Simulating a droplet of liquid Argon

Midterm Project 2 *Comp. Methods Physics* ASU PHY494 (2019)*

March 12, 2019 – March 31, 2019

Abstract Your project is to write, in teams of three students, a Python molecular dynamics (MD) code to simulate a droplet of liquid Argon. We want to study how varying some of the simulation parameters affects the accuracy of the simulations and extract thermodynamic properties from the simulations. You will write a short "letter"-style paper to communicate, discuss and summarize your reasoning and your results.

Due Sunday, March 31, 2019, 11:59pm.

- Students work in teams of two or three students.
- **Admissible Collaboration:** Students are allowed to talk to other students in the class about the project and exchange ideas and tips. However, sharing/copying reports or full code solutions is not allowed. **Help from other students must be acknowledged in an Acknowledgments section**. Direct help from outside the class is not allowed (except instructor/TA), e.g., you cannot ask for solutions (online or in person) but you can use

*Current version of this document: March 12, 2019. See Appendix E for a list of changes since v1 from March 11, 2019.

books and resources on the internet to solve problems. **Cite all sources**. Code from the class can be used without explicit citation or acknowledgement.

- Each team should commit their report (see Section 4) in **PDF** format to the team's **GitHub repository**; alternatively, combining report and code in a Jupyter notebook is also possible as long as the notebook can be read like a report (i.e., not just bullet points or short comments). If possible, also generate a PDF from your notebook and commit it together with everything else.
- The report *must* contain a section **Contributions** at the end where the contributions of all team members are summarized.
- Each team should commit and push **all code** (see Section 2.2) that is required to reproduce the results in the report to their **GitHub repository**. Include a text file **`README.txt`** that describes the commands to run calculations. In particular, it must show how to run and analyze *Simulation 1* in Table 2. The code must run in the standard anaconda-based environment used for the class. If it is a Jupyter notebook then it should be possible to *Kernel* → *Restart & Run All* and to produce all the required figures and output.

Grading will take the following into consideration:

- The code runs and produces correct output.
- The report clearly and succinctly describes the question, approach, and results and contains sufficient evidence that the requirements (see below) have been met.
- All team members contributed to the work: assessed by (1) Contributions section in the report, (2) commit history of the repository and comments in code, (3) short oral examination of team members (at instructor's discretion if deemed necessary).

- Code from outside sources (see Admissible Collaboration) and help is thoroughly attributed (Acknowledgements and References).
- BONUS: Additional work that you want to include in an appendix to the report or additional simulations for the main report will be treated as bonus material and can be awarded bonus points.
- BONUS: Elegant and fast code can be awarded bonus points.

Contents

1. Submission instructions

Submission is to your private **team GitHub repository**. Follow the link provided to you by the instructor in order for the repository to be set up: It will have the name *ASU-CompMethodsPhysics-PHY494/project-2-2019-*YourTeamName and will only be visible your team and the instructor/TA. Follow the instructions below to submit this project.

Read the following instructions carefully. Ask if anything is unclear.

1. `git clone` your project repository (change *YourTeamName* to your team's name)

   ```
   repo="project-2-2019-YourTeamName.git"
   git clone https://github.com/ASU-CompMethodsPhysics-PHY494/${repo}
   ```

   or, if you already have done so, `git pull` from within your assignments directory.

2. Create three sub-directories `Submission`, `Grade`, and `Work`.

3. You can try out code in the `Work` directory but you don't have to use it if you don't want to. Your grade with comments will appear in `Grade`.

4. Create your solution in `Submission`. Use Git to `git add` files and `git commit` changes.

   You can create a PDF file or Jupyter notebook inside the `Submission` directory as well as Python code (if required). **Name your files `project2.pdf` or or `project2.ipynb`**, depending on how you format your work. Files with code (if requested) should be named exactly as required in the assignment.

5. When you are ready to submit your solution, do a final `git status` to check that you haven't forgotten anything, commit any uncommitted changes, and `git push` to your GitHub repository. Check

on *your* GitHub repository web page[1] that your files were properly submitted.

You can push more updates up until the deadline. Changes after the deadline will not be taken into account for grading.

Work must be legible and intelligible or may otherwise be returned ungraded with 0 points.

