Institute for Microelectronics

Numerical Simulation and Scientific Computing II - Exercise 2

Member:
Mario Hiti, 01327428
Markus Hickel, 01026362
Markus Renoldner, 01621777

Submission: May. 06, 2020

Contents

1 Task 1 - Theory

(a) *Is the fourth-order Runge-Kutta method a good integrator in the context of molecular dynamics? Why? Give at least two reasons.*

- RK4 is **not** symplectic and **not** energy conserving. Both a are characteristics that we usually demand from a good solver.
- In order to solve a first order problem using RK4 for N particles in 3 dimensions we need to integrate $3 \cdot N$ equations. In order to solve second order problems, the equations have to be rewritten into two first-order ODEs thus doublinmg the amount of work. In molecular dynamics N is usually very high. Therefore RK4 can be very expensive.

(b) *Describe the advantages of automatic differentiation as a method to calculate forces.*

- AD allows to differentiate most Numpy as Scipy functions with just one line
- In contrary to finite differences the programmer does not need to think about step sizes
- Finite differences can only compute the derivative of a given scalar field. If a new scalar field is computed the whole operation has to be repeated. AD analyzes the transformations a functions performs on a given input. To calculate the derivative, a composition of the derivatives of these transformations is applied to the input
- jax is able to perform optimizations on the sequence of operations to achieve better results instead of naively calculating the finite difference on each point.
- in order to get derivatives of higher order finite differences need to repeat the complete calculation thus doubling the cost. AD should be able to calculate the second derivative as fast as the first derivative.
- The function to calculate forces does not change at runtime. Therefore the optimized AD fucntion can be reused every iteration to calculate the forces. Furthermore the precision is not limited by the step size.

(c) *When the potential energy of a system is described using the Lennard-Jones model, how does the number of calculations per time step scale with the number of particles? Why?*

The potential energy is being computed using:

$$E_{pot}(x_i) = \sum_i \sum_j V_{LJ}(|x_i - x_j|)$$

with i, j being particle indices and V_{LJ} being the Lennard-Jones Potential.
For n particles, we get a total number of iterations, that scales with

$$\mathcal{O}\Big(\frac{n^2}{2}\Big)$$

This comes from the fact that we have a product of two sums over n particles each, but only need to compute the potential from particle i to j and not the reverse [1].

(d) *Describe a strategy to make that number of calculations proportional to the number of particles.*

We could only compute the interaction with the closest particle
This reduces the number of neighbour particles, that one would need to take into account depending on the cutoff distance. Alternatively one could ignore particles beyond a certian distance, however this can still be expensive for high particle densities.

2 Task 2 - Defining the simulation Space

2.1 Project structure

The workspace has two important folders:

- */src* contains all code written for this assignment. Most of the code concerning the simulation is located in the module domain.py. Tasks 2-4 are implemented in separate files which make use us of the API provided by domain.py
- */out* contains all generated output

We also provide a Makefile to quickly test all tasks or install the virtual environment. The following section will outline the basic algorithms we used to implemetn our solution.

2.2 Initializing the Positions and Velocities

As it was proposed in the problem statement, the positions and velocities for a given particle are stored in one array. For n particles, the whole current state of the system can then be stored in n of these arrays. The trajectories of the particles then can be computed using the Lennard Jones potential as well as a symplectic integrator, that computes the next position and velocity for each particle after a given time step.
To initialize the starting positions of n particles, a cubic domain with the following number of particles m in one dimension of the domain is being used:

$$m = \text{roundup}(\sqrt[3]{n})$$

In the next step, we compute the spacing s between to particles on one axis with length l:

$$s = \frac{l}{m+1}$$

This ensures, that each particle is at least one spacing away of the next domain wall, which helped to prevent simulation errors at the start of the simulation.
Next, all n particles are being assigned a coordinate in the domain, which starts at $(0, 0, 0)$ and goes up to (m, m, m) in the case of a fully filled domain. If $m < \sqrt[3]{n}$, a part of the cubic domain will be empty at the initialization.
After that, a small random distance in each direction will be added to each coordinate of each particle.

The velocities are being computed using a random number generator that produces a normal distributed set of values. After that, the mean velocity in each coordinate direction is being subtracted from each particles velocity value, so that the velocity of the center of mass of all particles is zero, as all particles have the same mass.

$$v_{i,\text{new}} = v - \frac{1}{n}\sum_{j=1}^{n} v_{i,j}$$

where i is the coordinate x, y, z.

