1 Summary

Our current implementation of Lammps handles non-metallic units in some ways precariously, in others simply wrongly.

The precariousness comes from the fact that the 'units' attribute of the `input.control` object of a Lammps job is updated only when the potential is assigned to the job (automatically based on some units property of the potential object). Currently, other aspects of the input, such as those written by `calc_md` and `calc_minimize`, rely on this units attribute to properly convert from [pyiron units (https://pyiron.github.io/source/faq.html)](https://pyiron.github.io/source/faq.html) to [Lammps units (https://lammps.sandia.gov/doc/units.html)](https://lammps.sandia.gov/doc/units.html). Thus, if these are called in an a-typical order (i.e. `calc` before setting the potential), errors follow. It is certainly canonical to set the potential first, but there are no rules and certainly no safety checks making sure it has been done this way.

The wrongness comes from the fact that, while carefully accounted for in most (now all?) of the `calc` methods, units are either ignored or hard-coded into the output interpretation. Thus, while the input is in pyiron units, and the output at the moment is ok for "metal" units, all other outputs have at least some non-pyiron units.

The rest of the notebook simply demonstrates these claims.

I'll use water and "real" units as an example, because there we have a public [example notebook (https://github.com/pyiron/pyiron/blob/master/notebooks/water_MD.ipynb)](https://github.com/pyiron/pyiron/blob/master/notebooks/water_MD.ipynb) from which I'll draw heavily, e.g. to set up a cell of water. (I also found that one of the statements in this notebook doesn't hold, although it's a fairly low-priority statement.)

In [1]:
```
import numpy as np
%matplotlib inline
import matplotlib.pylab as plt
from pyiron.project import Project
import ase.units as units
```
executed in 2.44s, finished 15:53:49 2019-12-02

```
/Users/huber/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWa
rning: Conversion of the second argument of issubdtype from `float` to `np.float
ing` is deprecated. In future, it will be treated as `np.float64 == np.dtype(flo
at).type`.
  from ._conv import register_converters as _register_converters
```

In [2]:
```
pr = Project("tmp")
pr.remove_jobs(recursive=True)
pr.get_repository_status()
```
executed in 1.06s, finished 15:53:50 2019-12-02

Out[2]:

	Module	Git head
0	pyiron_mpie	d7f60587f8482d09931c3373c848b038308db07b
1	pyiron	cd2d24374c041808a175a8f8933eee308afe715e

In [3]:
```
density = 1.0e-24  # g/A^3
n_mols = 27
mol_mass_water = 18.015 # g/mol

# Determining the supercell size size
mass = mol_mass_water * n_mols / units.mol  # g
vol_h2o = mass / density # in A^3
a = vol_h2o ** (1./3.) # A

# Constructing the unitcell
n = int(round(n_mols ** (1. / 3.)))

dx = 0.7
r_O = [0, 0, 0]
r_H1 = [dx, dx, 0]
r_H2 = [-dx, dx, 0]
unit_cell = (a / n) * np.eye(3)
water = pr.create_atoms(elements=['H', 'H', 'O'],
                        positions=[r_H1, r_H2, r_O],
                        cell=unit_cell)
water.set_repeat([n, n, n])

potential = 'H2O_tip3p'
```
executed in 169ms, finished 15:53:50 2019-12-02

In [4]:
```
# water.plot3d()
```
executed in 3ms, finished 15:53:50 2019-12-02

2 Order dependency

Let's show that invoking `job.potential = X` and `job.calc_md` in the wrong order is a problem.

"real" units are extremely similar to "metal" units, differing only in the time, energy, velocity, force, and pressure.

In [5]:
```
temperature = 300
n_steps = 2000
n_print = 10
time_step = 1
time = np.linspace(start=0, stop=n_steps, num=int(n_steps/n_print) + 1, endpoint=T
```
executed in 5ms, finished 15:53:51 2019-12-02

In [6]:
```
job_normal_order = pr.create_job(pr.job_type.Lammps, 'normal_order')
print('created\t\tunits=', job_normal_order.input.control["units"])
job_normal_order.structure = water.copy()
print('structure set\tunits=', job_normal_order.input.control["units"])
job_normal_order.potential = potential
print('potential set\tunits=', job_normal_order.input.control["units"])
job_normal_order.calc_md(temperature=temperature, n_ionic_steps=n_steps, n_print=n
print('calc set\tunits=', job_normal_order.input.control["units"])
job_normal_order.run()
```
executed in 3.26s, finished 15:53:54 2019-12-02

```
created         units= metal
structure set   units= metal
potential set   units= real
calc set        units= real
The job normal_order was saved and received the ID: 113
```

