Exercise 4: Implementing energies

Potential energy for central-force potentials

Image that all N atoms would be interconnected by springs. Then it is clear that such a stretched "spring network" contains a certain amount of energy. Each spring contributes to the resulting potential energy.

A naive programming approach might be to simply go through the list of all atoms and compute the distance $\|\vec{r}_{ij}\|$ to all other atoms. For each of these distances sum up the resulting potential values ϕ:

$$??? \qquad E_{\text{pot}} = \sum_{i=1}^{N} \sum_{j=1, i \neq j}^{N} \phi(\|\vec{r}_{ij}\|) \qquad ???$$

The problem with this formulation is that each interaction (or "spring") is contributing twice: we have, e.g., $\phi(\|\vec{r}_{23}\|)$ as well as $\phi(\|\vec{r}_{32}\|)$, which obviously have the same value. With regards to the image of the spring: there is only one spring between each pair of atoms! Hence, we need a factor of one half:

$$E_{\text{pot}} = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1, i \neq j}^{N} \phi(\|\vec{r}_{ij}\|)$$

The two summations result in altogether $N(N-1)$ summands ("-1" because we skip the "i==j" element. But we can do better than that when we only consider the following sum (note that in the inner sum the lower limit is now i):

$$E_{\text{pot}} = \sum_{i=1}^{N-1} \sum_{j=i}^{N} \phi(\|\vec{r}_{ij}\|).$$

For $i = 1$ there are $N - 1$ elements, for $i = 2$ there are $N - 2$ elements left, ..., until for $i = N - 1$ there is only one element left. Thus, we only need to sum up $N(N-1)/2$ elements.

Last but not least, for implementing the summations in Python one should keep in mind that all indices start at 0!

Kinetic energy

The kinetic energy E_{kin} of a single atom/particle is given by

$$E_{\text{kin}} = \frac{1}{2} m \langle v^2 \rangle,$$

where $\langle v^2 \rangle$ is the average (squared) atom velocity. In the following, you do not have to differentiate between an average and an actual velocity -- just *the* velocity will do. The total kinetic energy of a system of N atoms is then just the sum over all individual contributions:

$$E_{\text{kin}} = \frac{1}{2} \sum_{i=1}^{N} m_i v_i^2.$$

Task 1: Write a function that computes the potential energy for an arbitrary number of atoms.

Assume that the number of atoms is `n_atoms` and the dimension of a simulations is given by `n_dim`. Then, a numpy array containing all positions could look like this:

```
import numpy as np

positions = np.array([
    [0., 0., 0.],
    [1., 1., 0.],
    [2., 0., 1.],
    [2., 2., 2.]
])

n_atoms, n_dim = positions.shape  # here: n_atoms = 4,  n_dim = 3
```

Use the following function template for computing the energy. The print statement can be used as a test for your code:

```
from potential import LJ_potential, LJ_force

def total_potential_energy(positions, sigma, epsilon):
    # `positions` is a ndarray of shape (n_atoms, n_dim) where n_dim = 2 for our example
    # sigma, epsilon are the Lennard-Jones parameter
    # ...
    # ...
    # ...
    return energy

# Use this to test your code:
sigma = 0.7
epsilon = 12.3
print("E_pot = {:1.4f}".format(total_potential_energy(positions, sigma, epsilon)))
>>> E_pot = 1443.0206
```

Task 2: Write a function that computes the kinetic energy for an arbitrary number of atoms.

```
def total_kinetic_energy(velocities, mass):
    # `velocities` is a ndarray of shape (n_atoms, n_dim) where n_dim = 2 for our example
    # `mass` is a scalar variable
    # ...
    # ...
    # ...
    return energy

# Use this to test your code:
mass = 120.1  # all atoms have the same mass

print("E_kin = {:1.4f}".format(total_kinetic_energy(velocities, mass)))
>>> E_kin = 10.3586
```

Task 3: Turn your code into a 2D code where each atom's position consists of a x and a y coordinate.

This is a preparation so that we then can use the two new energy functions. It is also a step towards a generic MD simulation code.

Hint 1: you have to adjust the arrays position and velocitiy in the program from the previous exercise -- we need to have a 2D array where the number of rows are the number of atoms and the **two columns** is the dimensionality. Set the y-coordinate and the y-velocity component simply to zero for both atoms.

Hint 2: You might also have to adjust the function `get_acceleration(...)` such that it contains the distance between the two atoms, e.g., `LJ_force(distance, sigma, epsilon)`.

Hint 3: In the plot part of your code you have to handle the positions as a `list` (time steps) of `numpy arrays` (n_atoms x n_dim). `positions[-1][1, 0]` then denotes the last step of the list, atom 1, x-component (this is the "0"). While it is very easy just to append to a list for each time step, this is a bit unconvenient for plotting. The easiest way to deal with this is to convert this into a 3d numpy array in the plotting part (`pos = np.array(positions)`). Then you can use the usual numpy indexing: `pos[n_step, n_atom, n_coordinate]` to access the data.

Task 4: Add a 3rd subplot to your "1D LJ-oscillator" and plot the kinetic, potential and total energy vs. time