2. Background and materials

2.1. MD simulations

For background reading the class notes, cited articles, and the book by Frenkel and Smit (2002) (and also Allen and Tildesley (1987) and Leach (1996)) are recommended). A good general article on molecular dynamics simulations is Mura and McAnany (2014). The original simulations of liquid argon were performed by Rahman (1964).

2.2. Code

Any code that has been developed or provided throughout the class can be used, either in full or in parts, as long as proper attribution is given (see below). In particular, code is provided to

- generate the initial positions of the argon atoms (Appendix A)
- generate the initial velocity distribution that correspons to a temperature T (see details in Appendix B)
- read and write XYZ files (Appendix C)

[1] `https://github.com/ASU-CompMethodsPhysics-PHY494/project-2-2019-`*`YourTeamName`*

3. Requirements

The following are requirements of the project and need to be demonstrated in the final report (e.g. by including appropriate graphs or mentioning in the Methods section).

3.1. MD program

Write a program to simulate the Lennard-Jones fluid (which we use as a model for liquid Argon) in Python.

The simulations are to be performed for constant particle number N and total energy E, for a given number density n and initial temperature T_0 (see Section 3.2 for values).

Simulate a **droplet of fluid argon in vacuum**. This means that you need to calculate the interactions of all atoms with each other, namely the force on atom i is the sum of all forces on this atom,

$$\mathbf{F}_i = \sum_{j \neq i} \mathbf{F}_{ij}. \tag{1}$$

There are no external forces acting on the fluid. This implies that in addition to the energy, total linear momentum is also conserved.[2] The forces are to be derived from a potential energy function in the usual way as

$$\mathbf{F}_i = -\frac{\partial}{\partial \mathbf{x}_i} U(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_n). \tag{2}$$

[2] Furthermore, total angular momentum also has to be conserved as there are no external torques acting on the fluid. In a more abstract sense, Noether's theorem states that due to time invariance of the equations of motions, energy is conserved. Translational invariance leads to momentum conservation, and rotational invariance to angular momentum conservation.

3.1.1. Potential energy function

Atoms are treated as point masses with positions $\mathbf{x}_i$ (in 3D space, i.e. $\mathbf{x}_i = (x_i, y_i, z_i)$) that interact through the pair-wise Lennard-Jones potential

$$v_{LJ}(r_{ij}) = 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right], \quad r_{ij} := \sqrt{(\mathbf{x}_j - \mathbf{x}_i)^2}. \tag{3}$$

The potential sets the energy scale through ϵ and the length scale through σ. The mass is set by the mass of an argon atom m. Using Lennard-Jones units (see Section 3.1.2 below), the potential energy function simplifies to

$$v^*_{LJ}(r^*_{ij}) = 4\left[(r^*_{ij})^{-12} - (r^*_{ij})^{-6} \right], \quad r^*_{ij} := \sqrt{(\mathbf{x}^*_j - \mathbf{x}^*_i)^2}, \tag{4}$$

which is the form to be used in the code.

The potential energy function for N interacting argon atoms is therefore

$$U(\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_N) = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} v_{LJ}(r_{ij}). \tag{5}$$

3.1.2. Units

As **units in the code** use **Lennard-Jones units** (denoted by an asterisk): We measure energy in units of ϵ,

$$E^* = \frac{E}{\epsilon}, \tag{6}$$

length in units of σ,

$$r^* = \frac{r}{\sigma}, \tag{7}$$

and mass in the mass m of an argon atom,

$$m^* = \frac{m}{m} = 1. \tag{8}$$

The **unit of time** is fixed with the units of energy, mass, and length, e.g., from the kinetic energy $T_{\text{kin}} \propto mv^2 = m(dr/dt)^2$ so the time in LJ units must be

$$t^* = \frac{t}{\tau}, \quad \tau := \sqrt{\frac{m\sigma^2}{\epsilon}} \tag{9}$$

where τ corresponds to $t^* = 1$. In SI units for the argon system (see Section 3.2 for the parameters in SI units), $\tau = 2.151388 \times 10^{-12}\,\text{s}$.

