Simulation Methods in Physics I

Worksheet 3: Molecular Dynamics 2 and Observables

Students:	Michael Marquardt	Cameron Stewart
matriculation numbers:	3122118	3216338

1 Command line parameters

For the simulation different command line options were defined. In the following code blocks you can see them. They will be explained during the report.

Code block 1: Commands for ljsim.py   script: src/ljsim.py

```
14 # command line arguments
15 parser = argparse.ArgumentParser()
16 parser.add_argument("--cont", type=double, help="continue
      calculation with for cont further time")
17 parser.add_argument("--time", type=double, help="How long
      do you want to run the simulation? | default time=10")
18 parser.add_argument("--tstat", type=double, help="Uses a
      thermostat with a given temperature")
19 parser.add_argument("--warm", type=double, help="Use the
      warming up | pass force")
20 parser.add_argument("--ctstat", type=double, help="
      continue for the simulation with tstat")
21 parser.add_argument("--cwarm", type=double, help="continue
       for the simulation with warm")
22
23 args = parser.parse_args()
```

Code block 2: Commands for ljanalyze.py   script: src/ljanalyze.py

```
 8 # command line arguments
 9 parser = argparse.ArgumentParser()
10 parser.add_argument("--M", type=int, default=10, help="
      window size | default: M=10")
11 parser.add_argument("--datafile", type=str, default='../
      dat/ljsim.dat', help="datafilename | default: '../dat/
      ljsim.dat'")
12 parser.add_argument("--teq", type=double, help="
      equilibration time | calculates averages after
      equilibration time")
13 parser.add_argument("--tlim", type=double, nargs='+', help
      ="type in the time limits you want to plot")
14
15 args = parser.parse_args()
```

2 Restart the program where left it

First we want to change the programm in a way, that it is able to restart the simulation where it ended last time. The first thing that we have to do is to store the necessary information about the end state of the system, the position x and velocity v of each particle, into the file ljsim.dat. This happens in code block 3. Furthermore there are the new variables Ts and Ps. The represent the temperature and pressure of the system over time and will be explained in the next chapter.

Code block 3: Storing data   script: src/ljsim.py

```python
221 # write out simulation data
222 print("Writing simulation data to {}.".format(datafilename
        ))
223 datafile = open(datafilename, 'w')
224 pickle.dump([ts, Es, Ts, Ps, x, v, hs], datafile)
225 datafile.close()
```

Now this data must be read by ljsim.py. Therefore we define a command line option −−cont which accepts as argument the time for which the simulation should be continued (code block 1). If the parameter is not used the simulation will start new. Furtehrmore there is another argument −−time which takes the simulation time for a new simulation. The code in code block 4 does exactly this.

Code block 4: Continue simulation   script: src/ljsim.py

```python
74 # Import previous data
75 if args.cont:
76         # open datafile
77         datafile = open(datafilename,'r')
78         ts, Es, Ts, Ps, x, v, hs = pickle.load(datafile)
79         datafile.close()
80
81         # length of run
82         t = ts[-1]
83         tmax = t+args.cont
84         step = 0
85
86 else:
87         # length of run
88         if args.time:
89                 tmax = args.time
90         else:
91                 tmax = 10.0
```

3 Simple Observables

3.1 Expanding Energy Calculation

First the energy calculation should be changed in order to also get the kinetic and potential energy. This is done by changing the compute_energy() function call and the parameter Es in ljsim.py (code block 5).

```
Code block 5: E_kin + E_pot                                  script: src/ljsim.py
189         if step % measurement_stride == 0:
190             E_pot, E_kin, E_tot = compute_energy(x, v)
191             T = 2*E_kin/(3*N)
192             P = compute_pressure(E_kin, x)
193             print("t={}:\n\tE_pot={}\n\tE_kin={}\n\tE_tot={}\n
                    \tT={}\n\tP={}".format(t, E_pot, E_kin, E_tot,
                    T, P))
194             compute_distances(x, r)
195             h = compute_histogram(r)
196
197             ts.append(t)
198             Es.append([E_pot,E_kin,E_tot])
199             Ts.append(T)
200             Ps.append(P)
201             hs.append(h)
```

3.2 Calculating the Temperature

The temperature of the system can be calculated from the kinetic Energy E_kin like in code block 5 shown. This is a single scalar calculation and therefore can be done in python.

