SciCompProject3

October 6, 2020

0.1 Project 3

Emily Wilbur

[219]:
```
import numpy as np
import matplotlib.pyplot as plt
```

[360]:
```
# (a)
# points: (N,3)-array of (x,y,z) coordinates for N points
# distance(points): returns (N,N)-array of inter-point distances:
def distance( points ):
    displacement = points[:,np.newaxis] - points[np.newaxis,:]
    return np.sqrt( np.sum(displacement*displacement, axis=-1) )

# computing Lennard-Jones ~*energy*~
def LJ(sigma,epsilon):
    def V( points ):
        r = distance(points)
        rij = np.where(r !=0)
        R = r[rij]
        vLJ = 4*epsilon*((sigma/R)**12 - (sigma/R)**6)
        V = np.sum((vLJ)/2)
        return V
    return V

# Do some computations! Parts 1 and 2
sigmaA = 3.401   # [A with a fun little hat]
epsilonA = 0.997 # [kJ/mol]
x = np.linspace(3,11,100) # added more points for smoother plots
listLJ1 = []     # initialize the list of lennard-jones potential for part 1
listLJ2 = []     # initialize the list of lennard-jones potential for part 2
for i in range(len(x)):
    X1 = np.array([[x[i], 0, 0],[0, 0, 0]])
    X2 = np.array([[x[i], 0, 0],[0, 0, 0],[14, 0, 0],[7, 3.2, 0]])
    LJ1 = LJ(sigmaA, epsilonA)(X1) # lennard-jones potential for part 1
    listLJ1.append(LJ1)            # list of lennard-jones potential for part 1
    LJ2 = LJ(sigmaA, epsilonA)(X2) # lennard-jones potential for part 2
    listLJ2.append(LJ2)            # list of lennard-jones potential for part 2
```

```
# plotting the figures for parts 1 and 2
plt.figure(figsize=(12,6))
plt.subplot(1,2,1)
plt.plot(listLJ1)
plt.xlabel('x, first part')
plt.ylabel('Lennard-Jones Potential')
plt.subplot(1,2,2)
plt.plot(listLJ2)
plt.xlabel('x, second part')
plt.show;
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[321]:
```
# (b)
# bisection root finding function
# x, n calls = bisection root(f,a,b,tolerance=1e-13)
def bisection_root(f,a,b,tolerance=10**(-13)):
    fa, fb = f(a), f(b)
    ncalls = 0 # open the counter for calls as a counter for how many times the␣
 ↪loop runs
    n = int(np.ceil(np.log2((b-a)/tolerance)))
    assert (np.sign(f(a)) != np.sign(f(b)))
    for i in range(n):
        c_i = a + (b - a)/2
        fc = f(c_i)
        if np.sign(f(a)) == np.sign(fc):
            a = c_i
            fa = f(c_i)
        else:
            b = c_i
            fb = f(b)
        ncalls += 1 # count the loops
    return c_i, ncalls # the c_i = x for this function. It was originally x but␣
 ↪I changed it for fun.

# test it out
a = 2
b = 6
sigma = 3.401
epsilon = 0.997
V = LJ(sigma, epsilon)
(xb, nb) = bisection_root(lambda X : V(np.array([[X, 0, 0],[0,0,0]])), a, b)
print(xb, nb)
```

```
3.4010000000000105 46
```

0.2 (b)

The bisection root finding function calls the potential function 46 times before it converges to the σ value. Technically, the bisection root solver uses the potential function 3 or 4 times (depending on who you ask), so you could count the calls as four times the number given. However, as far as I know, the function is only actually *called* once, and then used for different inputs. This may just be my lack of Python knowledge, though. Both the x value (approximately the σ value) and the number of calls, nb, are printed above.

[313]:
```
# (c)
# newton-rhapson root finding function
# x, n calls = newton root(f,df,x0,tolerance,max iterations)
def newton_root(f,df,x0,tolerance=10**(-12),maxiterations=100):
    ncalls = 0 # open the counter for calls as a counter for how many times the␣
 ↪loop runs
    for i in range(maxiterations):
        x0 = x0 - f(x0)/df(x0)
        ncalls += 1
        if abs(f(x0)) < tolerance:
            break
    return x0, ncalls

