****NOTE: For all code files, if not directly shown, the following 2 lines are run prior**

Import numpy as np

Import matplotlib.pyplot as plt

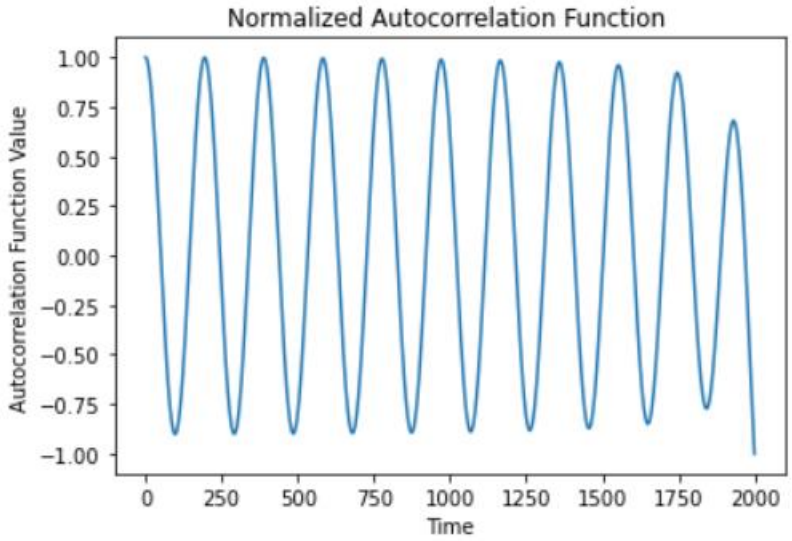
1a

```
#problem 1a
import numpy as np
import matplotlib.pyplot as plt
def autoc(data):
    array = np.load(data);

    time_length = array.shape[0]
    obs_avg = np.average(array) #<A>
    delta_array = np.empty((array.shape[0],)) # initialize empty delta array

    #calculate A-<A> for each row, and populate the empty array
    for row in range(0, array.shape[0]):
        delta_array[row] = array[row]-obs_avg
    #calculate C(t) for all the 2000 timesteps.
    #this means calculating C(0) = avg(A(0)*A(0), A(1)*A(1)......A(N)*A(N))
    # and C(1) = avg(A(0)A(1), A(1)A(2), .......A(N-1)A(N))
    # so on and so forth until C(2000)

    Cd_unnorm = np.empty((time_length,))
    #loop over each row
    for row in range(0, time_length):
        #interim list holding all the products for the Correlation function for time i, C(i)
        placeholder = []
        for value in range(0,time_length-row):
            #calculate each item in the set for C(i); each item is a product of C(value) and C(value+t)
            product = delta_array[value]*delta_array[value+row]
            placeholder.append(product)
        #calculate the unnormalized value for C(i): this is the average of all the items in the set created in the inner loop
        Cd_unnorm[row] = np.average(placeholder)


    Cd_unnorm_min = min(Cd_unnorm)
    Cd_unnorm_max = max(Cd_unnorm)
    Cd_norm = [ 2*(row-Cd_unnorm_min)/(Cd_unnorm_max-Cd_unnorm_min) -1 for row in Cd_unnorm]

    return Cd_norm #list
```