In [7]:
```
# job_normal_order.animate_structure()
```
executed in 3ms, finished 15:53:54 2019-12-02

In [8]:
```
job_wrong_order = pr.create_job(pr.job_type.Lammps, 'wrong_order')
print('created\t\tunits=', job_wrong_order.input.control["units"])
job_wrong_order.structure = water.copy()
print('structure set\tunits=', job_wrong_order.input.control["units"])
job_wrong_order.calc_md(temperature=temperature, n_ionic_steps=n_steps, n_print=n_
print('calc set\tunits=', job_wrong_order.input.control["units"])
job_wrong_order.potential = potential
print('potential set\tunits=', job_wrong_order.input.control["units"])
job_wrong_order.run()
```
executed in 3.19s, finished 15:53:57 2019-12-02

```
created         units= metal
structure set   units= metal
calc set        units= metal
potential set   units= real
The job wrong_order was saved and received the ID: 114
```

First, we see that the input in the control really is wrong. The 'normal' order, where we define the potential before setting the calculation type correctly shows the timestep and thermostat damping timescale in fs, which are both "real" units and pyiron units. However, if we happen to declare the calculation and potential in the opposite order then the thermostat and timestep are locked in at their "metal" unit values, which are in ps instead of fs!

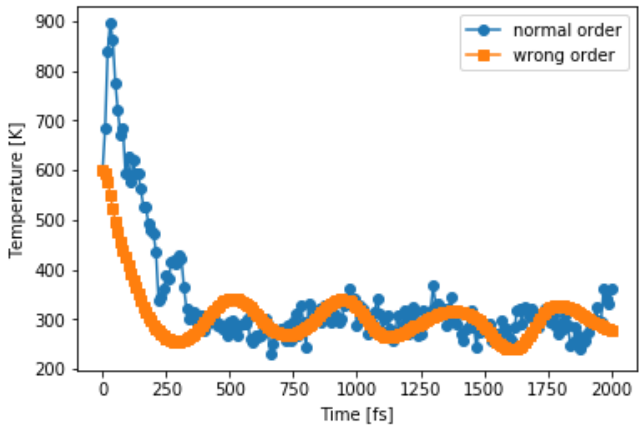
In [9]:
```
# The order really did change the input
print("Normal input:\n\tfix___ensemble = {}\n\ttimestep = {}".format(
    job_normal_order.input.control["fix___ensemble"],
    job_normal_order.input.control["timestep"]
))
print("Wrong order input:\n\tfix___ensemble = {}\n\ttimestep = {}".format(
    job_wrong_order.input.control["fix___ensemble"],
    job_wrong_order.input.control["timestep"]
))
```
executed in 9ms, finished 15:53:57 2019-12-02

```
Normal input:
        fix___ensemble = all nvt temp 300 300 100.0
        timestep = 1
Wrong order input:
        fix___ensemble = all nvt temp 300 300 0.1
        timestep = 0.001
```

This obviously has an impact on the run. For instance, below we see that with the wrong order the thermostat is much to strong (very rapid convergence to target temperature) and we simulate much less time than we intended (timescale of oscillations still perfectly clear). And, of course, simulating so short a time means that the 'wrong order' simulation hasn't actually had a chance to relax, and it's energy output remains much higher than we'd like and expect.

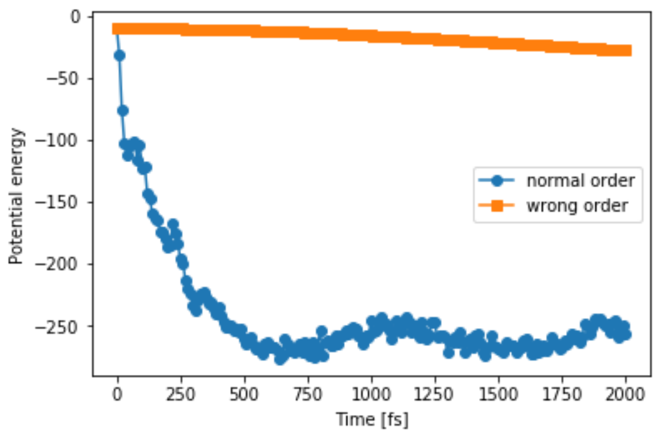
In [10]:
```
plt.plot(time, job_normal_order.output.temperature, marker='o', label='normal orde
plt.plot(time, job_wrong_order.output.temperature, marker='s', label='wrong order'
plt.legend()
plt.xlabel('Time [fs]')
```
executed in 298ms, finished 15:53:57 2019-12-02

Out[10]:
```
Text(0,0.5,'Temperature [K]')
```



In [11]:
```
plt.plot(time, job_normal_order.output.energy_pot, marker='o', label='normal order
plt.plot(time, job_wrong_order.output.energy_pot, marker='s', label='wrong order')
plt.legend()
plt.xlabel('Time [fs]')
```
executed in 281ms, finished 15:53:58 2019-12-02

Out[11]:
```
Text(0,0.5,'Potential energy')
```



As an aside, the example notebook claims that a time step of 1 fs is too high and uses 0.01 fs instead.

