freud Documentation

Release 2.4.0

The Regents of the University of Michigan

Nov 10, 2020

GETTING STARTED

CHAPTER

ONE

OVERVIEW

The **freud** Python library provides a simple, flexible, powerful set of tools for analyzing trajectories obtained from molecular dynamics or Monte Carlo simulations. High performance, parallelized C++ is used to compute standard tools such as radial distribution functions, correlation functions, order parameters, and clusters, as well as original analysis methods including potentials of mean force and torque (PMFTs) and local environment matching. The **freud** library supports many input formats and outputs NumPy arrays, enabling integration with the scientific Python ecosystem for many typical materials science workflows.

CHAPTER

TWO

RESOURCES

- Reference Documentation: Examples, tutorials, topic guides, and package Python APIs.
- Installation Guide: Instructions for installing and compiling **freud**.
- freud-users Google Group: Ask questions to the **freud** user community.
- GitHub repository: Download the **freud** source code.
- Issue tracker: Report issues or request features.

CHAPTER

THREE

CITATION

When using **freud** to process data for publication, please use this citation.

CHAPTER

FOUR

INSTALLATION

The easiest ways to install **freud** are using pip:

```
pip install freud-analysis
```

or conda:

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

freud is also available via containers for Docker or Singularity. If you need more detailed information or wish to install **freud** from source, please refer to the Installation Guide to compile **freud** from source.

CHAPTER

FIVE

EXAMPLES

The **freud** library is called using Python scripts. Many core features are demonstrated in the freud documentation. The examples come in the form of Jupyter notebooks, which can also be downloaded from the freud examples repository or launched interactively on Binder. Below is a sample script that computes the radial distribution function for a simulation run with HOOMD-blue and saved into a GSD file.

```
import freud
import gsd.hoomd

# Create a freud compute object (RDF is the canonical example)
rdf = freud.density.RDF(bins=50, r_max=5)

# Load a GSD trajectory (see docs for other formats)
traj = gsd.hoomd.open('trajectory.gsd', 'rb')
for frame in traj:
    rdf.compute(system=frame, reset=False)

# Get bin centers, RDF data from attributes
r = rdf.bin_centers
y = rdf.rdf
```

CHAPTER

SIX

SUPPORT AND CONTRIBUTION

Please visit our repository on GitHub for the library source code. Any issues or bugs may be reported at our issue tracker, while questions and discussion can be directed to our user forum. All contributions to **freud** are welcomed via pull requests!

CHAPTER

SEVEN

TABLE OF CONTENTS

7.1 Introduction

The **freud** library is a Python package for analyzing particle simulations. The package is designed to directly use numerical arrays of data, making it easy to use for a wide range of use-cases. The most common use-case of **freud** is for computing quantities from molecular dynamics simulation trajectories, but it can be used for analyzing any type of particle simulation. By operating directly on numerical arrays of data, **freud** allows users to parse custom simulation outputs into a suitable structure for input, rather than relying specific file types or data structures.

The core of **freud** is analysis of periodic systems, which are represented through the `freud.box.Box` class. The `freud.box.Box` supports arbitrary triclinic systems for maximum flexibility, and is used throughout the package to ensure consistent treatment of these systems. The package's many methods are encapsulated in various *compute classes*, which perform computations and populate class attributes for access. Of particular note are the various computations based on nearest neighbor finding in order to characterize particle environments. Such methods are simplified and accelerated through a centralized neighbor finding interface defined in the `freud.locality.NeighborQuery` family of classes in the `freud.locality` module of freud.

7.2 Installation

7.2.1 Installing freud

The **freud** library can be installed via conda or pip, or compiled from source.

Install via conda

The code below will install **freud** from conda-forge.

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

Install via pip

The code below will install **freud** from PyPI.

```
pip install freud-analysis
```

Compile from source

The following are **required** for installing **freud**:

- A C++14-compliant compiler
- Python (>=Python 3.6)
- NumPy
- Intel Threading Building Blocks
- Cython (>=0.29)
- scikit-build (>=0.10.0)
- CMake (>=3.3.0)

Note: Depending on the generator you are using, you may require a newer version of CMake. In particular, on Windows Visual Studio 2017 requires at least CMake 3.7.1, while Visual Studio 2019 requires CMake 3.14. For more information on specific generators, see the CMake generator documentation.

