ep)

        # Energy for each nearest neighbour distance is the potential for
        # each atom in the shells divided by 2, so interactions aren't
        # counted twice
        E_LJ[i] += pot_LJ * shell_at[j]/2
        E_Mie[i] += pot_Mie * shell_at[j]/2
        E_Mo[i] += pot_Mo * shell_at[j]/2

# volume per atom
v = (r * np.sqrt(2)) ** (3/4)

plt.figure()
plt.plot(r, E_LJ, label='LJ')
plt.plot(r, E_Mie, label='Mie')
plt.plot(r, E_Mo, label='Morse')
plt.xlabel('Lattice parameter (Angstroms)')
plt.ylabel('System energy (eV/atom)')
plt.legend()

plt.figure()
plt.plot(v, E_LJ, label='LJ')
plt.plot(v, E_Mie, label='Mie')
plt.plot(v, E_Mo, label='Morse')
plt.xlabel('Volume per atom (Angstroms^3)')
plt.ylabel('System energy (eV/atom)')
plt.legend()
```

```
plt.show()
```