Grid-Search Implementation in PyAMFF

Background

Python Atom-Centered Machine Learning Force Field (PyAMFF) is a set of atomistic machine learning tools developed by the Henkelman and Lei Li Groups at UT Austin. The training data for these processes is handled using the Atomic Simulation Environment (ASE) Python library's Trajectory object–stored on a .traj file–which defines a time series of one for more ase.Atoms objects. When loading training data, a list of atomic neighbors within a specified cutoff radius to each Atom in the Trajectory is calculated; as the size of training data grows, the computational cost of these calculations increase exponentially.

By implementing a 3-dimensional bin sort before these calculations are made, using the cutoff as a guide for the bin sizes, I hoped to reduce the computational complexity of this process to save time costs when loading training data.

Methods

I have written code in isolation from PyAMFF itself, to test solely the time savings of this algorithm apart from the rest of the training data processing.

I created a series of test Trajectory files of varying numbers of identical atoms randomly uniformly generated. I kept the relative density of atoms equal, so the number of atoms in the system are proportional to the size of the unit cell. As I ran this program locally, I kept the total number of atoms low to account for processing speed on my personal machine.

I wrote code for two implementations: one with a simple version of the atom nearest neighbor calculation, and another with a bin-sort algorithm filtering the atom list before the neighbor calculations occur. In the bin-sort algorithm, I filtered the list of atoms down to only those within the same bin as the atom, as well as the surrounding bins in each of the three dimensions, with periodicity accounted for.

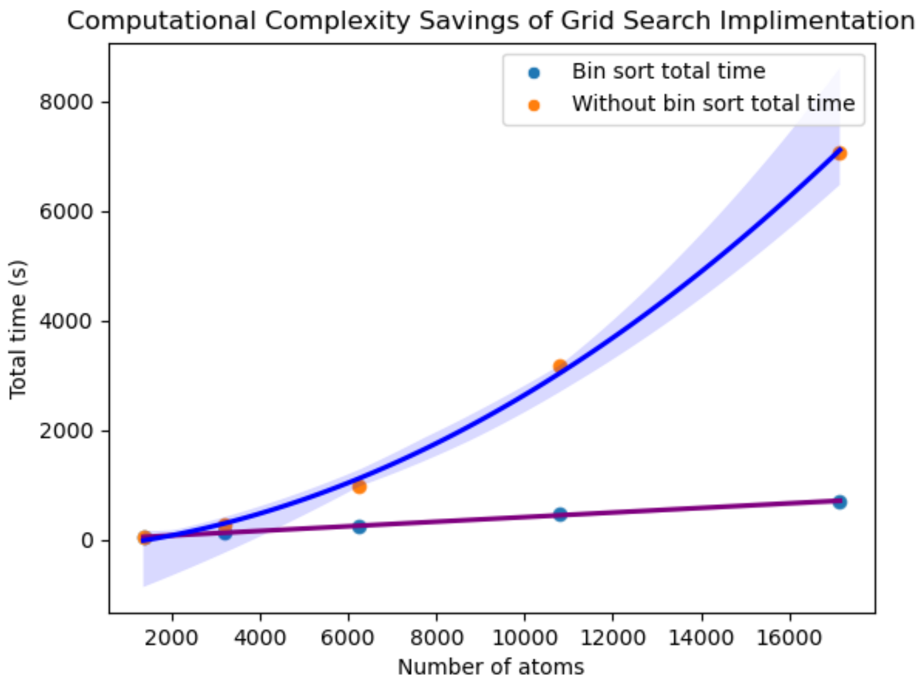
Current code only calls the get_nighbors method, which forces the machine to run the calculation but does not do anything with this call or save it to any location. Logging or printing the results of this call made negligible changes to the time when tested; the most costly aspect of this code is the get_distance calculation itself.

I have also included the code for my graph creation.

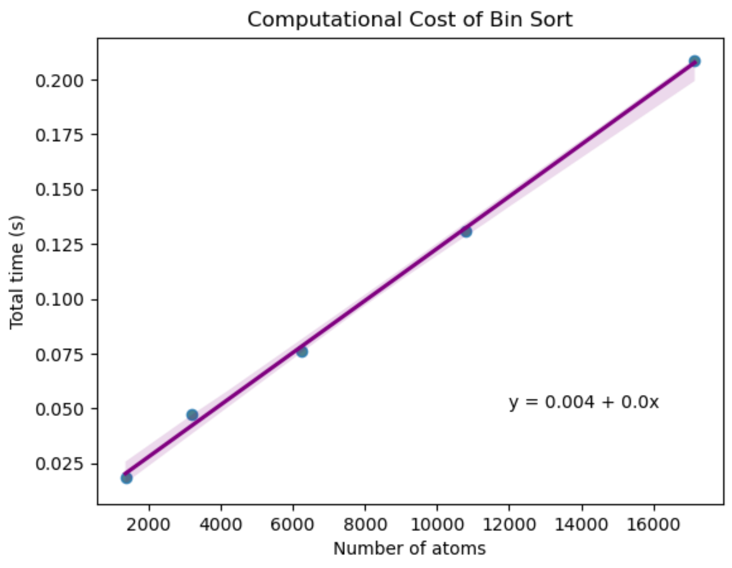
Results

The original neighbor calculations are of complexity O(n^2): every atom is compared to every other atom in the system that is not itself. The bin-sort neighbor calculations are of O(kn): in a system with uniform density, with equal sized bins, the number of atoms in each neighbor calculation are roughly equal.

This computational complexity can be easily see in the following graph:



You can see that, looking at the time cost of the bin_sort and bin_cull methods alone, the time cost is both linear and negligible compared to the cost of the get_neighbors method–on the order of tenths of seconds, versus the order of thousands of seconds.



Continuation

There’s a number of directions to take this project from here:

- I want to develop unit tests for this code: a series of trajectory files with known results by which I can evaluate future changes for accuracy. While my current code gives results
- As the code stands, periodicity is handled with a modulus in the bin_cull method. This fails to account for small systems, where any dimension is less than 3 times the size of the cutoff. I can likely accomplish this with a quick check on the number of bins in the system, and passing the method if there are less bins in a specific dimension than 3 times the cutoff value. I’ll have to rework my code to account for systems with unit cells that are significantly smaller in one dimension than the other two (i.e., a<<<b,c).
- Dr. Chai had suggested a variation of this algorithm, which would involve sorting the entire Atoms object by each of its three dimensions to narrow the list before the bin sort. While the bin sort itself is of a negligible time cost versus the get_distance method call, it would be interesting to try implementing–if time allows–to check time savings. If nothing else, it would be a demonstration of a different sort method.
- I’d like to integrate this algorithm into PyAMFF itself. I have limited experience adding code to an existing project, so I’d have to work with the group members in charge of the codebase to integrate this with the existing neighbor_list.py file, including the angle calculations. I’d only attempt this once I’m satisfied with the final implementation and am granted permission to do so.