



freud: A software suite for high throughput analysis of particle simulation data^{☆,☆☆}



Vyas Ramasubramani^a, Bradley D. Dice^b, Eric S. Harper^c, Matthew P. Spellings^a,
Joshua A. Anderson^a, Sharon C. Glotzer^{a,b,c,d,*}

^a Department of Chemical Engineering, University of Michigan, Ann Arbor, MI 48109, United States of America

^b Department of Physics, University of Michigan, Ann Arbor, MI 48109, United States of America

^c Department of Materials Science and Engineering, University of Michigan, Ann Arbor, MI 48109, United States of America

^d Biointerfacing Institute, University of Michigan, Ann Arbor, MI 48109, United States of America

ARTICLE INFO

Article history:

Received 12 June 2019

Received in revised form 24 February 2020

Accepted 5 March 2020

Available online 17 March 2020

Keywords:

Simulation analysis

Molecular dynamics

Monte Carlo

Computational materials science

ABSTRACT

The *freud* Python package is a library for analyzing simulation data. Written with modern simulation and data analysis workflows in mind, *freud* provides a Python interface to fast, parallelized C++ routines that run efficiently on laptops, workstations, and supercomputing clusters. The package provides the core tools for finding particle neighbors in periodic systems, and offers a uniform API to a wide variety of methods implemented using these tools. As such, *freud* users can access standard methods such as the radial distribution function as well as newer, more specialized methods such as the potential of mean force and torque and local crystal environment analysis with equal ease. Rather than providing its own trajectory data structure, *freud* operates either directly on NumPy arrays or on trajectory data structures provided by other Python packages. This design allows *freud* to transparently interface with many trajectory file formats by leveraging the file parsing abilities of other trajectory management tools. By remaining agnostic to its data source, *freud* is suitable for analyzing any particle simulation, regardless of the original data representation or simulation method. When used for on-the-fly analysis in conjunction with scriptable simulation software such as HOOMD-blue, *freud* enables smart simulations that adapt to the current state of the system, allowing users to study phenomena such as nucleation and growth.

Program summary

Program Title: *freud*

Program Files doi: <http://dx.doi.org/10.17632/v7wmv9xcct.1>

Licensing provisions: BSD 3-Clause

Programming language: Python, C++

Nature of problem: Simulations of coarse-grained, nano-scale, and colloidal particle systems typically require analyses specialized to a particular system. Certain more standardized techniques – including correlation functions, order parameters, and clustering – are computationally intensive tasks that must be carefully implemented to scale to the larger systems common in modern simulations.

Solution method: *freud* performs a wide variety of particle system analyses, offering a Python API that interfaces with many other tools in computational molecular sciences via NumPy array inputs and outputs. The algorithms in *freud* leverage parallelized C++ to scale to large systems and enable real-time analysis. The library's broad set of features encode few assumptions compared to other analysis packages, enabling analysis of a broader class of data ranging from biomolecular simulations to colloidal experiments.

[☆] The review of this paper was arranged by Prof. D.P. Landau.

^{☆☆} This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (<http://www.sciencedirect.com/science/journal/00104655>).

* Corresponding author at: Department of Chemical Engineering, University of Michigan, Ann Arbor, MI 48109, United States of America.

E-mail addresses: vramasub@umich.edu (V. Ramasubramani), bdice@umich.edu (B.D. Dice), harperic@umich.edu (E.S. Harper), mspellings@umich.edu (M.P. Spellings), joaander@umich.edu (J.A. Anderson), sglotzer@umich.edu (S.C. Glotzer).

Additional comments including restrictions and unusual features:

1. `freud` provides very fast parallel implementations of standard analysis methods like RDFs and correlation functions.
2. `freud` includes the reference implementation for the potential of mean force and torque (PMFT).
3. `freud` provides various novel methods for characterizing particle environments, including the calculation of descriptors useful for machine learning. The source code is hosted on GitHub (<https://github.com/glutzerlab/freud>), and documentation is available online (<https://freud.readthedocs.io/>). The package may be installed via `pip install freud-analysis` or `conda install -c conda-forge freud`.

© 2020 Elsevier B.V. All rights reserved.

1. Introduction

Molecular simulation is a crucial pillar in the investigation of scientific phenomena. Increased computational resources, better algorithms, and new hardware architectures have made it possible to simulate complex systems over longer timescales than ever before [1–5]. The sheer volume of data necessitates computationally efficient analysis tools, while the diversity of data requires flexible tools that can be adapted for specific systems. Additionally, to support scientists with limited prior computing experience, tools must be usable without extensive knowledge of the underlying code.

Numerous software packages that satisfy these requirements have been developed in recent years. Tools such as MDTraj [6], MDAAnalysis [7], LOOS [8], MMTK [9], and VMD [10] provide efficient implementations of various standard analysis methods. Although powerful, such tools are generally limited in scope to all-atom simulations, particularly biomolecular simulations. This focus is manifested not only through the features these tools provide, but also in their general design philosophies.

Perhaps the most pronounced characteristic of such tools is a strong emphasis on trajectory management, which includes parsing trajectory files and supporting extensive topology selection features to enable, for instance, selecting all residues or atoms in a protein backbone. Although such tools are crucial for working with topologies in atomistic simulations, they are frequently cumbersome for working with coarse-grained simulation data where the trivial selection (all particles in the system) is the most common selection for various analyses. Moreover, such topology selection tools make assumptions that are inappropriate for non-atomistic systems: “bonding” in colloidal systems, for instance, is typically based on whether two particles are found to be in the same neighborhood by some distance-based metric, not by the presence of a true chemical bond. Since such determination of nearest neighbors is highly dynamic and parameter-dependent, it must be calculated on-the-fly and cannot be stored in a trajectory.

Another inconvenient but almost universal implementation choice is to directly tie analysis methods to trajectories by writing code that acts directly on some in-memory representation of a trajectory. This direct linkage is generally inflexible because it inhibits pre-processing of the data before running the analysis, which is often crucial to analyzing more specialized systems. More importantly, existing tools emphasize implementations of highly specific analyses involving, for instance, hydrogen bonding and protein secondary structure (using, e.g., DSSP [11]), which are far less useful for analyzing non-biomolecular systems. The predominant analyses of coarse-grained, colloidal-scale, or nanoparticle simulations usually involve measurements like numbers of nearest neighbors, diffraction patterns, or bond-orientational order parameters. These analyses bear little relation to the analyses performed for atomistic systems. These considerations suggest a need for a different type of analysis package that offers different methods than most existing tools.

In this paper we introduce `freud`, an open-source simulation analysis toolkit that addresses these needs. All inputs to and outputs from `freud` are numerical arrays of data, and the package makes no reference to predefined notions of atoms or molecules. As a result, `freud` can analyze particle-based data from both experiments and simulations regardless of the specific tools, methods, or software that were used to generate it. The package provides a Python Application Programming Interface (API) for accessing fast methods implemented in C++, and it implements numerous specific methods such as radial distribution functions and correlation functions that are common in the field of soft-matter physics (see Fig. 1). Prior works have used `freud` for: determining spatial correlation functions and potentials of mean force and torque (PMFTs) in two dimensions [1]; calculating Steinhardt order parameters for identifying solid-like particles [12,13]; computing spherical harmonics for machine learning on crystal structures [14]; optimizing pair potentials for designing complex crystals [15]; calculating strain fields by finding neighbors of particles against a uniform grid [16]; finding PMFTs in depletion-mediated self-assembly of hard cuboctahedra [17]; measuring rotational degrees of freedom in entropically ordered systems [18]; umbrella sampling of solid–solid phase transitions using Steinhardt order parameters [19]; evaluating PMFTs in analysis of two-dimensional shape allophiles [20]; and more. The `freud` library is designed to work well with coarse-grained particle models, such as those used in simulations of anisotropic nanoparticles, colloidal crystals, and polymers, and its methods are particularly useful for studies of phase transitions and critical phenomena in such systems. The package is likely to be of greatest interest to scientific communities in materials science, chemical engineering, and physics, though many of its analysis methods would be useful in generic particle systems. The `freud` library also integrates well into the scientific Python ecosystem, especially in data pipelines for machine learning and visualization [21].

The paper is organized as follows. We first address the core design principles that went into building `freud` in Section 2. Section 3 focuses more specifically on the details of the code, including information on class structures. Section 4 describes the various analysis methods in `freud` and details their uses. Finally, in Section 5 we provide some example code demonstrating the usage of `freud`.¹ The figures in this paper are rendered using Matplotlib [22] unless otherwise noted.

2. Design

Many of the best known tools for analyzing molecular simulations are built into either simulation toolkits (such as LAMMPS [23], GROMACS [24], or the `cpptraj` [25] plugin to Amber [26]) or visualization toolkits (such as VMD [10], PyMOL [27], or OVITO [28]). Although most of these have introduced varying degrees

¹ The code for these examples and many others is available at <https://github.com/glutzerlab/freud-examples> and in our online documentation.