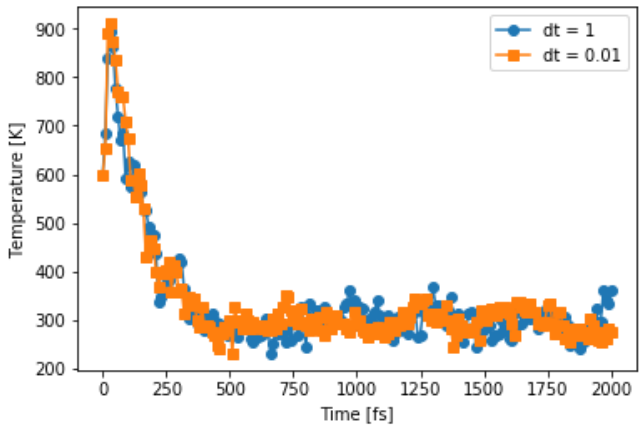
I don't think this is at all the case, and the example should be updated at some point:

In [12]:
```
rescale_time = 100
job_normal_order_slow = pr.create_job(pr.job_type.Lammps, 'normal_order_slow')
job_normal_order_slow.structure = water.copy()
job_normal_order_slow.potential = potential
job_normal_order_slow.calc_md(
    temperature=temperature,
    n_ionic_steps=n_steps*rescale_time,
    n_print=n_print*rescale_time,
    time_step=time_step/rescale_time
)
```
executed in 2m 42s, finished 15:56:40 2019-12-02

```
The job normal_order_slow was saved and received the ID: 115
```

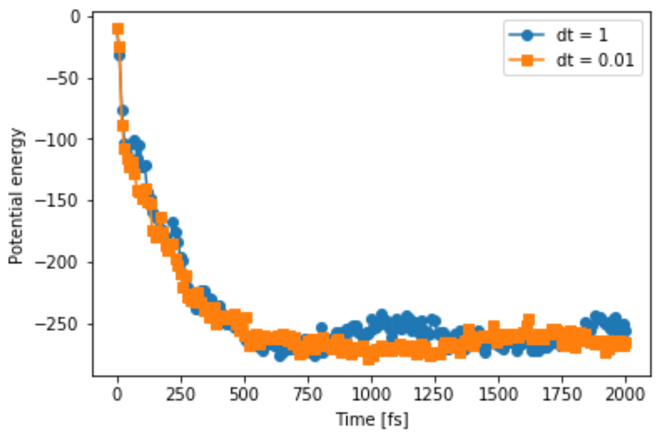
In [13]:
```
plt.plot(time, job_normal_order.output.temperature, marker='o', label='dt = 1')
plt.plot(time, job_normal_order_slow.output.temperature, marker='s', label='dt = 0
plt.legend()
plt.xlabel('Time [fs]')
```
executed in 244ms, finished 15:56:40 2019-12-02

Out[13]:
```
Text(0,0.5,'Temperature [K]')
```



In [14]:
```
plt.plot(time, job_normal_order.output.energy_pot, marker='o', label='dt = 1')
plt.plot(time, job_normal_order_slow.output.energy_pot, marker='s', label='dt = 0.
plt.legend()
plt.xlabel('Time [fs]')
```
executed in 235ms, finished 15:56:40 2019-12-02

Out[14]:
```
Text(0,0.5,'Potential energy')
```



3 Output parsing

Next, let's demonstrate that we account for units only on the way in, and not (properly) on the way out. Both "metal" and "real" units agree with pyiron units for a good deal of output, so this is a little tricky.

To demonstrate the point, we'll run an NPT ensemble and look at pressure, since "metal" and "real" use bar and atm, respectively, while pyiron uses GPa. Now, in fact, `pyiron/lammps/base/LammpsBase.collect_output_log` **does** convert units back to pyiron! But it hard-codes in the metal transition by calling `pressures *= 0.0001  # bar -> GPa` (line 543 at time of writing).

However, bar at atm are not identical; rather 1 bar = 0.986923 atm. So if we run NPT targetting a non-zero pressure for both a "metal"-using job and a "real"-using job (i.e. water), we should see that the metallic job hits the target pressure more less spot on, while the water underestimates by the ratio of atm to bar.

