Please refer to files "mdsim_micro-canonical.py" and "mdsim_canonical.py" for more information about the coding, if necessary.

- Instantaneous Temperature

Q: Add code to output the instantaneous temperature and output it along with the kinetic energy.

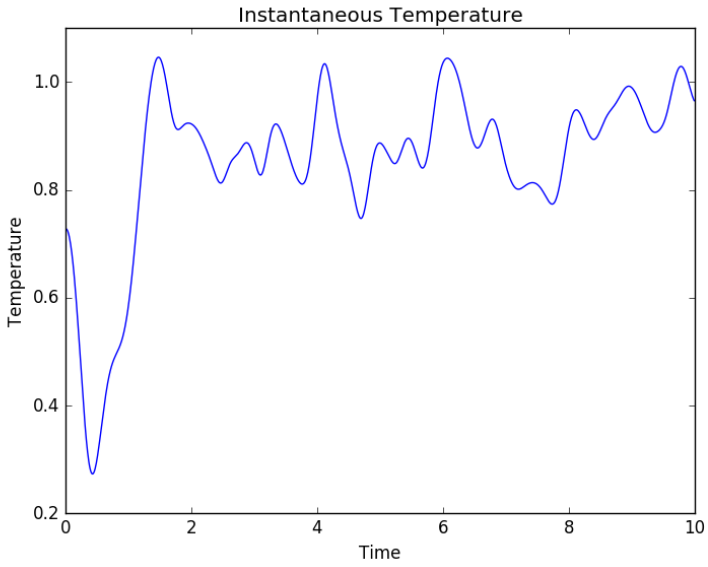
In my code this part is written into function compute_energy(). Numerically the instantaneous temperature is the instantaneous kinetic energy divided by 1.5N.

```
def compute_energy(pset,box_length):
    natom = pset.size()  # number of particles
    vel = pset.all_vel() # all particle velocities
    mass = pset.mass()   # particle mass
    ndim = pset.ndim()   # dimension

    tot_kinetic   = 0.0
    tot_potential = 0.0

    # calculate total kinetic energy
    for atom in range(natom):
        for idim in range(ndim):
            tot_kinetic += 0.5*mass*vel[atom][idim]*vel[atom][idim]
    # end calculating total kinetic energy

    # calculate total potential energy
    for iat in range(natom):
        for jat in range(iat):
            tot_potential += LJ_Potential(iat,jat,pset,box_length)
    # end calculating total potential energy

    tot_energy = tot_kinetic + tot_potential

    temperature = tot_kinetic / (1.5*pset.size()) # instantaneous temperature
    return (tot_kinetic, tot_potential, tot_energy, temperature)
# end def compute_energy
```

In the MD loop in the main function, instantaneous temperature is outputted along with the kinetic energy (and potential, total energies) as the simulation goes.

```
        """calculate properties of the particles, printing kinetic, potential
           and total energies, along with the system temperature
        """
        print(istep, compute_energy(pset,box_length))
```

Plot the instantaneous temperature over time, we can see that temperature isn't stable and has a tendency to increase to about 1.0 when time approaches 10.

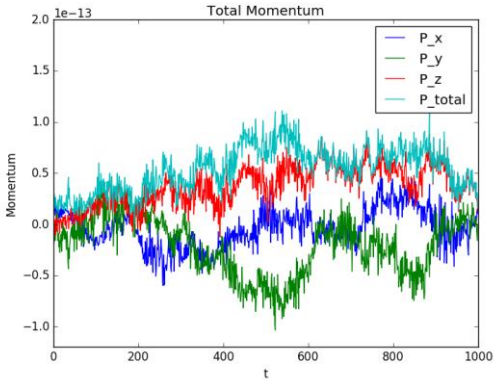


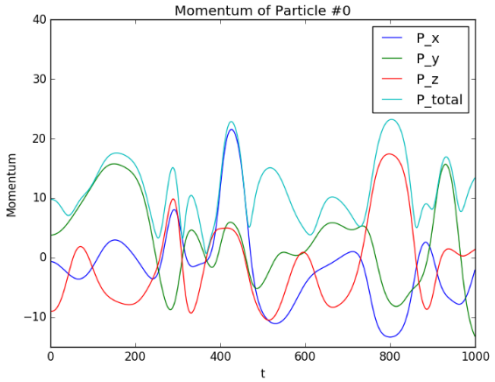
- Momentum

Q: Measure the total momentum of your system as it evolves in time. Is momentum conserved? Do you expect it to be conserved?