3.3 Calcating Pressure

Derivation of the pressure

The virial of a system is defined as:

$$G = \sum_i^N \boldsymbol{p}_i \cdot \boldsymbol{r}_i \tag{1}$$

In the case

$$0 = \left\langle \frac{\mathrm{d}G}{\mathrm{d}t} \right\rangle = \left\langle \sum_i^N \frac{\boldsymbol{p}_i^2}{m_i} \right\rangle + \sum_i^N \left\langle \boldsymbol{F}_i \cdot \boldsymbol{r}_i \right\rangle \tag{2}$$

the following equation can be derived:

$$\left\langle \sum_i^N \frac{\boldsymbol{p}_i^2}{m_i} \right\rangle = -\sum_i^N \langle \boldsymbol{F}_i \cdot \boldsymbol{r}_i \rangle = 3Nk_BT \overset{id.gas}{=} 3PV \tag{3}$$

For an pair interaction (like in the simulation) let the forces $\boldsymbol{f}_{ij}$ and the distance vectors $\boldsymbol{r}_{ij}$ be defined as follows. Thereby $f^{lj}(|\boldsymbol{r}_{ij}|)$ is the scalar lennard jones force for the distance $|\boldsymbol{r}_{ij}|$, where a positive value means repulsion.

$$\boldsymbol{r}_{ij} = \boldsymbol{r}_j - \boldsymbol{r}_i \tag{4}$$

$$\boldsymbol{f}_{ij} = f^{lj}(|\boldsymbol{r}_{ij}|) \cdot \frac{\boldsymbol{r}_{ij}}{|\boldsymbol{r}_{ij}|} \tag{5}$$

$$\boldsymbol{f}_{ij} = -\boldsymbol{f}_{ji} \tag{6}$$

With this knowldege we can derive the pressure of the system by splitting up the force $\boldsymbol{F}$ in equation (3) into an ideal gas part $\boldsymbol{F}^{id.gas}$ and a lennard jones part $\boldsymbol{F}^{lj}$.

$$-\sum_i^N \langle \boldsymbol{F}_i \cdot \boldsymbol{r}_i \rangle = -\sum_i^N \left\langle \boldsymbol{F}_i^{id.gas} \cdot \boldsymbol{r}_i \right\rangle - \sum_i^N \left\langle \boldsymbol{F}_i^{lj} \cdot \boldsymbol{r}_i \right\rangle \tag{7}$$

$$\left\langle \sum_i^N \frac{\boldsymbol{p}_i^2}{m_i} \right\rangle = 3PV - \sum_i^N \left\langle \boldsymbol{F}_i^{lj} \cdot \boldsymbol{r}_i \right\rangle \tag{8}$$

$$P = \frac{Nk_BT}{V} + \frac{1}{3V} \sum_i^N \langle \boldsymbol{F}_{i,\text{interaction}} \cdot \boldsymbol{r}_i \rangle \tag{9}$$

$$= \frac{1}{3V} \left[\left\langle \sum_i^N \frac{\boldsymbol{p}_i^2}{m_i} \right\rangle + \left\langle \sum_{i,j\neq i}^N -\boldsymbol{f}_{ij} \cdot \boldsymbol{r}_i \right\rangle \right] \tag{10}$$

$$= \frac{1}{3V} \left[\left\langle \sum_i^N \frac{\boldsymbol{p}_i^2}{m_i} \right\rangle + \left\langle \sum_{i,j>i}^N \boldsymbol{f}_{ij} \cdot \boldsymbol{r}_j - \boldsymbol{f}_{ij} \cdot \boldsymbol{r}_i \right\rangle \right] \tag{11}$$