2.3 Calculation the potential energy

To calculate the potential Energy we are using the Lennard-Jones model with natural dimensions

$$E_{\text{pot}}\left(\boldsymbol{x}_1 \dots \boldsymbol{x}_M\right) = \sum_{i=1}^{M-1} \sum_{j=i+1}^{M} V_{\text{LJ}}\left(|\boldsymbol{x}_i - \boldsymbol{x}_j|\right), \text{ where}$$

$$V_{\text{LJ}}(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

Translating these equations into an efficient algorithm is the most crucial part in order to achieve good performance. In fact using regular python loops results in unacceptable long runtimes for even small domains. In order to achieve good performance we have used the algorithm from the lecture slides and adapted it for our use:

Algorithm 1: Algorithm for calculating the potential Energy

```
1     def Epot_source(self, positions):
2         """Lennard-Jones potential in reduced units.
3         In this system of units, epsilon=1 and sigma=2**(-1. / 6.).
4         """
5         positions = to3D(positions)
6         if positions.ndim != 2 or positions.shape[1] != 3:
7             raise ValueError(positions must be an Mx3 array)
8
9         delta = positions[:, np.newaxis, :] - positions
10
11        # Minimum image transformation
12        delta = delta - self.length * np.round(delta/self.length)
13        r2 = (delta * delta).sum(axis=2)
14        # Take only the upper triangle (combinations of two atoms).
15        indices = np.triu_indices(r2.shape[0], k=1)
16        rm2 = 1. / r2[indices]
17        # Compute the potental energy recycling as many calculations as possible.
18        rm6 = rm2 * rm2 * rm2
19        rm12 = rm6 * rm6
20        return (rm12 - 2. * rm6).sum()
```

Some functions such as `scipy.minimize()` can only provide data in 1D structures. Therefore we have added `to3D()` to transform the data back to a Mx3 matrix. This is achieved by changing the shape-parameter of the numpy-array. Since no data has has to be moced this process has O(1) complexity and thus no noteworthy impact on performance. Next we are using the minimum image convention to map coordinates back into the box (Line 12).

To further increase performance we are using the just-in-time compiler from jax. Combining these techniques results in a reduction of runtime from several seconds down to 10^{-4} to 10^{-5} seconds.

3 Task 3 - Velocity Verlet integration

The verlet algorithm is provided as follow:

$$\boldsymbol{x}_i(t+\Delta t) = \boldsymbol{x}_i(t) + \boldsymbol{v}_i(t)\Delta t + \frac{1}{2}\boldsymbol{f}_i(t)(\Delta t)^2$$
$$\boldsymbol{v}_i(t+\Delta t) = \boldsymbol{v}_i(t) + \frac{\boldsymbol{f}_i(t)+\boldsymbol{f}_i(t+\Delta t)}{2}\Delta t$$

Which has been implemented as shown in Alg. 3. Note the minimum image conversion is performed within the Epo-fucntion.

Algorithm 2: velocity verlet

```
def verlet_advance(self, dt):
    if(dt == 0): return

    f = -self.grad_Epot(self.pos)
    new_pos = self.pos + (self.vel + 0.5 * f * dt) * dt

    new_f = -self.grad_Epot(new_pos)
    new_vel = self.vel + 0.5 * (f + new_f ) * dt

    self.pos = new_pos
    self.vel = new_vel
```

The output can be ghenerated by using task2.py.

4 Task 4 - Analysis

4.1 Conservation of energy

For a 100 timesteps, we simulated a system of 200 particles with a timestep size of 0.01 for different values of σ. σ, the standard deviation of the initial velocity, is also a metric for the kinetic energy. This is because the mean velocity is set to zero and the deviation from zero corresponds with a higher total velocity and henceforth higher kinetic energy.
For very small values of $\sigma = 0.5$ or even $\sigma = 0.2$, we get $E_{pot} \to 0$, and therefore of course $E_{tot} \to E_{pot}$.