1b

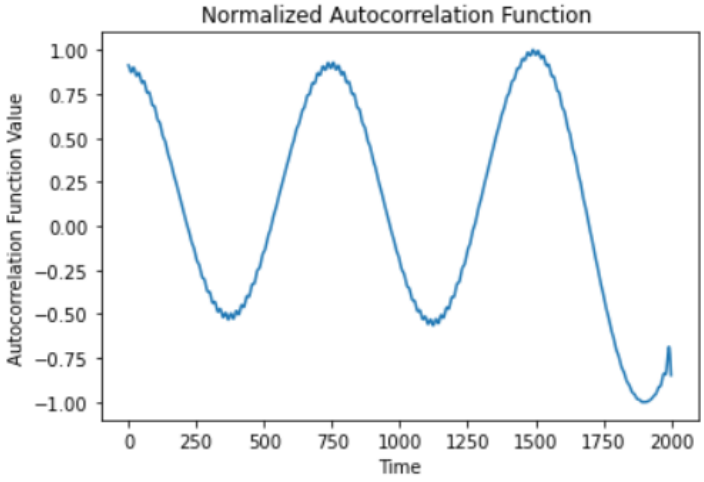
```
#Problem 1b
data_1b = np.load('HO_v.npy')
autoc_data_1b = autoc('HO_v.npy')
#print(autoc_data_1b)

time_vals = list(range(len(autoc_data_1b)))
plt.plot(time_vals, autoc_data_1b)
plt.title('Normalized Autocorrelation Function ')
plt.xlabel('Time')
```

```
10. plt.ylabel('Autocorrelation Function Value')
```



1c

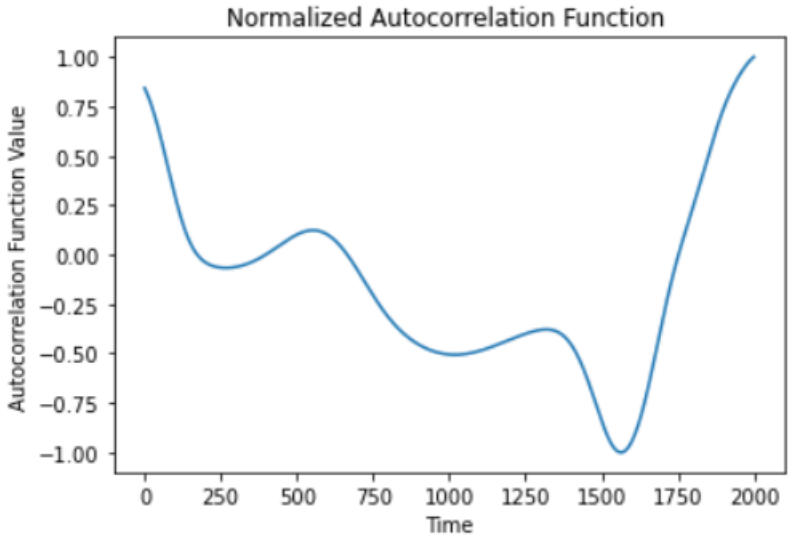
```
1.  #Problem 1c
2.  autoc_data_1c = autoc('ar2_v.npy')
3.  time_vals = list(range(len(autoc_data_1c)))
4.  plt.plot(time_vals, autoc_data_1c)
5.  plt.title('Normalized Autocorrelation Function ')
6.  plt.xlabel('Time')
7.  plt.ylabel('Autocorrelation Function Value')
```



1d

```
1.  #Problem 1d
2.  autoc_data_1d = autoc('ar_box_v.npy')
3.  time_vals = list(range(len(autoc_data_1d)))
4.  plt.plot(time_vals, autoc_data_1d)
```

```
plt.title('Normalized Autocorrelation Function ')
plt.xlabel('Time')
```



1e

```
#Problem 1e
#Calculate Correlation time
sum_autoc_data_1d = np.sum(np.abs(autoc_data_1d[0:876]))
t_corr = 1+2*sum_autoc_data_1d
print("Correlation Time: {}".format(t_corr))

#Calculate standard error: sigma/sqrt(N) using the reduced time series
#<A^2>

data_sq= [row*row for row in autoc_data_1d]
avg_data_sq = np.average(data_sq)

#<A>^2
data_avg = np.average(autoc_data_1d)
data_avg_sq = np.power(data_avg,2)

error = np.sqrt(avg_data_sq - data_avg_sq)/np.sqrt(len(data_sq))
print("Standard Error: {}".format(error))

#reduced dataset
data_red = [autoc_data_1d[0], autoc_data_1d[int(t_corr)]]
data_red_sq= [row*row for row in data_red]
avg_data_red_sq = np.average(data_red_sq)

#<A>^2
data_red_avg = np.average(data_red)
data_red_avg_sq = np.power(data_avg,2)

error = np.sqrt(avg_data_red_sq - data_red_avg_sq)/np.sqrt(len(data_red))
print("Standard Error from Reduced Set: {}".format(error))
```

```
Correlation Time: 300.9498958833786
Standard Error: 0.010139813129531222
```

```
Standard Error from Reduced Set: 0.409164577825459
```