# try it out
sigma = 3.401
epsilon = 0.997
dv = lambda R : 4*epsilon*((6*sigma**6)/R**7 - (12*sigma**12)/R**13) #␣
 ↪derivative of potential
x0 = 2
(xc, nc) = newton_root(lambda X : V(np.array([[X, 0, 0],[0,0,0]])),dv,x0)
print(xc,nc)
```

```
3.400999999999998 12
```

0.3 (c)

The Newton-Rhapson solver calls the actual function 12 times, but if you count the calls to the derivative of the function as well, then it would be 24 times. Since I computed the derivative by hand and then plugged it in, I am only counting the calls to the actual LJ potential function. Both the x value (approximately the σ value), and the number of calls, nc, are printed above.

[358]:
```
# (d)
# newton-rhapson w/ bisection, guaranteed convergence
# taken to mean that you should initially try bisection root finding, and then␣
 ↪move to newton-rhapson after
# first few steps. We'll see how it goes.
def combo(f, a, b, df, tolerance = 10**(-13), maxiterations = 100):
```

```
    nb = 0 # open the counter for number of calls
    ncalls = 0
    for i in range(2):
        c_i = a + (b - a)/2
        fc = f(c_i)
        if np.sign(f(a)) == np.sign(fc):
            a = c_i
            fa = f(c_i)
        else:
            b = c_i
            fb = f(b)
        c_i = a + (b - a)/2
        nb += 1
    (x, nc) = newton_root(f, df, c_i, tolerance, maxiterations)
    ncalls += nb + nc # count the calls to the original function as sum of␣
 ↪calls from both functions
    return x, ncalls

# testing !!!
sigma = 3.401
epsilon = 0.997
a = 2
b = 6
dv = lambda R : 4*epsilon*((6*sigma**6)/R**7 - (12*sigma**12)/R**13)
(xd, nd) = combo(lambda X : V(np.array([[X, 0, 0],[0,0,0]])),a,b,dv)
print(xd,nd)
```

```
3.4010000000000002 18
```

0.4 (d)

This function says it calls the LJ potential function 7 times, but if I am being honest, I think it's lying. If I multiply the nb (number of calls for the bisection part) by four and the nc (number of calls for the Newton-Rhapson function) by 2 (theoretically scaling the number of calls to the maximum number of possible calls for each function) then it says it takes 18 calls to the LJ function for it to converge. Both the x value and the number of calls, nd, are printed above.

[224]:
```
from numpy import newaxis, fill_diagonal, sum, sqrt
NA = newaxis

# \grad_k V(\xx_k) = 4\epsilon \sum_{j\ne k} (6\sigma^6/r_{kj}^7 - 12\sigma^12/
 ↪r_{kj}^13) u_{kj}
#                  = 4\epsilon \sum_{j=1}^N  (6\sigma^6/\tilde{r}_{kj}^7 -␣
 ↪12\sigma^12/\tilde{r}_{kj}^13) u_{kj}
# u_{kk} = (0,0,0), \tilde{r}_{kk} = 1
def LJgradient(sigma, epsilon):
```

```
    def gradV(X):
        d = X[:,NA] - X[NA,:]         # (N,N,3) displacement vectors
        r = sqrt( sum(d*d,axis=-1) ) # (N,N)   distances

        fill_diagonal(r,1)           # Don't divide by zero
        u = d/r[:,:,NA]              # (N,N,3) unit vectors in direction of␣
↪\xx_i - \xx_j

        T = 6*(sigma**6) * (r**-7) - 12*(sigma**12) * (r**-13) # NxN matrix of␣
↪r-derivatives

