simple_run

June 6, 2022

1 Single run Lennard-Jones simulation

Here we demo how to run a single simulation, and the sort of information you get out of it. Larger simulations will be built out of many runs over a range of density and temperature parameters.

1.1 Running the simulation

For running the actual simulation, the only dependency is the `lennardjones` package. One could also use `numpy` as a source of random seeds, but `lennardjones` already exposes the C++ `std::random_device` seed generator, which is good enough for physics simulations.

[ ]:
```
import lennardjonesium as lj
```

1.1.1 Setup

First, we will create an `lj.Configuration` object. This is a `dataclass` which simply contains the various parameters of the simulation. We will write it to a file so that the information is easy to recall, in case we want to repeat the experiment.

[ ]:
```
c = lj.Configuration()

c.system.temperature = 0.5
c.system.density = 0.8
c.system.particle_count = 500
c.system.cutoff_distance = 2.5
c.system.time_delta = 0.005
c.system.random_seed = 42

c.equilibration.tolerance = 0.10
c.equilibration.sample_size = 50
c.equilibration.adjustment_interval = 50
c.equilibration.steady_state_time = 500
c.equilibration.timeout = 2000

c.observation.tolerance = 0.20
c.observation.sample_size = 50
c.observation.observation_interval = 50
c.observation.observation_count = 20
```

```
c.filepaths.event_log = 'events.log'
c.filepaths.thermodynamic_log = 'thermodynamics.csv'
c.filepaths.observation_log = 'observations.csv'
c.filepaths.snapshot_log = 'snapshots.csv'

c.write('data/run.ini')
```

1.1.2 Using the `run()` function

Next, load the `.ini` file and run the simulation. I've chosen a specific random seed because it produces an interesting result. To use a newly-generated random seed, you can write

`random_seed=lj.SeedGenerator().get`

instead, or provide your own function (or one from `numpy`).

[ ]:
```
result = lj.run('data/run.ini', random_seed=117208264)
```

```
==================== Lennard-Jones Simulation ====================

Temperature: 0.5
Density: 0.8
Number of Particles: 500
Time Step: 0.005
Random Seed: 117208264

Simulation will run in 2 phases:
Phase 1: Equilibration Phase
Phase 2: Observation Phase

Results to be output to the following files:
Events: events.log
Thermodynamics: thermodynamics.csv
Observations: observations.csv
Snapshots: snapshots.csv

Begin simulation…
0: Phase started: Equilibration Phase
50: Temperature measured at: 0.302, adjusted to: 0.5
100: Temperature measured at: 0.4128, adjusted to: 0.5
600: Phase complete: Equilibration Phase
600: Phase started: Observation Phase
650: Observation recorded
700: Observation recorded
750: Observation recorded
800: Observation recorded
850: Observation recorded
900: Observation recorded
950: Observation recorded
```

```
1000: Observation recorded
1050: Observation recorded
1100: Observation recorded
1150: Observation recorded
1200: Observation recorded
1250: Observation recorded
1300: Observation recorded
1350: Observation recorded
1400: Observation recorded
1450: Observation recorded
1500: Observation recorded
1550: Observation recorded
1600: Observation recorded
1600: Phase complete: Observation Phase
End simulation

1600 time steps computed in 2.455 seconds
0.002 seconds per step, or 1.534 milliseconds
Framerate: 651.687 fps
```

Here we just check how much data was written. For this small example, I want to put the entire `data/` directory on GitHub, so I wanted to make sure it was not too large.

[ ]:
```python
import os
_ = os.system('du -sh data')
```

```
328K    data
```

1.2 Analysing the output

Now let's load the data we've created and visualize it. First we need some additional packages:

[ ]:
```python
import pandas as pd
from mpl_toolkits import mplot3d
pd.plotting.register_matplotlib_converters()
import matplotlib.pyplot as plt
import seaborn as sns
```