Velocity v and linear momentum $p = mv$ are the same in LJ units because the only mass in the system is $m^* = 1$,

$$v^* = \frac{v}{\sigma\tau^{-1}} \tag{10}$$

$$p^* = \frac{p}{m\sigma\tau^{-1}} = \frac{v}{\sigma\tau^{-1}} = v^*. \tag{11}$$

The particle density $n = N/V$ (with the particle number N and the volume V) is

$$n^* = \frac{n}{\sigma^{-3}}. \tag{12}$$

Lennard-Jones temperature is measured on the energy scale via Boltzmann's constant $k_B = 1.3806488 \times 10^{-23}\text{J} \cdot \text{K}^{-1}$,

$$T^* = \frac{k_B T}{\epsilon}. \tag{13}$$

The MD code should use LJ units throughout, i.e., all lengths are considered to be multiples of σ. Effectively, $\sigma = \epsilon = m = \tau = 1$.

3.1.3. Required program capabilities

Your code needs to be able to do at least the following in order to obtain full marks:

System setup and initialization This stage only needs to be completed once before running the actual simulation.

1. Initialize the simulation to a system of the prescribed R and n^*, using the provided function `system.generate_droplet()`. Have the code print the number of atoms in the simulation. Appendix A contains more information.
2. Assign initial velocities, corresponding to the prescribed temperature T_0^*, in such a way that the total linear momentum is 0. Have the code print T_0^* and initial temperature $T^*(t=0)$, calculated from the initial velocities. Appendix B contains further information.

Molecular dynamics simulation This is the "core" of the code, which performs the force evaluation and integration of motion in time steps of size Δt^*.

1. Simulate a fluid droplet in vacuum.
2. Calculate forces between all atoms.
3. Integrate the equations of motion using the *velocity Verlet* integrator.
4. Calculate and store for each time step
   - the potential energy $U^*(t)$ of the system
   - the kinetic energy $T^*_{\mathrm{kin}}(t)$ and system temperature $\mathcal{T}^*(t)$ (see Eq. 18)
   - the total energy $\mathcal{H}^*(t) = T^*_{\mathrm{kin}} + U^*$.
   - BONUS: the magnitude of the total linear momentum $|\mathbf{P}^*(t)| = \left|\sum_{i=1}^{N} \mathbf{v}_i^*(t)\right|$
5. BONUS: Write your energies to a file; e.g., if all your energy data are in a big numpy array, use `numpy.savetxt()`. You can then load the data with `numpy.loadtxt()` and analyze them separately.
6. Write out the coordinates for every frame to a trajectory in XYZ format (see Appendix C).

7. At the end of the run, your code should report timing/performance statistics: total run time ("wall time"), wall time per time step, performance in steps per h and simulated LJ time per h.[3]

Your code should be able to run in Python 3.4–3.7 and should make use of NumPy. You can use any other standard Python modules if you want to [4]. Your code must run and produce output when run on a machine with the standard installation.[5] You will need to *analyze* the output from your code. See Section 3.3 for what analysis is required for your report (Section 4).

Include all the code that is needed to generate the results shown in your report. This can consist of Python programs, modules, a Jupyter notebook, or a mixture thereof. Include a separate file `README.txt` that explains how to run your code in order to generate the results in your report, in particular how to run and analyze *Simulation 1* in Table 2. **Your code must run without errors in order for you to be awarded full marks.**

3.1.4. Skeleton code

Skeleton code is provided for the MD program itself (`mdlj.py`), system building (`system.py`, see Appendix A), for initialization (`mdInit.py`, see also Appendix B), and for reading and writing coordinates (`mdIO.py`, see Appendix C). You may use as much or as little of this code as you like but you need to include any code that is required to run your program(s).

unit	SI	Lennard-Jones
mass m	39.948 u	1
mass density ρ	$1.374\ \mathrm{g \cdot cm^{-3}}$	0.8141
number density n	$2.07 \times 10^{28}\ \mathrm{m^{-3}}$	0.8141
temperature T_0	94.4 K	0.787
LJ ϵ	$120\ \mathrm{K} \cdot k_B$	1
LJ σ	0.34 nm	1
LJ τ	2.1514 ps	1

Table 1. Simulation parameters for liquid argon in SI (or SI-related) units and Lennard-Jones units. Your code should use Lennard-Jones units. Note that τ is a derived quantity (Eq. 9) and listed here for convenience.