I expect the total momentum to conserve as the system of the group of atoms is not influenced by any external force or interaction.

The total momentum is calculated as the system evolves and the result is shown below. The total momentum, along with its projections in x, y and z directions, though fluctuating, stays within the range of 1e-13. Comparing to the momentum change of a specific particle, e.g. particle No.0 which varies from approximately -10 to 25, the fluctuation of the total momentum is negligible. So the calculation can also verify that the total momentum is conserved. The fluctuation mainly comes from errors, including but not limited to, round-off errors.





Function compute_momentum() is used to compute total momentum in each MD step.

```
def compute_momentum(pset,box_length):
    natom = pset.size()  # number of particles
    vel = pset.all_vel() # all particle velocities
    mass = pset.mass()   # particle mass
    ndim = pset.ndim()   # dimension

    tot_momentum = np.zeros(ndim) # initializing momentum

    # calculating momentum
    for iatom in range(natom):
        for idim in range(ndim):
            tot_momentum[idim] += mass*vel[iatom][idim]
        # end for
    # end for
    return tot_momentum
```

- Pair Correlation Function

Q: Implement the calculation of g(r) in your code and submit a plot of g(r) for the Lennard-Jones simulations described above. Be sure to normalize your function correctly!

For each MD step after the system reaches equilibrium, which is marked by parameter step_equil, I use function compute_PCF() to compute the unnormalized g(r)

```
# calculate the pair correlation function g(r)
def compute_PCF(pset, box_length, nhis):
    rmax = box_length/2     # maximum r considered in g(r)
    dr = rmax/nhis          # resolution
    num_atoms = pset.size() # number of atoms
    g = np.zeros(nhis)      # the histogram of g(r)

    #scan over the possitions of the particles to determine the histogram
    for iat in range(0, num_atoms-1):
        for jat in range(iat+1, num_atoms):
            rij = distance(iat, jat, pset, box_length)
            if (rij < rmax):
                ihis = int(rij/dr) # determine the location in the histogram
                g[ihis] += 2       # contribution of particle iat and jat
            # end if
        # end for
    # end for

    return g
# end def compute_PCF
```

In the main function, add up g(r) for every MD step (S(k) and VVC are also shown here):

```
"""After equilibrium, calculate g(r) and S(k)
   record velocities for calculating VVC
"""
if istep >= step_equil:
    # calculate g(r)
    g_renewal = compute_PCF(pset, box_length, nhis)
    for ihis in range(nhis):
        g[ihis] += g_renewal[ihis]
    # end for

    # calculate S(k)
    S_renewal = Sk(kvecs, pset)
    for ik in range(S_size):
        S[ik] += S_renewal[ik]
    # end for

    # record velocities of all particles for calculating VVC
    v_record.insert(istep-step_equil, pset.all_vel())
```

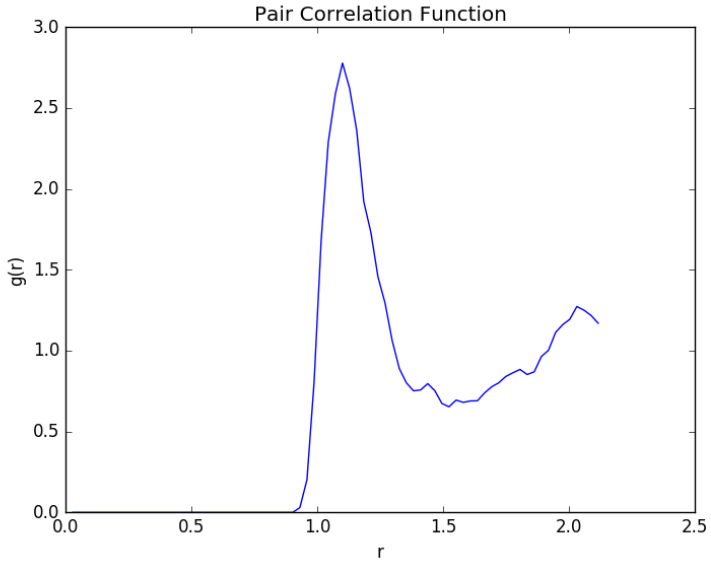
After the MD loop ends, normalize and plot g(r):

```
# normalize and plot g(r)
for ihis in range(nhis):
    dV = ((ihis+1)**3-ihis**3)*dr**3
    dV *= (4/3)*math.pi # the volume of the shell considered
    num_ideal = num_atoms*dV/V  # number of ideal gas atoms in dV
    g[ihis] /= (num_ideal*num_atoms*(nsteps-step_equil)) # normalization
plot_PCF(nhis, dr, g)
```