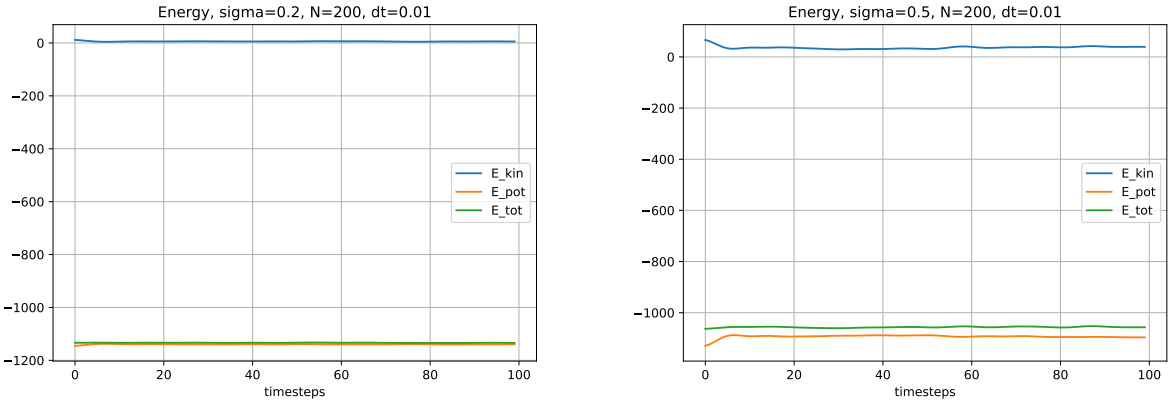

Fig. 1: Energy plot using a sigma=0.2 and sigma=0.5

For higher σ, we can observe how the two parts of the energy take a longer time until they start to converge to a certain optimum state.

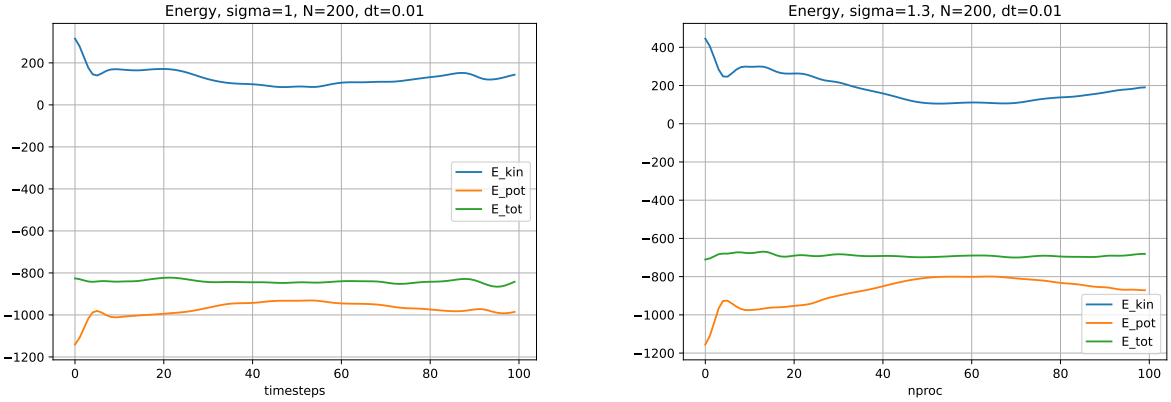

Fig. 2: Energy plot using a sigma=1 and sigma=1.3

Here in this last plot we tried to observe the conservation of energy on a larger time scale, with a slightly larger time step size.
The change in energy seems more erratic or "twitchy" but this is due to a longer time span being compressed within the graph. One would expect the total energy to be perfectly smooth but just like any other simulation we have to deal with numerical errors. Furthermore 200 particles interacting with each other results in a highly chaotic system which results in very random fluctuations. This can very clearly be seen in the animation we provided.