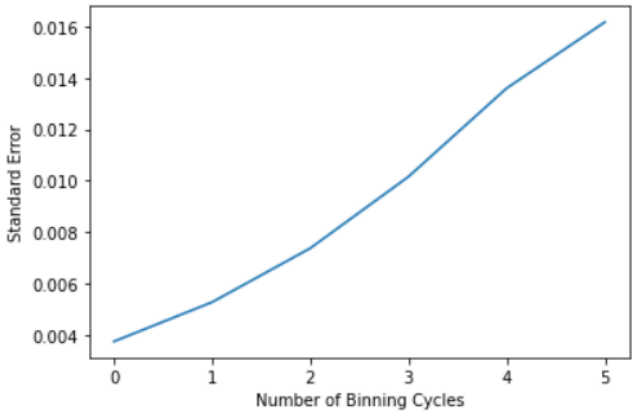
Comments on 1e

For this simulation it is important to truncate in the summation for your correlation time, as summing over the whole correlation function (which steeply increases past t=1600) will only skew your correlation time and increase the standard error. It is necessary to decide on time step for truncation which resembles the exponential decay expected in an ideal correlation function. Aside from the first peak at ~t=625, the correlation function follows the expected global decay until ~t=875. As such, t=876 is chosen as the upper bound in the summation for calculating the correlation time. Doing this, rather than summing over all 2000 timesteps, reduces the correlation time from 1527 to 301.

1f

```
#problem 1f
import numpy as np
import matplotlib.pyplot as plt
def binning(data):
    binlist = []
    for i in range(1, int(np.floor(len(data)/2))):
        binlist.append(.5*(data[(2*i) - 1] + data[2*i]))
    return binlist

data_1f = np.load('ar_box_v.npy')
bin_1f=data_1f

#initial value for the #items, which will become smaller and approach 30 as binning progresses
N_l = np.inf
print('Initial Standard Error: {}'.format(np.std(data_1f)/(np.sqrt(len(data_1f)))))
stderr_1f = []
while (N_l > 30):
    bin_1f = binning(bin_1f)
    delta_f = np.std(bin_1f)/(np.sqrt(len(bin_1f)))
    N_l = len(bin_1f)
    print('Standard Error After {} Binning: {}'.format(int(np.floor(np.log2(len(data_1f)/len(bin_1f)))), delta_f))
    stderr_1f.append(delta_f)

yvals_1f = stderr_1f
xvals_1f = list(range(len(stderr_1f)))
plt.plot(xvals_1f, yvals_1f)
plt.xlabel('Number of Binning C
```

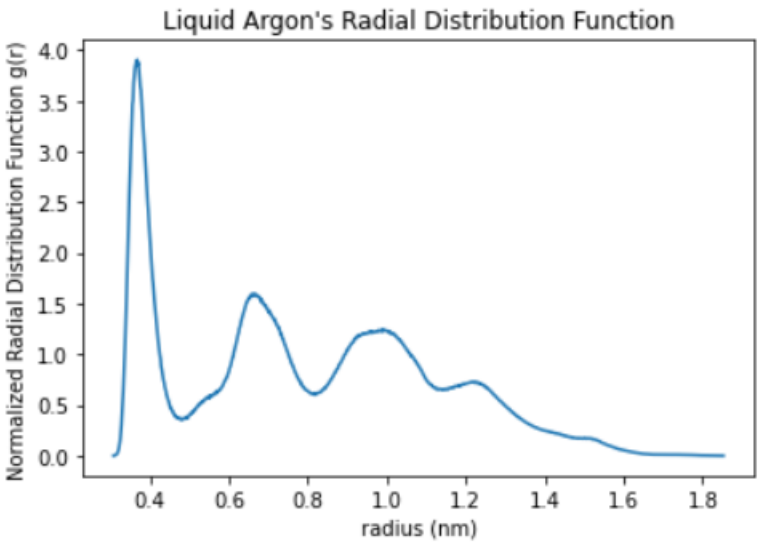
```
Initial Standard Error: 0.0026609797394669056
Standard Error After 1 Binning: 0.003752286468509703
Standard Error After 2 Binning: 0.005280250849790534
Standard Error After 3 Binning: 0.007375907272658209
Standard Error After 4 Binning: 0.010156074361279641
Standard Error After 5 Binning: 0.013592861509651055
Standard Error After 6 Binning: 0.016161251168909582
```