where some parameters are defined at the beginning of the main function:

```
# parameters useful in calculating g(r)
nhis = 75           # the number of intervals in the histogram
g = np.zeros(nhis)  # pair correlation function g(r)
rmax = box_length/2 # maximum r considered in g(r)
dr = rmax/nhis      # resolution
V = box_length**3   # volume of the box
```

The plot of g(r):



- Structure Factor

Q: Implement the calculation of S(k) in your code and submit a plot of S(k) from a run using the Lennard-Jones system parameters described above. Be sure to normalize your function correctly! Describe qualitatively how S(k) and g(r) should look for a liquid and a solid.

For a solid, the curve of S(k) or g(r) is likely to be composed of "spikes", which means the magnitude of S(k) or g(r) becomes great sharply at some values of k or r, but remains approximately 0 for the rest. For a liquid, the curve may have several peaks with a larger width (hence less sharp), and keeps fluctuating. It is much smoother compared to that of a solid.

First, find out the legal k-vectors for computing S(k) and ρ_k in function legal_kvecs(). Legal k-vectors satisfy

$$\vec{k} = \frac{2\pi}{L}\left(n_x, n_y, n_z\right)$$

where $n_x, n_y, n_z = 0,1,2...maxk$ and cannot be 0 at the same time.

```
# compute a list of the legal k-vectors for computing S(k)
def legal_kvecs(maxk, pset, box_length):
    kvecs=[]
    # calculate a list of legal k vectors
    for vec in itertools.product(range(maxk+1),repeat=pset.ndim()):
        if np.linalg.norm(vec)<1e-5:
            continue
        kvecs.insert(0, np.array(vec) * (2*math.pi/box_length))
    return np.array(kvecs)
# end def legal_kvecs
```

Then compute ρ_k for a given vector kvec in function rhok():

```
# Fourier transform to calculate rho_k
def rhok(kvec, pset):
    value = 0.0
    #computes \sum_j \exp(i * k \dot r_j)
    for iatom in range(pset.size()):
        kr = np.dot(kvec, pset.pos(iatom)) # dot product of k and atom position
        value += np.cos(kr) + 1j*np.sin(kr)
    return value
# end def
```

Then we can compute an instantaneous S(k) for one certain time step in MD.

```
# calculate the structure factor S(k)
def Sk(kvecs, pset):
  """ compute structure factor for all k vectors in klist
   and return a list of them. """
  sk_list = np.zeros(len(kvecs))
  index = 0
  for ivec in kvecs:
    sk_list[index] += (rhok(ivec,pset) * rhok(-ivec,pset)).real/pset.size()
    index += 1
  return sk_list
# end def Sk
```