3.2. Input parameters

In order to simulate liquid argon we are using the Lennard-Jones parameters introduced by Rahman (1964). Simulate liquid Ar ($m = 39.948$ u) at a density $\rho = 1.374\ \mathrm{g \cdot cm^{-3}}$ and an initial temperature of $T_0 = 94.4$ K. For the Lennard-Jones potential Eq. 3 choose[6] $\epsilon = 120\ \mathrm{K} \cdot k_B = 0.99774\ \mathrm{kJ \cdot mol^{-1}}$ and $\sigma = 0.34$ nm. However, because using LJ units (see Section 3.1.2) makes our code much simpler, we convert the above parameters in "real" units to LJ units as in Table 1.

For this project you should run a number of different simulations that differ by run parameters as explained in Section 3.4 (see also Table 2).

[3] The Python `time.time()` function from the **time** module can be used to get the time *before* the start of the main loop and *after* the main loop.

[4] "standard Python modules" refers to the packages installed by *anaconda* and Python Standard Library 3.4–3.7.

[5] Generally, *no C or FORTRAN compiler is available* so your will have to work with Python/NumPy alone. However, you may also submit an *additional* accelerated code and use it for production, as long as you demonstrate that both codes deliver the same results.

[6] If we use an energy unit of kJ/mol then $k_B = 1.3806488 \times 10^{-23}\ \mathrm{J \cdot K^{-1}} = 8.3144621\ \mathrm{J \cdot mol^{-1} \cdot K^{-1}}$

3.3. Required Analysis

After the simulation has finished you should

1. Plot the time series $\mathcal{H}^*(t)$ (Hamiltonian, i.e., the total energy, $\mathcal{H}^* = T^*_{\text{kin}} + U^*$), the average energies per particle $\mathcal{H}^*(t)/N$, $U^*(t)/N$, $T^*_{\text{kin}}(t)/N$ and $\mathcal{T}^*(t)$ (instantaneous temperature, defined in Eq. 17).

2. Report the time-averaged values $\langle \mathcal{T}^* \rangle$, $\langle U^*/N \rangle$, $\langle E^*/N \rangle$ and standard deviations over a trajectory.[7] *Report these quantities in a table*, with one line for each simulation. Include the simulation number (from Table 2) and the number of atoms in the table as well.

3. Calculate the *energy drift* (Martyna et al., 1996)

$$\Delta E^* = \frac{1}{N_{\text{steps}}} \sum_{i=1}^{N_{\text{steps}}} \left| \frac{E^*(0) - E^*(i\Delta t^*)}{E(0)} \right| = \left\langle \left| 1 - \frac{E^*(t)}{E^*(0)} \right| \right\rangle \tag{15}$$

   ($E^*(i\Delta t^*)$ is simply $E^*(t)$). Report ΔE^* in the same table with the time averages.

   Based on your data, discuss the choices of simulation parameters that lead to good or bad energy conservation.

4. Summarize the performance of your code. How does performance change with N and Δt^*? Can you suggest parameters that are a good trade-off between simulation accuracy and performance?

[7] The time-average of a quantity A is defined as

$$\langle A \rangle = \frac{1}{t_{\text{tot}}} \int_0^{t_{\text{tot}}} A(t)\, dt = \frac{1}{N_{\text{steps}}} \sum_{i=0}^{N_{\text{steps}}} A_i \tag{14}$$

where the second equality assumes that A is computed for each time step i over a trajectory of total length $t_{\text{tot}} = N_{\text{steps}}\Delta t$. The standard deviation is as usual $\sigma_A = \sqrt{\langle (A - \langle A \rangle)^2 \rangle}$.

5. Visualize the trajectory with matplotlib 3D (or another tool of your choice) and prepare an image of the *first* and *last* frame of the trajectory for some of your simulations.[8] Comment on the behavior of your system, e.g., does the droplet retain its general shape. Explain and discuss what you observe.

6. BONUS: Plot the average radial density $n^*(r)$ of droplets. See Appendix D for further details.

3.4. Simulations

Perform the simulations with the parameters listed in Table 2. Simulations 6* and 7* are optional and you can do them to explore how system size affects the energies. You can of course do further/longer simulations if you think they are necessary or if you want to add more data points to a graph.

simulation	R	t_{tot}	Δt
1	3	10	0.01
2	3	100	0.01
3	3	100	0.02
4	3	100	0.04
5	3	10	0.005
6*	4	10	?
7*	5	10	?