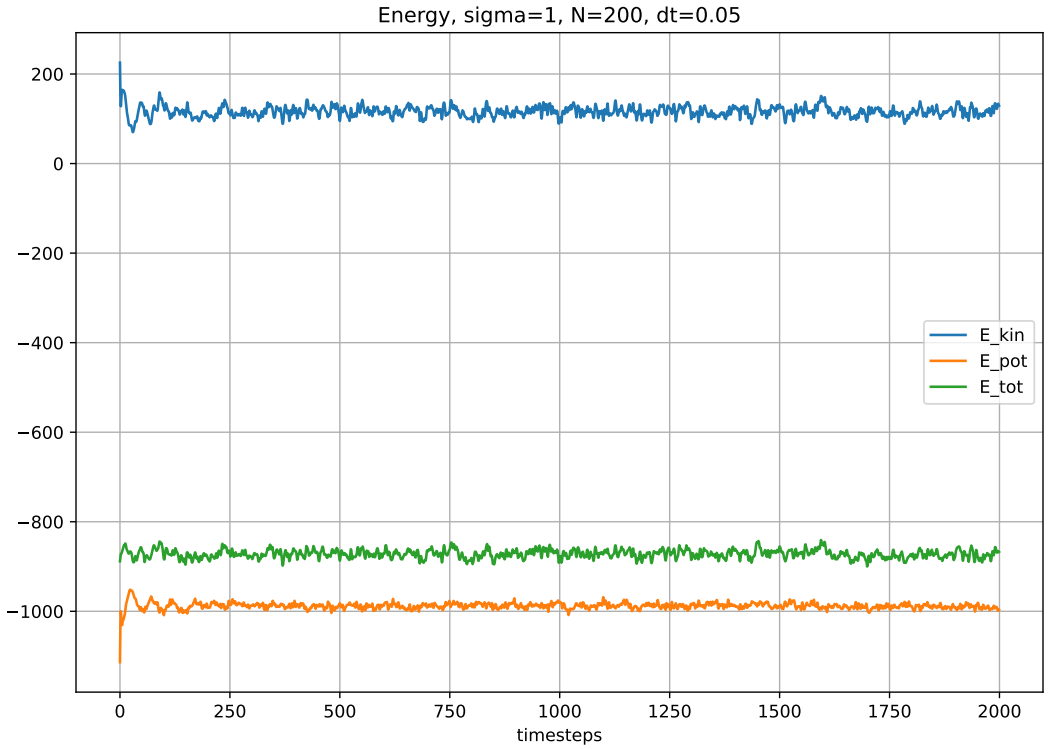

Fig. 3: Energy plot over 2000 timesteps

If we assume no interaction between the particles which is just $f_i = 0$, then the velocity will stay constant over time. Thus, the kinetic energy will also be constant, while the potential

$$V_{\mathrm{LJ}}(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

will be zero per definition. This also results in a potential energy of zero.

4.2 Volumetric density

To estimate the volumetric density of particles we use 3 Alg. 3.

Algorithm 3: code density

```
def volumetric_density(self, sample_cnt):

    r = np.linspace(0, self.length/2., sample_cnt)

    // set 0th particle as the origin
    origin = self.pos[0]

    // calculate eudlicdian distances to that particle
    distances = np.array(np.linalg.norm(self.pos[1:] - origin, axis=1))

    // perform minimum image transformation
    distances = abs(MI_transform(distances, self.length))
    assert(len(distances) == self.particle_count-1)

    // use histogram() to count particles in each shell
    hist, edges = np.histogram(distances, bins=r)

    densities = []
    for i in range(1,len(hist)):
        // calculate volume of the shell
        V = 4/3 * math.pi * (r[i]**3 - r[i-1]**3)

        // divide number of particles in that
        densities.append(hist[i]  / V)

    return densities
```

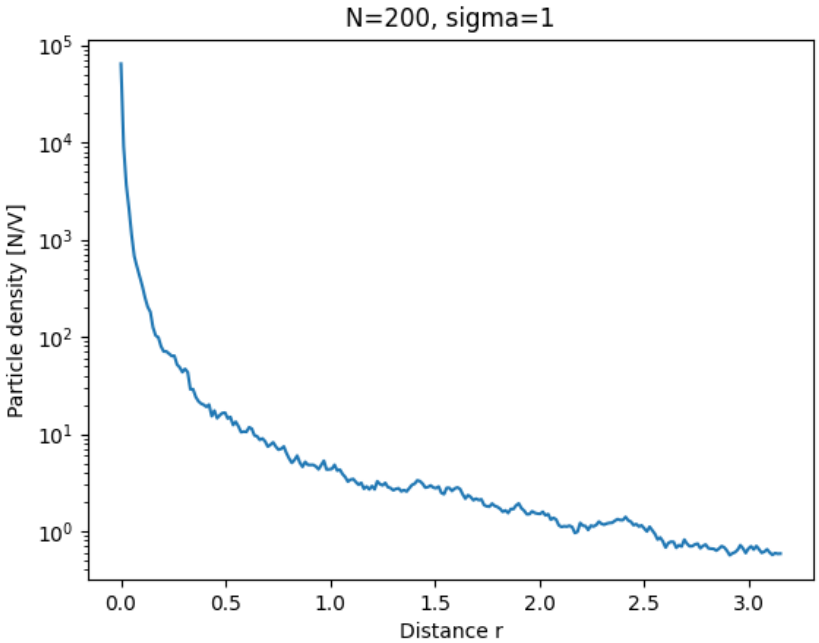

Fig. 4: Average Volumetric density over 2000 iterations. The first 25% are discarded. The sharp rise in density can be explained by $\rho = \frac{N}{V} \implies \lim_{r \to 0^+} = +\infty$

5 Conclusion

We have implemented a simulation of N particles in a cubic box using a Lennard-Jones Potential in a natural unit system. Furthermore a velocity verlet integrator is used to advance the system in time.

The results match our expectations:

- The system reaches an energy equilibrium after a short amount of time
- After convergence Ekin, Epot and Etot are constant within an expected numerical error
- In addition to colorful graphs we also provide a pretty animation

References

[1] Jesús Carrete Montana PhD. *Introduction to Molecular Dynamics - Numerical Simulation and Scientific Computing II*. 2021.