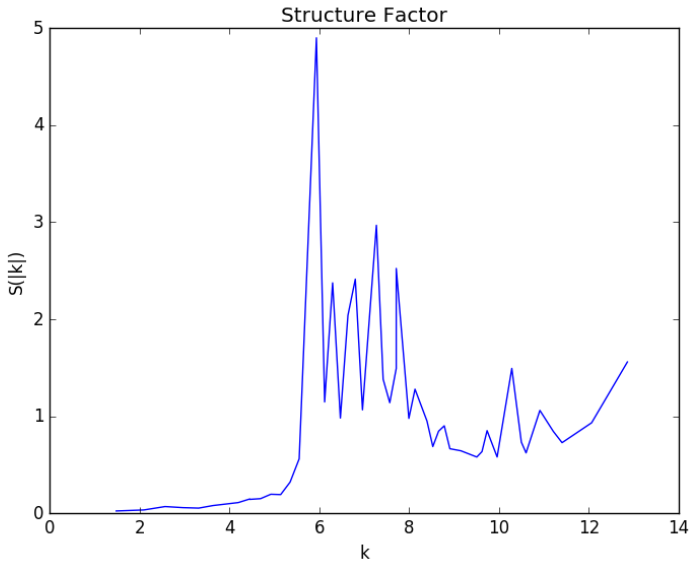
In the main function, with the parameters defined in the beginning:

```
# parameters useful in calculating S(k)
maxk = 5                # the maximum of n_x, n_y and n_z in the k vector
S_size = (maxk+1)**3-1 # the size of S list or the number
                        # of legal k vectors, eliminating vector [0,0,0]
S = np.zeros(S_size)    # structure factor
kvecs = legal_kvecs(maxk,pset,box_length) # legal k vectors
```

we can add up S(k) in the MD loop (where we also add up g(r) and record particle velocities for computing VVC), and normalize it after MD loop ends

```
# average and plot S(k)
for ik in range(S_size):
    S[ik] /= (nsteps-step_equil)
plot_Sk(kvecs, S)
```

The plot of S(k):



- Velocity-Velocity Correlation and Diffusion Constant

Q: Implement a calculation of this quantity as a function of time t; be sure to normalize your function.

Q: Implement a diffusion constant observable. Using the simulation parameters described above, produce a plot of the velocity-velocity correlation as a function of time. Also report your value for the diffusion constant.

First, in the MD loop, record all the particle velocities in each MD step. This is done along with the add-ups of g(r) and S(k).

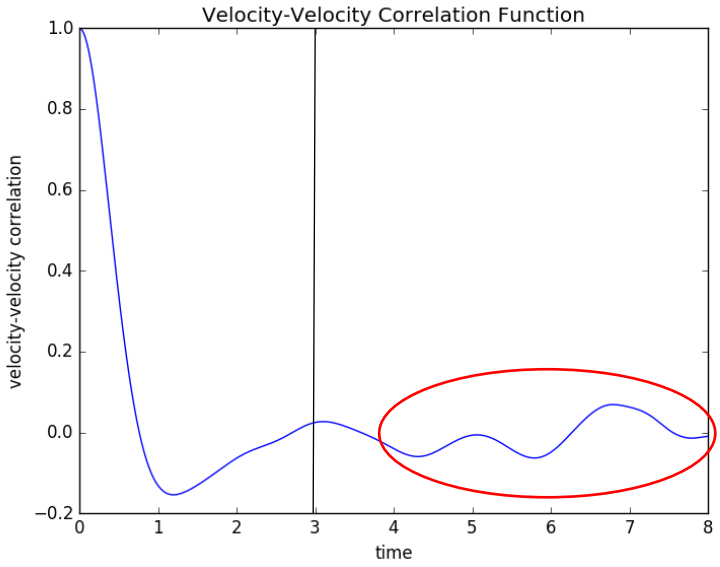
Then, after the MD loop, calculate, normalize and plot VVC. And then calculate diffusion constant.

```
# calculate normalized velocity-velocity correlation
VVC = VVCorr(v_record, N_stored, pset.size())

# plot velocity-velocity correlation function
plot_VVCorr(VVC, dt)

# calculate diffusion constant
D = Diffusion_Constant(VVC, N_stored, dt)
print("Diffusion Constant = ", D)
```

