Worksheet 2: Statistical Mechanics and Molecular Dynamics

April Cooper, Patrick Kreissl und Sebastian Weber

November 26, 2012
University of Stuttgart

Contents

1 Statistical Mechanics

1.1 Task 1

First we had to consider a system A consisting of two subsystems A_1 and A_2 with the related numbers of configurations $\Omega_1 = 10^20$ and $\Omega_2 = 10^22$.
The number of configurations available to the combined system is $\Omega = \Omega_1 \cdot \Omega_2 = 10^{42}$.
As the entropy is defined as $S = ln(\Omega)k_B$ the entropies are:
$S_1 = k_B ln(\Omega_1) = k_B ln(10^{20})$, $S_2 = k_B ln(\Omega_2) = k_B ln(10^{22})$, $S = k_B ln(\Omega) = k_B ln(10^{42})$.

Then were asked to give the factor by which the number of available configurations increases in a system with given initial V_1, T, p_1 if V_1 was expanded isothermal by 0.001%.
With the help of the first law of thermodynamics it follows that for T = const it is:
$dS = \frac{p}{T}dV$, which leads with the ideal gas equation to

$$\begin{aligned}\Delta S &= \int_{V1}^{V2} \frac{k_B N}{V} dV \\ &= k_B N ln\left(\frac{V_2}{V_1}\right) \\ &= 10^{-7} k_B N \end{aligned} \tag{1}$$

If you consider now the definition of S: $\Delta S = ln(\hat{\Delta}\Omega)k_B$ you get with equation (1):

$$\hat{\Delta}\Omega = exp(\frac{\Delta S}{k_B}) = exp(10^{-7}N) = \frac{\Omega_2}{\Omega_1}$$

Finally we were asked to give the factor by which the number of available configurations increases in a system with given initial N, V, T_1 when an energy of 150 kJ is added to the system at constant Volume.
Analogous to the task before it can be derived that $\Delta S = c_v N ln\left(\frac{T_2}{T_1}\right)$. Using again that the factor by which the number of configurations is given by $\hat{\Delta}\Omega = exp(\frac{S}{k_B})$ it follows that:

$$\hat{\Delta}\Omega = exp\left(\frac{c_v N ln(\frac{T_2}{T_1})}{k_B}\right) = exp\left(\frac{c_v N}{k_B}\right)\frac{T_2}{T_1}$$

Using that $T_2 = \frac{\Delta Q}{c_v N} - T_1$ and plugging in the given values leads finally to:

$$\hat{\Delta}\Omega = exp\left(\frac{2c_v}{k_B}\right)\left(\frac{78}{300c_v} - 1\right) = \frac{\Omega_2}{\Omega_1}$$

1.2 Task 2 - Thermodynamic Variables in the Canonical Ensemble

Given the Helmholtz free energy F we were asked to derive expressions for U,p,S. Using the Maxwell relations it follows:

$$S = \frac{-\partial F}{\partial T} = ln(Q(N,V,T))k_B + \frac{k_B T}{Q(N,V,T)}\frac{\partial Q(N,V,T)}{\partial T}$$
$$p = \frac{-\partial F}{\partial V} = \frac{1}{\beta Q(N,V,T)}\frac{\partial Q(N,V,T)}{\partial V}$$

The derivation of the equation for U is as follows:
It is known from statistical mechanics that $Q(N,V,T) = \frac{1}{h^{3N}N!}\int d\Gamma exp(-\beta H)$. The thermodynamic properties of the system can be obtained by $Q(N,V,T) = exp(-\beta F(N,V,T))$. To justify this identification we show fist that F is extensive and then that $F = U - TS$, where $U = \langle H \rangle$.
That F is extensive can be derived directly from $Q(N,V,T) = \frac{1}{h^{3N}N!}\int d\Gamma exp(-\beta H)$. When the system is split up into two systems with a very weak corelation, then is Q a product of two factors. To show the second equivalence we rewrite $F = U - TS$ to $U = \langle H \rangle = A - T\left(\frac{\partial A}{\partial T}\right)$. To show this we divide the two expressions of Q that we stated before and therefore get:

$$\frac{1}{h^{3N}N!}\int d\Gamma exp(\beta(F - H)) = 1$$

Deriving both sides by β leads to:

$$\frac{1}{h^{3N}N!}\int d\Gamma exp(\beta(F - H))(F - H + \beta\left(\frac{\partial F}{\partial \beta}\right)) = 0$$

This is equivalent to $F - U - T\left(\frac{\partial F}{\partial T}\right)$, wherefore it is: $U = F + TS$.

