The Journal of Open Source Software

pylattica: a package for prototyping lattice models in chemistry and materials science

Max C. Gallant [1,2] **and Kristin A. Persson** [1,2]

1 Lawrence Berkeley National Laboratory, U.S.A. **2** University of California, Berkeley, U.S.A.

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Richard Gowers

Reviewers:

- @riesben
- @amkrajewski

Submitted: 14 November 2023
Published: unpublished

Summary

pylattica provides a simple and flexible framework for prototyping lattice-based simulations such as atomistic Monte Carlo simulations or cellular automata. It is differentiated from other lattice simulation packages by i) its agnosticism toward the form of the update rule, simulation structure, neighborhood structure, and simulation state and ii) its interoperability with the pymatgen package which allows modeling of arbitrary crystalline systems (i.e. not only two or three dimensional square grids), and makes it particularly well suited to applications in materials science and chemistry.

Statement of need

Cellular automata (Bays, 2010), lattice-gas automata (Boghosian, 1999), and atomistic Monte Carlo models (Andersen et al., 2019) are all simulations in which a system, represented by an arrangement of connected sites, evolves over time according to an update rule which determines the future state of a site by considering its current state and the state of each of its neighbors. For example, in the classic "Game of Life" cellular automaton (Gardner, 1970), sites in a 2D square grid switch between "dead" and "alive" during each timestep based on the number of living neighbors surrounding them. In lattice Monte Carlo simulations for vacancy diffusion in crystalline solid materials, atoms move between neighboring sites at rates partially determined by the occupancy of their neighbors (Haley et al., 2006).

These simulation classes have been implemented many times in various programming languages for a range of applications (Andersen et al., 2019; Raabe, 2002). However, these implementations typically focus on tuning an existing simulation form within a relatively narrow range of focus. For instance, `CellPyLib` (L. Antunes, 2023), `netomaton` (L. M. Antunes, 2019), and `cellular_automaton` (Feistenauer, 2021) are all libraries for simulating cellular automata, but they each are limited in the simulation geometry, the data type for the simulation state, the geometry of the neighborhood, or the strategy for applying the update rule. Similarly, `lattice_mc` (Morgan, 2017) is an excellent Monte Carlo program that focuses solely on diffusion in ionic solids. While `KMCLib` (Leetmaa & Skorodumova, 2014) is a more generic alternative, it is still (appropriately) limited in the form of the state and the update rule.

The goal of pylattica is to synthesize the essential elements of these valuable simulation classes into a flexible and user-friendly framework for developing lattice models that do not fit neatly into the target use case of one of the existing packages. It accomplishes this by providing implementations of common lattice model features (e.g. various neighborhoods, methods for applying evolution rules, simulation structures, and analysis tools) while remaining unopinionated with regard to the ways these pieces are used in new models. It is implemented in python to maximize accessibility and interoperability with other scientific software tools, in particular, pymatgen, a package containing utilities for analysis in materials science (Ong et al., 2013).

Because `pylattica` is focused on enabling fast iteration on simulation features during development, it prioritizes flexibility and application agnosticism over performance. Therefore, it is better suited for cases in which the developer needs to prototype and experiment with various forms of their simulation as opposed to honing in a hardened production model.

Package Overview

Simulation representation

In lattice models, a system is represented by a network of connected sites, frequently a two or three dimensional square grid, with some state value assigned to each site. In `pylattica`, this representation is accomplished by the combination of three entities, which separate the dominant concerns (illustrated schematically in Figure 1):

- A `Structure`, which enumerates the sites and their physical locations with no limitations on periodicity or dimensionality
- A `SimulationState`, which acts as an index of sites and stores the state of each site as an arbitrary key-value mapping
- A `Neighborhood`, which encodes the connectedness of the sites

Of these three entities, only a `SimulationState` is required to run a simulation. The user can freely utilize `Structures` and `Neighborhoods` as required by their use case in the preparation or evolution of the system.