where the functions called are as follows

```
# calculate the normalized velocity-velocity correlation function
def VVCorr(v_record, N_stored, num_atoms):
    VVC = np.zeros(N_stored) # velocity-velocity correlation

    """calculate velocity-velocity correlation(unnormalized)
       VVC[t] is defined as <V(t0)V(t0+t)>
       where the angular bracket means averaging over t0 and particles
    """
    for t in range(N_stored):
        for t0 in range(N_stored-t):
            for iat in range(num_atoms):
                VVC[t] += np.dot(v_record[t0][iat], v_record[t0+t][iat])
            # end for
        # end for
        VVC[t] /= num_atoms    # average over all the particles
        VVC[t] /= (N_stored-t) # average over all the t0's
    # end for

    # normalize velocity-velocity correlation
    vvc0 = VVC[0] # normalize by dividing <v(t0)v(t0)>
    for t in range(N_stored):
        VVC[t] /= vvc0
    # end for

    return VVC
# end def VVCorr

# plot velocity-velocity correlation function over time
def plot_VVCorr(VVC, dt):
    tmax=len(VVC)
    time = np.array(range(tmax))*dt # real time
    plt.plot(time, VVC)
    plt.xlabel("time")
    plt.ylabel("velocity-velocity correlation")
    plt.title("Velocity-Velocity Correlation Function")
    plt.show()
# end def plot_VVCorr

def Diffusion_Constant(VVC, N_stored, dt):
    D = 0.0 # diffusion constant
    for itau in range(N_stored):
        D += VVC[itau]*dt
    # end for
    return D
# end def
```

Simulation result:



Everything in the red circle is simply noise and should be thrown out when integrating for the diffusion constant.

Output: Diffusion Constant = 0.26600401343

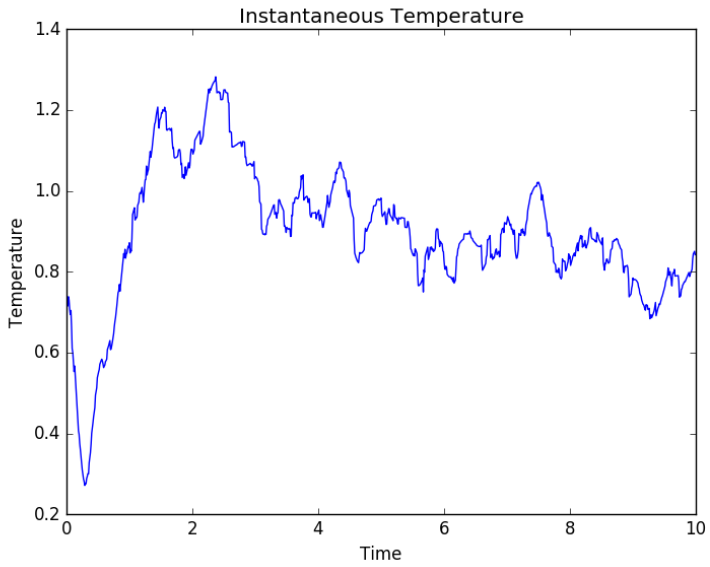
Anderson Thermostat

Q: Implement this new technique in your code. To check that it is working, re-run the Lennard-Jones simulation described above but with your thermostat, and produce a plot of the instantaneous temperature as a function of simulation time.

Add Anderson Thermostat part in the MD loop

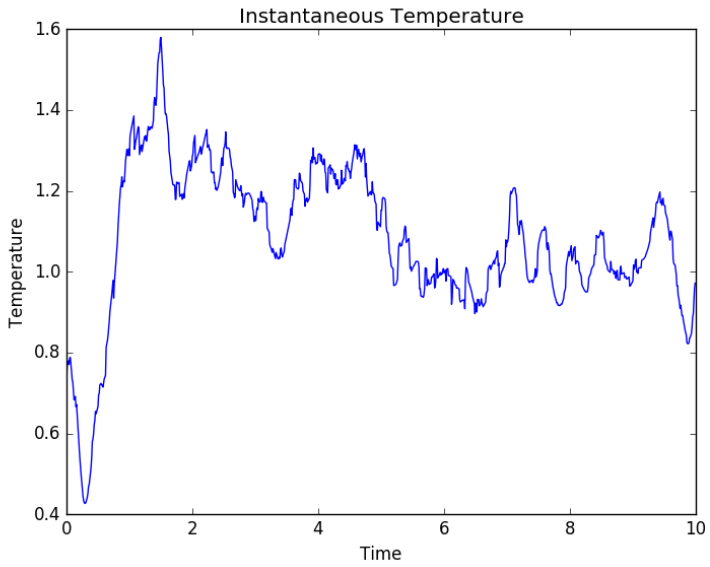
```
# Anderson Thermostat
for iat in range(num_atoms):
    # collision occurs
    if (random.random() < coll_prob):
        for idim in range(pset.ndim()):
            rand_velocity[idim] = random.gauss(0, sigma)
        pset.change_vel(iat, rand_velocity)
# end for iat
```

With the temperature of the heat bath equal to 0.728, plot the instantaneous temperature:



We can see that compared to the instantaneous temperature plot in the micro-canonical case, the temperature here tends to be stabilized around about 0.7.

