

## 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

```
9 # command line arguments
10 parser = argparse.ArgumentParser()
11 parser.add_argument("--cont", type=double, help="continue
    calculation with for cont further time")
12 parser.add_argument("--time", type=double, help="How long
    do you want to run the simulation? | default time=10s")
13 parser.add_argument("--tstat", type=double, help="Uses a
    thermostat with a given temperature")
14 parser.add_argument("--warm", type=double, help="Use the
    warming up | pass force")
15 parser.add_argument("--ctstat", type=double, help="
    continue for the simulation with tstat")
16 parser.add_argument("--cwarm", type=double, help="continue
    for the simulation with warm")
17
18 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 program 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  $T_s$  and  $P_s$ . They 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`

```
208 # write out simulation data
209 print("Writing simulation data to {}".format(datafilename
      ))
210 datafile = open(datafilename, 'w')
211 pickle.dump([ts, Es, Ts, Ps, x, v], datafile)
212 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. Furthermore 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`

```
65 # Import previous data
66 if args.cont:
67     # open datafile
68     datafile = open(datafilename, 'r')
69     ts, Es, Ts, Ps, x, v = pickle.load(datafile)
70     datafile.close()
71
72     # length of run
73     t = ts[-1]
74     tmax = t+args.cont
75     step = 0
76
77 else:
78     # length of run
79     if args.time:
80         tmax = args.time
81     else:
82         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:  $E_{\text{kin}} + E_{\text{pot}}$

script: `src/ljsim.py`

```

179     if step % measurement_stride == 0:
180         E_pot, E_kin, E_tot = compute_energy(x, v)
181         T = 2*E_kin/(3*N)
182         P = compute_pressure(E_kin, x)
183         print("t={}: \n\tE_pot={}\n\tE_kin={}\n\tE_tot={}\n\tT={}\n\tP={}".format(t, E_pot, E_kin, E_tot,
184                                                     T, P))
185         ts.append(t)
186         Es.append([E_pot, E_kin, E_tot])
187         Ts.append(T)
188         Ps.append(P)

```

#### 3.2 Calculating the Temperature

The temperature of the system can be calculated from the kinetic Energy  $E_{\text{kin}}$  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:

$$G = \sum_i^N \mathbf{p}_i \cdot \mathbf{r}_i \quad (1)$$

In the case

$$0 = \left\langle \frac{dG}{dt} \right\rangle = \left\langle \sum_i^N \frac{\mathbf{p}_i^2}{m_i} \right\rangle + \sum_i^N \langle \mathbf{F}_i \cdot \mathbf{r}_i \rangle \quad (2)$$

the following equation can be derived:

$$-\sum_i^N \langle \mathbf{F}_i \cdot \mathbf{r}_i \rangle = 2 \left\langle \sum_i^N \frac{\mathbf{p}_i^2}{2m_i} \right\rangle = 2 \langle E_{\text{kin}} \rangle = 3Nk_B T \stackrel{\text{id. gas}}{=} 3PV \quad (3)$$

For an pair interaction (like in the simulation) let the forces  $\mathbf{f}_{ij}$  and the vectors  $\mathbf{r}_{ij}$  be defined as follows:

$$\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i \quad (4)$$

$$\mathbf{f}_{ij} = f_{ij}(\mathbf{r}_{ij}^2) \cdot \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|} \quad (5)$$

With this knowledge we can derive the pressure of the system:

$$P = P_{\text{id. gas}} + P_{\text{interaction}} \quad (6)$$

$$= \frac{Nk_B T}{V} + \frac{1}{3V} \sum_i^N \langle \mathbf{F}_{i,\text{interaction}} \cdot \mathbf{r}_i \rangle \quad (7)$$

$$= \frac{1}{3V} \left[ \left\langle \sum_i^N \frac{\mathbf{p}_i^2}{2m_i} \right\rangle + \left\langle \sum_{i,j \neq i}^N -\mathbf{f}_{ij} \cdot \mathbf{r}_i \right\rangle \right] \quad (8)$$

$$= \frac{1}{3V} \left[ \left\langle \sum_i^N \frac{\mathbf{p}_i^2}{2m_i} \right\rangle + \left\langle \sum_{i,j > i}^N \mathbf{f}_{ij} \cdot \mathbf{r}_j - \mathbf{f}_{ij} \cdot \mathbf{r}_i \right\rangle \right] \quad (9)$$

$$= \frac{1}{3V} \left[ \left\langle \sum_i^N \frac{\mathbf{p}_i^2}{2m_i} \right\rangle + \left\langle \sum_{i,j > i}^N \mathbf{f}_{ij} \cdot \mathbf{r}_{ij} \right\rangle \right] \quad (10)$$

$$= \frac{1}{3V} \left[ 2 \langle E_{\text{kin}} \rangle + \left\langle \sum_{i,j > i}^N \mathbf{f}_{ij} \cdot \mathbf{r}_{ij} \right\rangle \right] \quad (11)$$

## Pressure in Cython

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

Code block 6:  $E_{\text{kin}} + E_{\text{pot}}$

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 calculation time the function takes the kinetic energy, which is already calculated, as an argument.

## 4 Molecular Dynamics at a Desired Temperature

For the *velocity rescaling* thermostat we can derive the rescaling-factor  $f_{\text{re}}$  from equation (12).

$$\frac{3}{2}k_B T_0 = \frac{E_{\text{kin},0}}{N} \quad (12)$$

$$= \frac{1}{N} \sum_{i=1}^N \frac{(f_{\text{re}} \cdot \mathbf{v}^{(i)})^2}{2m} \quad (13)$$

$$= f_{\text{re}}^2 \frac{E_{\text{kin}}}{N} \quad (14)$$

$$= f_{\text{re}}^2 \frac{3}{2} k_B T \quad (15)$$

$$f_{\text{re}} = \sqrt{\frac{T_0}{T}} \quad (16)$$

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`

```

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 measuring the observables.

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 program saves the results of the simulation with different names for different temperatures.

## 4.1 Warming up the System

When warming up the system you want to state a initial maximum force. Therefore the command line option `--warm` can be used (code block 1). The option `--cwarm` works analogous 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  $f_{cap}$ , but conserves the direction of 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  $f_{cap}$  to zero.

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

Code block 9: Limiting forces

script: `src/ljsim.py`

```
154     # compute new forces
155     # we assume that m=1 for all particles
156     f = compute_forces(x)
157     if args.warm:
158         force_capping(f, args.warm)
```

---

As you can see, the function will not be activated any more if  $f_{cap} = \text{args.warm}$  is set to zero.

At least the  $f_{cap}$  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`

```
200     # rescale fcap
201     if args.warm:
202         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`

```
89     # Initialize particle position
90     if args.warm:
91         # The warming up random positions
92         x = L*random.random((3,N))
```

---