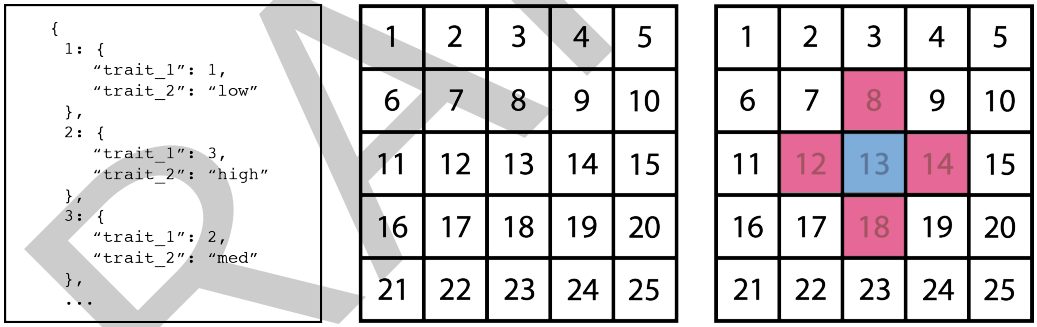

Figure 1: Schematic showing an example state, a structure labeled with site IDs, and a possible neighborhood for site 13 in a simulation with a two dimensional grid structure.

Constructing Neighborhoods

`pylattica` supports two and three dimensional square grid simulation structures out of the box (though any simulation structure can be created), and provides convenience methods for building them. Additionally, it provides a number of `NeighborhoodBuilder` classes which encode methods for specifying site neighbors in `Structures`. The two most flexible `NeighborhoodBuilder` classes are the `DistanceNeighborhoodBuilder` and the `MotifNeighborhoodBuilder`. Using the `DistanceNeighborhoodBuilder`, the neighbors of a site are defined as all other sites falling within a particular cutoff distance. Using the `MotifNeighborhoodBuilder`, the locations of a site's neighbors are specified by providing a list of offset vectors from that site (one for each neighbor). While these two classes can be used to construct practically any neighborhood, builder classes for the following common neighborhoods are also provided:

- Moore (square grid) (Packard & Wolfram, 1985)
- Von Neumann (square grid) (Packard & Wolfram, 1985)
- Pseudopentagonal (square grid) (Sieradzki & Madej, 2013)
- Pseudohexagonal (square grid) (Sieradzki & Madej, 2013)
- Annular (arbitrary structure)

Simulation Execution

Running a simulation entails applying an "update rule" to sites in the simulation. `pylattica` only requires that the update rule accept a site identifier and the current simulation state as input and provide a collection of intended state changes as output. This rule is implemented by the user in the `get_state_update` method on a `Controller` class. In most cases, a `Neighborhood` object will be used to consider the state of neighboring sites when calculating the intended changes, though this is not required. The flexibility provided by this arrangement makes it straightforward to iterate on the definition of the rule while developing a simulation.

The simulation is evolved by providing the `Controller` and a desired number of steps to an instance of the `Runner` class. The `Runner` passes sites to the `Controller`, and keeps track of updates as they are returned and accumulated over the course of the simulation. Two modes of evolution are supported by `pylattica` (Fatès, 2013):

- Synchronous - at each simulation step, the rule is applied to every site
- Asynchronous - at each simulation step, the rule is applied to a single random site

The result of a simulation run is an instance of `SimulationResult`, which stores the state at every step in the simulation as a list of `SimulationStates`. It can be easily serialized for storage on a filesystem or a document store, like MongoDB.