1.3 Task 3 - Ideal Gas

Given the partition function $Q(N,V,T)$ we had to derive expressions for the free Helmholtz energy F(N,V,T) and the pressure p(N,V,T).
As we know from task 2 it is $F = \frac{-ln(Q(N,V,T))}{\beta}$. Plugging in the $Q(N,V,T)$ given on the sheet and using the hint leads to:

$$\begin{aligned}
F &= -\frac{ln(Q(N,V,T)}{\beta}\\
&= -k_B T ln\left(\frac{V^N}{\lambda^{3N}N!}\right)\\
&= -k_B T\left(Nln(V) - Nln(\lambda^3) - ln(N!)\right)\\
&= -k_B T N\left(ln(V) - ln(\lambda^3) - ln(N) + 1\right)\\
&= k_B T N\left(ln\left(\frac{N\lambda^3}{V}\right) - 1\right)
\end{aligned} \tag{2}$$

Applying the Maxwell relation $\frac{-\partial F}{\partial V} = p$ on equation (1) it follows:

$$\begin{aligned} p &= \frac{-\partial F}{\partial V} \\ &= \frac{-\partial k_B T N \left(ln \left(\frac{N\lambda^3}{V} \right) - 1 \right)}{\partial V} \\ &= \frac{k_B T N}{V} \end{aligned} \tag{3}$$

As it is obvious equation(3) leads to the ideal gas law : $pV = Nk_BT$.

2 Molecular Dynamics: Lennard-Jones Fluid

2.1 Implementation of the Lennard Jones Potential

The Lennard Jones Potential is $V_{LJ}(r) = 4\epsilon \left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^6 \right)$ and it's straight forward implementation works as follows:

Listing 1: Function compute_lj_potential

```
def compute_lj_potential(rij , eps = EPS, sig = SIG):
    q = sig/np.linalg.norm(rij)
    return 4*eps*(q**12-q**6)
```

Consequently the force being the negative gradient of the potential V_{LJ} is computed by the following function:

Listing 2: Function compute_lj_force

```
def compute_lj_force(rij , eps = EPS, sig = SIG):
    norm = np.linalg.norm(rij)
    q = sig/norm
    return 4*eps*(12*q**11-6*q**5)*q/norm**2*rij
```

To compute the plot of the potential and the first component of the force against d the following routine was implemented:

Listing 3: Computation of the potential and force

```
d = np.zeros((NParticles,3))
d[:,0] = np.linspace(0.85,2.5,NParticles)
potential = np.array(map(compute_lj_potential, d))
force = np.array(map(compute_lj_force, d))
```