The **freud** library uses scikit-build and CMake to handle the build process itself, while the other requirements are required for compiling code in **freud**. These requirements can be met by installing the following packages from the conda-forge channel:

```
conda install -c conda-forge tbb tbb-devel numpy cython scikit-build cmake
```

All requirements other than TBB can also be installed via the Python Package Index

```
pip install numpy cython scikit-build cmake
```

Wheels for tbb and tbb-devel exist on PyPI, but only for certain operating systems, so your mileage may vary. For non-conda users, we recommend using OS-specific package managers (e.g. Homebrew for Mac) to install TBB. As in the snippets above, it may be necessary to install both both a TBB and a "devel" package in order to get both the headers and the shared libraries.

The code that follows builds **freud** and installs it for all users (append `--user` if you wish to install it to your user site directory):

```
git clone --recurse-submodules https://github.com/glotzerlab/freud.git
cd freud
python setup.py install
```

You can also build **freud** in place so that you can run from within the folder:

```
# Run tests from the tests directory
python setup.py build_ext --inplace
```

Building **freud** in place has certain advantages, since it does not affect your Python behavior except within the **freud** directory itself (where **freud** can be imported after building). Additionally, due to limitations inherent to the distutils/setuptools infrastructure, building extension modules can only be parallelized using the build_ext subcommand of

setup.py, not with install. As a result, it will be faster to manually run build_ext and then install (which normally calls build_ext under the hood anyway) the built packages.

CMake Options

The scikit-build tool allows setup.py to accept three different sets of options separated by `--`, where each set is provided directly to scikit-build, to CMake, or to the code generator of choice, respectively. For example, the command `python setup.py build_ext --inplace -- -DCOVERAGE=ON -G Ninja -- -j 4` tell scikit-build to perform an in-place build, it tells CMake to turn on the `COVERAGE` option and use Ninja for compilation, and it tells Ninja to compile with 4 parallel threads. For more information on these options, see the scikit-build docs.

In addition to standard CMake flags, the following CMake options are available for **freud**:

--COVERAGE Build the Cython files with coverage support to check unit test coverage.

The **freud** CMake configuration also respects the following environment variables (in addition to standards like `LD_LIBRARY_PATH`).

TBB_ROOT The root directory where TBB is installed. Useful if TBB is installed in a non-standard location or cannot be located by Python for some other reason.

TBB_INCLUDE The directory where the TBB headers (e.g. `tbb.h`) are located. Useful if TBB is installed in a non-standard location or cannot be located by Python for some other reason.

TBB_LINK The directory where the TBB shared library (e.g. `libtbb.so` or `libtbb.dylib`) is located. Useful if TBB is installed in a non-standard location or cannot be located by Python for some other reason.

Note: **freud** makes use of git submodules. If you ever wish to manually update these, you can execute:

```
git submodule update --init
```

7.2.2 Unit Tests

The unit tests for **freud** are included in the repository and are configured to be run using the Python `unittest` library:

```
# Run tests from the tests directory
cd tests
python -m unittest discover .
```

Note that because **freud** is designed to require installation to run (i.e. it cannot be run directly out of the build directory), importing **freud** from the root of the repository will fail because it will try and import the package folder. As a result, unit tests must be run from outside the root directory if you wish to test the installed version of **freud**. If you want to run tests within the root directory, you can instead build **freud** in place:

```
# Run tests from the tests directory
python setup.py build_ext --inplace
```

This build will place the necessary files alongside the **freud** source files so that **freud** can be imported from the root of the repository.

7.2.3 Documentation

The documentation for **freud** is hosted online at ReadTheDocs. You may also build the documentation yourself.