Overview

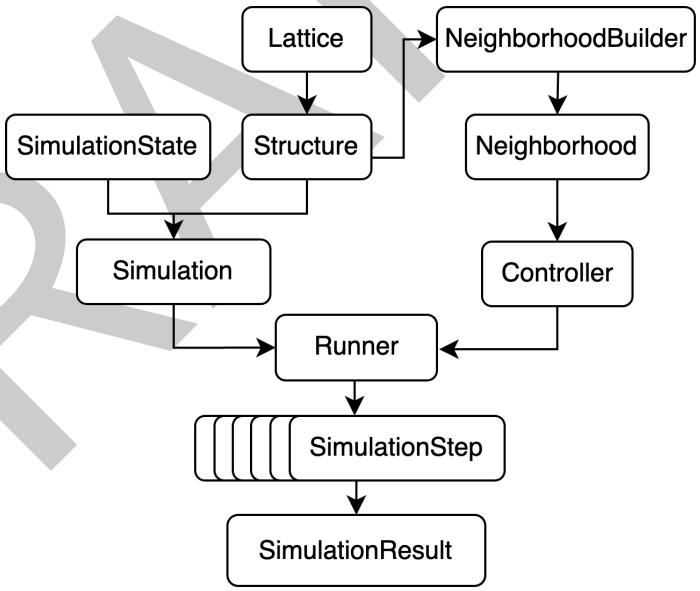

Figure 2: Diagram showing relationships between `pylattica` entities.

Figure 2 shows the relationship between the entities discussed so far, and how they are connected in producing a `SimulationResult`. To summarize, a `Lattice` is used to create a `Structure`, which is paired with an initial `SimulationState` to create a `Simulation`, or the starting point for simulation execution. The `Structure` is also fed to a `NeighborhoodBuilder` to construct a `Neighborhood` object, which is used in the update rule implemented by the `Controller` to determine how the simulation evolves. Finally, the `Simulation` and `Controller` are passed to a `Runner`, which applies the update rule repeatedly, producing a series of `SimulationStates`, which are concatenated to form a `SimulationResult`.

Visualization and Analysis

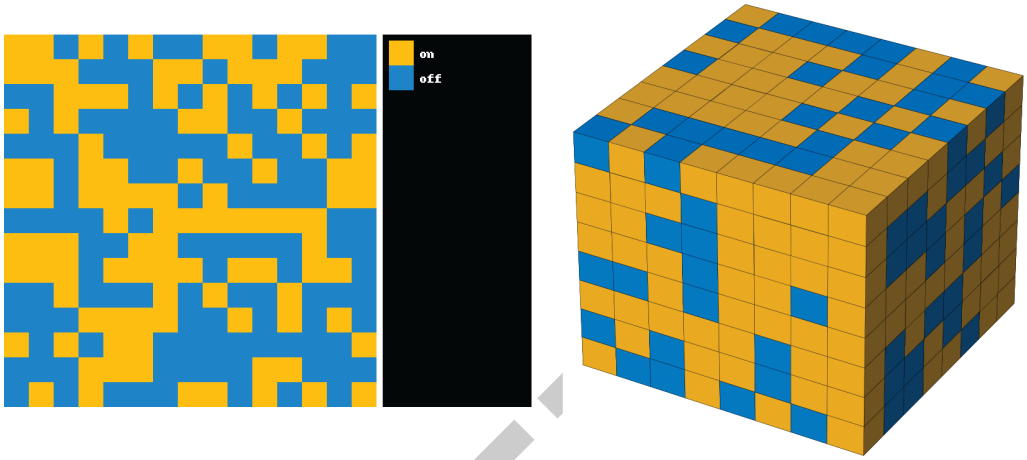

Figure 3: Example visualizations of two and three dimensional square grid simulation states.

`pylattica` provides basic utilities for analyzing the state of the simulation. These tools provide functionality for filtering and counting sites in a `SimulationState` by arbitrary criteria (implemented as a function of the site's state). Further specialized support is provided for simulation states in which the state of each site is a single discrete label (as is the case in traditional cellular automata).

In the case of simulations with two- and three-dimensional square grid structures, `pylattica` provides visualization tools which convert `SimulationStates` into PNG images (as shown in Figure 3) and `SimulationResults` into animated GIFs.

Crystal Structure Support and pymatgen

`pylattica` was developed with simulations of crystalline materials in mind. As a result, it supports simulation `Structures` defined with periodic boundaries and lattices with arbitrarily shaped unit cells. These `Structures` are supported by a `Lattice` class which was cloned from `pymatgen` and then adapted to the needs of `pylattica`, primarily because `pymatgen`'s implementation is hard-coded to use 3-dimensions, while `pylattica` strives for generality and enforces no such constraint. In service of developing simulations of real crystalline materials, `pylattica` also provides utility functions for defining neighborhoods in periodic space based on displacement motifs (e.g. octahedral or tetrahedral neighbors) and supports converting `pymatgen.Structure` objects to `pylattica Structures`. This feature is intended to enable more seamless integration with existing materials science workflows.