Table 2. Simulations to be performed for the project. t_{tot} is the total simulation time, Δt is the time step, R the initial radius of the spherical droplet. All quantities are in Lennard-Jones units. *Simulations marked with an asterisk are optional and you need to choose a suitable time step.

As explained in Appendix A, the number of atoms in each simulation

[8] You can also try using VMD (Humphrey et al., 1996), which can be freely downloaded from http://www.ks.uiuc.edu/Research/vmd/. VMD can read xyz trajectories.

depends on the initial radius of the droplet R and the density n^*. Report the number of atoms N for each simulation in your results section.

4. Report

Prepare a "letter-style" paper in which you report what you accomplished. The report should contain a brief introduction, including a description of the problem, and overview over the methods used and implemented, the results obtained, and what the results mean (e.g. comment on the question of some of the simulation parameters affect the accuracy of the simulations). The report must be written in full sentences and read as a coherent piece of work.

4.1. Report structure and formatting

The report be structured like a scientific article. Follow the following instructions:

- *length*: approximately 5 pages, *including figures and tables* and excluding references, acknowledgements or appendices; use 11 pt font size (captions and tables can have smaller fonts), single spaced, minimum 1 inch margins. If the report is produced from a jupyter notebook (which is less dense than a formatted paper) the length can be somewhat longer but the content should be equivalent to a typed report of about 5 pages.
- Include a *title*.
- *author list*: list all authors on the team and "star" ("*") each person who wrote parts of the report; also list the writing contribution in Acknowledgments (see below)
- *abstract*: summarize what you did and what you found in about 200 words or less

- **include the following sections**: Introduction, Methods, Results and Discussion, Conclusions, Acknowledgements, References (see this handout for how to format references)
- appendix for any bonus work (but your report must be readable without it)
- All figures must be properly labeled (axes, units, individual lines distinguishable). Figures should have captions.
- Include tables; make sure it is clear what is shown in tables (e.g. state units).
- Include equations; make sure you explained your symbols.
- In the *Acknowledgements* section mention any help you got from outside your team. Also mention briefly who in the team contributed to which part of the project. For instance,

  > "All authors designed the project together. M.N. wrote code (mdInit.py) to set up the system and contributed the force calculation and performed simulations. X.Y. wrote the MD code (mdLJ.py) and performed simulations. Q.Z. wrote the analysis code (LJanalysis.py), analyzed data together with M.N. and X.Y., and contributed the overlap detection routine in mdInit.py. All authors discussed the results."

  (Many journals require attributions of this kind. You don't have to follow the example exactly but you need to spell out everyone's major contributions to the success of the project.)

If you find it difficult to keep within the page limit then try to be concise, combine multiple graphs into one, e.g. all per-particle energies (but make sure that each line is properly labeled). Graphs can be small but must still

be readable.[9] You don't have to include all graphs for all simulations in the paper but there must be sufficient data shown to support your conclusions. For instance, you could show the energy time series for a typical and a extreme case and summarize the results by plotting the averages, standard deviations, and drifts from all simulations.

4.2. Notes on scientific writing

You should *describe* data that you show and *explain* what the data *mean*. *Discuss* your results compared to what you expect to see based on your understanding of the physics of the problem.

Show any *equations* that you derive or use. For instance, in the *Methods* section show the explicit functional form of the force on atom i (derived from Eqs. 2–5).

The *Introduction* briefly describes the problem (including references to previous work, which you cite), states the question to be answered, and summarizes the approach. This section answers the question "Why did we do this work?".

The *Methods* section describes how you solved the problem. You typically cite other work for details. You answer the question "How did we approach the problem?"

Results and Discussion contains your results (figures and tables) together with your description and interpretation of your findings. If you compare to other work, you cite these papers. The *Conclusions* summarizes your work. These two sections answer the question "What did you do?".

[9] Hint: Generate graphs in matplotlib at final size by using `plt.figure(figsize=(5, 5))`; font sizes can be changed with `import matplotlib; matplotlib.rc('font', size=8)`; consider plotting graphs with `linewidth=3` to make them better visible.

5. Code re-use and collaboration

You will carry out the project in teams. In the authors list, add a star "*" to each person who wrote parts of the report.