$$= \frac{1}{3V} \left[\left\langle \sum_i^N \frac{\boldsymbol{p}_i^2}{m_i} \right\rangle + \left\langle \sum_{i,j>i}^N \boldsymbol{f}_{ij} \cdot \boldsymbol{r}_{ij} \right\rangle \right] \tag{12}$$

$$= \frac{1}{3V} \left[2 \langle E_{\text{kin}} \rangle + \left\langle \sum_{i,j>i}^N \boldsymbol{f}_{ij} \cdot \boldsymbol{r}_{ij} \right\rangle \right] \tag{13}$$

Pressure in Cython

Becaus of the vectorial calculations for each particle pair the calculation has to be done in C if it should be fast. Therefor the function c_compute_pressure() is written in c_lj.cpp.

Code block 6: Pressure calculation in C	script: src/c_lj.cpp

```
253    double c_compute_pressure(double E_kin, double* x){
254      double rij[3];
255      double fij[3];
256      double interaction = 0.0;
257
258      // for (int i = 1; i < N; i++)
259      //   for (int j = 0; j < i; j++) {
260
261      // add up fij*rij
262      vector<int>::iterator it = verlet_list.begin();
263      vector<int>::iterator end = verlet_list.end();
264      while (it != end) {
265        int i = *it;
266        ++it;
267        int j = *it;
268        ++it;
269
270        if (i!=j){
271          minimum_image(x, i, j, rij);
272          compute_lj_force(rij,fij);
273
274          for (int k = 0; k < 3; k++)
275            interaction += fij[k]*rij[k];
276        }
277      }
278      return (2.*E_kin+interaction)/(3*L*L*L);
279    }
```

As you can see the first part is the same like in c_compute_forces(), but the last part differs. In order to save run time the function takes the kinetic energy, which is already calculated, as an argument.

4 Molecular Dynamics at a Desired Temperature

Programming

For the *velocity rescaling* thermostat we can derive the rescaling-factor f_{re} from equation (14).

$$\frac{3}{2}k_B T_0 = \frac{E_{\mathrm{kin},0}}{N} \tag{14}$$

$$= \frac{1}{N}\sum_{i=1}^{N}\frac{(f_{\mathrm{re}}\cdot \boldsymbol{v}^{(i)})^2}{2m} \tag{15}$$

$$= f_{\mathrm{re}}^2 \frac{E_{\mathrm{kin}}}{N} \tag{16}$$

$$= f_{\mathrm{re}}^2 \frac{3}{2}k_B T \tag{17}$$

$$f_{\mathrm{re}} = \sqrt{\frac{T_0}{T}} \tag{18}$$

This rescaling is implemented in C. In fact the implementation in C is not much faster than in python, but the thermostat in python did some crazy stuff. The function c_velocity_rescaling can be seen in code block 7.

Code block 7: Velocity rescaling	script: src/c_lj.cpp

```cpp
281    void c_velocity_rescaling(double T0, double T, double* v
          ){
282      double f = sqrt(T0/T);
283      for (int k = 0; k < 3*N; k++){
284          v[k] *= f;
285      }
286    }
```

This function is used in the main loop in ljsim.py every time after meausuring the observalbles.

To start the thermostat you can use the command line option −−tstat which accepts the desired temperature as argument (see code block 1). The option −−ctstat is there for continuing the simulation for the interrupted simulation with temperature T thermostat, but now with deactivated thermostat. This is necessary because the programm saves the results of the simulation with different names for different temperatures.

Measurement

Now we can test the simulation for desired temperatures of $T_0 \epsilon \{0.3, 1.0, 2.0\}$. The following figures show the results.