Acknowledgments

This work was primarily funded and intellectually led by the Materials Project, which is funded by the U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences, Materials Sciences and Engineering Division, under Contract no. DE-AC02-05-CH11231: Materials Project program KC23MP. It also received support from the U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences, Materials Sciences and Engineering Division, under Contract No. DE-AC02-05CH11231 within the Data Science for Data-Driven Synthesis Science grant (KCD2S2). MCG acknowledges Matthew J. McDermott and Bryant Li for useful discussions during the development of this work.

Andersen, M., Panosetti, C., & Reuter, K. (2019). A Practical Guide to Surface Kinetic Monte Carlo Simulations. *Frontiers in Chemistry*, *7*. https://doi.org/10.3389/fchem.2019.00202

Antunes, L. (2023). *CellPyLib*. https://github.com/lantunes/cellpylib

Antunes, L. M. (2019). *Netomaton: A Python Library for working with Network Automata*. Zenodo. https://doi.org/10.5281/ZENODO.3893141

Bays, C. (2010). Introduction to Cellular Automata and Conway's Game of Life. In A. Adamatzky (Ed.), *Game of Life Cellular Automata* (pp. 1–7). Springer. https://doi.org/10.1007/978-1-84996-217-9_1

Boghosian, B. M. (1999). Lattice gases and cellular automata. *Future Generation Computer Systems*, *16*(2), 171–185. https://doi.org/10.1016/S0167-739X(99)00045-X

Fatès, N. (2013). A Guided Tour of Asynchronous Cellular Automata. In J. Kari, M. Kutrib, & A. Malcher (Eds.), *Cellular Automata and Discrete Complex Systems* (pp. 15–30). Springer. https://doi.org/10.1007/978-3-642-40867-0_2

Feistenauer, R. (2021). Cellular_automaton. In *GitLab*. https://gitlab.com/DamKoVosh/cellular_automaton

Gardner, M. (1970). Mathematical Games. *Scientific American*, *223*(4), 120–123. https://www.jstor.org/stable/24927642

Haley, B. P., Beardmore, K. M., & Grønbech-Jensen, N. (2006). Vacancy clustering and diffusion in silicon: Kinetic lattice Monte Carlo simulations. *Physical Review B*, *74*(4), 045217. https://doi.org/10.1103/PhysRevB.74.045217

Leetmaa, M., & Skorodumova, N. V. (2014). KMCLib: A general framework for lattice kinetic Monte Carlo (KMC) simulations. *Computer Physics Communications*, *185*(9), 2340–2349. https://doi.org/10.1016/j.cpc.2014.04.017

Morgan, B. J. (2017). Lattice_mc: A python lattice-gas monte carlo module. *Journal of Open Source Software*, *2*(13), 247. https://doi.org/10.21105/joss.00247

Ong, S. P., Richards, W. D., Jain, A., Hautier, G., Kocher, M., Cholia, S., Gunter, D., Chevrier, V. L., Persson, K. A., & Ceder, G. (2013). Python Materials Genomics (pymatgen): A robust, open-source python library for materials analysis. *Computational Materials Science*, *68*, 314–319. https://doi.org/10.1016/j.commatsci.2012.10.028

Packard, N. H., & Wolfram, S. (1985). Two-dimensional cellular automata. *Journal of Statistical Physics*, *38*(5), 901–946. https://doi.org/10.1007/BF01010423

Raabe, D. (2002). Cellular Automata in Materials Science with Particular Reference to Recrystallization Simulation. *Annual Review of Materials Research*, *32*(1), 53–76. https://doi.org/10.1146/annurev.matsci.32.090601.152855

Sieradzki, L., & Madej, L. (2013). A perceptive comparison of the cellular automata and Monte Carlo techniques in application to static recrystallization modeling in polycrystalline materials. *Computational Materials Science*, *67*, 156–173. https://doi.org/10.1016/j.commatsci.2012.08.047