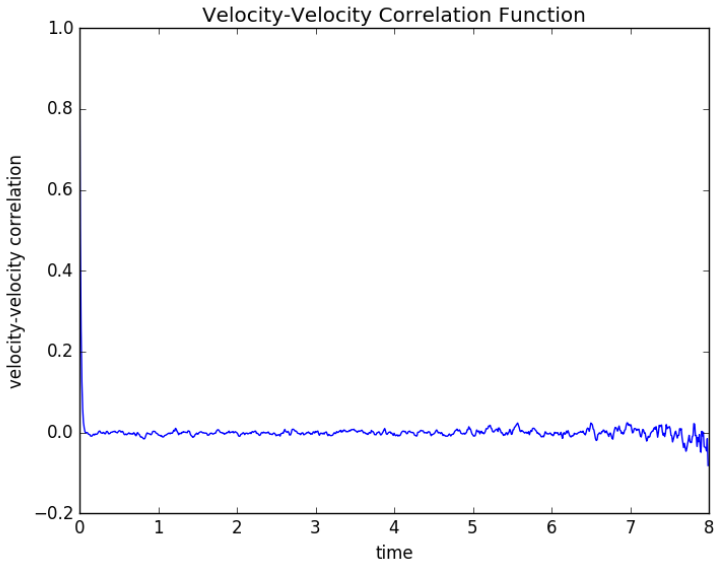
Q: Now, increase the temperature to T=1.0, but still pass the value 0.728 as an argument to the "InitVelocity" function. Re-run and produce a plot of temperature vs. time for this simulation.
The plot of temperature vs time is as below



It is obvious that the temperature has the tendency to change to approximately 1.0 from 0.728.

Q: To see this quantitatively, adjust the value of η so that your collision probability is around 50%. Produce a plot of the velocity-velocity correlation function with thermostatting on and report your calculation of the diffusion constant. Compare these results with your data from the microcanonical simulation (above). Specifically, you should discuss what differences you expect to see in these observables and whether or not you do, in fact, see them.
I expect to see velocity-velocity correlation converging to 0 faster here than in the micro-canonical situation. Physically that's because the particles are under some random influence, which will cause $v(t_0)$ and $v(t_0+t)$ less correlated. Diffusion constant, being the integral of velocity-velocity correlation over time, is consequently expected to decrease here.

The plot of velocity-velocity correlation is as below. It converges to 0 at a much faster speed compared to that in the micro-canonical situation.



Here the output gives Diffusion constant = 0.0170334873189, much smaller than that in the micro-canonical situation. The simulation result is in line with my expectation.

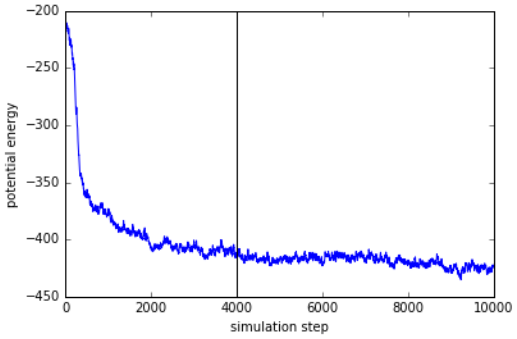
Calculating phase transitions

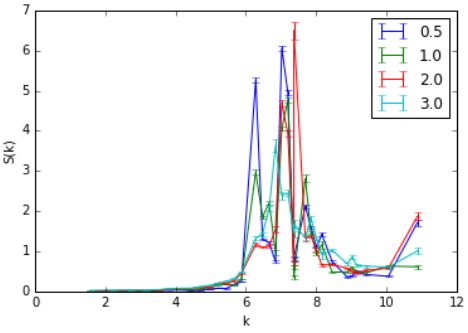
Q: Choose 5 temperatures in the range between 0 and 4 and run MD simulations at those temperatures.