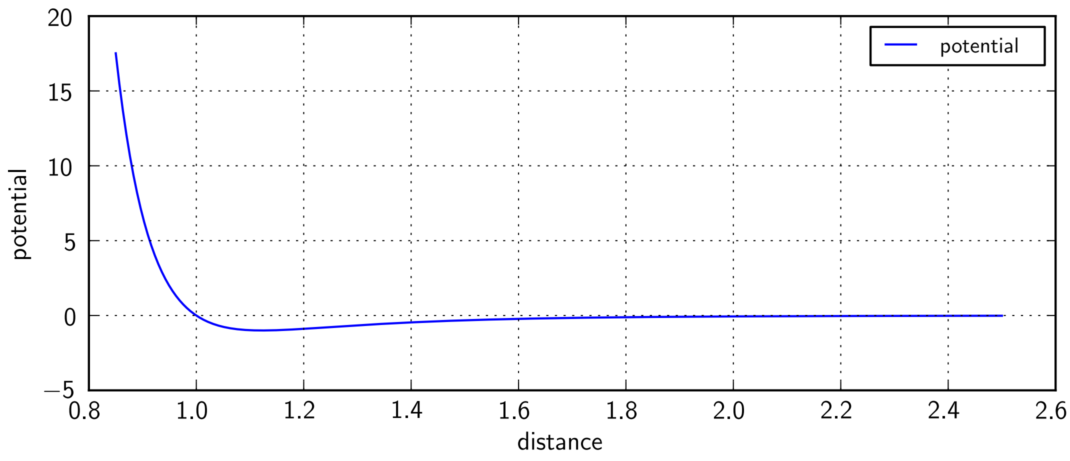


Figure 1: LJ Potential

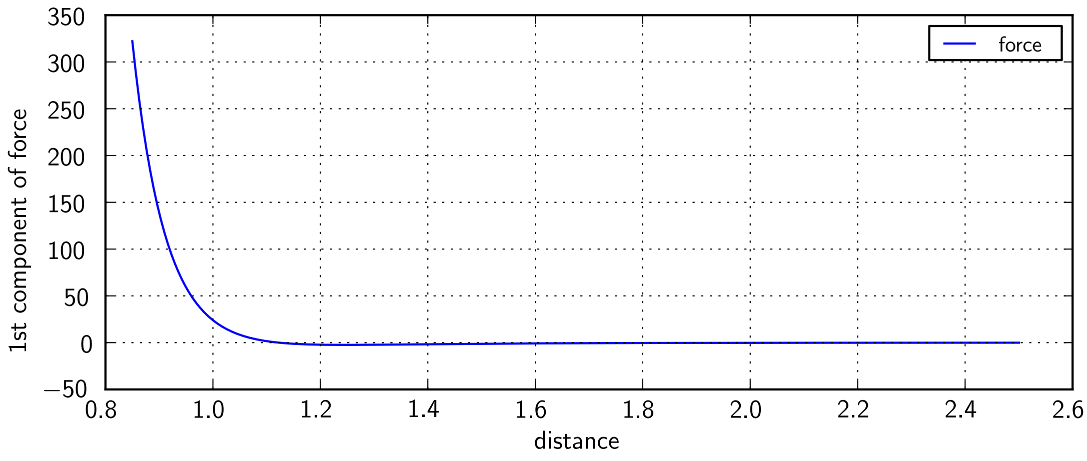
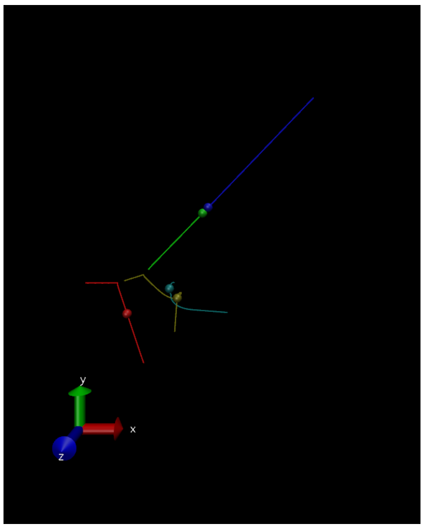
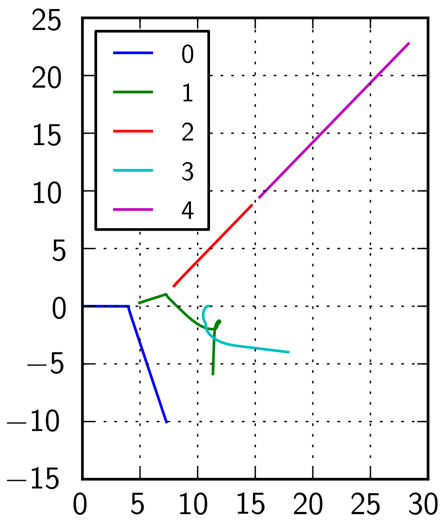


Figure 2: LJ Force

2.2 Lennard-Jones Billards

In this task we first had to insert the already programmed routines into the given templates. After that we had to run the program and to visualize the system. The visualization via VMD and the resulting trajectories in the xy-plane are:





As it can be seen in this picture ball 4 has already been moved so that it hits ball 2. The position needed, that this occurs is: $x[:,4] = [15.4324, 9.51146, 0.0]$