- You can use any code that was developed or provided during class or as part of the project.
- You are allowed to discuss the problem with other teams, and you are allowed to share individual pieces of code, *provided that each piece of code is attributed to the original author (use full names)*. However, if more than 50% of code appear to be from other sources than the team, marks will be deducted. (Code from class is exempt from the 50% rule.)
- Copying text (report and code) verbatim from other sources without attribution constitutes plagiarism. Plagiarism is a very serious offence and will carry penalties ranging from 0 points to referral to the College for an XE in the permanent record (see Syllabus).

  The report should be in your own words but it is perfectly acceptable to cite other works instead of explaining in detail how, for instance, the integrator works.
- You can use the Acknowledgements section to highlight major external contributions (in addition to comments in the code).

A. Initial coordinate generation

In order to start a simulation we need to place N argon atoms at defined positions in space $\mathbf{x}_i$. The atoms should have a density n^* (see Table 1). Because we want to simulate a droplet in vacuum, we start out with a sphere of radius R in which we pack argon atoms on a regular grid. You are provided with code in `system.py`, namely the `system.generate_droplet(n, R)` function, that solves this problem. As input it requires the density n^* and the radius R:

```
import system
atoms, coordinates = system.generate_droplet(n, R)
```

It returns a list of N atom names (e.g., `['Ar', 'Ar', 'Ar', ...]`) and $N \times 3$ numpy array with the coordinates of atoms in a rough spherical shape with maximum distance R from the center and an approximate density n^*. The number of atoms N is determined by the geometry and the density. You should report the number of atoms in each simulation.

The following briefly outlines how this works. The basic approach is to first generate a regular lattice of the correct density and the carve out a sphere of atoms from the lattice. For simplicity we use a simple cubic lattice where each argon atom occupies the center of a cube (unit cell) of length a. We need to determine a so that on average our lattice of atoms has the density n^*. On the primitive cubic lattice, each unitcell of volume $V = a^3$ contains a single atom so the density of the lattice is

$$n^*_{\text{cubic}}(a) = \frac{1}{a^3}$$

In order to match the fluid density

$$n^* = n^*_{\text{cubic}}(a) = a^{-3}$$
$$a = (n^*)^{-\frac{1}{3}}$$

We then generate lattice sites for a cubic supercell of at least $L = 1.2R$ (i.e., we need L/a primitive unitcells in each direction). Give then supercell, we calculate the center of mass, $\mathbf{r}_0 = N^{-1} \sum_{i=1}^{N} \mathbf{x}_i$, calculate the radial distance of each atom from the center, $r_i = \sqrt{(\mathbf{x}_i - \mathbf{r}_0)^2}$, and select only those atoms for which $r_i < R$.

B. Initial velocity distribution

In order to start a simulation we need to assign an initial set of velocities $\mathbf{v}_i$. The velocities should correspond to a given system temperature T. Here we are taking a very simple approach and use a *uniform* distribution

of velocities[10] that we then scale to obtain the average kinetic energy corresponding to our target temperature.

The material in this appendix should be considered background reading for the interested student who wants to better understand what the functions do that were provided in `mdInit.py`.

B.1. Random velocities

We start out by assigning random velocities in an arbitrary range. The NumPy function `numpy.random.rand` produces random numbers ξ uniformly distributed in the interval $0 \leqslant \xi < 1$, and it can produce arrays of arbitrary shape.

To create an array `velocities` with shape $(N, 3)$ where each entry $-0.5 \leqslant v_{ij} < 0.5$ use

```
velocities = np.random.rand(N, 3) - 0.5
```

B.2. Temperature and kinetic energy

The temperature is related to the kinetic energy of one degree of freedom α via the equipartition theorem:

$$\left\langle \frac{1}{2} m v_\alpha^2 \right\rangle = \frac{1}{2} k_B T \tag{16}$$

We operationally *define* the *instantaneous temperature* at a time step t

$$\mathcal{T}(t) := \sum_{i=1}^{N} \frac{m_i \mathbf{v}_i^2}{k_B N_f}, \quad N_f = 3N - 6. \tag{17}$$

[10] A uniform distribution does not conform to any known thermodynamic ensemble but this is not a problem because the MD simulation will quickly redistribute energy in such a way that the microcanonical ensemble is obtained. One could assign an initial Maxwell-Boltzmann distribution of velocities (which would be more realistic) but that is too much effort.