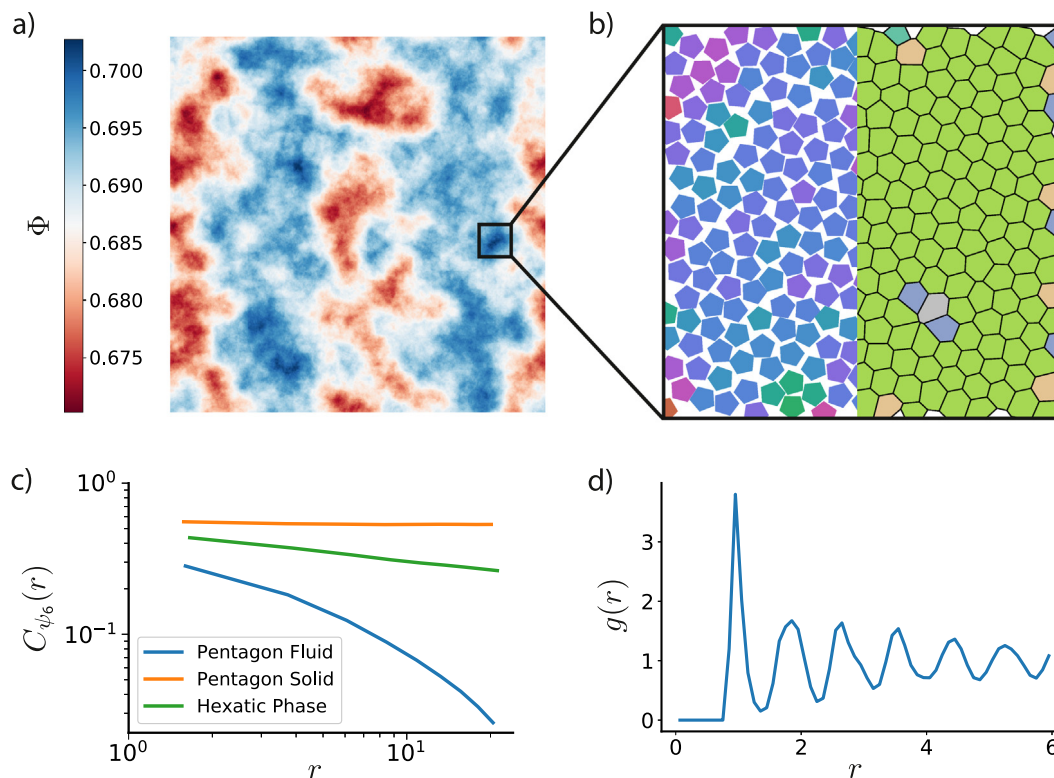


Fig. 1. The `freud` library is capable of computing a number of characteristics of a system of particles. Here, we demonstrate some of those features on a 2D Monte Carlo simulation of polygons that exhibits hexatic ordering [1]. a) Phase separation is clearly evident in this system of 512² pentagons colored by local density Φ ; the system is divided into denser (blue) and less dense (red) regions. b) Zooming into a particularly dense region shows that the hexatic ordering (left half) is generally uniform across the region. The Voronoi diagram of the system (right half) is also largely defect-free, with just a few pentagons having more or fewer than 6 nearest neighbors. c) The spatial correlation of the hexatic order parameter $C_{\psi_6}(r)$ is nearly constant for a nearly perfect crystal of pentagons (orange), whereas it decays very quickly in a fluid (blue). For a comparable system of hexagons, however, we see a power-law decay (green) in the hexatic order parameter due to the presence of a hexatic phase between the solid and fluid phases. d) The radial distribution function $g(r)$ for the system of pentagons shows the expected sequence of neighbor shells as a function of distance. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

of scripting support over the years, the analyses built into simulation toolkits are primarily focused on performing one-shot analyses on trajectory files directly from the command line. The visualization toolkits tend to have more full-featured scripting interfaces, but they are frequently difficult (if not impossible) to use outside their own sandboxed environments, complicating or even prohibiting integration with other tools. More recently, many newer tools such as MDTraj [6], MDAAnalysis [7], LOOS [8], and Pterios [29] have aimed to decouple analysis from simulation and visualization, making scriptability a primary focus to increase flexibility. Among such tools, Python is the most common language of choice due to its ease of use and the fact that it can be naturally extended with high performance languages like C, C++, and FORTRAN.

`freud` follows in the footsteps of these tools, providing a full-featured Python API to access all of its routines. However, while most other tools focus on calculating properties involving molecular topologies, `freud` is fundamentally designed for analyzing the local neighborhoods of particles, particularly where such local analyses provide global insight about the system. Such analyses are typically far more varied and system-dependent than the standard analyses of molecular topologies and therefore require more flexible tools. To meet this need, `freud` eschews any form of trajectory object encoding system topology and is instead designed such that each analysis method is an independent Python class that performs computations directly on NumPy arrays [30] of data.

This design makes it possible to use a much wider range of data with `freud` than is possible with tools that are tied to simulation trajectories. For instance, calculating Voronoi diagrams and

computing spatial correlation functions with `freud` is possible for essentially arbitrary spatial data, not just the result of a molecular simulation. Another major benefit is that NumPy arrays (the de facto standard for numerical data in Python) can be easily passed between multiple tools, making `freud` equally easy to use for one-off analyses or as part of a larger analysis pipeline involving many steps and using various software packages. As a result, `freud` is a much more flexible choice both for analyzing disparate sources of data and for incorporating into Python workflows. For example, most of `freud`'s analyses can be used within the OVITO visualization environment for real-time visualization with almost no noticeable performance cost.

Producing such array data from simulation trajectories for input to `freud` is straightforward because high quality file parsers with Python APIs already exist for all common trajectory file formats. Through integration with tools like MDAAnalysis, GSD [31], and garnett [32], `freud` can be used with data from over 25 different file formats, including common formats like DCD, XTC, and TRJ files. `freud` integrates with the trajectory objects produced by many of these tools, but if necessary, users can read trajectories into arrays and modify them before passing the data to `freud` for analysis. By using data read by other tools, `freud`'s analyses can be made aware of molecular topology if needed, but only when the analysis method requires such information. Similarly, since the outputs of `freud`'s analyses are also NumPy arrays, they can be passed to almost any tool in the scientific Python software stack. For example, constructing a Pandas [33] DataFrame from the outputs of any `freud` analysis requires just one line of code, and it immediately enables writing the output to text, CSV, or HDF5, or saving into an SQL database.

Beyond differences in trajectory and data handling, the most significant design choice in `freud` stems from the most common pattern followed by its many analysis methods. Since the first task in characterizing local neighborhoods is often the identification of neighboring particles, `freud` provides efficient methods for finding neighbors in arbitrary system geometries. The nearest-neighbor finding routines are designed to be as fast and flexible as possible, supporting various algorithms optimized for different system configurations and offering different criteria for neighbor selection. In general, queries can be based on either a cutoff distance or a desired number of nearest neighbors. These tools are optimized to provide cheap access to neighbors even in highly performance-critical loops in C++ analysis routines, but the package's system box representation and neighbor finding tools also have Python APIs, so users can implement custom analyses directly in Python (for an example, see Section 5.4).

The analysis methods in `freud` are essentially independent tools that make use of these objects to efficiently perform various calculations. These features are all presented with a common API, easing the transition between the different types of analyses needed for different simulations. All methods in `freud` are accelerated through extensive parallelization.

3. Implementation

The `freud` package is entirely object-oriented, with two core C++ classes: the `Box` class, which encapsulates all logic associated with periodicity in arbitrary triclinic boxes (boxes with 3 linearly independent basis vectors); and the `NeighborQuery` class, which facilitates efficiently finding, storing, and iterating over nearest neighbors. In keeping with the Python ethos, box objects in `freud` may be constructed from a variety of inputs. Any method in `freud` that accepts a box object also accepts a number of objects that can be interpreted as a box, such as a 3×3 NumPy array of box vectors or a list of three numbers representing the edge lengths of an orthorhombic box. There are two subclasses of `NeighborQuery` in `freud` that each implement different neighbor search algorithms: one implements a bounding volume hierarchy (BVH) [34], while the other implements a cell list [35]. The `NeighborList` class is a lightweight storage mechanism for `NeighborQuery` results that accelerates performing multiple analyses on the same set of neighbor pairs.

The analysis methods in `freud` are encapsulated by *Compute* classes, which are loosely defined as classes providing a compute method that populates class attributes after performing some computation. Compute classes, such as the density module's RDF class, are usually configured with constructor arguments, after which they can be used multiple times to perform distinct calculations. Some classes in `freud` (e.g. the RDF, PMFT, or bond-orientational order diagram) represent histogram-like quantities, and therefore allow the user to specify `reset=False` as an argument to compute in order to accumulate and average data over many calls.

Compute classes can be divided into two groups, those that depend on finding neighbors and those that do not. A majority of calculations in `freud` require neighbors, and the compute methods of such classes all share two arguments, `system` and `neighbors` (in addition to analysis-specific arguments like particle orientations for PMFTs; such arguments are also typically NumPy arrays). The `system` parameter accepts a `NeighborQuery` or any object that can be interpreted as a tuple (`box`, `points`), where the `box` is any valid box-like object (as described above) and the `points` argument is anything that can be interpreted as an $N \times 3$ NumPy array of positions. Valid systems include simulation frame objects from tools such as MD-Analysis, GSD, garnett, OVITO, or the particle simulation engine HOOMD-blue [36–38].

When performance is critical, providing a `NeighborQuery` object is advantageous because many compute methods can reuse these neighbor search data structures. For all other system inputs, `freud` internally constructs a `NeighborQuery` if the compute method requires neighbor pairs. The `neighbors` argument is a dictionary of query arguments, such as `dict(num_neighbors=12)` or `dict(r_max=3.0)` (the complete specification for `freud`'s Query API is provided in the documentation). Alternatively, users may precompute a `NeighborList` and provide it as the `neighbors`. In this case, whether `system` is a `NeighborQuery` or not has no impact on performance because the calculation will be carried out directly on the provided set of neighbor pairs and no additional spatial searches are required. Fig. 2 shows a flowchart demonstrating how these classes and data structures are used.

Some methods in `freud` do not operate on neighboring pairs of particles. For those that still depend on particle positions (such as `GaussianDensity`), the first argument is still any valid system, but no neighbors are provided. Some methods do not depend on positions at all; for instance, the `Nematic` order parameter only requires particle orientations. In such cases, the user can simply pass that quantity alone to the calculation. This mode of operation is particularly useful when performing high-throughput analysis of large files; using file formats like GSD that permit reading only certain properties of the trajectory, users can minimize I/O operations by only reading the required arrays from memory.

All Compute classes use efficient, thread-parallel C++ implementations for performance-critical components. The Python bindings for these C++ classes are generated using Cython [39], and the C++ methods are mirrored in Python using thin Cython classes that dispatch calls to the underlying C++ class instances. The Cython classes have limited responsibilities: managing the memory of the underlying C++ instances, sanitizing inputs when necessary, and providing transparent access via memory views on C++ arrays.