        # Using the chain rule, we turn the (N,N)-matrix of r-derivatives into␣
↪the (N,3)-array
        # of derivatives to Cartesian coordinates, i.e.:the gradient.
        return 4*epsilon*sum(T[:,:,NA] * u, axis=1)

    return gradV
```

[359]:
```
# (e)
# looking at gradients of potential
# from part (a):

sigma = 3.401 # [A with a fun little hat]
epsilon = 0.997 # [kJ/mol]
x = np.linspace(3,11,100)
listLJ1grad = []
listLJ2grad = []
for i in range(len(x)):
    X1 = np.array([[x[i], 0, 0], [0, 0, 0]])
    X2 = np.array([[x[i], 0, 0], [0, 0, 0],[14, 0, 0],[7, 3.2, 0]])
    LJ1grad = LJgradient(sigma, epsilon)(X1) # lennard-jones potential gradient␣
↪for part 1
    listLJ1grad.append(LJ1grad)              # list of lennard-jones potential␣
↪gradient for part 1
    LJ2grad = LJgradient(sigma, epsilon)(X2) # lennard-jones potential gradient␣
↪for part 2
    listLJ2grad.append(LJ2grad)              # list of lennard-jones potential␣
↪gradient for part 2

plt.figure(figsize=(6,6))
plt.plot(listLJ1)
plt.plot(np.array(listLJ1grad)[:,0,0])
plt.xlabel('x for X1')
plt.ylabel('Lennard-Jones Potential')
plt.show;
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

0.5 (e)

The two components of the gradient are non-zero because they are theoretically representing the slopes of the LJ potential toward the minimum. They are equal and opposite because, presumably, the minimum is represented by a dip in the function, and so the slope will dip downward and in a potitive direction toward the dip, while the other side will dip "upward" (directionally) and negative toward the dip (or vice versa). The zero of the derivative lines up with the zero of the potential, but the approach zero from opposite sides of the x axis.

The gradient for the four partical system is not zero where it is zero for the two particle version because when you add more particles, you alter the gradient of the entire vector field, and create new minima and maxima.

[226]:
```
# (f)
# line searching function
def linesearch(F, X0, d, alphamax, tolerance, maxiterations):
    a = lambda alpha : np.dot(d.flatten(), F(X0+alpha*d).flatten())
    (x, ncalls) = bisection_root(a, 0, alphamax, tolerance)
    return x, ncalls

# test
sigma = 3.401 # [A with a fun little hat]
epsilon = 0.997 # [kJ/mol]
X0 = np.array([[4,0,0], [0,0,0], [14,0,0], [7,3.2,0]])
d = - LJgradient(sigma,epsilon)(X0)
alphamax = 1
alpha = linesearch(LJgradient(sigma,epsilon), X0, d, alphamax, 10**(-6), 1000)
print(alpha)
```

```
(0.4517068862915039, 20)
```

[291]:
```
# (g)
ArSt = np.load('ArStart.npz')

# Golden ratio minimizer
def golden_section_min(f,a,b,tolerance=10**(-3)):
    golden = (np.sqrt(5)-1)/2
    ncalls = 0
    x1 = a + (1-golden)*(b-a)
    f1 = f(x1)
    x2 = a + golden*(b-a)
    f2 = f(x2)
    while (b - a) > tolerance:
        if f(x1) > f(x2):
```

```
            a = x1
            x1 = x2
            f1 = f2
            x2 = a + golden*(b-a)
            f2 = f(x2)
        else:
            b = x2
            x2 = x1
            f2 = f1
            x1 = a + (1-golden)*(b-a)
            f1 = f(x1)
        ncalls += 1
    xopt = (b+a)/2
    return xopt, ncalls


# testing!
a = 2
b = 6
sigma = 3.401
epsilon = 0.997
something = lambda alpha : V(X0+alpha*d)
alpha = golden_section_min(something, 0,1)
r0 = golden_section_min(lambda X : V(np.array([[X, 0, 0],[0,0,0]])), a, b)
print(alpha) # same alpha as in part (f)
print(r0)    # this is the minimal energy distance between the particles in the␣
 ↪2 particle system
```

```
(0.4516693913190848, 15)
(3.8172876299217666, 18)
```