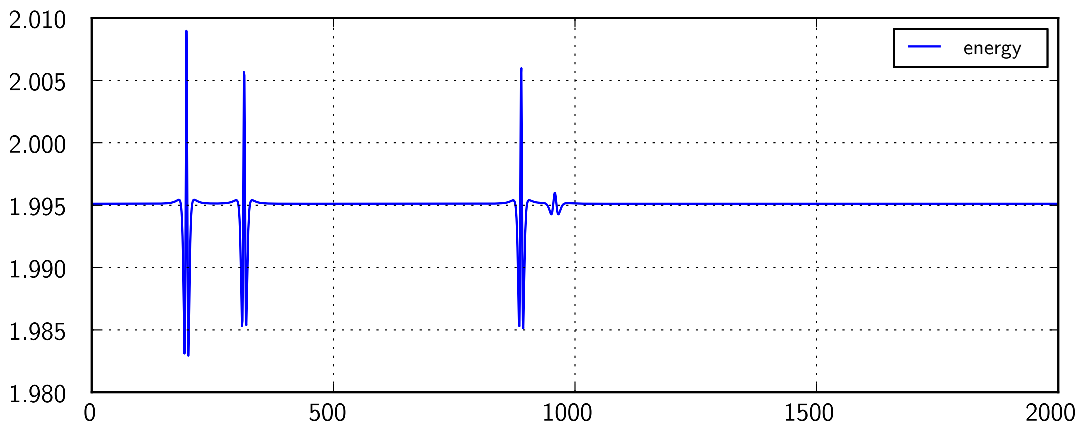


Figure 3: Energy

The energy is constant except from the times when the balls hit each other. In these cases occur relatively small variations in the energy. As it can be seen in Figure 3 the particles 0, 2 and 4 act just like normal billards balls. However the trajectories of the particles 1 and 3 show that the interaction is not based on a hard sphere potential, which would be exact for a billards ball, but on a potential with an attractive part for certain distances. This is obvious as the trajectories show that particle 1 influences particle 3 without hitting it directly which results in a curved trajectory of these two balls. After the distance between the balls becomes big enough the attraction is not acting anymore and the particles follow straight trajectories afterwards.

2.3 Minimum Image Convention and PBC

First we had to implement the potential and the force such that it implements the truncated LJ potential for $r_cutoff = 2.5$. The straight forward implementation is analogous to the function implemented before:

Listing 4: Function compute_lj_potential

```
def compute_lj_potential(rij, eps = EPS, sig = SIG, rcoff = RCOFF,\
                            vcoff = VCOFF):
    norm = np.linalg.norm(rij)
    if norm > rcoff: return 0
    q = sig/norm
    return 4*eps*(q**12-q**6) - vcoff
```

Listing 5: Function compute_lj_force

```
def compute_lj_force(rij, eps = EPS, sig = SIG, rcoff = RCOFF):
    norm = np.linalg.norm(rij)
    if norm > rcoff: return np.zeros(3)
    q = sig/norm
    return 4*eps*(12*q**11-6*q**5)*q/norm**2*rij
```

In order to implement PBC some of the routines have to be changed. First of all the function compute_forces had to be updated in order to fulfill the minimum image convention.

Listing 6: Function compute_forces

```
def compute_forces(x, L = L):
    """Compute and return the forces acting onto the particles,
    depending on the positions x."""
    global epsilon, sigma
    _, N = x.shape
    f = np.zeros_like(x)
    for i in range(1,N):
        for j in range(i):
            # distance vector
            rij = x[:,j] - x[:,i]
            rij -= np.rint(rij/L)*L #BECAUSE OF PBC
            fij = compute_lj_force(rij)
            f[:,i] -= fij
            f[:,j] += fij
    return f
```

In this case the function `rint()`takes an array and rounds the entries to the closest integer. The same modification had to be done in the function compute_energy, meaning that `rij`had again to be rounded with the help of `rint`. As no other things are changed we ask the reader to have a look in the source code in the appendix in order to see the whole function. Another change had to be done in the main loop. As not all particles considered in the box the positions x had to be folded back into the central box. This is done as follows:

Listing 7: Folding x back into the central box

```
    xfolded = x.copy()
    xfolded -= np.floor(x/L)*L  #BECAUSE OF PBC
```

Finally also the trajectories had to be folded back before being plotted. As it works just as the back folding of x we again ask the reader to have a closer look at the source code for further information. Now the implemented PBC were tested using a simulation of LJ billards balls. The main loop of the program is implemented as