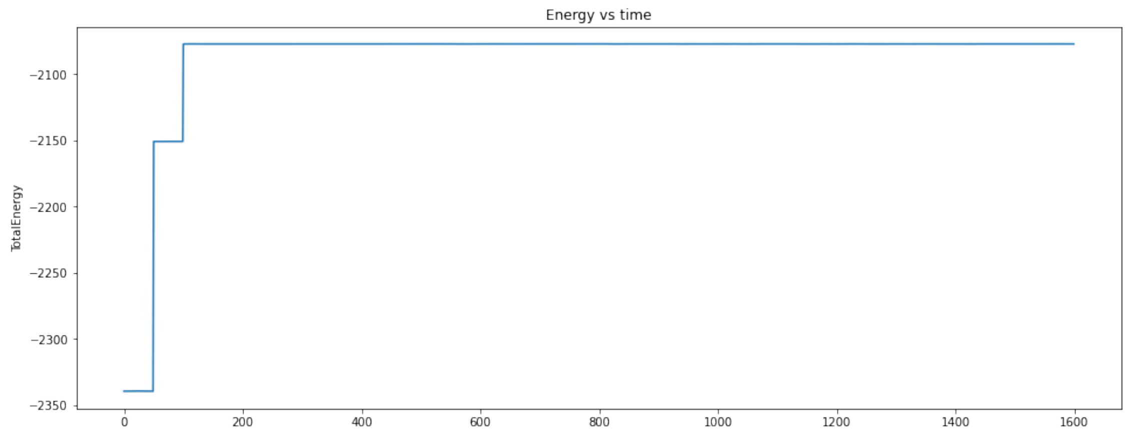
The data is output conveniently in `.csv` format, making it easy to load into `pandas`. The `snapshots.csv` file is special, however: it uses multi-indexing, and requires the additional `header=` argument to load it properly.

[ ]:
```python
thermoynamic_data = pd.read_csv("data/thermodynamics.csv")
observation_data = pd.read_csv("data/observations.csv")
snapshot_data = pd.read_csv("data/snapshots.csv", header=[0,1])
```

1.2.1 Visualization of the simulation process

Let's start by looking at what happens in the simulation over time. The most informative thing to plot is the total energy:

[ ]:
```
plt.figure(figsize=(16,6))
plt.title("Energy vs time")
sns.lineplot(data=thermoynamic_data['TotalEnergy'])
plt.show()
```

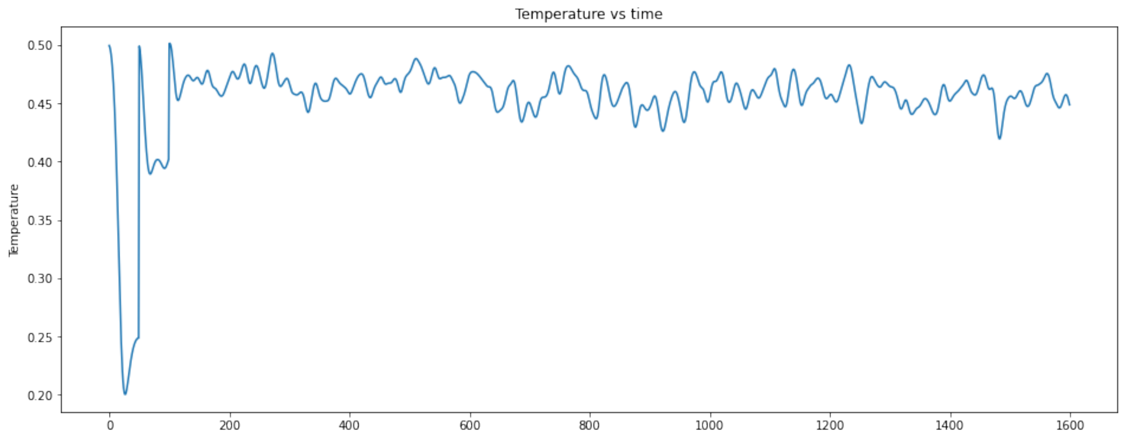


We see a number of interesting features. First of all, aside from a few “jumps” at the beginning, the total energy is roughly *constant* (it does actually have small fluctuations, but these are not visible at this scale). This is good, because we are simulating a closed system at constant total energy, and we were careful to choose a symplectic integrator so that the energy should remain close to constant. So, this plot reveals that the symplectic integrator is working.