[398]:
```
# (h)
# write the BFGS function

# these are from the assignment instructions
def flattenfunction(f):
    return lambda X: f(X.reshape(-1,3)) # Reshape with '-1' means "whatever is␣
 ↪left".

def flattengradient(f):
    return lambda X: f(X.reshape(-1,3)).reshape(-1)

# back to code that I wrote:
# BFGS
def BFGS(f,gradf,X, tolerance = 10**(-6), maxiterations=10000):
    n = len(X)
    x0 = X
```

```
    B = np.identity(n) # *(np.linalg.norm(gradf(x0)))/(0.1) # initialize B,␣
 ↪tried to scale it but it didn't work
    ncalls = 0         # open the counter
    for k in range(maxiterations):
        sk = - B@gradf(x0) # opted to do the inverse BFGS for speed and␣
 ↪accuracy and also because it is easier
        xk = x0 + sk       # this is the x_next
        yk = gradf(xk) - gradf(x0)
        By = B@yk          # getting B_y
        B += np.outer(sk, sk)/np.dot(sk, yk) - np.outer(By, By)/np.dot(By, yk)
        x0 = xk            # update x0
        ncalls += 3        # counting calls to f, gradf calls f once per time␣
 ↪it's used
        if np.linalg.norm(gradf(x0)) < tolerance: # this is the test for␣
 ↪convergence, if 'true' then it converged
            return ('True'), x0, ncalls           # this will only print if it␣
 ↪converged, otherwise it will
    return ('oof'), x0, ncalls                    # try to print this line and␣
 ↪fail

# test it out
# this is technically question (i)
ArSt = np.load('ArStart.npz')
ArSt['Xstart2']
sigma = 3.401
epsilon = 0.997

# BFGS LJ for X2
(w2, x2, ncalls2) = BFGS(LJ(sigma,epsilon),␣
 ↪flattengradient(LJgradient(sigma,epsilon)), ArSt['Xstart2'])
fff2 = flattenfunction(LJ(sigma,epsilon))(x2)
print(fff2) # flattened LJ potential for the X2 values
(r0,nr) = golden_section_min(lambda X : V(np.array([[X, 0, 0],[0,0,0]])), a, b)
optdist2 = np.sum(abs(distance(x2.reshape(-1,3)) - r0)/r0 <= 0.01)/2 # optimal␣
 ↪distance for X2
print(optdist2)
print(ncalls2)

(w3, x3, ncalls3) = BFGS(LJ(sigma,epsilon),␣
 ↪flattengradient(LJgradient(sigma,epsilon)), ArSt['Xstart3'])
fff3 = flattenfunction(LJ(sigma,epsilon))(x3)
print(fff3) # flattened LJ potential for the X3 values
(r0,nr) = golden_section_min(lambda X : V(np.array([[X, 0, 0],[0,0,0]])), a, b)
optdist3 = np.sum(abs(distance(x3.reshape(-1,3)) - r0)/r0 <= 0.01)/2 # optimal␣
 ↪distance for X3
print(optdist3)
```

```
print(ncalls3)

(w3, x4, ncalls3) = BFGS(LJ(sigma,epsilon),␣
 ↪flattengradient(LJgradient(sigma,epsilon)), ArSt['Xstart4'])
fff3 = flattenfunction(LJ(sigma,epsilon))(x4)
print(fff3) # regular BFGS has a really hard time for N = 4 :( LJ potential␣
 ↪goes to zero

# GRAPHING!
# graphing the stuff for up to N = 3 and then maybe in the next question I will␣
 ↪graph for N = 4
# start up some loading items
from mpl_toolkits.mplot3d import Axes3D
# and
plot_dict = dict(projection='3d')
%matplotlib notebook
# this is a very steep learning curve for me personally, so bear with me here
x3new = x3.reshape(3,-1)
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.scatter3D(x3[0], x3[1], x3[2])
ax.scatter3D(x3[3], x3[4], x3[5])
ax.scatter3D(x3[6], x3[7], x3[8])
ax.plot_trisurf(x3new[:,0], x3new[:,1], x3new[:,2],
                color = 'magenta', edgecolor='none')
ax.set_title('3D for N=3');
```

```
-0.997
1.0
18
-2.9909999999999988
3.0
213
-2.5567656185157e-06
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

0.6 (h)

This does indeed yield the same r0 as previous.

0.7 (i)

For the N = 2 system, the number of particles within optimal distance is two. For N = 3, it is three. Since my highest converged N is N = 3, I graphed it. I will graph N = 9 below for the line

search BFGS as well. My apologies for the weird graph, the only way I could get the points to connect was with a surface triangulation.

[397]:
```
# (j)