While this data appears to be diverging, because these errors are increasing only by thousandths, it can be said that the error is not changing by much. As well, if we had more datapoints than just 2000, it could be shown that this convergence is more definitive.

2a Part 1

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.  #problem 2a: Liquid Argon
4.  #part i
5.  num_bins = 1000 #fxn becomes noiser for 10k, 100k
6.  data_2a = np.load('ar_liquid_250_pairs.npy')
7.  #flatten into 31125*1000 array to make it easier to histogram
8.  data_2a = data_2a.flatten()
9.
10. #array returned with hist entries as type int64 and binedge as type float32
11. liq_hist, liq_binedge= np.histogram(data_2a, bins=num_bins)
12. #change datatype to np.float 64 or you'll get a graph w/ square waves
13. liq_hist = liq_hist.astype(np.float64)
14. print(liq_binedge.shape)
15.
16. #normalization by dividing by the density
17. #each enumerate item has 2 member tuple (i the index, and the indexed item)
18. for i, value in enumerate(liq_hist):
19.     #dividing by the #particles in the differential volume element divided by box
    volume (dV*N / s^3)
20.     liq_hist[i] = liq_hist[i] / ((4/3)*np.pi*(np.power(liq_binedge[i+1],3) -
    np.power(liq_binedge[i],3)) * 31125000 / (np.power(2.14186,3)))
21.     #note ** is faster than np.power, but np.power I think is more precise
22. plt.plot( liq_binedge[1:], liq_hist)
23. plt.xlabel('radius (nm)')
24. plt.ylabel('Normalized Radial Distribution Function g(r)')
25. plt.title('Liquid Argon\'s Radial Distribution Function')
```



2a Part 2

```
1.  #part ii
2.  rc = .51 # cutoff radius
3.  index = 0 #index for returning radial coordinate to definite integral.
4.  integ =0 # initializing blank variable for integral (or point along the xaxis)
5.  #while the value of the indexed binedges are less than rc
6.  while (liq_binedge[index] < rc):
7.      # dr * r^2 * g(r). Note because this is discretized, you're just performing a
    summation
8.      integ += ((liq_binedge[index+1] - liq_binedge[index]) * (liq_binedge[index]**2) *
    (liq_hist[index]))
9.      #increment the binedge index (moving right along the x-axis or to the next
    subsequent larger radial point)
10.     index += 1
11.
12.     #n(rc) = 4*pi*rho (the density)*integrand
13. nn_rc = (4*np.pi*(250/2.14186**3)) * integ
14. print("Liquid Argon has {} Nearest Neighbors ".format(np.round(nn_rc,2)))
```

Output: Liquid Argon has 13.37 Nearest Neighbors

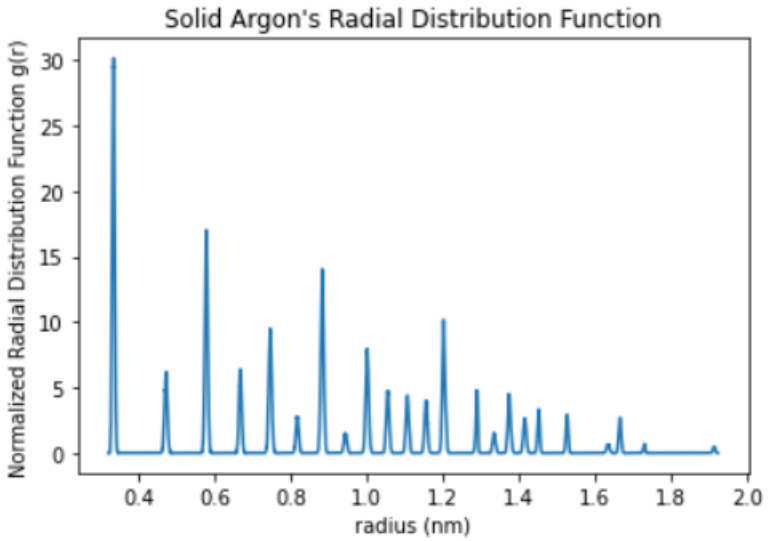
Comments on 2a

13 nearest neighbors (henceforth abbreviated NN) is quite large when comparing with NN=7 in literature at 91.8K and P=1.8atm (DOI: 10.1103/PhysRev.67.285). This is likely due to the Lennard-Jones (LJ) potential from OpenMM differing from the LJ potential modified by Corner (J. Corner, Trans. Faraday Soc. 35, 711 (1939)). It's important to note that the simulation's infinite limit does not approach 1 like an ideal liquid as seen in lecture, which also confirms the simulated potential should be refined. Interesting to note is we see the Function starting to globally decay after half the box edge length.

2b Part 1