A gas or fluid droplet consisting of N point-like particles has $N_f = 3N - 6$ degrees of freedom (only translation in x, y, and z and no rotations; minus 2×3 degrees of freedom to account for the overall system translation and rotation). In LJ units this equation simplifies to

$$\mathcal{T}^*(t) := \sum_{i=1}^{N} \frac{\mathbf{v}_i^{*2}}{N_f}. \tag{18}$$

Given the velocities $\mathbf{v}_i^*$ in an array `velocities`, we define a function `mdInit.kinetic_temperature(velocities)` that calculates the temperature according to Eq. 18:

```
def kinetic_temperature(velocities):
    N = len(velocities)
    Nf = 3*N - 6
    return np.sum(velocities**2)/Nf
```

B.3. Removing linear momentum

Newton's equations of motion in vacuum or under periodic boundaries conserve the linear momentum $\mathbf{P} = \sum_i m_i \mathbf{v}_i$ of the system. It is convenient to have $\mathbf{P} = 0$ throughout the simulation, therefore we want to start out with a system that already has $\mathbf{P} = 0$. This can be achieved by subtracting the mean velocity $\langle \mathbf{v}_i \rangle := \langle \mathbf{P} \rangle / m_i$ from each initial velocity $\mathbf{v}_i$: The total momentum vanishes, $\mathbf{P}' = \sum_i m_i \mathbf{v}_i' = 0$, when

$$\mathbf{v}_i' = \mathbf{v}_i - \langle \mathbf{v}_i \rangle \tag{19}$$

The function `new_velocities = mdInit.remove_linear_momentum(velocities)` resets total linear momentum to 0 by performing the transformation in Eq. 19 (and using LJ units)

```
def remove_linear_momentum(velocities):
    Pavg = np.mean(velocities, axis=0)
    vavg = Pavg  # same in LJ units
    return velocities - vavg
```

B.4. Setting velocities for a given temperature by rescaling

Following Eq. 17, we can generate velocities for any given system temperature T' by simply scaling all velocities by a constant factor:

$$\mathbf{v}' = \left(\frac{T'}{\mathcal{T}(t)}\right)^{\frac{1}{2}} \mathbf{v}. \tag{20}$$

or in LJ units

$$\mathbf{v}^{*\prime} = \left(\frac{T^{*\prime}}{\mathcal{T}^*(t)}\right)^{\frac{1}{2}} \mathbf{v}^*. \tag{21}$$

The function `velocities = mdInit.rescale(velocities, temperature)` performs the velocity rescaling:

```
def rescale(velocities, temperature):
    """Rescale velocities so that they correspond to temperature T*"""
    current_temperature = kinetic_temperature(velocities)
    return np.sqrt(temperature/current_temperature) * velocities
```

C. I/O with XYZ files

Trajectories, i.e., time series of positions should be stored on disk for future analysis because typically it is much more costly to generate those data than to analyze them. A very simple trajectory file format is the *XYZ* format. It is read by many widely used analysis and visualization programs [11]. A `.xyz` file is simple text file. Each frame (time step) of a trajectory is stored with a two-line header and a list of atom names and coordinates such as

[11] For example, VMD (Humphrey et al., 1996) is an excellent tool, which can be freely downloaded from http://www.ks.uiuc.edu/Research/vmd/. Two short tutorials on VMD (from the PHY542 class) are Protein visualization and analysis with VMD and Visualizing and analysing Molecular Dynamics trajectories with VMD.

```
80
frame 0  simulation
      Ar    -2.12506   -0.42501   -1.27503
      Ar    -2.12506   -1.27503   -0.42501
      ...
80
frame 1  simulation
      Ar    -2.11053   -0.42770   -1.29023
      Ar    -2.12239   -1.28893   -0.42623
      ...
...
...
80
frame 999  simulation
      Ar    -2.85456    1.10555    0.61041
      Ar    -2.01884   -0.85398    0.73630
      ...
      Ar     1.63796    0.04548   -0.04969
```

The first line in each frame contains the number of atoms. The second line is a comment and may contain the frame number as shown plus additional text. After the two header lines there is one line for each atom in the simulation. Entries are white-space separated. The first entry is the *atom name*, the following three numbers are the *coordinates* (x, y, z).