As for the “jumps” at the beginning, those happen during the Equilibration phase of the simulation. We try to initialize the simulation with a Maxwell distribution of velocities at the correct temperature; however, this process is not perfect, and there will initially be quite a lot of temperature drift. During the Equilibration phase, we periodically measure the temperature of the system and adjust it by globally rescaling all velocities. Eventually, the system should settle down and remain within our given temperature tolerance for a sufficiently long amount of time. At that point, the Equilibration phase ends, and we move on to the Observation phase, where we can measure relevant quantities for equilibrium physics.

We can get further insight into this initial temperature drift by plotting the temperature of the system over time:

[ ]:
```
plt.figure(figsize=(16,6))
plt.title("Temperature vs time")
sns.lineplot(data=thermoynamic_data['Temperature'])
plt.show()
```



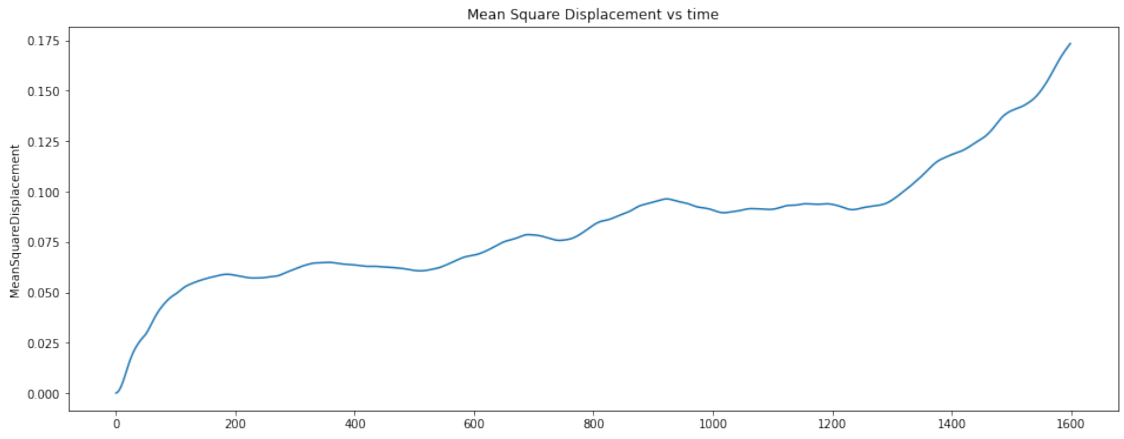
As you can see, the temperature near the beginning tries to fall below the initial value, and needs to be corrected a few times. Once we've reached "equilibrium", we can also see that the temperature continues to drift slowly downwards. For the sake of simulation speed (i.e., time taken to reach equilibrium), we have chosen a fairly wide temperature tolerance. But it is obvious from this graph that we are not close enough to equilibrium to make valid measurements in this demo experiment.

1.2.2 Physical observations

Next let's try to get an understanding of the physics. We chose to simulate the Lennard-Jones system at a density of 0.8 and a temperature of 0.5 in reduced units. I chose these values because they are very close to a triple point, which gives us something interesting to talk about.

For a single simulation like this, we would like to understand a basic question like "Is the system in a solid or fluid state?" A nice way to answer this question is to look at the mean square displacement:

[ ]:
```
plt.figure(figsize=(16,6))
plt.title("Mean Square Displacement vs time")
sns.lineplot(data=thermoynamic_data['MeanSquareDisplacement'])
plt.show()
```