Building the documentation

The following are **required** for building **freud** documentation:

- Sphinx
- Read the Docs Sphinx Theme
- nbsphinx
- jupyter_sphinx
- sphinxcontrib-bibtex

You can install these dependencies using conda:

```
conda install -c conda-forge sphinx sphinx_rtd_theme nbsphinx jupyter_sphinx␣
↪sphinxcontrib-bibtex
```

or pip:

```
pip install sphinx sphinx-rtd-theme nbsphinx jupyter-sphinx sphinxcontrib-bibtex
```

To build the documentation, run the following commands in the source directory:

```
cd doc
make html
# Then open build/html/index.html
```

To build a PDF of the documentation (requires LaTeX and/or PDFLaTeX):

```
cd doc
make latexpdf
# Then open build/latex/freud.pdf
```

7.3 Quickstart Guide

Once you have *installed freud*, you can start using **freud** with any simulation data that you have on hand. As an example, we'll assume that you have run a simulation using the HOOMD-blue and used the `hoomd.dump.gsd` command to output the trajectory into a file `trajectory.gsd`. The GSD file format provides its own convenient Python file reader that offers access to data in the form of NumPy arrays, making it immediately suitable for calculation with **freud**. Many other file readers and data formats are supported, see *Reading Simulation Data for freud* for a full list and more examples.

We start by reading the data into a NumPy array:

```
import gsd.hoomd
traj = gsd.hoomd.open('trajectory.gsd', 'rb')
```

We can now immediately calculate important quantities. Here, we will compute the radial distribution function $g(r)$ using the `freud.density.RDF` compute class. Since the radial distribution function is in practice computed as a histogram, we must specify the histogram bin widths and the largest interparticle distance to include in our calculation.

To do so, we simply instantiate the class with the appropriate parameters and then perform a computation on the given data:

```
import freud
rdf = freud.density.RDF(bins=50, r_max=5)
rdf.compute(system=traj[-1])
```

We can now access the data through properties of the `rdf` object.

```
r = rdf.bin_centers
y = rdf.rdf
```

Many classes in **freud** natively support plotting their data using *Matplotlib <https://matplotlib.org/>*:

```
import matplotlib as plt
fig, ax = plt.subplots()
rdf.plot(ax=ax)
```

You will note that in the above example, we computed $g(r)$ only using the final frame of the simulation trajectory, `traj[-1]`. However, in many cases, radial distributions and other similar quantities may be noisy in simulations due to the natural fluctuations present. In general, what we are interested in are *time-averaged* quantities once a system has equilibrated. To perform such a calculation, we can easily modify our original calculation to take advantage of **freud**'s *accumulation* features. To accumulate, just add the argument `reset=False` with a supported compute object (such as histogram-like computations). Assuming that you have some method for identifying the frames you wish to include in your sample, our original code snippet would be modified as follows:

```
import freud
rdf = freud.density.RDF(bins=50, r_max=5)
for frame in traj:
    rdf.compute(frame, reset=False)
```

You can then access the data exactly as we previously did. And that's it!

Now that you've seen a brief example of reading data and computing a radial distribution function, you're ready to learn more. If you'd like a complete walkthrough please see the *Tutorial*. The tutorial walks through many of the core concepts in **freud** in greater detail, starting with the basics of the simulation systems we analyze and describing the details of the neighbor finding logic in **freud**. To see specific features of **freud** in action, look through the *Examples*. More detailed documentation on specific classes and functions can be found in the API documentation.

7.4 Tutorial

This tutorial provides a complete introduction to **freud**. Rather than attempting to touch on all features in **freud**, it focuses on common core concepts that will help understand how **freud** works with data and exposes computations to the user. The tutorial begins by introducing the fundamental concepts of periodic systems as implemented in **freud** and the concept of `Compute classes`, which consitute the primary API for performing calculations with **freud**. The tutorial then discusses the most common calculation performed in **freud**, finding neighboring points in periodic systems. The package's neighbor finding tools are tuned for high performance neighbor finding, which is what enables most of other calculations in **freud**, which typically involve characterizing local environments of points in some way. The next part of the tutorial discusses the role of histograms in **freud**, focusing on the common features and properties that all histograms share. Finally, the tutorial includes a few more complete demonstrations of using **freud** that should provide reasonable templates for use with almost any other features in **freud**.