The file `mdIO.py` contains code for *writing* and *reading* XYZ files. For example, to **write trajectory data** in array `x`, have the atom names in a list `atoms` (see Appendix A), open the file, and write every step with the `mdIO.write_xyz_frame()` function:

```
import mdIO

with open('trajectory.xyz', 'w') as xyzfile:
    ...
    # integration loop
    for istep in range(1, nsteps):
```

```
    # produce new positions x as a Nx3 array
    mdIO.write_xyz_frame(xyzfile, atoms, x, istep)
```

You can **read** the first frame of a trajectory with the `mdIO.read_xyz_single()` function (e.g., to get the list of atoms and initial coordinates) or use `mdIO.read_xyz()` to read a whole trajectory into an array:

```
import mdIO

atoms, x0 = mdIO.read_xyz_single('trajectory.xyz')
traj = mdIO.read_xyz('trajectory.xyz')
```

D. Radial density: algorithm

The radial density can be defined as

$$n^*(r) = \left\langle \sum_{i=1}^{N} \delta(r - \sqrt{\mathbf{x}_i(t)^2}) \right\rangle,$$

where the delta-functions count how often a particle is seen at a distance r from the origin so that

$$\int_0^{2\pi} d\phi \int_0^{\pi} d\theta \int_0^{\infty} dr r^2 n^*(r) = N.$$

The typical algorithm *histograms* the radial distances of all atoms. It contains the following steps:

1. Decide on the number of bins and the bin size of the histograms. A good choice might be to divide the droplet of initial radial size R into 50 bins, but you should experiment with settings to get plots that seem meaningful to you.

2. Initialize an empty array s that you will use to accumulate histograms. You can do this by calling the `numpy.histogram()` function with your bins from 1 on a single fake data point

```
s, edges = np.histogram([0], bins=50, range=(0, R))
s *= 0
```

and then erasing the histogram by multiplying all entries with zero.

3. For each time frame t in the trajectory:
   a) For each atom i in the simulation:
      i. calculate the distance from the origin $r_i = \sqrt{\mathbf{x}_i(t)^2}$
   b) histogram all r_i (e.g., use the `numpy.histogram()` function with your fixed bins from 1) in histogram h_t
   c) add the histogram h_t to the previous histograms, $s_t = \sum_{t=1}^{t_\mathrm{total}} h_t$
4. Average the counts in the histogram by dividing by the number of frames t_tot.
5. Normalize the counts by dividing the count in each bin with lower edge r_L and upper edge r_U by the volume of the radial shell volume and $\mathcal{N}$ is a normalization[12] $\Delta V(r_L) = \frac{4\pi}{3}(r_U^3 - r_L^3)$.

Plot the resulting array over the middles of the bins $((r_L + r_U)/2)$.

E. History

Changes to this document are listed here.

2019-03-11 Initial version.

References

Frenkel D and Smit B, 2002 *Understanding Molecular Simulations*, 2nd edn. (Academic Press, San Diego).

[12] The normalization by the volume of each spherical shell is often written as "$dV(r) = 4\pi r^2 dr$" for the continuous limit.

Allen M P and Tildesley D J, 1987 *Computer Simulations of Liquids* (Oxford University Press, Oxford).

Leach A R, 1996 *Molecular Modelling. Principles and Applications* (Longman).

Mura C and McAnany C E, 2014 An introduction to biomolecular simulations and docking. *Molecular Simulation* **40** 732–764, http://dx.doi.org/10.1080/08927022.2014.935372.

Rahman A, 1964 Correlations in the motion of atoms in liquid argon. *Phys. Rev.* **136** 405–411.

Martyna G J, Tuckerman M E, Tobias D J and Klein M L, 1996 Explicit reversible integrators for extended systems dynamics. *Molecular Physics* **87** 1117–1157, http://www.tandfonline.com/doi/abs/10.1080/00268979600100761.

Humphrey W, Dalke A and Schulten K, 1996 VMD – Visual Molecular Dynamics. *J. Mol. Graph.* **14** 33–38, http://www.ks.uiuc.edu/Research/vmd/.