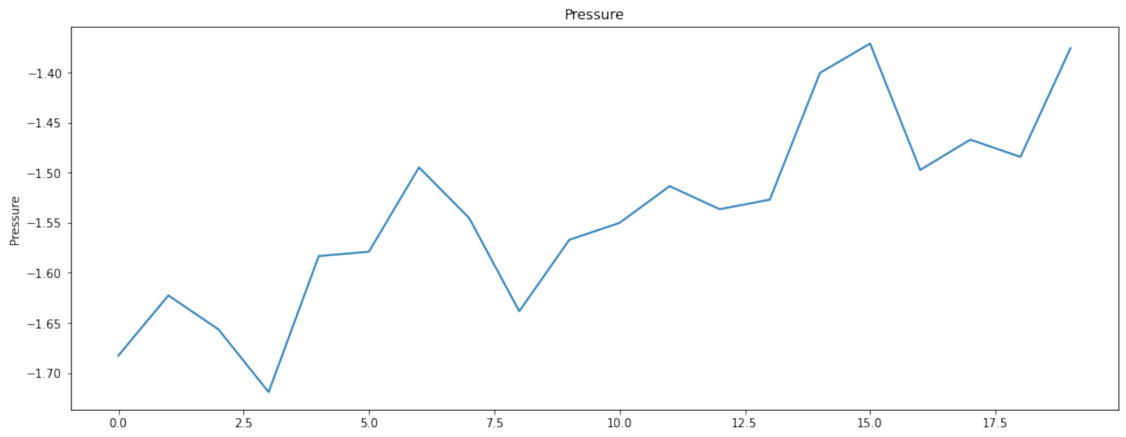


This mean square displacement actually exhibits very strange behavior! We should ignore the first part of it, since that was during the Equilibration phase and is not properly part of the experiment. But after the initial settling, we see that the mean square displacement alternates between being rougly flat, or increasing linearly.

Of course, the mean square displacement being flat is an indicator that atoms are staying still (i.e., a solid phase), whereas linearly increasing means that the atoms are engaged in diffusion (i.e., a fluid phase). It appears that the atoms in this simulation are constantly freezing and then melting again: a telltale sign that we are sitting at a phase transition point!

Let's also take a look at the pressure. In this simulation, we have taken 20 pressure measurements, obtained by averaging several physical properties over some given sliding time window over the course of the simulation:

[ ]:
```
plt.figure(figsize=(16,6))
plt.title("Pressure")
sns.lineplot(data=observation_data['Pressure'])
plt.show()
```



We see that the pressure is *negative*, which suggests that the atoms are not dispersing to fill their "container". Therefore we should conclude that during the times when this simulation exhibits fluid behavior, it is behaving as a *liquid*, since a gas would not support negative pressure.

1.2.3 Visualizing the final state

The simulation also records the final state in a `snapshots.csv` file. In principle, the format of this file could allow us to store several snapshots. In practice, it would be an impractical amount of data, so we only store the final state. In any case, this allows us to *look* at the system and understand what's going on.