Obs.	$T_0 = 0.3$	$T_0 = 1.0$	$T_0 = 2.0$

T

E

U

P
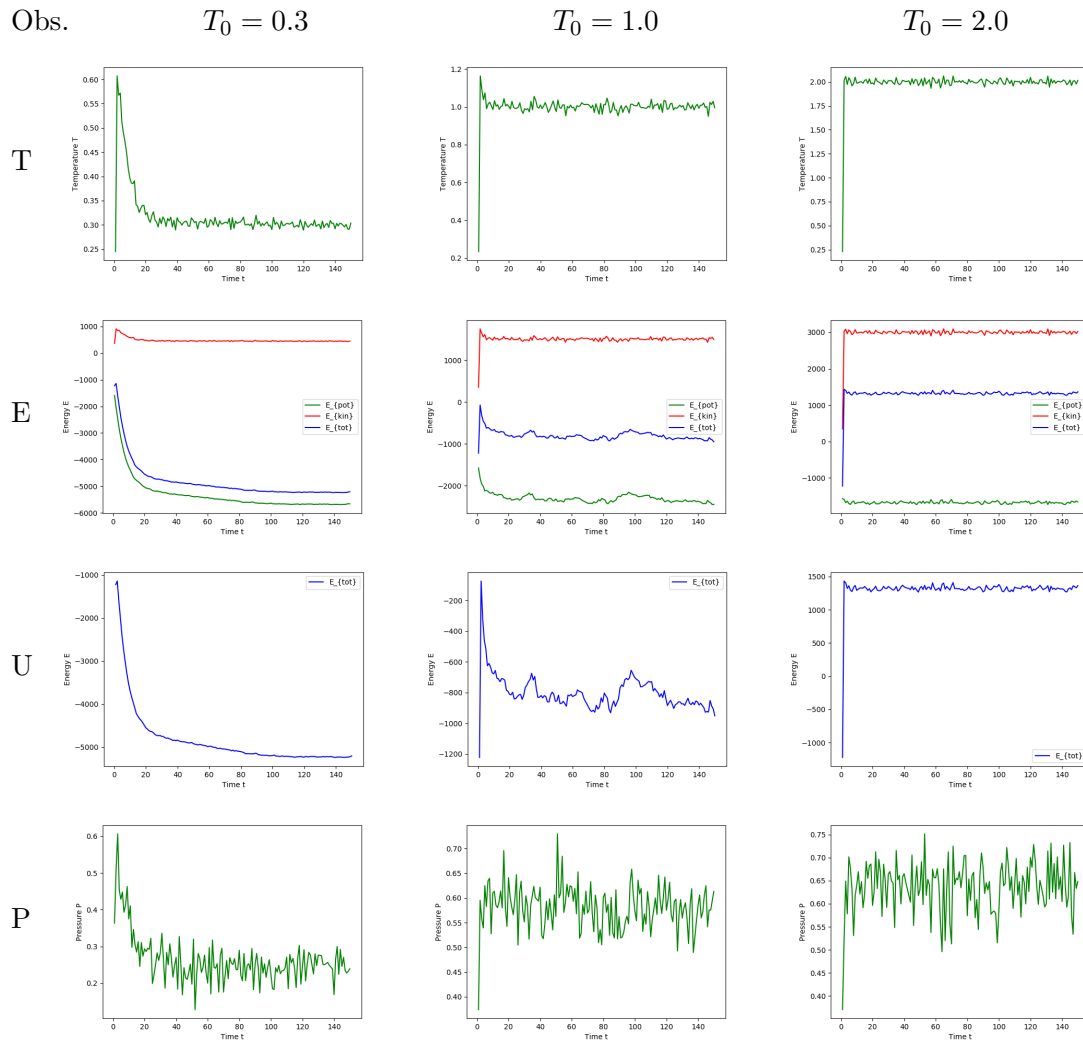
5 Warming up the System

When warming up the system you want to state a initial maximum force. Therefor the command line option $--$warm can be used (code block 1). The option $--$cwarm works analougus to $--$ctstat.