```
1.  #problem 2b: Solid Argon
2.  import numpy as np
```

```
3.  import matplotlib.pyplot as plt
4.  #part i
5.  num_bins = 1000 #fxn becomes noiser for 10k, 100k
6.  data_2b = np.load('ar_solid_500_pairs.npy')
7.  #flatten into 31125*1000 array to make it easier to histogram
8.  data_2b = data_2b.flatten()
9.  num_pts = data_2b.shape[0]
10. l_box = 2.35743 #nm
11.
12. #array returned with hist entries as type int64 and binedge as type float32
13. sol_hist, sol_binedge= np.histogram(data_2b, bins=num_bins)
14. #change datatype to np.float 64 or you'll get a graph w/ square waves
15. sol_hist = sol_hist.astype(np.float64)
16.
17.
18. #normalization by dividing by the density
19. #each enumerate item has 2 member tuple (i the index, and the indexed item)
20. for i, value in enumerate(sol_hist):
21.     #dividing by the #particles in the differential volume element divided by box
    volume (dV*N / s^3)
22.     sol_hist[i] = sol_hist[i] / ((4/3)*np.pi*(np.power(sol_binedge[i+1],3) -
    np.power(sol_binedge[i],3)) * num_pts / (np.power(l_box,3)))
23.     #note ** is faster than np.power, but np.power I think is more precise
24. plt.plot(sol_binedge[1:], sol_hist)
25. plt.xlabel('radius (nm)')
26. plt.ylabel('Normalized Radial Distribution Function g(r)')
27. plt.title('Solid Argon\'s Radial Distribution Function')
```

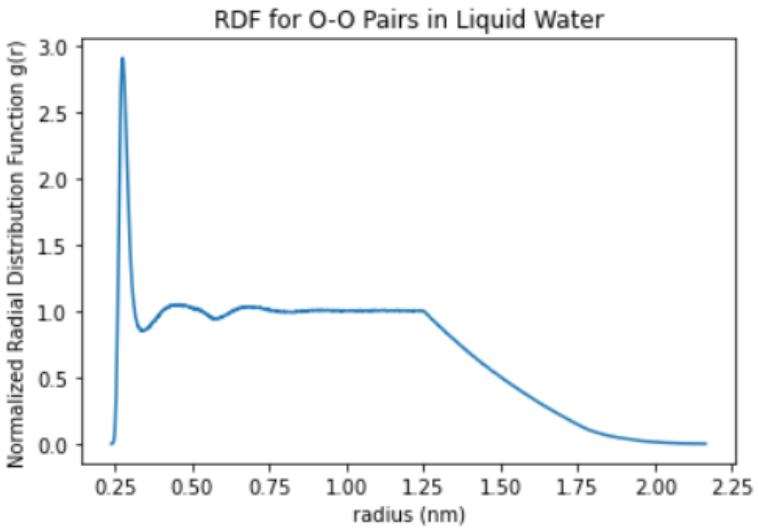


Comments on 2b

Compared to a similar Function developed for solid argon at 80K (DOI: 10.1007/BF02738574) the general decline of the most prominent peaks is observed, but shifted .1-.2nm right, with much smaller Function values (maxing out at ~3.7 for the first peak, for example), and with blurred minor peaks. Since our simulation was run at 10K, it's expected that the Function peak values will be larger. As well, we anticipate too that the peaks could be shifted to larger r values, and the minor peaks are blurred/smoothened, owing to the increased thermal energy at 80K. Unlike the referenced paper, our simulation does not tend to 1 in the radial infinite limit, suggesting that the model requires refining.