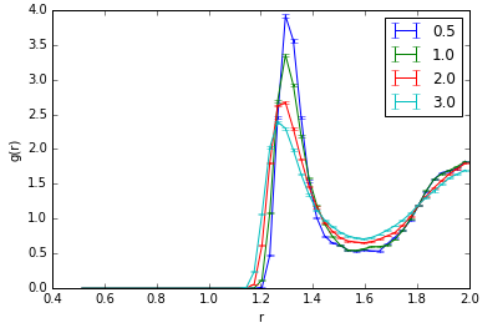
Q: Now, try to refine your estimate by choosing three more temperature points in this vicinity.

Q: Submit your estimate of the transition temperature along with data that support this claim. Minimally, you should include graphs of the pair correlation function and structure factor for temperatures above and below your estimated transition temperature.

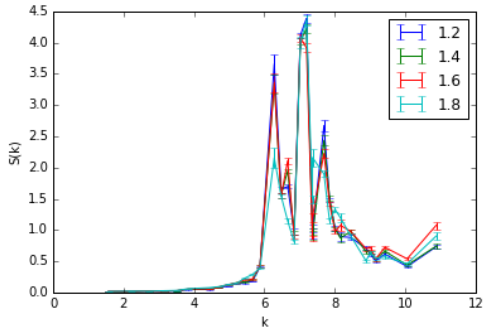
STEP 1 Make runs at T=0.5, 1.0, 2.0, 3.0, ensure equilibration

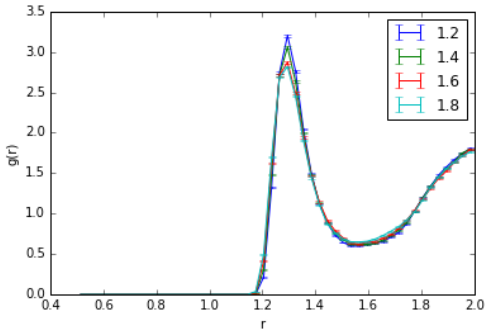




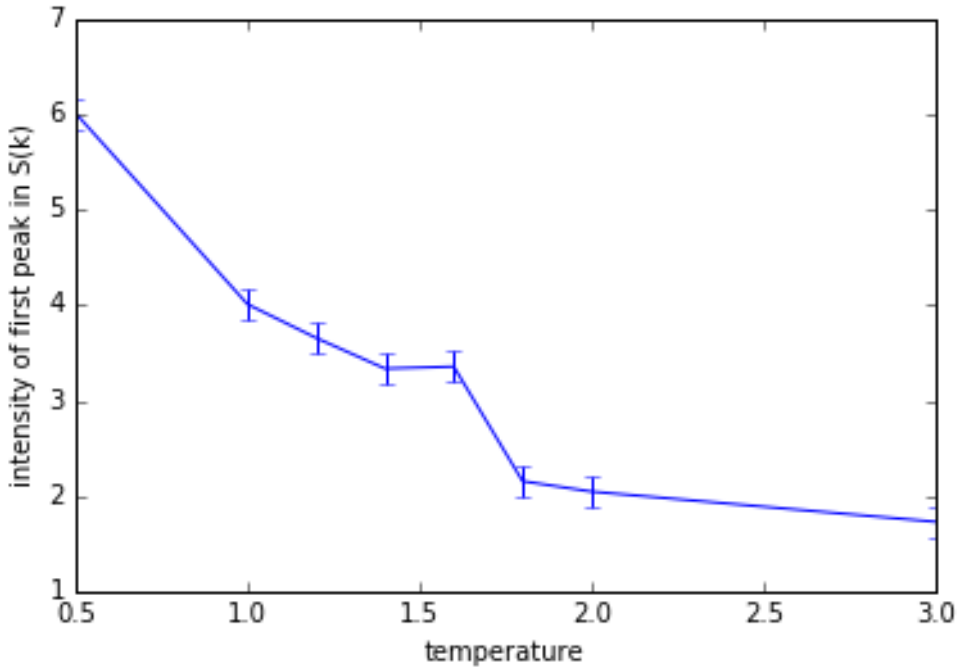


STEP 2 Refine search in the range T=1.0-2.0





STEP 3 Make a convincing plot of the transition



The increase at low temperatures is due to insufficient equilibration