The main exception to this design is the `msd` module, which is implemented in pure Python in `freud`. The mean squared displacement (MSD) is a measure of, on average, how far particles move in a given window of time. In a simulation trajectory of N_f frames, the MSD of particle i over a window of length m frames is given by:

$$MSD(i, m) = \frac{1}{N_f - m} \sum_{k=0}^{N_f - m - 1} \|\vec{r}_i(k + m) - \vec{r}_i(k)\|^2 \quad (1)$$

Therefore, the total MSD is given by:

$$MSD(m) = \frac{1}{N_p} \sum_{i=1}^{N_p} MSD(i, m) \quad (2)$$

Direct computation of the MSD is an $\mathcal{O}(N_p N_f^2)$ operation, but by using a fast Fourier transform (FFT) this cost can be reduced to $\mathcal{O}(N_p N_f \log(N_f))$ [40]. When using this approach, the FFT is responsible for most of the computation time, and since packages like NumPy [30] and SciPy [41] already expose fast C and FORTRAN FFT routines to Python, `freud` simply leverages them directly and implements the rest of the MSD in pure Python.

Calculations in `freud` are generally parallelized over *particles* (e.g. the `Nematic` order parameter class) or over *pairs of particles* (e.g. computing inter-particle distances with the RDF class). Both the number of particles and the number of particle-particle pairs increase with system size, ensuring that the work is load-balanced well among threads because the number of threads

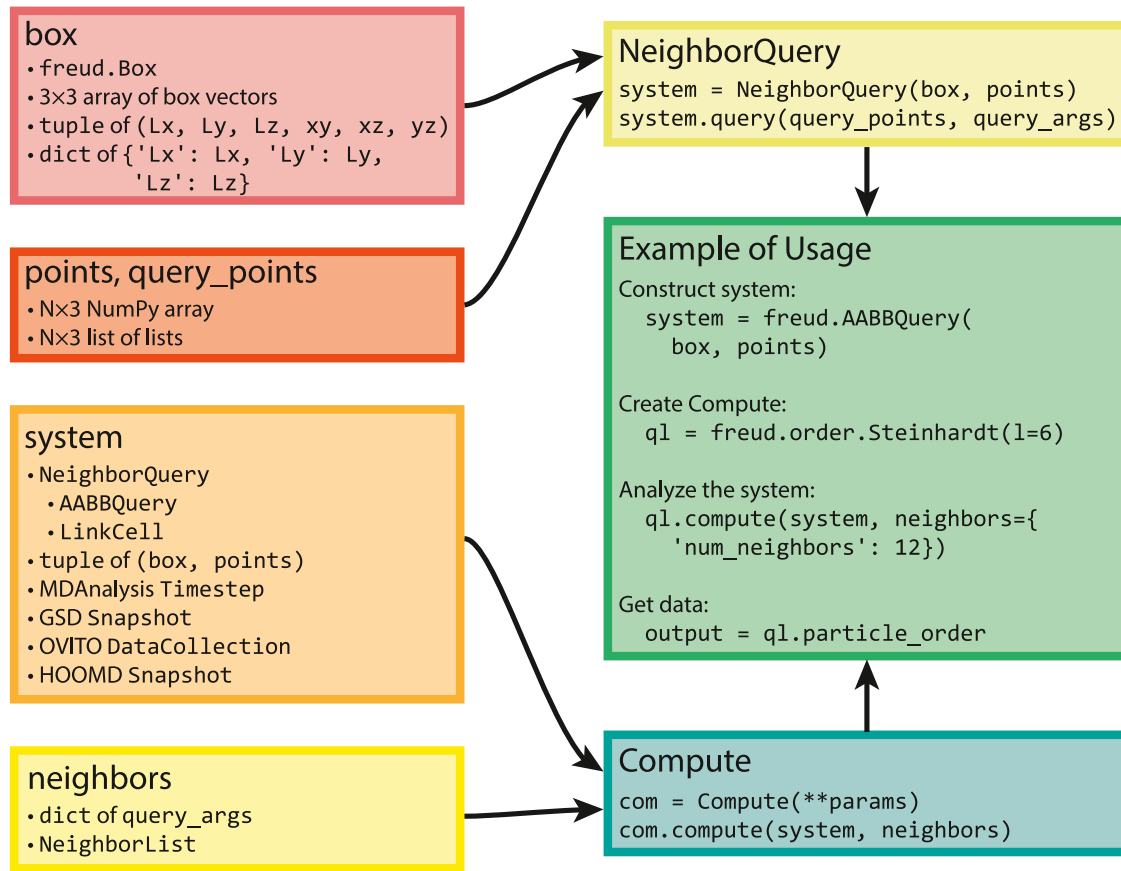


Fig. 2. Here we show the flow of various types of inputs into `freud`. Boxes can be constructed based on a variety of inputs, all of which can also directly be provided anywhere a box object is required. Similarly, any object that can be interpreted as an $N \times 3$ array can be provided where particle positions are required. Any valid pair of box and points can be used to construct a `NeighborQuery` object, which is one of the types of systems that `freud` accepts. In addition to a `NeighborQuery`, `freud` can also interpret raw tuples of boxes and points as system objects, or use simulation frames from numerous external tools (a subset are shown in the figure). Any computation that involves finding nearest neighbors also requires a specification of neighbors, which can be a `NeighborList` or a dictionary of query arguments. The “Example of Usage” box on the right shows a typical use case of `freud` that combines these concepts.

is much less than the number of particles or pairs. Parallelism in `freud` is accomplished using Intel Threading Building Blocks (TBB) [42]. Analysis routines are written as lambda functions operating on a particle or a pair of particles; `freud` provides wrappers that then automatically parallelize these functions appropriately using TBB. Modern compilers aggressively inline such lambda functions, thereby optimizing away any additional cost that could arise from the extra function calls. `freud` uses thread-local storage extensively to avoid any parallel writes to data containers. For histograms that accumulate over many frames of simulation data, `freud` performs reduction over thread-local containers lazily.

Currently, `freud` is at version 2.1.0 and supports Python versions 3.5.0 or greater. The package is distributed through the Python Package Index (PyPI) and the `conda-forge` channel of the Anaconda package manager [43], making it easy to install on any Unix-based operating system (e.g. Linux or macOS). Builds for the Windows operating system are also available on `conda-forge`. `freud` depends on NumPy and TBB libraries, which are automatically installed with `freud`. The library can also be compiled from source using a C++11 compliant compiler. Compilation requires NumPy and TBB headers as well as a Cython installation. Code documentation is written using Google-style docstrings rendered using Sphinx and hosted on ReadTheDocs. The `freud` library is released open source under the BSD 3-Clause License, and the source code is available in a GitHub repository [44]. Continuous integration testing is performed using CircleCI.

4. Features

4.1. General utilities

The general utilities in `freud` are contained in two modules: `box` and `locality`. The `box` module contains the core `Box` class. The `locality` module contains the `NeighborQuery` abstract class, which defines the standardized query API. `NeighborQuery` results (neighboring particle pairs) can be obtained dynamically or stored in the `NeighborList` class provided by the `locality` module.

Box periodicity is built in at the lowest level of the `NeighborQuery` subclasses, which are highly optimized for this use case. The `AABBQuery` subclass implements a tree data structure of Axis-Aligned Bounding Boxes (AABBs), a type of BVH which greatly accelerates the process of finding particles' neighbors [13, 34]. A second approach is implemented in the `LinkCell` subclass [35], which uses linked cell lists to find particle neighbors. Both of these classes can find neighboring particle pairs based on a distance cutoff or a desired number of neighbors, and both were adapted from HOOMD-blue.

As a performance benchmark, we compare `freud`'s `AABBQuery` class with the `cKDTree` implementation in SciPy [41]. As part of SciPy, this implementation is the most readily available alternative to `AABBQuery`. Fig. 3 shows that `freud`'s `AABBQuery` routines clearly outperform the `cKDTree` as system sizes increase to thousands of points. Moreover, we note that while `freud` supports general triclinic boxes, SciPy's `cKDTree` only supports

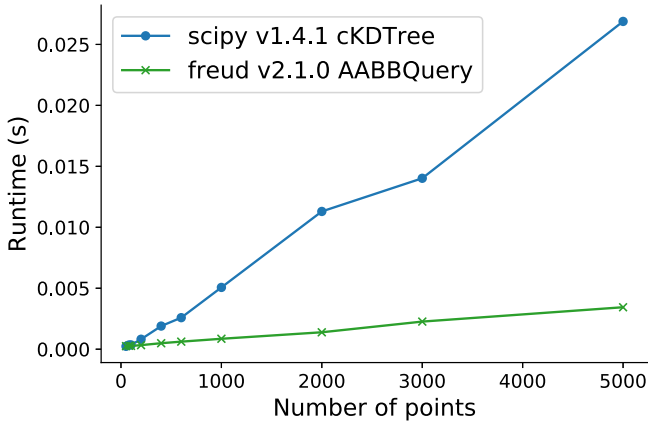


Fig. 3. Here we benchmark the AABQuery implementation in *freud* against the cKDTree implementation in SciPy. We construct randomly generated sets of points such that each particle would have, on average, 12 neighbors within a distance of 1. We then measure the performance of finding all neighbors within this distance using both SciPy's cKDTree and *freud*'s AABQuery. The benchmarks were performed on a system with an Intel® Xeon® CPU E5-2680 v2 @ 2.80 GHz. The AABQuery implementation in *freud* scales much better than SciPy's cKDTree for larger system sizes. We do not report error bars due to the extremely low variance in the data. The exact details are available at <https://github.com/gloterlab/freud-examples>.

periodic orthorhombic boxes (i.e. a cuboid, a rectangular prism where all angles are right angles).

In addition to these performance gains, the *NeighborQuery* objects in *freud* are designed to interoperate seamlessly with analysis routines. Since analyses in *freud* are written in C++, using a Python API to find neighbors and then pass them into other C++ routines would require storing an intermediate representation of the neighbors. Furthermore, while a Python API should make certain promises, such as sorting the resulting neighbors, the analyses using neighbors simply loop over all pairs and therefore do not require such ordering. To avoid these costs, the features of the *NeighborQuery* classes are directly accessible in C++ in the form of iterators that lazily produce neighbors. In practice, using the *NeighborQuery* classes in this manner speeds up computations by a factor of two or more depending on the system size. To make use of these iterators, developers implement analysis methods as lambda functions that are passed as arguments to *freud*'s internal TBB wrappers that apply these functions to neighbor pairs in parallel.

The final feature of the *locality* module is the *Voronoi* class, which uses the *voro++* library [45] to generate Voronoi diagrams for systems of particles. Voronoi diagrams are a standard method for characterizing the local geometric arrangements in the system, and they also provide a parameter-free method for defining nearest-neighbor relationships [46]. The *Voronoi* class produces a *NeighborList* object that can then be used as the *neighbors* argument for other compute classes.

4.2. Analysis modules

The remaining modules in *freud* are independent of one another and contain groups of classes that implement related features. While some of *freud*'s features are unique, many others are standard techniques. However, implementations of these methods commonly lack support for periodicity. For example, the SciPy library [41] has functions for computing Voronoi diagrams and correlation functions, but these are restricted to aperiodic systems.

The *cluster* module of *freud* can be used to find clusters of particles – where cluster membership is defined by neighbor

bonds – and then compute properties of these clusters such as gyration tensors. The *density* module contains features for calculating radial distribution functions as well as spatial correlation functions of arbitrary quantities. Additionally, the *density* module can estimate local particle density and interpolate particle density onto a regular grid suitable for, e.g., computing discrete Fourier transforms for diffraction patterns. The *interface* module provides a quick tool for identifying interfaces between two mutually exclusive sets of points (e.g. a solid and a liquid phase). The *msd* module enables the calculation of mean squared displacements of particles over the course of a trajectory.

The *order* module is the most extensive one in *freud*, containing a large number of different order parameters commonly used to measure ordering and identify phase transitions in crystalline systems. Of particular note are the bond-orientational order parameters Q_l and W_l [48] and the cubic order parameter [49] (see Fig. 4). The module also contains the nematic order parameter for identifying orientationally ordered, translationally disordered phases, as well as a solid-liquid order parameter for identifying generic ordered phases [50].

The other features of *freud* are analysis methods developed by researchers in our group and not yet implemented anywhere else. In particular, the *pmft* and *environment* modules implement features unique to *freud* that we now discuss in greater detail.

4.3. Potentials of mean force and torque

The potential of mean force and torque (PMFT) is a generalization of the classical potential of mean force (PMF) that was recently developed to quantify directional entropic forces that emerge in crowded systems [51,52]. Given the canonical partition function as a function of particle positions $\{q\}$ and particle orientations $\{Q\}$, Ref. [52] derives the PMFT by separating out a component corresponding to the relative coordinates of a pair of particles $\Delta\xi_{12}$:

$$Z = \int d\Delta\xi_{12} J(\Delta\xi_{12}) e^{-\beta U(\Delta\xi_{12})} \int [d\tilde{q}][d\tilde{Q}] e^{-\beta U(\{\tilde{q}\}, \{\tilde{Q}\}, \Delta\xi_{12})} \quad (3)$$

$$= \int d\Delta\xi_{12} J(\Delta\xi_{12}) e^{-\beta U(\Delta\xi_{12})} e^{-\beta \tilde{F}_{12}(\Delta\xi_{12})} \quad (4)$$

where J is the Jacobian transforming to the local coordinate system and \tilde{F}_{12} is the free energy of the other particles, which have been integrated over in Eq. (4). The PMFT F_{12} is defined by the relation

$$Z \equiv \int d\Delta\xi_{12} e^{-\beta F_{12}(\Delta\xi_{12})} \quad (5)$$

Combining Eqs. (4) and (5) gives an expression for the PMFT

$$\beta F_{12}(\Delta\xi_{12}) = \beta U(\Delta\xi_{12}) - \log J(\Delta\xi_{12}) + \beta \tilde{F}_{12}(\Delta\xi_{12}) \quad (6)$$

In hard particle systems governed exclusively by excluded volume interactions, the potential energy term becomes an infinite Heaviside function H and the PMFT can be simplified to

$$F_{12}(\Delta\xi_{12}) = -k_B T \log(H(d(\Delta\xi_{12}))J(\Delta\xi_{12})) + \tilde{F}_{12}(\Delta\xi_{12}) \quad (7)$$

To contextualize the PMFT, we note that if in Eq. (3) we redefine $\Delta\xi_{12}$ to only include the center-to-center distance of the pair of particles and otherwise follow the same steps, the resulting potential F_{12} reduces to the classical PMF with the usual RDF relation $g(r) = e^{-\beta F_{12}(r)}$. This suggests that although the PMFT is a function of all degrees of freedom required to characterize the relative configuration of a pair of particles, examining a more limited coordinate system can still be informative. Fig. 5 shows two examples of PMFTs that contain more information than a PMF without containing all available degrees of freedom. In the

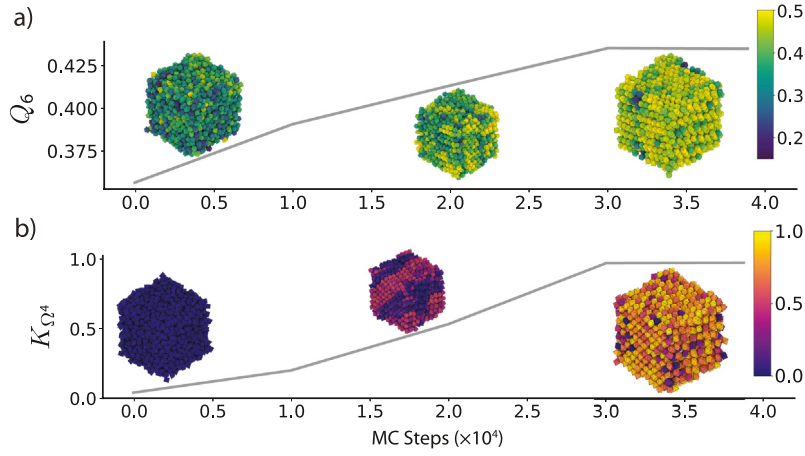


Fig. 4. Various order parameters can be used to characterize the degree of ordering in a system. The per-particle order parameter values eventually converge to a uniform global value as the system becomes globally well-ordered. These plots show the evolution of two order parameters over the course of a Monte Carlo simulation of hard particles, which over time rearrange into an ordered phase under compression. Simulation snapshots are colored by the per-particle order parameter and rendered with *fresnel* [47]. a) The Steinhardt Q_6 order parameter is an appropriate scalar descriptor for systems forming a BCC (cI2-W) structure. Systems of cuboctahedra in the fluid phase show a distinctly different characteristic value of the order parameter than in the solid phase. b) The cubic order parameter K_{Q^4} is useful for characterizing ordering in these systems of octahedra.

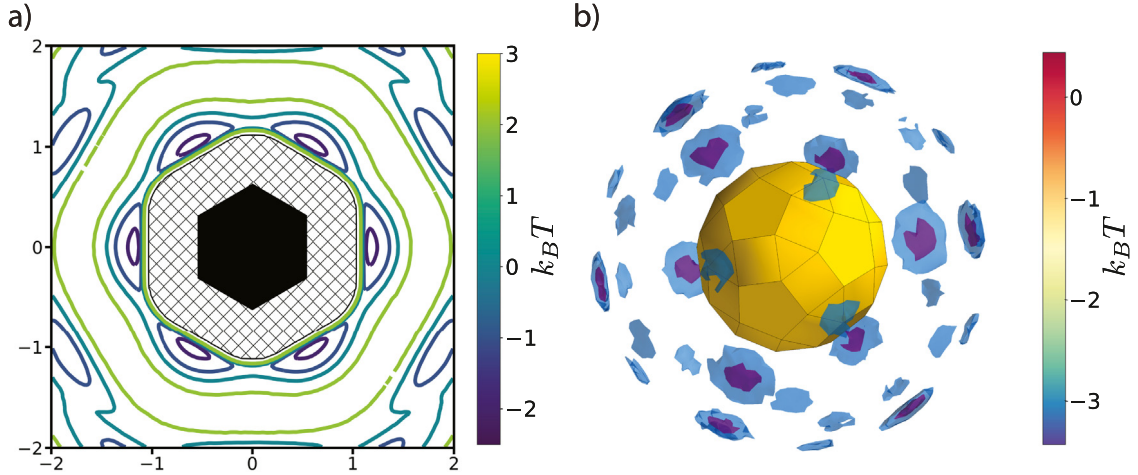


Fig. 5. The PMFT is related to the probability of finding particles at a given position and orientation relative to one another. a) The PMFT of an ordered system of hexagons [20], where the locations of the wells indicate that particles are much more likely to sit next to the edges of their neighbors than the corners. In two-dimensional systems, the full PMFT is 3-dimensional, since it also must account for the orientation of the second particle relative to the first; for clarity, in this figure we have integrated out that degree of freedom. b) A PMFT computed from a system of rhombicosidodecahedra shows distributions of neighboring particles in three dimensions (figure rendered using Mayavi [53]). There are six degrees of freedom in 3D systems, three translational and three rotational. This PMFT only shows the three translational degrees of freedom. The wells representing the deepest energy isosurfaces of the PMFT align with the largest (pentagonal) facets of the polyhedron.

2D PMFT in the left panel, the orientation of the second particle is ignored, but its angular position relative to the reference particle is sufficient to illustrate the clear preference for facet-to-facet alignment. Similarly, the right panel ignores the orientation of the second particle (which encodes three degrees of freedom in 3D, as represented by e.g. Euler angles), but once again the preference for facet-to-facet alignment is clear. For an example of a case where analyzing the full, high-dimensional PMFT is necessary, see Ref. [54].

freud calculates the PMFT by accumulating a histogram of the configurations of all other particles and then taking the negative logarithm of the counts. PMFTs may be accumulated over many frames to generate smoother energy surfaces. This method of computing the PMFT is very similar to that of computing an RDF, so we compare their scaling behavior in a two-dimensional system in Fig. 6. The calculations scale almost identically to many threads, with a constant scaling factor between them. There are two components contributing to the absolute difference in their

performance: 1) the extra operations required to compute the orientation of the local coordinate system in the PMFT, and 2) the extra cost of binning in multiple dimensions.

In addition to the metric shown in Fig. 6, we also tested performance as a function of the parameters of these two methods, namely the maximum interparticle distance and the number of bins. In the former case, both methods show the expected quadratic growth, since the number of particles included in the calculation increases as the square of this cutoff distance in 2D. The behavior with respect to the number of bins is more interesting: this parameter has no effect on performance until it becomes sufficiently large, at which point performance begins to degrade. This degradation can be understood as the result of two things: 1) poor cache performance as the histograms become too large to fit in memory, and 2) increasing costs of reduction, which can eventually affect performance. Since the PMFT shown is a two-dimensional histogram, the number of bins that can be used along each dimension before experiencing this performance drop

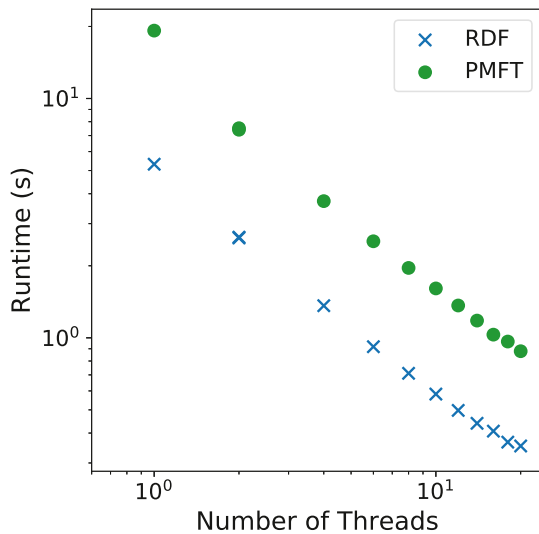


Fig. 6. This benchmark compares the performance of the 2D PMFT to that of an RDF on the same two-dimensional system of hexagons used in Fig. 5. This computation was performed with a randomly generated trajectory of consisting of 10 frames of 20000 particles. Particle positions were constructed such that each particle would have, on average, 12 neighbors within a distance of 1. The benchmarks were performed on a system with an Intel® Xeon® CPU E5-2680 v2 @ 2.80 GHz. Both methods have essentially the same performance characteristics, with the PMFT approximately three times slower than the RDF. We do not report error bars due to the extremely low variance in the data.

is commensurately smaller than can be used with the RDF; this effect would be even more pronounced for a three-dimensional PMFT.

4.4. Local environments

The `environment` module provides methods for characterizing the local environments of particles that we now illustrate in greater detail.

4.4.1. Bond-orientational order diagrams

The `BondOrder` class enables the calculation of bond-orientational order diagrams (BOODs) [55–59]. Inspired by the bond-orientational order parameters defined by Steinhardt et al. [48], BOODs characterize the local ordering of systems by calculating the vectors between all neighboring particles in a system and then projecting these vectors onto a sphere. One example of how BOODs can be used is to identify n -fold ordering in a system; in simple crystal structures with n -fold coordination, the BOOD will show n peaks corresponding to the average location of nearest neighbors.

In addition to the standard BOOD calculation, the class offers some additional modes of operation that can be useful in specific cases. One mode involves finding the positions of nearest neighbors in the local coordinate system of a given particle rather than the global coordinate system, which can prevent misidentifying systems with multiple grains [55]. Another mode modifies the BOOD to help identify plastic crystals, which appear crystalline due to having translational order but lack orientational ordering. In this mode, the positions of the nearest neighbors of each particle are modified by the relative orientations of the neighboring particles, creating a BOOD in which positional ordering will no longer appear except when orientational ordering is also present.

4.4.2. Spherical harmonic descriptors

The BOOD is closely related to the Steinhardt order parameters Q_l and W_l , which measure rotational order in a system

using spherical harmonics [48]. While the BOOD is essentially a histogram of nearest-neighbor bonds, the Steinhardt order parameters take this one step further, measuring l -fold order by constructing scalar quantities from rotationally invariant combinations of spherical harmonics of degree l calculated from the locations of nearest-neighbor bonds. However, spherical harmonic representations can also be used in a variety of different ways. For example, distinguishing different grains of the same crystal structure could be done using descriptors that are not rotationally invariant. Alternatively, we can often obtain rotationally-invariant descriptions of local environments for crystal structure identification via the principal axes of the moment of inertia tensor of the environments, or by using particle orientations of anisotropic particles [14,49,60]. To support such spherical harmonic analyses, the `LocalDescriptors` class in `freud` computes spherical harmonics characterizing particle neighborhoods. These harmonics can then be combined in arbitrary ways to generate custom descriptors of local particle environments. Such descriptors have proven useful in identifying multiple complex crystals (see Fig. 7). One method for identifying these structures is to use the information contained in this array of spherical harmonics as a set of per-particle features in an artificial neural network (ANN) for structure classification [14].

4.4.3. Environment matching

Methods like the spherical harmonic descriptors and the BOOD characterize ordering in systems by calculating system-averaged quantities from neighbor bonds. The `EnvironmentCluster` class takes a different approach by defining environments according to the nearest neighbors of each particle and performing point set registration to identify and cluster similar environments [61]. This type of analysis is particularly useful because it emphasizes local information for each particle. As a result, it can be used for tasks such as identifying different Wyckoff positions in a crystal. The complementary `EnvironmentMotifMatch` class can be used to match clusters to specific motifs, allowing deeper analysis of a given structural motif.

Since this method performs a direct pairwise comparison of all motifs, it is substantially more expensive than common methods used for structure identification. For instance, the performance is at least an order of magnitude slower than the implementations of Polyhedral Template Matching and Common Neighbor Analysis in OVITO. Unlike these methods, however, the environment matching algorithm does not depend on previous knowledge of possible structures, and instead infers possible structures entirely from the local motifs present in a system. Moreover, it can be tuned much more finely than the other methods, allowing not only the identification of crystal structures present, but also the precise identification of stacking faults like those found in Fig. 8. As a result, it is a good complement to existing methods for identifying crystal structures in various systems.

4.4.4. Angular separation

The `AngularSeparation` classes provide a way to characterize typical particle orientations in a system. The `AngularSeparationGlobal` class allows comparison of particle orientations to a set of reference orientations, which can be used to characterize orientational order relative to the reference input. This metric can be used as an order parameter for measuring orientational disorder in plastic crystals, which exhibit translational order and orientational disorder [62]. Alternatively, the `AngularSeparationNeighbor` computes minimum separation angles between neighboring particles, allowing a more fine-grained analysis of the orientational ordering in local motifs. Both of these methods account for symmetry by accepting an array of equivalent

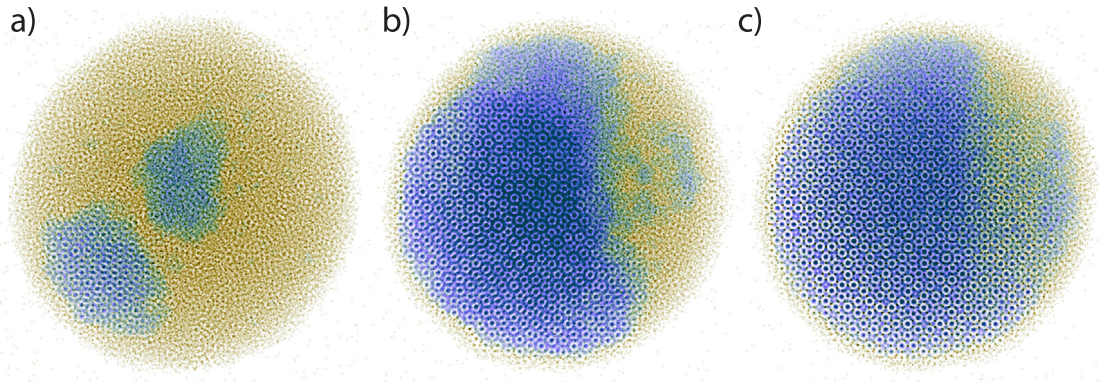


Fig. 7. Spherical harmonic descriptors can be used to identify the nucleation and growth of *tP30*-CrFe (Frank-Kasper σ phase). a–c) As time progresses, crystallites nucleate and grow. Solid-like particles (blue) are identified via a feedforward artificial neural network using spherical harmonic descriptors (described in more detail in [14]).

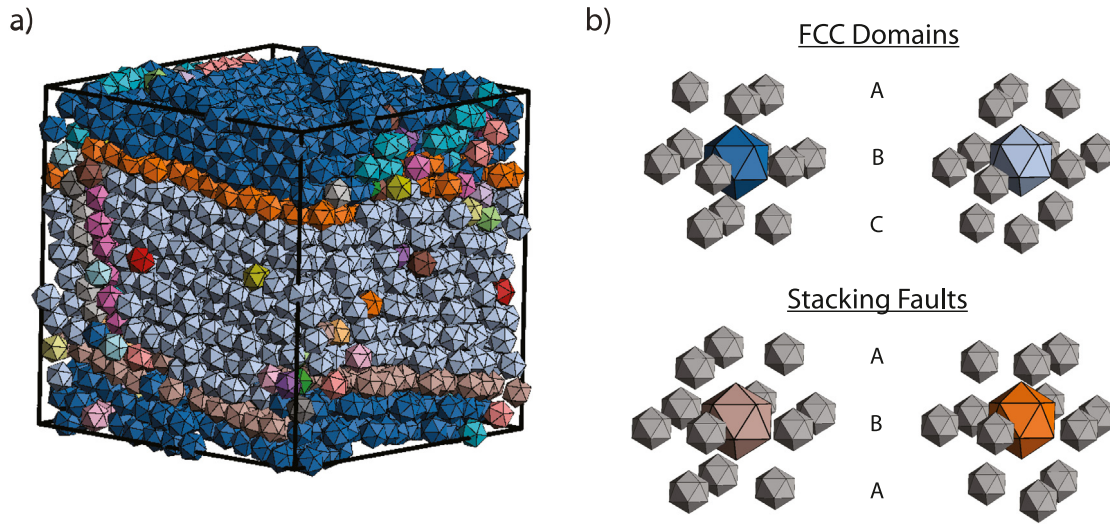


Fig. 8. Environment matching allows us to detect variations in the local environments of particles. a) The presence of grain boundaries in this system (rendered with *fresnel*) is clearly visible due to the different coloring according to local environments. b) The two distinct domains (blue and gray particles) are clustered separately, but we can see that they both exhibit FCC-like (*cF4*-Cu) ordering in their stacking pattern (upper-right). The environment matching method can also detect the different environments of the stacking faults themselves (mauve and orange), which exhibit an ABA stacking pattern instead of the expected ABC pattern (lower-right).

quaternions corresponding to all symmetry-preserving transformations of the particle (i.e. the particle's point group).

4.5. Data generation and plotting

For the purposes of teaching via code examples and testing *freud*'s analyses, *freud* includes the *freud.data* module. It includes the *UnitCell* class for representing arbitrary unit cells with user-provided box vectors and basis positions. The *UnitCell* class includes class methods that generate common crystal structures like face-centered cubic, body-centered cubic, and simple cubic. The data module also includes a method for generating a random system with uniformly distributed points in a periodic box.

Many analysis modules in *freud* implement a *plot()* method, which can be used for rapid data visualization. The *Compute* classes (e.g. instances of *freud.density.RDF*) also define a *_repr_png_()* method that allows their data to be automatically plotted in IPython environments (such as Jupyter notebooks) using Matplotlib [22], when the last line in a code cell returns that analysis object.

5. Examples

In this section, we demonstrate the use of *freud* in conjunction with the broader scientific software ecosystem. The code for these examples and many others is available at <https://github.com/glotzerlab/freud-examples>.

5.1. Computing an RDF and an MSD from a LAMMPS simulation

Here, we consider the problem of calculating the RDF and the MSD of a system simulated using LAMMPS (version 5 Jun 2019) [23]. LAMMPS is a standard tool for particle simulation used in many fields, and it supports multiple output formats, including those used by other simulation codes (e.g., the DCD format from CHARMM [63] and the XTC format from GROMACS [24]). In this case, we demonstrate the case of using the output of a custom dump format in LAMMPS, which allows users to dump selected quantities into a text file. Although the default XYZ file format lacks sufficient information to calculate an MSD, the necessary particle image information can be included as shown.

```

import numpy as np
import freud

# For the MSD we also need images, which can be dumped
# using the LAMMPS dump custom command as follows:
# dump 2 all custom 100 output_custom.xyz x y z ix iy iz

# We read the number of particles, the system box, and the
# particle positions into 3 separate arrays.
N = int(np.genfromtxt(
    'output_custom.xyz', skip_header=3, max_rows=1))
box_data = np.genfromtxt(
    'output_custom.xyz', skip_header=5, max_rows=3)
data = np.genfromtxt(
    'output_custom.xyz', skip_header=9,
    invalid_raise=False)

# Remove the unwanted text rows
data = data[~np.isnan(data).all(axis=1)].reshape(-1, N, 6)

box = freud.box.Box.from_box(
    box_data[:, 1] - box_data[:, 0])

# We shift the system by half the box lengths to match the
# freud coordinate system, which is centered at the origin.
# Since all methods support periodicity, this shift is
# simply for consistency but does not affect any analyses.
data[:, :, :3] -= box.L/2
rdf = freud.density.RDF(bins=100, r_max=4, r_min=1)
for frame in data:
    rdf.compute(system=(box, frame[:, :3]), reset=False)

msd = freud.msd.MSD(box)
msd.compute(
    positions=data[:, :, :3], images=data[:, :, 3:])

# The object contains all the data we need to plot the RDF.
from matplotlib import pyplot as plt
plt.plot(rdf.bin_centers, rdf.rdf)
plt.title('Radial Distribution Function')
plt.xlabel('$r$')
plt.ylabel('$g(r)$')
plt.show()

```

If our trajectory was stored in a DCD file, we could modify our code above to read the input data using MDAnalysis (version 0.20.1):

```

reader = MDAnalysis.coordinates.DCD.DCDReader(
    'output.dcd')
rdf = freud.density.RDF(bins=100, r_max=4, r_min=1)
for frame in reader:
    rdf.compute(system=frame, reset=False)
...

```

5.2. On-the-fly analysis with HOOMD-blue

A major strength of freud is that it can also be used for on-the-fly analysis. For example, freud can be used to terminate a simulation based on some additional condition, or to log a quantity at a higher frequency than we want to save the full

system trajectory. In our previous example, we demonstrated the calculation of an RDF using freud. An RDF can be noisy when calculated with limited data, so we would like to average it over a large number of simulation frames; however, storing many frames can lead to unreasonably large simulation trajectory files. Using the simulation engine HOOMD-blue (v2.9.0), we can accumulate RDF data during a simulation without storing the entire output. Additionally, we show that we can log an order parameter over the course of the simulation:

```

import hoomd
from hoomd import hpmc
import freud
import numpy as np

hoomd.context.initialize()
system = hoomd.init.create_lattice(
    hoomd.lattice.sc(a=1), n=10)
mc = hpmc.integrate.sphere(seed=42, d=0.1, a=0.1)
mc.shape_param.set('A', diameter=0.5)

rdf = freud.density.RDF(bins=50, r_max=4)
q6 = freud.order.Steinhardt(l=6)

def calc_rdf(timestep):
    snap = system.take_snapshot()
    rdf.compute(system=snap, reset=False)

def calc_Q6(timestep):
    snap = system.take_snapshot()
    q6.compute(system=snap,
        neighbors={'num_neighbors': 12})
    return np.mean(q6.particle_order)

# Equilibrate the system before accumulating the RDF.
hoomd.run(1e4)
hoomd.analyze.callback(calc_rdf, period=100)

logger = hoomd.analyze.log(filename='output.log',
    quantities=['q6'],
    period=100,
    header_prefix='#',
    overwrite=True)

logger.register_callback('q6', calc_Q6)

hoomd.run(1e4)

# Store the computed RDF in a file.
np.savetxt('rdf.csv',
    np.vstack((rdf.bin_centers, rdf.rdf)).T,
    delimiter=',', header='r, g(r)')

```

5.3. Analyzing atomistic trajectories from GROMACS

As discussed in Sections 1 and 2, freud's design focus differs from that of many similar tools in the lack of focus on trajectory management. The example below is based on a simulation trajectory of water molecules in a box generated using GROMACS (version 2020) [24]. We use MDTraj (version 1.9.3) [6] to read in an XTC trajectory file and then compute an RDF of the oxygen atoms in the water molecules using freud. In the process,

we demonstrate how the sophisticated subsetting functionality offered by tools like MDTraj can be replicated with Python code, which is very useful when such subsets must be computed from coarse-grained trajectories with highly customized topology definitions that standard trajectory management tools cannot handle.

```
import mdtraj
import freud
import numpy as np

traj = mdtraj.load_xtc(
    'output/prd.xtc', top='output/prd.gro')
bins = 300
r_max = 1
r_min = 0.01

# Expression selection, a common feature of analysis tools
# for atomistic systems, can be used to identify all
# oxygen atoms.
oxygen_pairs = traj.top.select_pairs('name O', 'name O')

# We can directly use the above selection in freud.
oxygen_indices = traj.top.select('name O')

# Alternatively, we can subset directly using Python logic.
# Such selectors require the user to define the nature of
# the selection, but can be more precisely tailored to a
# specific system.
oxygen_indices = [atom.index for atom in traj.top.atoms
                  if atom.name == 'O']

freud_rdf = freud.density.RDF(
    bins=bins, r_min=r_min, r_max=r_max)
for system in zip(np.asarray(traj.unitcell_vectors),
                  traj.xyz[:, oxygen_indices, :]):
    freud_rdf.compute(system, reset=False)

# We can plot these RDFs to verify that they are equivalent.
from matplotlib import pyplot as plt
fig, ax = plt.subplots()
ax.plot(freud_rdf.bin_centers, freud_rdf.rdf, 'o',
        label='freud', alpha=0.5)
ax.plot(*mdtraj_rdf, 'x', label='mdtraj', alpha=0.5)
ax.set_xlabel('$r$')
ax.set_ylabel('$g(r)$')
ax.set_title('Radial Distribution Function')
ax.legend()
```

5.4. Common neighbor analysis

Common Neighbor Analysis (CNA) [64] is a standard technique for analyzing the local neighborhoods of particles in a crystal. The method involves a classification of local neighborhoods based on a number of features. Using freud's NeighborList, however, the method is straightforward to implement in Python.

We first consider the simpler problem of identifying all common neighbors between any pair of points. This is equivalent to searching for the **second-nearest** neighbor pairs, which can be done using freud as follows (note that this code is primarily written for clarity and could easily be optimized):

```
import freud
import numpy as np
from collections import defaultdict

# Use a face-centered cubic (fcc) system.
box, points = freud.data.UnitCell.fcc().generate_system(4)
aq = freud.AABBQuery(box, points)
query = aq.query(
    points, {'num_neighbors': 12, 'exclude_ii': True})
nl = query.toNeighborList()

# Get all sets of common neighbors.
common_neighbors = defaultdict(list)
for i, p in enumerate(points):
    selection1 = nl.query_point_indices == i
    for j in nl.point_indices[selection1]:
        selection2 = nl.query_point_indices == j
        for k in nl.point_indices[selection2]:
            if i != k:
                common_neighbors[(i, k)].append(j)
```

Our dictionary `common_neighbors` now contains lists of common neighbors `j` for every pair of points `(i, k)`. This information could itself be useful for performing some analysis on the system. If we are interested in actually implementing CNA, then we need to use this information to build local graphs, which we can do with the `networkx` (version 2.4) [65] Python package. Combined with the code above, the CNA algorithm can be implemented as follows:

```
import networkx as nx
from collections import Counter

diagrams = defaultdict(list)
particle_counts = defaultdict(Counter)

for (a, b), neighbors in common_neighbors.items():
    # Build up the graph of connections between the
    # common neighbors of a and b.
    g = nx.Graph()
    for i in neighbors:
        for j in set(nl.point_indices[
            nl.query_point_indices == i]
            ).intersection(neighbors):
            g.add_edge(i, j)

# Define the four identifiers for a CNA diagram:
# 1. 1 if particles are bonded, 0 if not.
# 2. Number of shared neighbors.
# 3. Number of bonds between shared neighbors.
# 4. Index guaranteeing diagram uniqueness.
diagram_type = 2 - int(
    b in nl.point_indices[nl.query_point_indices == a])
key = (diagram_type, len(neighbors),
       g.number_of_edges())

# If we've seen any neighborhood graphs with this
# signature, we explicitly check if the two graphs are
# identical to determine whether to save this one.
# Otherwise, we always add the new graph.
if key in diagrams:
    isomorphs = [
        nx.is_isomorphic(g, h) for h in diagrams[key]]
    if any(isomorphs):
        idx = isomorphs.index(True)
    else:
        diagrams[key].append(g)
        idx = diagrams[key].index(g)
    cna_signature = key + (idx,)
    particle_counts[a].update([cna_signature])
```


In this code, we are looping over all pairs of previously identified second neighbor shells, and finding bonds between the common neighbors of these pairs. The graph of these bonds then uniquely identifies a new environment.

6. Conclusion

`freud` is a high-performance Python library for analyzing particle simulations. Among simulation analysis packages, `freud` is unique due to its emphasis on coarse-grained simulations and its flexibility. Its high-performance C++ back-end makes `freud` a suitable solution for large-scale, high-throughput simulation analysis, while its simple, compact API is highly amenable to integration with other tools for, e.g., machine learning applications. The package's API also promotes the prototyping of new analyses directly in Python, and the intuitive design of `freud`'s internals ensures that translating such analyses into C++ is a relatively painless process.

The package's design is general enough to work with any particle-based system. However, `freud` is primarily targeted at communities of materials scientists, chemical engineers, and physicists analyzing molecular dynamics and Monte Carlo for which existing tools are too specialized to be convenient. Since it makes no assumptions about the types of its input data or the system topology, `freud` can be used with arbitrary simulation outputs based on topologies defined by the user. As a result, `freud` can find wide use in these areas to simplify workflows that require consideration of periodic systems without the complexity associated with specific atomistic features. Contributions to this open-source toolkit are highly encouraged as new methods are developed in future research applications.

Acknowledgments

Support for the design and development of `freud` has evolved over time and with programmatic research directions. Conceptualization and early implementations were supported in part by the DOD/ASD(R&E) under Award No. N00244-09-1-0062 and also by the National Science Foundation, Integrative Graduate Education and Research Traineeship, Award # DGE 0903629 (E.S.H. and M.P.S.). A majority of the code development including all public code releases was supported by the National Science Foundation, Division of Materials Research under a Computational and Data-Enabled Science & Engineering Award # DMR 1409620 (2014–2018) and the Office of Advanced Cyberinfrastructure Award # OAC 1835612 (2018–2021). V.R. holds the 2019–2020 J. Robert Beyster Computational Innovation Graduate Fellowship at the University of Michigan. B.D. acknowledges fellowship support from the National Science Foundation under ACI-1547580, S212: Impl: The Molecular Sciences Software Institute [66,67] and an earlier National Science Foundation Graduate Research Fellowship Grant #DGE 1256260 (2016–2019). M.P.S. also acknowledges support from the University of Michigan Rackham Predoctoral Fellowship program. Computational resources and services supported in part by Advanced Research Computing at the University of Michigan, Ann Arbor.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

- [1] J.A. Anderson, J. Antonaglia, J.A. Millan, M. Engel, S.C. Glotzer, *Phys. Rev. X* (ISSN: 21603308) 7 (2) (2017) 021001, <http://dx.doi.org/10.1103/PhysRevX.7.021001>.
- [2] A.J. Simon, Y. Zhou, V. Ramasubramani, J. Glaser, A. Pothukuchy, J. Gollihar, J.C. Gerberich, J.C. Leggere, B.R. Morrow, C. Jung, S.C. Glotzer, D.W. Taylor, A.D. Ellington, *Nature Chem.* (ISSN: 17554349) 11 (3) (2019) 204–212, <http://dx.doi.org/10.1038/s41557-018-0196-3>.
- [3] C. Niethammer, S. Becker, M. Bernreuther, M. Buchholz, W. Eckhardt, A. Heinecke, S. Werth, H.J. Bungartz, C.W. Glass, H. Hasse, J. Vrabec, M. Horsch, *J. Chem. Theory Comput.* (ISSN: 15499626) 10 (10) (2014) 4455–4464, <http://dx.doi.org/10.1021/ct500169q>.
- [4] P.L. Freddolino, F. Liu, M. Gruebele, K. Schulten, *Biophys. J.* (ISSN: 15420086) 94 (10) (2008) L75–L77, <http://dx.doi.org/10.1529/biophysj.108.131565>.
- [5] D.E. Shaw, K.J. Bowers, E. Chow, M.P. Eastwood, D.J. Ierardi, J.L. Klepeis, J.S. Kuskin, R.H. Larson, K. Lindorff-Larsen, P. Maragakis, M.A. Moraes, R.O. Dror, S. Piana, Y. Shan, B. Towles, J.K. Salmon, J.P. Grossman, K.M. Mackenzie, J.A. Bank, C. Young, M.M. Denneroff, B. Batson, *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09*, ACM Press, New York, New York, USA, 2009, p. 1, <http://dx.doi.org/10.1145/1654059.1654099>.
- [6] R.T. McGibbon, K.A. Beauchamp, M.P. Harrigan, C. Klein, J.M. Swails, C.X. Hernández, C.R. Schwantes, L.-P. Wang, T.J. Lane, V.S. Pande, *Biophys. J.* (ISSN: 0006-3495) 109 (8) (2015) 1528–1532, <http://dx.doi.org/10.1016/j.bpj.2015.08.015>.
- [7] N. Michaud-Agrawal, E.J. Denning, T.B. Woolf, O. Beckstein, *J. Comput. Chem.* (ISSN: 01928651) 32 (10) (2011) 2319–2327, <http://dx.doi.org/10.1002/jcc.21787>.
- [8] T. Romo, A. Grossfield, 2009 Annual International Conference of the IEEE Engineering in Medicine and Biology Society, IEEE, 2009, pp. 2332–2335, <http://dx.doi.org/10.1109/IEMBS.2009.5335065>.
- [9] K. Hinsen, *J. Comput. Chem.* (ISSN: 01928651) 21 (2) (2000) 79–85, [http://dx.doi.org/10.1002/\(SICI\)1096-987X\(20000130\)21:2<79::AID-JCC1>3.0.CO;2-B](http://dx.doi.org/10.1002/(SICI)1096-987X(20000130)21:2<79::AID-JCC1>3.0.CO;2-B).
- [10] W. Humphrey, A. Dalke, K. Schulten, *J. Mol. Graph.* 14 (1996) 33–38, [http://dx.doi.org/10.1016/0263-7855\(96\)00018-5](http://dx.doi.org/10.1016/0263-7855(96)00018-5).
- [11] W. Kabsch, C. Sander, *Biopolymers* (ISSN: 10970282) 22 (12) (1983) 2577–2637, <http://dx.doi.org/10.1002/bip.360221211>.
- [12] W.F. Reinhart, A.Z. Panagiotopoulos, *J. Chem. Phys.* (ISSN: 00219606) 148 (12) (2018) 124506, <http://dx.doi.org/10.1063/1.5021347>.
- [13] M.P. Howard, W.F. Reinhart, T. Sanyal, M.S. Shell, A. Nikoubashman, A.Z. Panagiotopoulos, *J. Chem. Phys.* (ISSN: 00219606) 149 (9) (2018) 094901, <http://dx.doi.org/10.1063/1.5043401>.
- [14] M. Spellings, S.C. Glotzer, *AIChE J.* (ISSN: 00011541) 64 (6) (2018) 2198–2206, <http://dx.doi.org/10.1002/aic.16157>.
- [15] C.S. Adorf, J. Antonaglia, J. Dshemuchadse, S.C. Glotzer, *J. Chem. Phys.* (ISSN: 00219606) 149 (20) (2018) 204102, <http://dx.doi.org/10.1063/1.5063802>.
- [16] B. Vansaders, J. Dshemuchadse, S.C. Glotzer, *Phys. Rev. Mater.* (ISSN: 24759953) 2 (6) (2018) 063604, <http://dx.doi.org/10.1103/PhysRevMaterials.2.063604>.
- [17] A.S. Karas, J. Glaser, S.C. Glotzer, *Soft Matter* (ISSN: 17446848) 12 (23) (2016) 5199–5204, <http://dx.doi.org/10.1039/c6sm00620e>.
- [18] J.A. Antonaglia, G. van Anders, S.C. Glotzer, Mapping disorder in entropically ordered crystals, 2018, arXiv preprint [arXiv:1803.05936](https://arxiv.org/abs/1803.05936).
- [19] C.X. Du, G. van Anders, R.S. Newman, S.C. Glotzer, *Proc. Natl. Acad. Sci. USA* (ISSN: 1091-6490) 114 (20) (2016) E3892–E3899, <http://dx.doi.org/10.1073/pnas.1621348114>.
- [20] E.S. Harper, R.L. Marson, J.A. Anderson, G. Van Anders, S.C. Glotzer, *Soft Matter* (ISSN: 17446848) 11 (37) (2015) 7250–7256, <http://dx.doi.org/10.1039/c5sm001351h>.
- [21] B. Dice, V. Ramasubramani, E. Harper, M. Spellings, J. Anderson, S. Glotzer, *Proceedings of the 18th Python in Science Conference, (Scipy) 2019*, pp. 27–33, <http://dx.doi.org/10.25080/Majora-7ddc1dd1-004>.
- [22] J.D. Hunter, *Comput. Sci. Eng.* 9 (3) (2007) 90–95, <http://dx.doi.org/10.1109/MCSE.2007.55>.
- [23] S. Plimpton, *J. Comput. Phys.* (ISSN: 0021-9991) 117 (1) (1995) 1–19, <http://dx.doi.org/10.1006/JCPH.1995.1039>.
- [24] H. Berendsen, D. van der Spoel, R. van Drunen, *Comput. Phys. Comm.* (ISSN: 0010-4655) 91 (1–3) (1995) 43–56, [http://dx.doi.org/10.1016/0010-4655\(95\)00042-E](http://dx.doi.org/10.1016/0010-4655(95)00042-E).
- [25] D.R. Roe, T.E. Cheatham, *J. Chem. Theory Comput.* (ISSN: 1549-9618) 9 (7) (2013) 3084–3095, <http://dx.doi.org/10.1021/ct400341p>.
- [26] D.A. Case, T.E. Cheatham, T. Darden, H. Gohlke, R. Luo, K.M. Merz, A. Onufriev, C. Simmerling, B. Wang, R.J. Woods, *J. Comput. Chem.* (ISSN: 0192-8651) 26 (16) (2005) 1668–1688, <http://dx.doi.org/10.1002/jcc.20290>.
- [27] L. Schrödinger, *The pymol molecular graphics system, version 2.3*, 2019.
- [28] A. Stukowski, *Modelling Simulation Mater. Sci. Eng.* (ISSN: 09650393) 18 (1) (2010) <http://dx.doi.org/10.1088/0965-0393/18/1/015012>.