2c Part 1

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.  #RDF plotting function
4.  def rdf(data, nbins, l_box, title):
5.      data = data.flatten()
6.      num_pts = data.shape[0]
7.
8.      #array returned with hist entries as type int64 and binedge as type float32
9.      sol_hist, sol_binedge= np.histogram(data, bins=num_bins)
10.     #change datatype to np.float 64 or you'll get a graph w/ square waves
11.     sol_hist = sol_hist.astype(np.float64)
12.
13.     #attributes for later accessing in part iii
14.     rdf.hist = sol_hist
15.     rdf.binedge = sol_binedge
16.
17.     #normalization by dividing by the density
18.     #each enumerate item has 2 member tuple (i the index, and the indexed item)
19.     for i, value in enumerate(sol_hist):
20.         #dividing by the #particles in the differential volume element divided by box
    volume (dV*N / s^3)
21.         sol_hist[i] = sol_hist[i] / ((4/3)*np.pi*(np.power(sol_binedge[i+1],3) -
    np.power(sol_binedge[i],3)) * num_pts / (np.power(l_box,3)))
22.         #note ** is faster than np.power, but np.power I think is more precise
23.     a = plt.plot(sol_binedge[1:], sol_hist)
24.     plt.xlabel('radius (nm)')
25.     plt.ylabel('Normalized Radial Distribution Function g(r)')
26.     plt.title(title)
27.     return a
28.
29. #Problem 2ci: O-O Distances in liquid water
30. num_bins = 1000
31. data_2ci = np.load('h2o_liquid_OO_pairs.npy')
32. box_length = 2.5 #nm
33. str = 'RDF for O-O Pairs in Liquid Water'
34. rdf(data_2ci, num_bins, box_length, str)
```

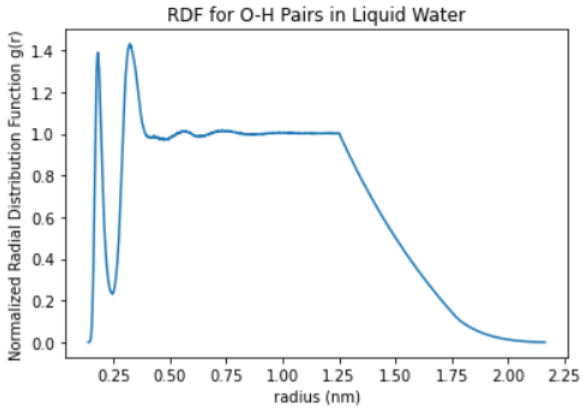


Comments on 2c Part 1

The peak at ~.3nm, and the subsequent two minima and maxima in this simulation are closely resembled in literature (DOI: 10.1021/acs.jpclett.5b01066), which was also simulated at ambient conditions. Similar to previous simulations in this assignment, the global decay is seen to commence at approximately half the box length.