# I had to make a fresh line-search function because the old one doesn't work␣
 ↪for this problem.
def linesearch2(F, X0, d, tolerance, maxiterations = 10000):
    a = lambda alpha : F(X0+alpha*d).flatten()
    (x, ncalls2) = golden_section_min(a, -1, 1, tolerance) # using the golden␣
 ↪section minimizer
    return x, ncalls2

# adding a line search the the above BFGS function
def BFGSlinesearch(f,gradf,X, tolerance = 10**(-7), maxiterations=10000):
    n = len(X)
    x0 = X
    B = np.identity(n) # *(np.linalg.norm(gradf(x0)))/(0.1) # initialize B
    ncalls1 = 0
    ncalls2 = 0
    ncalls = 0
    for k in range(maxiterations):
        sk = - B@gradf(x0)                                  # I did the inverse␣
 ↪BFGS
        (alpha, ncalls2) = linesearch2(f, x0, sk, 10**(-3)) # woohoo here's the␣
 ↪line search
        sk = sk*alpha                                       # re-adjusting the␣
 ↪step size sk using line search
        xk = x0 + sk
        yk = gradf(xk) - gradf(x0)
        By = B@yk
        B += np.outer(sk, sk)/np.dot(sk, yk) - np.outer(By, By)/np.dot(By, yk)
        x0 = xk
        ncalls1 += 1
        ncalls2 += 1
        ncalls += ncalls1 + ncalls2
        if np.linalg.norm(gradf(x0)) < tolerance: # this is the test for␣
 ↪convergence, if 'true' then it converged
            return ('True'), x0, ncalls           # this will only print if it␣
 ↪converged, otherwise it will
    return ('False') , x0, ncalls                 # try to print this line and␣
 ↪fail

# testing testing 1 2 3
ArSt = np.load('ArStart.npz')
ArSt['Xstart2']
sigma = 3.401
```

```
epsilon = 0.997

# Hi. I was going to write a for loop for this but it seemed like more fun to␣
 ↪write all the calls out one by one
# and it seemed like you would have more fun reading all these lines of code. :)

(wj, xj2, ncallsj) = BFGSlinesearch(flattenfunction(LJ(sigma,epsilon)),␣
 ↪flattengradient(LJgradient(sigma,epsilon)), ArSt['Xstart2'])
fffj = flattenfunction(LJ(sigma,epsilon))(xj2)
print(fffj)

(wj3, xj3, ncallsj3) = BFGSlinesearch(flattenfunction(LJ(sigma,epsilon)),␣
 ↪flattengradient(LJgradient(sigma,epsilon)), ArSt['Xstart3'])
fffj3 = flattenfunction(LJ(sigma,epsilon))(xj3)
print(fffj3)