The first thing needed is the function c_force_capping() which is implemented in C. The reason for this is because you have to do many if requests and calculations for each particle.

```
Code block 8: Force capping                                    script: src/c_lj.cpp
288    double c_force_capping(double* f, double fcap){
289      double fcap2 = fcap*fcap;
290      double test_cap = true; // if true: no force is capped
```

```
                      any more
291       for (int k = 0; k < N; k++) {
292         double fabs2 = f[k]*f[k] + f[k+1]*f[k+1] + f[k+2]*f[
              k+2];
293         if (fabs2 > fcap2) {
294            test_cap = false;
295            double ff = sqrt(fcap2/fabs2);
296            f[k] *= ff;
297            f[k+N] *= ff;
298            f[k+2*N] *= ff;
299         }
300       }
301       if (test_cap){
302          fcap = 0;
303       }
304    return fcap;
305    }
```

The function in code block 8 limits the forces to a given maximum force fcap, but conserves the direction if the forces. Although this process is not physically correct. Furthermore the function tests if no force is capped any more. When this is the case it sets fcap to zero.

The function is called in the vv_step() function in ljsim.py.

Code block 9: Limiting forces   script: src/ljsim.py

```
164        # compute new forces
165        # we assume that m=1 for all particles
166        f = compute_forces(x)
167        if args.warm:
168            force_capping(f,args.warm)
```

As you can see, the function will not be activated any more if fcap = args.warm is set to zero.

At least the fcap should be increased by ten percent every time the measurements are done. This is done in code block 10.

Code block 10: Increase maximum force   script: src/ljsim.py

```
213            # rescale fcap
214            if args.warm:
215                args.warm *= 1.1
```

But all the capping is useless without random initial conditions. The velocity is already random, so only the initial position has to be changed. You can do this with a similar command like in 11.

Code block 11: Random initial position	script: src/ljsim.py

```
98          # Initialize particle position
99          if args.warm:
100                 # The warming up random positions
101                 x = L*random.random((3,N))
```

6 Radial Distribution Function

An important observable for our simulation is the radial distribution function. It is the probability to find a particle at a given radial distance away from another particle compared to an ideal gas. Mathematically it is written as

$$g(r) = \frac{1}{\rho 4\pi \mathbf{r}^2 dr} \sum_{i,j} \langle \delta(r - |\mathbf{r}_{ij}|) \rangle \tag{19}$$

For an ideal gas we expect $g(r) = 1$ if we normalize correctly.

In our simulation we implement the rdf using both C and python. First we calculate all the pair distances for our particles in C.

Code block 12: Radial Distances	script: src/c_lj.cpp

```
307     void c_compute_distances(double* x, double* r){
308        double rij[3];
309        int k = 0;
310        for ( int i = 0; i < N; i++){
311           for ( int j = i+1; j < N; j++){
312              minimum_image(x, i, j, rij);
313              r[k] = sqrt(rij[0]*rij[0]+rij[1]*rij[1]+rij
                     [2]*rij[2]);
314              k++;
315           }
316        }
317     }
```

We then send this to an array in python and compute a histogram using numpy and normalize it in the correct way

Code block 13: Numpy Histogram	script: src/ljsim.py

```
10 def compute_histogram(r):
```

```
11          h, bins = histogram(r, bins=100, range=(0.8, 5.),
               density=True)
12          return  h*vol/(4*pi*bins[:−1]*bins[:−1])
```

Finally we implement an average RDF after equilibrium in `ljanalyze.py` . We compute this using the following function

Code block 14: Compute Mean RDF   script: src/ljanalyze.py

```
51 def compute_mean_rdf(O,keq):
52     N = O.shape[0]
53     Om = zeros(O.shape[1])
54     for k in xrange(keq,N):
55         Om += O[k,:]
56     return Om/(N-keq)
```

and plot it.