2c Part 2

```
#Problem 2cii: O-H Distances in liquid water
#The same function used in Part 1 of 2c is called here
num_bins = 1000
data_2cii = np.load('h2o_liquid_OH_pairs.npy')
box_length = 2.5 #nm
str = 'RDF for O-H Pairs in Liquid Water'
rdf(data_2cii, num_bins, box_length, str)
```



Comments on 2C Part 2

The first two peaks and primary local minima match literature remarkably well in both Function and radius values (DOI: 10.1063/1.1783871). The paper's simulation was done with 256 molecules and a TIP4P model, which resembles an LJ potential summed with a Coulombic potential. The global decay seen resembles that of the O-O liquid water Function.

2c part 3

```
#problem 2ciii: NN for O-O rdf

#run code from Problem 2ci again to get hist and bin_edges
num_bins = 1000
data_2ci = np.load('h2o_liquid_OO_pairs.npy')
box_length = 2.5 #nm
str = 'RDF for O-O Pairs in Liquid Water'
rdf(data_2ci, num_bins, box_length, str)
#suppress the output of rdf
plt.close()

#these attributes are from the function in 2c Part 1
hist_2c3 = rdf.hist;
binedge_2c3 = rdf.binedge;

```

```
16.
17. rc = .4725 #nm
18. index = 0 #index for returning radial coordinate to definite integral.
19. integ =0 # initializing blank variable for integral (or point along the xaxis)
20. #while the value of the indexed binedges are less than rc
21. while (binedge_2c3[index] < rc):
22.     # dr * r^2 * g(r). Note because this is discretized, you're just performing a
    summation
23.     integ += ((binedge_2c3[index+1] - binedge_2c3[index]) * (binedge_2c3[index]**2) *
    (hist_2c3[index]))
24.     #increment the binedge index (moving right along the x-axis or to the next
    subsequent larger radial point)
25.     index += 1
26.
27.     #n(rc) = 4*pi*rho (the density)*integrand
28. nn_rc = (4*np.pi*(501/2.5**3)) * integ
29. print("Liquid Water has {} Nearest Neighbors ".format(np.round(nn_rc,2)))
```

Output: Liquid Water has 13.5 Nearest Neighbors

Comments on 2C Part 3

This value seems intuitively quite large, which could mean that the simulation needs to be improved, whether with a better thermostat, more time steps, or a more comprehensive initialization and equilibration. Though this does contradict the surprisingly accurate resemblance seen in 2C Part 1.

3

VE 452 Assignment 2 Question 3

$$n(\vec{r}_c) = \int_0^{\vec{r}_c} d\vec{r}\, \rho g(\vec{r}) = n(r_c) = 4\pi\rho \int_0^{r_c} dr\, r^2 g(r)$$

particles within $\vec{r}_c$ (within the cutoff vector's distance from the tagged particle) = # neighbors within distance r_c from tagged atom

Since the variable of integration in the left integral is $\vec{r}$, meaning in the left integral we're integrating over all vectors from the tagged particle, $\vec{r}$ is effectively 3 dimensions for a sphere around the tagged particle).

This differential volume element can be expanded in terms of r, the radial coordinate:

$$d\vec{r} = \frac{4\pi}{3}(r+dr)^3 - \frac{4\pi r^3}{3} = \frac{4\pi}{3}\left(r^3 + 2r^2dr + rdr^2 + r^2dr + 2rdr^2 + dr^3 - r^3\right)$$

$$= \frac{4\pi}{3}\left(3r^2dr + 3rdr^2 + dr^3\right) = 4\pi r^2 dr$$

($dr \to 0$, $dr^3 \to 0$)

In terms of the radial coordinate r, instead of the 3D cartesian vector $\vec{r}$, the top left integral now reads:

$$n(\vec{r}_c) = \int_0^{\vec{r}_c} d\vec{r}\, \rho g(\vec{r}) = \int_0^{r_c} \rho g(r)\, 4\pi r^2 dr = \underbrace{4\pi\rho \int_0^{r_c} r^2 dr\, g(r)}$$

This is the top right integral