Indeed, this is exactly what we see below. ...And most other properties don't have *any* conversion for output.

In [15]:
```
temperature = 300
n_steps = 40000
n_print = 100
time_step = 1
time = np.linspace(start=0, stop=n_steps, num=int(n_steps/n_print) + 1, endpoint=T
pressure = 50
```
executed in 4ms, finished 15:56:40 2019-12-02

In [16]:
```
job_npt_water = pr.create_job(pr.job_type.Lammps, 'npt_water')
job_npt_water.structure = water.copy()
job_npt_water.potential = potential
job_npt_water.calc_md(
    temperature=temperature,
    n_ionic_steps=n_steps,
    n_print=n_print,
    time_step=time_step,
    pressure=pressure
)
job_npt_water.run()
```
executed in 1m 10.1s, finished 15:57:51 2019-12-02

```
The job npt_water was saved and received the ID: 116
```

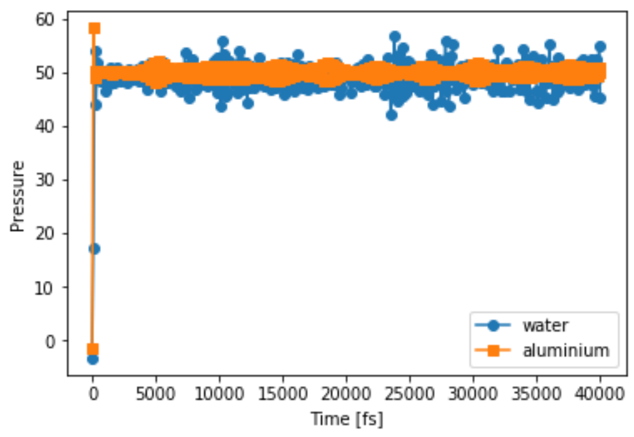
In [17]:
```
job_npt_Al = pr.create_job(pr.job_type.Lammps, 'npt_Al')
job_npt_Al.structure = pr.create_ase_bulk('Al', cubic=True).repeat(4)
job_npt_Al.potential = job_npt_Al.list_potentials()[0]
job_npt_Al.calc_md(
    temperature=temperature,
    n_ionic_steps=n_steps,
    n_print=n_print,
    time_step=time_step,
    pressure=pressure
)
job_npt_Al.run()
```
executed in 39.2s, finished 15:58:30 2019-12-02

```
The job npt_Al was saved and received the ID: 117
```

In [18]:
```
def pressure_tensor_to_mean(pressures):
    return np.mean(np.diagonal(pressures, axis1=1, axis2=2), axis=-1)
```
executed in 4ms, finished 15:58:30 2019-12-02

In [19]:
```
plt.plot(time, pressure_tensor_to_mean(job_npt_water.output.pressures), marker='o'
plt.plot(time, pressure_tensor_to_mean(job_npt_Al.output.pressures), marker='s', l
plt.legend()
plt.xlabel('Time [fs]')
plt.ylabel('Pressure')
```
executed in 402ms, finished 15:58:30 2019-12-02

Out[19]:
```
Text(0,0.5,'Pressure')
```



In [20]:
```
n_equil = 10
n_samples = (n_steps / n_print) - n_equil

water_P = pressure_tensor_to_mean(job_npt_water.output.pressures)[n_equil:]
water_P_ave = np.mean(water_P)
water_P_std = np.std(water_P)
water_P_stderr = water_P_std / np.sqrt(n_samples)

Al_P = pressure_tensor_to_mean(job_npt_Al.output.pressures)[n_equil:]
Al_P_ave = np.mean(Al_P)
Al_P_std = np.std(Al_P)
Al_P_stderr = Al_P_std / np.sqrt(n_samples)

print("Water pressure = {} +- {}".format(water_P_ave, water_P_stderr))
print("Al pressure = {} += {}".format(Al_P_ave, Al_P_stderr))
```
executed in 119ms, finished 15:58:30 2019-12-02

```
Water pressure = 49.300583936991394 +- 0.10872091056588776
Al pressure = 50.001873204879345 += 0.039081594852438784
```

In [21]:
```
from scipy.constants import bar, atm
print("Water pressure / Al pressure \t = {}".format(water_P_ave/Al_P_ave))
print("bar/atm \t\t\t = {}".format(bar/atm))
```
executed in 7ms, finished 15:58:30 2019-12-02

```
Water pressure / Al pressure     = 0.9859747400859471
bar/atm                          = 0.9869232667160128
```