Listing 8: Test of the PBC using ljbillards

```
# main loop
while t < tmax:
    xfolded = x.copy()
    xfolded -= np.floor(x/L)*L  #BECAUSE OF PBC

    x, v, f = step_vv(xfolded, v, f, dt)
    t += dt

    traj.append(x.copy())
    Es.append(compute_energy(xfolded, v))

    # write out that a new timestep starts
    vtffile.write('timestep\n')
    # write out the coordinates of the particles
    for i in range(N):
        vtffile.write("%s %s %s\n" % (x[0,i], x[1,i], x[2,i]))

vtffile.close()

traj = np.array(traj)
```

The resulting trajectories are:

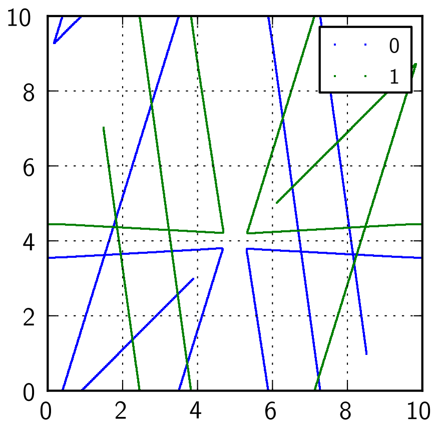


Figure 4: Trajectories of the two particles for 20 timesteps

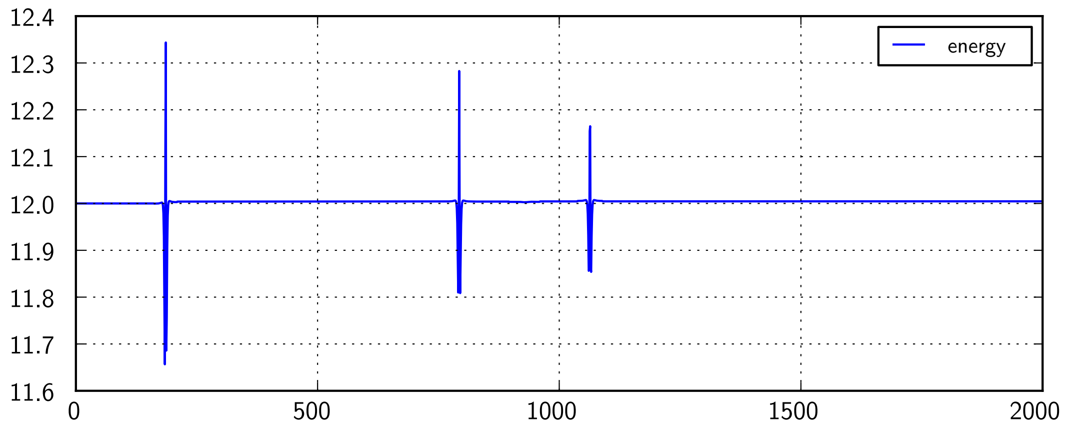


Figure 5: Energy - truncated potential

In the plot of the trajectories it becomes obvious that the PBC work, as when a particle leaves the box on one side the trajectory seems, after being folded back into the central box, to continue on the opposite side. Again the energy stays constant and shows that the particles hit each other thrice during the simulation time, which can be seen at the sharp energy peaks that have already been described before.

3 Lennard- Jones Fluid

3.1 Pure Python

First we had to extend the program `ljfluid.py`to set up the particles on a cubic lattice given the Number n of particles per side and their density ρ. The implementation of this is the following:

Listing 9: Placing Particles on a Cubic Lattice

```
# total number of particles
N = n*n*n

# particle positions on a cubic lattice
x = np.zeros((3,N))

# compute system size
L = n/density**(1/3)

# - set up n*n*n particles on a cubic lattice
positions = np.arange(0.5, n + 0.5)/density**(1/3)
for a in range(n):
    for b in range(n):
        for c in range(n):
            x[0,(a*n*n)+(b*n)+c] = positions[a]
            x[1,(a*n*n)+(b*n)+c] = positions[b]
            x[2,(a*n*n)+(b*n)+c] = positions[c]
```

3.1.1 Timings

The measured timings are given in the following table:

n	time [s]
3	2.628
4	13.081
5	44.11

It can be seen that the time needed to do the simulations is basically proportional to n^3 which would be expected as we have three nested for loops over n in our computation of the positions on the cubic lattice and no other function has a higher complexity.

3.2 Pure C/C++

First we had to implement the function `compute_energy()`in C: