Analyzing Particle Systems for Machine Learning and Data Visualization with freud

Bradley D. Dice[‡*], Vyas Ramasubramani[§], Eric S. Harper[**], Matthew P. Spellings[§], Joshua A. Anderson[§], Sharon C. Glotzer[‡§¶‖]

Abstract—The `freud` Python library analyzes particle data output from molecular dynamics simulations. The library's design and its variety of high-performance methods make it a powerful tool for many modern applications. In particular, `freud` can be used as part of the data generation pipeline for machine learning (ML) algorithms for analyzing particle simulations, and it can be easily integrated with various simulation visualization tools for simultaneous visualization and real-time analysis. Here, we present numerous examples both of using `freud` to analyze nano-scale particle systems by coupling traditional simulational analyses to machine learning libraries and of visualizing per-particle quantities calculated by `freud` analysis methods. We include code and examples of this visualization, showing that in general the introduction of `freud` into existing ML and visualization workflows is smooth and unintrusive. We demonstrate that among Python packages used in the computational molecular sciences, `freud` offers a unique set of analysis methods with efficient computations and seamless coupling into powerful data analysis pipelines.

Index Terms—molecular dynamics, analysis, particle simulation, particle system, computational physics, computational chemistry

Introduction

The availability of "off-the-shelf" molecular dynamics engines (e.g. HOOMD-blue [ALT08], [GNA+15], LAMMPS [Pli95], GROMACS [BvdSvD95]) has made simulating complex systems possible across many scientific fields. Simulations of systems ranging from large biomolecules to colloids are now common, allowing researchers to ask new questions about reconfigurable materials [CDA+18] and develop coarse-graining approaches to access increasing timescales [SZR+19]. Various tools have arisen to facilitate the analysis of these simulations, many of which are immediately interoperable with the most popular simulation tools. The `freud` library is one such analysis package that differentiates itself from others through its focus on colloidal and nano-scale systems.

Due to their diversity and adaptability, colloidal materials are a powerful model system for exploring soft matter physics [GS07]. Such materials are also a viable platform for harnessing photonic [CDA+18], plasmonic [TCLC11], and other useful structurally-derived properties. In colloidal systems, features like particle anisotropy play an important role in creating complex crystal structures, some of which have no atomic analogues [DEG12]. Design spaces encompassing wide ranges of particle morphology [DEG12] and interparticle interactions [AADG18] have been shown to yield phase diagrams filled with complex behavior.

The `freud` Python package offers a unique feature set that targets the analysis of colloidal systems. The library avoids trajectory management and the analysis of chemically bonded structures, which are the province of most other analysis platforms like MDAnalysis and MDTraj (see also 1) [MADWB11], [MBH+15]. In particular, `freud` excels at performing analyses based on characterizing local particle environments, which makes it a powerful tool for tasks such as calculating order parameters to track crystallization or finding prenucleation clusters. Among the unique methods present in `freud` are the potential of mean force and torque, which allows users to understand the effects of particle anisotropy on entropic self-assembly [vAAS+14], [vAKA+14], [KGG16], [HMA+15], [AAM+17], and various tools for identifying and clustering particles by their local crystal environments [TvAG19]. All such tasks are accelerated by `freud`'s extremely fast neighbor finding routines and are automatically parallelized, making it an ideal tool for researchers performing peta- or exascale simulations of particle systems. The `freud` library's scalability is exemplified by its use in computing correlation functions on systems of over a million particles, calculations that were used to illuminate the elusive hexatic phase transition in two-dimensional systems of hard polygons [AAM+17]. More details on the use of `freud` can be found in [RDH+19]. In this paper, we will demonstrate that `freud` is uniquely well-suited to usage in the context of data pipelines for visualization and machine learning applications.

Data Pipelines

The `freud` package is especially useful because it can be organically integrated into a data pipeline. Many research tasks in computational molecular sciences can be expressed in terms of data pipelines; in molecular simulations, such a pipeline typically involves:

1) **Generating** an input file that defines a simulation.
2) **Simulating** the system of interest, saving its trajectory to a file.

* *Corresponding author: bdice@umich.edu*
‡ *Department of Physics, University of Michigan, Ann Arbor*
§ *Department of Chemical Engineering, University of Michigan, Ann Arbor*
** *Department of Materials Science & Engineering, University of Michigan, Ann Arbor*
¶ *Department of Materials Science and Engineering, University of Michigan, Ann Arbor*
‖ *Biointerfaces Institute, University of Michigan, Ann Arbor*

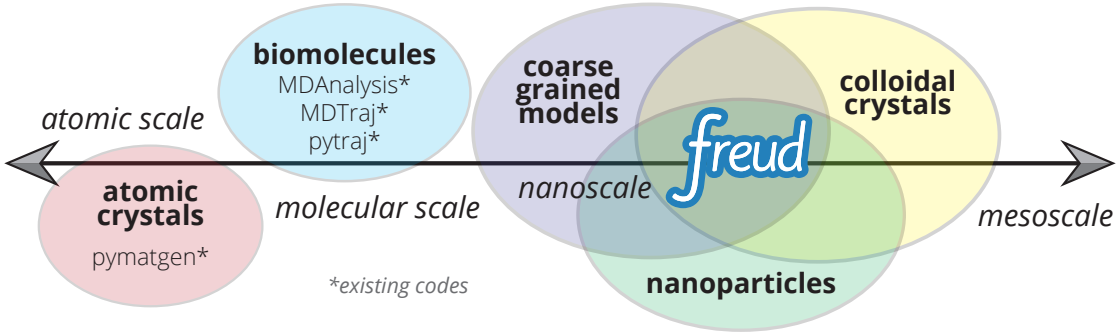

Fig. 1: *Common Python tools for simulation analysis at varying length scales. The freud library is designed for nanoscale systems, such as colloidal crystals and nanoparticle assemblies. In such systems, interactions are described by coarse-grained models where particles' atomic constituents are often irrelevant and particle anisotropy (non-spherical shape) is common, thus requiring a generalized concept of particle "types" and orientation-sensitive analyses. These features contrast the assumptions of most analysis tools designed for biomolecular simulations and materials science.*

3) **Analyzing** the resulting data by computing and storing various quantities.
4) **Visualizing** the trajectory, using colors or styles determined from previous analyses.

However, in modern workflows the lines between these stages is typically blurred, particularly with respect to analysis. While direct visualization of simulation trajectories can provide insights into the behavior of a system, integrating higher-order analyses is often necessary to provide real-time interpretable visualizations in that allow researchers to identify meaningful features like defects and ordered domains of self-assembled structures. Studies of complex systems are also often aided or accelerated by a real-time coupling of simulations with on-the-fly analysis. This simultaneous usage of simulation and analysis is especially relevant because modern machine learning techniques frequently involve wrapping this pipeline entirely within a higher-level optimization problem, since analysis methods can be used to construct objective functions targeting a specific materials design problem, for instance.

Following, we provide demonstrations of how `freud` can be integrated with popular tools in the scientific Python ecosystem like TensorFlow, Scikit-learn, SciPy, or Matplotlib. In the context of machine learning algorithms, we will discuss how the analyses in `freud` can reduce the 6N-dimensional space of particle positions and orientations into a tractable set of features that can be fed into machine learning algorithms. We will further show that `freud` can be used for visualizations even outside of scripting contexts, enabling a wide range of forward-thinking applications including Jupyter notebook integrations, versatile 3D renderings, and integration with various standard tools for visualizing simulation trajectories. These topics are aimed at computational molecular scientists and data scientists alike, with discussions of real-world usage as well as theoretical motivation and conceptual exploration. The full source code of all examples in this paper can be found online[1].

Performance and Integrability

Using `freud` to compute features for machine learning algorithms and visualization is straightforward because it adheres to a UNIX-like philosophy of providing modular, composable features. This design is evidenced by the library's reliance on NumPy arrays [Oli06] for all inputs and outputs, a format that is naturally integrated with most other tools in the scientific Python ecosystem. In general, the analyses in `freud` are designed around analyses of raw particle trajectories, meaning that the inputs are typically $(N,3)$ arrays of particle positions and $(N,4)$ arrays of particle orientations, and analyses that involve many frames over time use *accumulate* methods that are called once for each frame. This general approach enables `freud` to be used for a range of input data, including molecular dynamics and Monte Carlo simulations as well as experimental data (e.g. positions extracted via particle tracking) in both 3D and 2D. The direct usage of numerical arrays indicates a different usage pattern than that of tools, such as MDAnalysis [MADWB11] and MDTraj [MBH+15], for which trajectory parsing is a core feature. Due to the existence of many such tools which are capable of reading simulation engines' output files, as well as certain formats like `gsd`[2] that provide their own parsers, `freud` eschews any form of trajectory management and instead relies on other tools to provide input arrays. If input data is to be read from a file, binary data formats such as `gsd` or NumPy's `npy` or `npz` are strongly preferred for efficient I/O. Though it is possible to use a library like Pandas to load data stored in a comma-separated value (CSV) or other text-based data format, such files are often much slower when reading and writing large numerical arrays. Decoupling `freud` from file parsing and specific trajectory representations allows it to be efficiently integrated into simulations, machine learning applications, and visualization toolkits with no I/O overhead and limited additional code complexity, while the universal usage of NumPy arrays makes such integrations very natural.

In keeping with this focus on composable features, `freud` also abstracts and directly exposes the task of finding particle neighbors, the task most central to all other analyses in `freud`. Since neighbor finding is a common need, the neighbor finding routines in `freud` are highly optimized and natively support periodic systems, a crucial feature for any analysis of particle simulations (which often employ periodic boundary conditions). In figure 2, a comparison is shown between the neighbor finding algorithms in `freud` and SciPy [JOPo01]. For each system size, N particles are uniformly distributed in a 3D periodic cube such that each particle has an average of 12 neighbors within a distance of $r_{cut} = 1.0$. Neighbors are found for each particle by

1. https://github.com/glotzerlab/freud-examples

2. https://github.com/glotzerlab/gsd

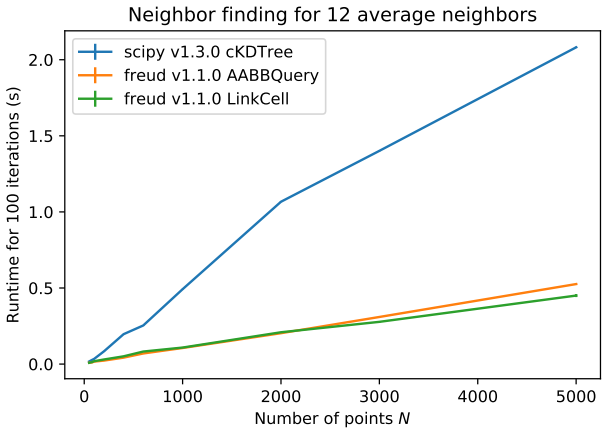

Fig. 2: *Comparison of runtime for neighbor finding algorithms in* `freud` *and SciPy for varied system sizes. See text for details.*

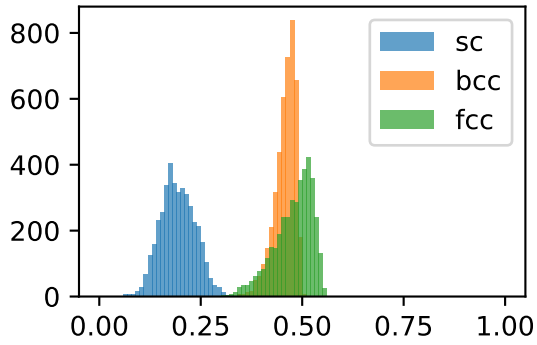

Fig. 3: *Histogram of the Steinhardt Q_6 order parameter for 4000 particles in simple cubic, body-centered cubic, and face-centered cubic structures with added Gaussian noise.*

searching within the cutoff distance r_{cut}. The methods compared are `scipy.spatial.cKDTree`'s `query_ball_tree`, `freud.locality.AABBQuery`'s `queryBall`, and `freud.locality.LinkCell`'s `compute`. The benchmarks were performed with 5 replicates on a 3.6 GHz Intel Core i3-8100B processor with 16 GB 2667 MHz DDR4 RAM.

Evidently, `freud` performs very well on this core task and scales well to larger systems. The parallel C++ backend implemented with Cython and Intel Threading Building Blocks makes `freud` perform quickly even for large systems [BBC+11], [Int18]. Furthermore, `freud` supports periodicity in arbitrary triclinic volumes, a common feature found in many simulations. This support distinguishes it from other tools like `scipy.spatial.cKDTree`, which only supports cubic boxes. The fast neighbor finding in `freud` and the ease of integrating its outputs into other analyses not only make it easy to add fast new analysis methods into `freud`, they are also central to why `freud` can be easily integrated into workflows for machine learning and visualization.

Machine Learning

A wide range of problems in soft matter and nano-scale simulations have been addressed using machine learning techniques, such as crystal structure identification [SG18]. In machine learning workflows, `freud` is used to generate features, which are then used in classification or regression models, clusterings, or dimensionality reduction methods. For example, Harper et al. used `freud` to compute the cubatic order parameter and generate high-dimensional descriptors of structural motifs, which were visualized with t-SNE dimensionality reduction [HWG19], [vdMH08]. The library has also been used in the optimization and inverse design of pair potentials [AADG18], to compute fitness functions based on the radial distribution function. The open-source `pythia`[3] library offers a number of descriptor sets useful for crystal structure identification, leveraging `freud` for fast computations. Included among the descriptors in `pythia` are quantities based on bond angles and distances, spherical harmonics, and Voronoi diagrams.

Computing a set of descriptors tuned for a particular system of interest (e.g. using values of Q_l, the higher-order Steinhardt W_l parameters, or other order parameters provided by `freud`) is possible with just a few lines of code. Descriptors like these (exemplified in the `pythia` library) have been used with TensorFlow for supervised and unsupervised learning of crystal structures in complex phase diagrams [SG18], [AAB+15].

Another useful module for machine learning with `freud` is `freud.cluster`, which uses a distance-based cutoff to locate clusters of particles while accounting for 2D or 3D periodicity. Locating clusters in this way can identify crystalline grains, helpful for building a training set for machine learning models.

To demonstrate a concrete example, we focus on a common challenge in molecular sciences: identifying crystal structures. Recently, several approaches have been developed that use machine learning for detecting ordered phases [SCKL15], [SG18], [FSM19], [SNR83], [LD08]. The Steinhardt order parameters are often used as a structural fingerprint, and are derived from rotationally invariant combinations of spherical harmonics. In the example below, we create face-centered cubic (fcc), body-centered cubic (bcc), and simple cubic (sc) crystals with added Gaussian noise, and use Steinhardt order parameters with a support vector machine to train a simple crystal structure identifier. Steinhardt order parameters characterize the spherical arrangement of neighbors around a central particle, and combining values of Q_l for a range of l often gives a unique signature for simple crystal structures. This example demonstrates a simple case of how `freud` can be used to help solve the problem of structural identification, which often requires a sophisticated approach for complex crystals.

In figure 3, we show the distribution of Q_6 values for sample structures with 4000 particles. Here, we demonstrate how to compute the Steinhardt Q_6, using neighbors found via a periodic Voronoi diagram. Neighbors with small facets in the Voronoi polytope are filtered out to reduce noise.

```
import freud
import numpy as np
from util import make_fcc

def get_features(box, positions, structure):
    # Create a Voronoi compute object
    voro = freud.voronoi.Voronoi(
        box, buff=max(box.L)/2)
    voro.computeNeighbors(positions)

    # Filter the Voronoi NeighborList
    nlist = voro.nlist
    nlist.filter(nlist.weights > 0.1)
```

3. https://github.com/glotzerlab/pythia

```
    # Compute Steinhardt order parameters
    features = {}
    for l in [4, 6, 8, 10, 12]:
        ql = freud.order.LocalQl(
             box, rmax=max(box.L)/2, l=l)
        ql.compute(positions, nlist)
        features['q{}'.format(l)] = ql.Ql.copy()

    return features

# Create a freud box object and an array of
# 3D positions for a face-centered cubic
# structure with 4000 particles
fcc_box, fcc_positions = make_fcc(
    nx=10, ny=10, nz=10, noise=0.1)

structures = {}
structures['fcc'] = get_features(
    fcc_box, fcc_positions, 'fcc')
# ... repeat for all structures
```

Then, using Pandas and Scikit-learn, we can train a support vector machine to identify these structures:

```
# Build dictionary of DataFrames,
# labeled by structure
structure_dfs = {}
for i, struct in enumerate(structures):
    df = pd.DataFrame.from_dict(structures[struct])
    df['class'] = i
    structure_dfs[struct] = df

# Combine DataFrames for input to SVM
df = pd.concat(structure_dfs.values())
df = df.reset_index(drop=True)

from sklearn.preprocessing import normalize
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

# We use the normalized Steinhardt order parameters
# to predict the crystal structure
X = df.drop('class', axis=1).values
X = normalize(X)
y = df['class'].values
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=42)

svm = SVC()
svm.fit(X_train, y_train)
print('Score:', svm.score(X_test, y_test))
# The model is ~98% accurate.
```

To interpret crystal identification models like this, it can be helpful to use a dimensionality reduction tool such as Uniform Manifold Approximation and Projection (UMAP) [MH18], as shown in figure 4. The low-dimensional UMAP projection shown is generated directly from the Pandas `DataFrame`:

```
from umap import UMAP
umap = UMAP()

# Project the high-dimensional descriptors
# to a two dimensional manifold
data = umap.fit_transform(df)
plt.plot(data[:, 0], data[:, 1])
```

Visualization

Many analyses performed by the `freud` library provide a `plot(ax=None)` method (new in v1.2.0) that allows their computed quantities to be visualized with Matplotlib. Additionally, these plottable analyses offer IPython representations, allowing Jupyter notebooks to render a graph such as a radial distribution function $g(r)$ just by returning the compute object at the end of a cell. Analyses like the radial distribution function or correlation functions return data that is binned as a one-dimensional histogram -- these are visualized with a line graph via `matplotlib.pyplot.plot`, with the bin locations and bin counts given by properties of the compute object. Other classes provide multi-dimensional histograms, like the Gaussian density or Potential of Mean Force and Torque, which are plotted with `matplotlib.pyplot.imshow`.

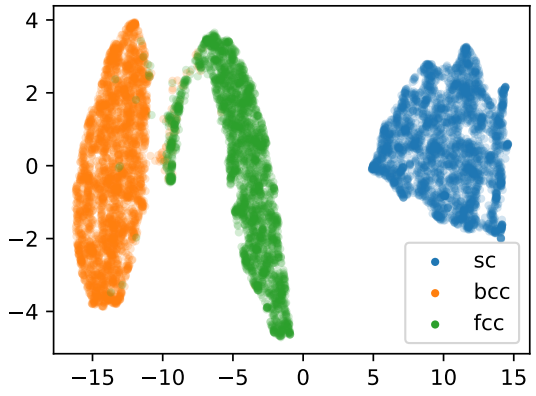

Fig. 4: *UMAP of particle descriptors computed for simple cubic, body-centered cubic, and face-centered cubic structures of 4000 particles with added Gaussian noise. The particle descriptors include Q_l for $l \in \{4, 6, 8, 10, 12\}$. Some noisy configurations of bcc can be confused as fcc and vice versa, which accounts for the small number of errors in the support vector machine's test classification.*

The most complex case for visualization is that of per-particle properties, which also comprises some of the most useful features in `freud`. Quantities that are computed on a per-particle level can be continuous (e.g. Steinhardt order parameters) or discrete (e.g. clustering, where the integer value corresponds to a unique cluster ID). Continuous quantities can be plotted as a histogram over particles, but typically the most helpful visualizations use these quantities with a color map assigned to particles in a two- or three-dimensional view of the system itself. For such particle visualizations, several open-source tools exist that interoperate well with `freud`. Below are examples of how one can integrate `freud` with `plato`[4], `fresnel`[5], and OVITO[6] [Stu10].

plato

`plato` is an open-source graphics package that expresses a common interface for defining two- or three-dimensional scenes which can be rendered as an interactive Jupyter widget or saved to a high-resolution image using one of several backends (PyThreejs, Matplotlib, `fresnel`, POVray[7], and Blender[8], among others). Below is an example of how to render particles from a HOOMD-blue snapshot, colored by the density of their local environment [ALT08], [GNA+15]. The result is shown in figure 5.

```
import plato
import plato.draw.pythreejs as draw
import numpy as np
```

4. https://github.com/glotzerlab/plato
5. https://github.com/glotzerlab/fresnel
6. https://ovito.org/
7. https://www.povray.org/
8. https://www.blender.org/

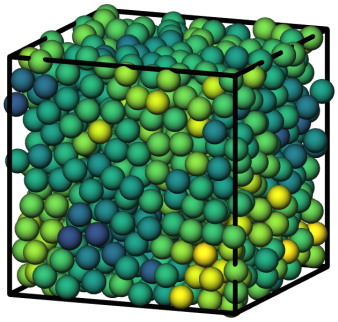

Fig. 5: *Interactive visualization of a Lennard-Jones particle system, rendered in a Jupyter notebook using* `plato` *with the* `pythreejs` *backend.*

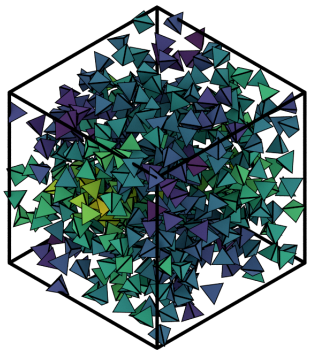

Fig. 6: *Hard tetrahedra colored by local density, path traced with* `fresnel`*.*

```
import matplotlib.cm
import freud
from sklearn.preprocessing import minmax_scale

# snap comes from a previous HOOMD-blue simulation
box = freud.box.Box.from_box(snap.box)
positions = snap.particles.position

# Compute the local density of each particle
ld = freud.density.LocalDensity(
    r_cut=3.0, volume=1.0, diameter=1.0)
ld.compute(box, positions)

# Create a scene for visualization,
# colored by local density
radii = 0.5 * np.ones(len(positions))
colors = matplotlib.cm.viridis(
    minmax_scale(ld.density))
spheres_primitive = draw.Spheres(
    positions=positions,
    radii=radii,
    colors=colors)
scene = draw.Scene(spheres_primitive, zoom=2)
scene.show()  # Interactive view in Jupyter
```

fresnel

`fresnel`[9] is a GPU-accelerated ray tracer designed for particle simulations, with customizable material types and scene lighting, as well as support for a set of common anisotropic shapes. Its feature set is especially well suited for publication-quality graphics. Its use of ray tracing also means that an image's rendering time scales most strongly with the image size, instead of the number of particles -- a desirable feature for extremely large simulations. An example of how to integrate `fresnel` is shown below and rendered in figure 6.

```
# Generate a snapshot of tetrahedra using HOOMD-blue
import hoomd
import hoomd.hpmc
hoomd.context.initialize('')

# Create an 8x8x8 simple cubic lattice
system = hoomd.init.create_lattice(
    unitcell=hoomd.lattice.sc(a=1.5), n=8)

# Create tetrahedra, configure HPMC integrator
mc = hoomd.hpmc.integrate.convex_polyhedron(seed=123)
mc.set_params(d=0.2, a=0.1)
vertices = [( 0.5, 0.5, 0.5),
            (-0.5,-0.5, 0.5),
            (-0.5, 0.5,-0.5),
            ( 0.5,-0.5,-0.5)]
mc.shape_param.set('A', vertices=vertices)

# Run for 5,000 steps
hoomd.run(5e3)
snap = system.take_snapshot()

# Import analysis & visualization libraries
import fresnel
import freud
import matplotlib.cm
from matplotlib.colors import Normalize
import numpy as np
device = fresnel.Device()

# Compute local density and prepare geometry
poly_info = \
    fresnel.util.convex_polyhedron_from_vertices(
        vertices)
positions = snap.particles.position
orientations = snap.particles.orientation
box = freud.box.Box.from_box(snap.box)
ld = freud.density.LocalDensity(3.0, 1.0, 1.0)
ld.compute(box, positions)
colors = matplotlib.cm.viridis(
    Normalize()(ld.density))
box_points = np.asarray([
    box.makeCoordinates(
        [[0, 0, 0], [0, 0, 0], [0, 0, 0],
         [1, 1, 0], [1, 1, 0], [1, 1, 0],
         [0, 1, 1], [0, 1, 1], [0, 1, 1],
         [1, 0, 1], [1, 0, 1], [1, 0, 1]]),
    box.makeCoordinates(
        [[1, 0, 0], [0, 1, 0], [0, 0, 1],
         [1, 0, 0], [0, 1, 0], [1, 1, 1],
         [1, 1, 1], [0, 1, 0], [0, 0, 1],
         [0, 0, 1], [1, 1, 1], [1, 0, 0]])])

# Create scene
scene = fresnel.Scene(device)
geometry = fresnel.geometry.ConvexPolyhedron(
    scene, poly_info,
    position=positions,
    orientation=orientations,
    color=fresnel.color.linear(colors))
geometry.material = fresnel.material.Material(
    color=fresnel.color.linear([0.25, 0.5, 0.9]),
    roughness=0.8, primitive_color_mix=1.0)
geometry.outline_width = 0.05
box_geometry = fresnel.geometry.Cylinder(
    scene, points=box_points.swapaxes(0, 1))
box_geometry.radius[:] = 0.1
box_geometry.color[:] = np.tile(
    [0, 0, 0], (12, 2, 1))
box_geometry.material.primitive_color_mix = 1.0
scene.camera = fresnel.camera.fit(
    scene, view='isometric', margin=0.1)
scene.lights = fresnel.light.lightbox()
```

9. https://github.com/glotzerlab/fresnel

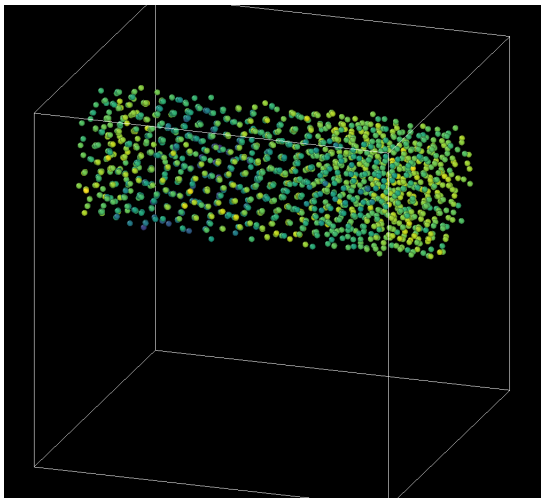

Fig. 7: *A crystalline grain identified using* `freud`*'s* `LocalDensity` *module and cut out for display using OVITO. The image shows a tP30-CrFe structure formed from an isotropic pair potential optimized to generate this structure [AADG18].*

```
# Path trace the scene
fresnel.pathtrace(scene, light_samples=64,
                  w=800, h=800)
```

OVITO

OVITO is a GUI application with features for particle selection, making movies, and support for many trajectory formats [Stu10]. OVITO has several built-in analysis functions (e.g. Polyhedral Template Matching), which complement the methods in `freud`. The Python scripting functionality built into OVITO enables the use of `freud` modules, demonstrated in the code below and shown in figure 7.

```
import freud

def modify(frame, input, output):

    if input.particles != None:
        box = freud.box.Box.from_matrix(
            input.cell.matrix)
        ld = freud.density.LocalDensity(
            r_cut=3, volume=1, diameter=0.05)
        ld.compute(box, input.particles.position)
        output.create_user_particle_property(
            name='LocalDensity',
            data_type=float,
            data=ld.density.copy())
```

Conclusions

The `freud` library offers a unique set of high-performance algorithms designed to accelerate the study of nanoscale and colloidal systems. These algorithms are enabled by a fast, easy-to-use set of tools for identifying particle neighbors, a common first step in nearly all such analyses. The efficiency of both the core neighbor finding algorithms and the higher-level analyses makes them suitable for incorporation into real-time visualization environments, and, in conjunction with the transparent NumPy-based interface, allows integration into machine learning workflows using iterative optimization routines that require frequent recomputation of these analyses. The use of `freud` for real-time visualization has the potential to simplify and accelerate existing simulation visualization pipelines, which typically involve slower and less easily integrable solutions to performing real-time analysis during visualization. The application of `freud` to machine learning, on the other hand, opens up entirely new avenues of research based on treating well-known analyses of particle simulations as descriptors or optimization targets. In these ways, `freud` can facilitate research in the field of computational molecular science, and we hope these examples will spark new ideas for scientific exploration in this field.

Getting `freud`

The `freud` library is tested for Python 2.7 and 3.5+ and is compatible with Linux, macOS, and Windows. To install `freud`, execute

```
conda install -c conda-forge freud
```

or

```
pip install freud-analysis
```

Its source code is available on GitHub[10] and its documentation is available via ReadTheDocs[11].

Acknowledgments

Thanks to Jin Soo Ihm for benchmarking the neighbor finding features of `freud` against SciPy. The `freud` library's code development and public code releases are 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). B.D. is supported by a National Science Foundation Graduate Research Fellowship Grant DGE 1256260. M.P.S acknowledges funding from the Toyota Research Institute; this article solely reflects the opinions and conclusions of its authors and not TRI or any other Toyota entity. Data for Figure 7 generated on the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575; XSEDE award DMR 140129.

References

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

[AADG18] Carl S. Adorf, James Antonaglia, Julia Dshemuchadse, and Sharon C. Glotzer. Inverse design of simple pair potentials for the self-assembly of complex structures. *The Journal of Chemical Physics*, 149(20):204102, 11 2018. doi:10.1063/1.5063802.

[AAM+17] Joshua A. Anderson, James Antonaglia, Jaime A. Millan, Michael Engel, and Sharon C. Glotzer. Shape and Symmetry Determine Two-Dimensional Melting Transitions of Hard Regular Polygons. *Physical Review X*, 7(2):021001, 4 2017. doi:10.1103/PhysRevX.7.021001.

10. https://github.com/glotzerlab/freud
11. https://freud.readthedocs.io/

[ALT08] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342 – 5359, 2008. doi:https://doi.org/10.1016/j.jcp.2008.01.047.

[BBC+11] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The Best of Both Worlds. *Computing in Science & Engineering*, 13(2):31–39, 3 2011. doi:10.1109/MCSE.2010.118.

[BvdSvD95] H.J.C. Berendsen, D. van der Spoel, and R. van Drunen. GROMACS: A message-passing parallel molecular dynamics implementation. *Computer Physics Communications*, 91(1-3):43–56, 9 1995. doi:10.1016/0010-4655(95)00042-E.

[CDA+18] Rose K. Cersonsky, Julia Dshemuchadse, James A. Antonaglia, Greg van Anders, and Sharon C. Glotzer. Pressure-Tunable Photonic Band Gaps in an Entropic Colloidal Crystal. *Physical Review Materials*, 2:125201, 2018. doi:10.1103/PhysRevMaterials.2.125201.

[DEG12] Pablo F. Damasceno, Michael Engel, and Sharon C. Glotzer. Predictive Self-Assembly of Polyhedra into Complex Structures. *Science*, 337(6093):453–457, 7 2012. doi:10.1126/science.1220869.

[FSM19] Maxwell Fulford, Matteo Salvalaglio, and Carla Molteni. DeepIce: a Deep Neural Network Approach to Identify Ice and Water Molecules. *Journal of Chemical Information and Modeling*, page acs.jcim.9b00005, 3 2019. doi:10.1021/acs.jcim.9b00005.

[GNA+15] Jens Glaser, Trung Dac Nguyen, Joshua A. Anderson, Pak Lui, Filippo Spiga, Jaime A. Millan, David C. Morse, and Sharon C. Glotzer. Strong scaling of general-purpose molecular dynamics simulations on GPUs. *Computer Physics Communications*, 192:97–107, 2015. doi:10.1016/j.cpc.2015.02.028.

[GS07] Sharon C. Glotzer and Michael J. Solomon. Anisotropy of building blocks and their assembly into complex structures. *Nature Materials*, 6:557–562, Aug 2007. URL: https://doi.org/10.1038/nmat1949.

[HMA+15] Eric S. Harper, Ryan L. Marson, Joshua A. Anderson, Greg van Anders, and Sharon C. Glotzer. Shape allophiles improve entropic assembly. *Soft Matter*, 11(37):7250–7256, 9 2015. doi:10.1039/C5SM01351H.

[HWG19] Eric S. Harper, Brendon Waters, and Sharon C. Glotzer. Hierarchical self-assembly of hard cube derivatives. *Soft Matter*, 15:3733–3739, 2019. doi:10.1039/C8SM02619J.

[Int18] Intel. Intel Threading Building Blocks, 2018. URL: https://www.threadingbuildingblocks.org/.

[JOPo01] Eric Jones, Travis Oliphant, Pearu Peterson, and others. SciPy: Open source scientific tools for Python, 2001. URL: https://www.scipy.org/.

[KGG16] Andrew S. Karas, Jens Glaser, and Sharon C. Glotzer. Using depletion to control colloidal crystal assemblies of hard cuboctahedra. *Soft Matter*, 12(23):5199–5204, 6 2016. doi:10.1039/C6SM00620E.

[LD08] Wolfgang Lechner and Christoph Dellago. Accurate determination of crystal structures based on averaged local bond order parameters. *Journal of Chemical Physics*, 129(11), 2008. doi:10.1063/1.2977970.

[MADWB11] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. *Journal of Computational Chemistry*, 32(10):2319–2327, 7 2011. doi:10.1002/jcc.21787.

[MBH+15] Robert T. McGibbon, Kyle A. Beauchamp, Matthew P. Harrigan, Christoph Klein, Jason M. Swails, Carlos X. Hernández, Christian R. Schwantes, Lee-Ping Wang, Thomas J. Lane, and Vijay S. Pande. MDTraj: A Modern Open Library for the Analysis of Molecular Dynamics Trajectories. *Biophysical Journal*, 109(8):1528–1532, 10 2015. doi:10.1016/J.BPJ.2015.08.015.

[MH18] Leland McInnes and John Healy. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. Feb 2018. arXiv:1802.03426.

[Oli06] Travis E. Oliphant. *A guide to NumPy*. Trelgol Publishing, 2006.

[Pli95] Steve Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, Mar 1995. doi:10.1006/JCPH.1995.1039.

[RDH+19] Vyas Ramasubramani, Bradley D. Dice, Eric S. Harper, Matthew P Spellings, Joshua A. Anderson, and Sharon C. Glotzer. freud: A Software Suite for High Throughput Analysis of Particle Simulation Data. June 2019. arXiv:1906.06317.

[SCKL15] S S Schoenholz, E D Cubuk, E Kaxiras, and a J Liu. A structural approach to relaxation in glassy liquids. *Nature Physics*, (February):1–11, 2015. doi:10.1038/nphys3644.

[SG18] Matthew Spellings and Sharon C. Glotzer. Machine learning for crystal identification and discovery. *AIChE Journal*, 64(6):2198–2206, 6 2018. doi:10.1002/aic.16157.

[SNR83] Paul J. Steinhardt, David R Nelson, and Marco Ronchetti. Bond-orientational order in liquids and glasses. *Physical Review B*, 28(2), 1983.

[Stu10] Alexander Stukowski. Visualization and analysis of atomistic simulation data with OVITO–the Open Visualization Tool. *Modelling and Simulation in Materials Science and Engineering*, 18(1):015012, 1 2010. doi:10.1088/0965-0393/18/1/015012.

[SZR+19] Anna J Simon, Yi Zhou, Vyas Ramasubramani, Jens Glaser, Arti Pothukuchy, Jimmy Gollihar, Jillian C. Gerberich, Janelle Leggere, Barrett R Morrow, Cheulhee Jung, Sharon C Glotzer, David W Taylor, and Andrew D Ellington. Supercharging enables organized assembly of synthetic biomolecules. *Nature Chemistry*, 11:204–212, 2019. doi:10.1038/s41557-018-0196-3.

[TCLC11] Shawn J. Tan, Michael J. Campolongo, Dan Luo, and Wenlong Cheng. Building plasmonic nanostructures with DNA. *Nature Nanotechnology*, 6(5):268–276, 5 2011. doi:10.1038/nnano.2011.49.

[TvAG19] Erin G. Teich, Greg van Anders, and Sharon C. Glotzer. Identity crisis in alchemical space drives the entropic colloidal glass transition. *Nature Communications*, 10(1):64, 12 2019. doi:10.1038/s41467-018-07977-2.

[vAAS+14] Greg van Anders, N. Khalid Ahmed, Ross Smith, Michael Engel, and Sharon C. Glotzer. Entropically patchy particles: Engineering valence through shape entropy. *ACS Nano*, 8(1):931–940, 2014. doi:10.1021/nn4057353.

[vAKA+14] Greg van Anders, Daphne Klotsa, N. Khalid Ahmed, Michael Engel, and Sharon C. Glotzer. Understanding shape entropy through local dense packing. *Proceedings of the National Academy of Sciences*, 111(45):E4812–E4821, 2014. doi:10.1073/pnas.1418159111.

[vdMH08] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.