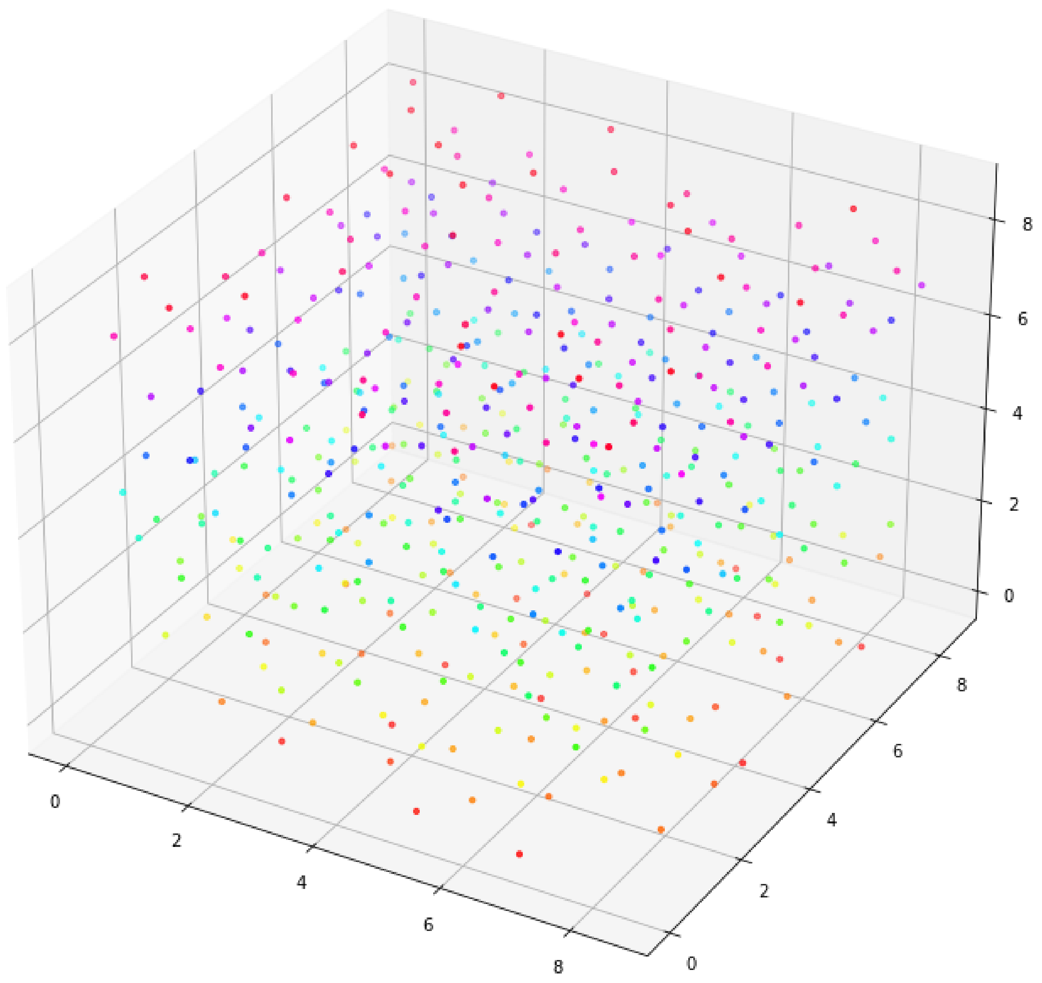
Here is what the snapshots file looks like:

```
[ ]: snapshot_data.head(10)
```

```
[ ]:  TimeStep ParticleID  Position                               Velocity            \
      TimeStep ParticleID         X         Y         Z         X         Y
    0     1600          0  0.233203  8.440631  8.282624 -0.567582  0.556043
    1     1600          1  0.990303  0.831183  8.419367  0.689747 -0.561095
    2     1600          2  0.990312  8.537178  0.907050  0.111305  0.256449
    3     1600          3  8.523888  0.906916  0.583649  1.029239  0.060980
    4     1600          4  0.083800  8.456665  1.653626 -0.418410  0.398105
    5     1600          5  0.875960  0.649243  1.674124  0.709302 -0.236276
    6     1600          6  0.844525  8.304534  2.413890 -0.196477 -0.449641
    7     1600          7  1.012698  0.882362  2.787505  0.133839 -0.250819
    8     1600          8  0.170776  8.400429  3.246523 -1.078942  1.364693
    9     1600          9  1.145364  1.230824  3.757346  0.697241  1.087242

                   Force
             Z         X          Y          Z
    0 -1.410407 -9.926590  -4.229977  15.258944
    1  0.913059 -9.272431 -10.680586  -1.025319
    2  0.744851 -9.120950 -16.252058 -16.388571
    3  0.749519  4.693217   0.710547   6.248287
    4  0.056528 -6.009218   7.399409 -11.877819
    5  0.128990  1.467884  13.711113  12.291029
    6  0.557142  6.904816   2.863215  -6.487414
    7  0.661273 -3.379601 -10.370031 -10.543975
    8 -0.397322 -10.494095  1.706656  -2.867521
    9 -0.478575 -12.528175  -6.417052  20.402740
```

Finally, let's plot the positions of the particles in three dimensions, to get an idea what is going on in this system:

```
[ ]: fig = plt.figure(figsize=(12, 12))
     ax = fig.add_subplot(projection='3d')

     ax.scatter3D(
         snapshot_data[('Position', 'X')],
         snapshot_data[('Position', 'Y')],
         snapshot_data[('Position', 'Z')],
         c=snapshot_data[('Position', 'Z')],
         cmap='hsv',
         s=10
     )

     plt.show()
```



We see that the atoms seem more or less randomly arranged, so this lins up with our earlier conclusion that we are in a fluid state (near the end of the simulation, the mean square displacement is growing linearly, rather than holding flat). We also see that the atoms are *not* evenly dispersed, but appear to have left “gaps”, which explains the negative pressure measurements. The attractive Lennard-Jones force in this temperature-density regime is not quite strong enough to hold the particles in a lattice, but it is strong enough to prevent them from filling their container.