(wj4, xj4, ncallsj4) = BFGSlinesearch(flattenfunction(LJ(sigma,epsilon)),␣
 ↪flattengradient(LJgradient(sigma,epsilon)), ArSt['Xstart4'])
fffj4 = flattenfunction(LJ(sigma,epsilon))(xj4)
print(fffj4)

(wj5, xj5, ncallsj5) = BFGSlinesearch(flattenfunction(LJ(sigma,epsilon)),␣
 ↪flattengradient(LJgradient(sigma,epsilon)), ArSt['Xstart5'])
fffj5 = flattenfunction(LJ(sigma,epsilon))(xj5)
print(fffj5)

(wj6, xj6, ncallsj6) = BFGSlinesearch(flattenfunction(LJ(sigma,epsilon)),␣
 ↪flattengradient(LJgradient(sigma,epsilon)), ArSt['Xstart6'])
fffj6 = flattenfunction(LJ(sigma,epsilon))(xj6)
print(fffj6)

(wj7, xj7, ncallsj7) = BFGSlinesearch(flattenfunction(LJ(sigma,epsilon)),␣
 ↪flattengradient(LJgradient(sigma,epsilon)), ArSt['Xstart7'])
fffj7 = flattenfunction(LJ(sigma,epsilon))(xj7)
print(fffj7)

(wj8, xj8, ncallsj8) = BFGSlinesearch(flattenfunction(LJ(sigma,epsilon)),␣
 ↪flattengradient(LJgradient(sigma,epsilon)), ArSt['Xstart8'])
fffj8 = flattenfunction(LJ(sigma,epsilon))(xj8)
print(fffj8)

(r0,nr) = golden_section_min(lambda X : V(np.array([[X, 0, 0],[0,0,0]])), a, b)
optdist8 = np.sum(abs(distance(xj8.reshape(-1,3)) - r0)/r0 <= 0.01)/2
print(optdist8)
```

```
(wj9, xj9, ncallsj9) = BFGSlinesearch(flattenfunction(LJ(sigma,epsilon)),␣
↪flattengradient(LJgradient(sigma,epsilon)), ArSt['Xstart9'])
fffj9 = flattenfunction(LJ(sigma,epsilon))(xj9)
print(fffj9)
optdist9 = np.sum(abs(distance(xj9.reshape(-1,3)) - r0)/r0 <= 0.01)/2
print(optdist9)
```

```
-0.9969999999999999
-2.9909999999999997
-5.981999999999999
-9.076540858460435
-12.193939574577536
-15.486460874411144
-19.753742034351745
15.0
-22.114190613056394
19.0
```

[400]:
```
print(ncallsj, ncallsj3)
```

```
57 343
```

[410]:
```
# plotting for N = 9
x9new = xj9.reshape(9,-1)
fig = plt.figure(figsize=(11,6))
ax = plt.axes(projection='3d')
ax = fig.add_subplot(1, 2, 1, projection='3d')
ax.scatter3D(x9new[:,0], x9new[:,1], x9new[:,2])
ax.set_title('3D for N=9, points');
ax = fig.add_subplot(1, 2, 2, projection='3d')
ax.scatter3D(x9new[:,0], x9new[:,1], x9new[:,2])
ax.plot_trisurf(x9new[:,0], x9new[:,1], x9new[:,2],
                color = 'magenta', edgecolor='none');
ax.set_title('3D for N=9, surface');
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

0.8 (j)

The number of steps needed to converge goes down for N = 2 and N = 3, but since nothing higher converged for me, I don't know if it takes less to converge for higher N. I can get all through N = 9 to converge with the line search added, which is new. (My apologies for the wait in getting the values for N = 7 and higher.) As far as I can tell, the number of Van der Waals bonds stays the same for this, but that likely has to do with how few N converged for me before. Since it

took 18 calls for N = 2, and 213 calls for N = 3 in the previous BFGS, and now 57 and 343 calls, respectively, it actually takes more calls to converge.

[ ]:
```
# (k)
# oh yeehaw! bonus task: meta-heuristics
```