- [29] S.O. Yesylevskyy, J. Comput. Chem. (ISSN: 01928651) 33 (19) (2012) 1632–1636, <http://dx.doi.org/10.1002/jcc.22989>.
- [30] T.E. Oliphant, *A Guide to NumPy*, Trelgol Publishing, 2006.
- [31] G. Lab, GSD V2.0.0, 2020, URL <https://github.com/glotzerlab/gsd>.
- [32] G. Lab, Garnett v0.6.1, 2020, URL <https://github.com/glotzerlab/garnett>.
- [33] W. McKinney, *Data Structures for Statistical Computing in Python*, Tech. Rep., 2010, p. 51.
- [34] J.A. Anderson, M. Eric Irrgang, S.C. Glotzer, Comput. Phys. Comm. (ISSN: 00104655) 204 (2016) 21–30, <http://dx.doi.org/10.1016/j.cpc.2016.02.024>.
- [35] M.P. Allen, D.J. Tildesley, *Computer Simulation of Liquids*, Clarendon Press, ISBN: 9780198556459, 1987, p. 385.
- [36] J.A. Anderson, C.D. Lorenz, A. Travesset, J. Comput. Phys. (ISSN: 0021-9991) 227 (10) (2008) 5342–5359, <http://dx.doi.org/10.1016/j.jcp.2008.01.047>.
- [37] J. Glaser, T.D. Nguyen, J.A. Anderson, P. Lui, F. Spiga, J.A. Millan, D.C. Morse, S.C. Glotzer, Comput. Phys. Comm. (ISSN: 0010-4655) 192 (2015) 97–107, <http://dx.doi.org/10.1016/j.cpc.2015.02.028>.
- [38] J.A. Anderson, M. Eric Irrgang, S.C. Glotzer, Comput. Phys. Comm. (ISSN: 00104655) 204 (2016) 21–30, <http://dx.doi.org/10.1016/j.cpc.2016.02.024>.
- [39] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, K. Smith, Comput. Sci. Eng. (ISSN: 1521-9615) 13 (2) (2011) 31–39, <http://dx.doi.org/10.1109/MCSE.2010.118>.
- [40] V. Calandrini, E. Pellegrini, P. Calligari, K. Hinsén, G. Kneller, Écol. Mat. Soc. Française de la Neutronique (ISSN: 2107-7223) 12 (2011) 201–232, <http://dx.doi.org/10.1051/sfn/201112010>.
- [41] E. Jones, T. Oliphant, P. Peterson, et al., *Scipy: Open source scientific tools for python*, 2001, URL <https://www.scipy.org/>.
- [42] Intel, Intel threading building blocks, 2020, URL <https://github.com/intel/tbb>.
- [43] Anaconda Software Distribution, Anaconda, 2020, <https://anaconda.com>.
- [44] Glotzer Lab, Freud source code repository, 2020, URL <https://github.com/glotzerlab/freud>.
- [45] C. Rycroft, Voro++: a Three-Dimensional Voronoi Cell Library in C++, Tech. Rep., Lawrence Berkeley National Laboratory (LBNL), Berkeley, CA, 2009, <http://dx.doi.org/10.2172/946741>.
- [46] E.A. Lazar, J. Han, D.J. Srolovitz, Proc. Natl. Acad. Sci. USA (ISSN: 10916490) 112 (43) (2015) E5769–E5776, <http://dx.doi.org/10.1073/pnas.1505788112>.
- [47] Glotzer Lab, Fresnel v0.11.0, 2020, URL <https://github.com/glotzerlab/fresnel>.
- [48] P.J. Steinhardt, D.R. Nelson, M. Ronchetti, Phys. Rev. B 28 (2) (1983) 784–805, <http://dx.doi.org/10.1103/PhysRevB.28.784>.
- [49] A. Haji-Akbari, S.C. Glotzer, J. Phys. A (ISSN: 17518121) 48 (48) (2015) 485201, <http://dx.doi.org/10.1088/1751-8113/48/48/485201>.
- [50] P. Rein ten Wolde, M.J. Ruiz-Montero, D. Frenkel, J. Chem. Phys. (ISSN: 0021-9606) 104 (24) (1996) 9932–9947, <http://dx.doi.org/10.1063/1.471721>.
- [51] G. van Anders, N.K. Ahmed, R. Smith, M. Engel, S.C. Glotzer, ACS Nano (ISSN: 19360851) 8 (1) (2014) 931–940, <http://dx.doi.org/10.1021/nn4057353>.
- [52] G. van Anders, D. Klotz, N.K. Ahmed, M. Engel, S.C. Glotzer, Proc. Natl. Acad. Sci. (ISSN: 0027-8424) 111 (45) (2014) E4812–E4821, <http://dx.doi.org/10.1073/pnas.1418159111>.
- [53] P. Ramachandran, G. Varoquaux, Comput. Sci. Eng. 13 (2) (2011) 40–51, <http://dx.doi.org/10.1109/MCSE.2011.35>.
- [54] E.S. Harper, G. van Anders, S.C. Glotzer, Proc. Natl. Acad. Sci. USA (ISSN: 10916490) 116 (34) (2019) 16703–16710, <http://dx.doi.org/10.1073/pnas.1822092116>.
- [55] P.F. Damasceno, M. Engel, S.C. Glotzer, Science (ISSN: 0036-8075) 337 (6093) (2012) 453–457, <http://dx.doi.org/10.1126/science.1220869>.
- [56] M. Dzugasov, Phys. Rev. Lett. (ISSN: 00319007) 70 (19) (1993) 2924–2927, <http://dx.doi.org/10.1103/PhysRevLett.70.2924>.
- [57] J.W. Roth, R. Schilling, H.R. Trebin, Phys. Rev. B (ISSN: 01631829) 51 (22) (1995) 15833–15840, <http://dx.doi.org/10.1103/PhysRevB.51.15833>.
- [58] J. Roth, A.R. Denton, Phys. Rev. E (ISSN: 1063-651X) 61 (6) (2000) 6845–6857, <http://dx.doi.org/10.1103/PhysRevE.61.6845>.
- [59] M. Engel, P.F. Damasceno, C.L. Phillips, S.C. Glotzer, Nature Mater. (ISSN: 1476-1122) 14 (1) (2015) 109–116, <http://dx.doi.org/10.1038/nmat4152>.
- [60] A.S. Keys, C.R. Iacovella, S.C. Glotzer, J. Comput. Phys. (ISSN: 0021-9991) 230 (17) (2011) 6438–6463, <http://dx.doi.org/10.1016/j.jcp.2011.04.017>.
- [61] E.G. Teich, G. van Anders, S.C. Glotzer, Nature Commun. (ISSN: 2041-1723) 10 (1) (2019) 64, <http://dx.doi.org/10.1038/s41467-018-07977-2>.
- [62] A.S. Karas, J. Dshemuchadse, G. van Anders, S.C. Glotzer, Soft Matter (ISSN: 1744-683X) (2019) <http://dx.doi.org/10.1039/C8SM02643B>.
- [63] B.R. Brooks, C.L. Brooks, A.D. Mackerell, L. Nilsson, R.J. Petrella, B. Roux, Y. Won, G. Archontis, C. Bartels, S. Boresch, A. Caflisch, L. Caves, Q. Cui, A.R. Dinner, M. Feig, S. Fischer, J. Gao, M. Hodoscek, W. Im, K. Kucsera, T. Lazaridis, J. Ma, V. Ovchinnikov, E. Paci, R.W. Pastor, C.B. Post, J.Z. Pu, M. Schaefer, B. Tidor, R.M. Venable, H.L. Woodcock, X. Wu, W. Yang, D.M. York, M. Karplus, J. Comput. Chem. (ISSN: 01928651) 30 (10) (2009) 1545–1614, <http://dx.doi.org/10.1002/jcc.21287>.
- [64] J.D. Honeycutt, H.C. Andersen, J. Phys. Chem. (ISSN: 0022-3654) 91 (19) (1987) 4950–4963, <http://dx.doi.org/10.1021/j100303a014>.
- [65] A.A. Hagberg, D.A. Schult, P.J. Swart, in: G. Varoquaux, T. Vaught, J. Millman (Eds.), *Proceedings of the 7th Python in Science Conference*, Pasadena, CA USA, pp. 11–15.
- [66] N. Wilkins-Diehr, D.T. Crawford, Comput. Sci. Eng. (ISSN: 1558366X) 20 (5) (2018) 26–38, <http://dx.doi.org/10.1109/MCSE.2018.05329813>.
- [67] A. Krylov, T.L. Windus, T. Barnes, E. Marin-Rimoldi, J.A. Nash, B. Pritchard, D.G. Smith, D. Altarawy, P. Saxe, C. Clementi, T.D. Crawford, R.J. Harrison, S. Jha, V.S. Pande, T. Head-Gordon, J. Chem. Phys. (ISSN: 00219606) 149 (18) (2018) 180901, <http://dx.doi.org/10.1063/1.5052551>.