7.4.1 Periodic Boundary Conditions

The central goal of **freud** is the analysis of simulations performed in periodic boxes. Periodic boundary conditions are ubiquitous in simulations because they permit the simulation of quasi-infinite systems with minimal computational effort. As long as simulation systems are sufficiently large, i.e. assuming that points in the system experience correlations over length scales substantially smaller than the system length scale, periodic boundary conditions ensure that the system appears effectively infinite to all points.

In order to consistently define the geometry of a simulation system with periodic boundaries, **freud** defines the `freud.box.Box` class. The class encapsulates the concept of a triclinic simulation box in a right-handed coordinate system. Triclinic boxes are defined as parallelepipeds: three-dimensional polyhedra where every face is a parallelogram. In general, any such box can be represented by three distinct, linearly independent box vectors. Enforcing the requirement of right-handedness guarantees that the box can be represented by a matrix of the form

$$\mathbf{h} = \begin{pmatrix} L_x & xyL_y & xzL_z \\ 0 & L_y & yzL_z \\ 0 & 0 & L_z \end{pmatrix}$$

where each column is one of the box vectors.

Note: All **freud** boxes are centered at the origin, so for a given box the range of possible positions is $[-L/2, L/2)$.

As such, the box is characterized by six parameters: the box vector lengths L_x, L_y, and L_z, and the tilt factors xy, xz, and yz. The tilt factors are directly related to the angles between the box vectors. All computations in **freud** are built around this class, ensuring that they naturally handle data from simulations conducted in non-cubic systems. There is also native support for two-dimensional (2D) systems when setting $L_z = 0$.

Boxes can be constructed in a variety of ways. For simple use-cases, one of the factory functions of the `freud.box.Box` provides the easiest possible interface:

```
# Make a 10x10 square box (for 2-dimensional systems).
freud.box.Box.square(10)

# Make a 10x10x10 cubic box.
freud.box.Box.cube(10)
```

For more complex use-cases, the `freud.box.Box.from_box()` method provides an interface to create boxes from any object that can easily be interpreted as a box.

```
# Create a 10x10 square box from a list of two items.
freud.box.Box.from_box([10, 10])

# Create a 10x10x10 cubic box from a list of three items.
freud.box.Box.from_box([10, 10, 10])

# Create a triclinic box from a list of six items (including tilt factors).
freud.box.Box.from_box([10, 5, 2, 0.1, 0.5, 0.7])

# Create a triclinic box from a dictionary.
freud.box.Box.from_box(dict(Lx=8, Ly=7, Lz=10, xy=0.5, xz=0.7, yz=0.2))

# Directly call the constructor.
freud.box.Box(Lx=8, Ly=7, Lz=10, xy=0.5, xz=0.7, yz=0.2, dimensions=3)
```

More examples on how boxes can be created may be found in the API documentation of the `Box` class.

7.4.2 Compute Classes

Calculations in **freud** are built around the concept of `Compute classes`, Python objects that encode a given method and expose it through a `compute` method. In general, these methods operate on a system composed of a triclinic box and a NumPy array of particle positions. The box can be provided as any object that can be interpreted as a **freud** box (as demonstrated in the examples above). We can look at the `freud.order.Hexatic` order parameter calculator as an example:

```
import freud
positions = ...  # Read positions from trajectory file.
op = freud.order.Hexatic(k=6)
op.compute(
    system=({'Lx': 5, 'Ly': 5, 'dimensions': 2}, positions),
    neighbors=dict(r_max=3)
)

# Plot the value of the order parameter.
from matplotlib import pyplot as plt
plt.hist(np.absolute(op.particle_order))
```

Here, we are calculating the hexatic order parameter, then using Matplotlib to plot. The `freud.order.Hexatic` class constructor accepts a single argument `k`, which represents the periodicity of the calculation. If you're unfamiliar with this order parameter, the most important piece of information here is that many compute methods in **freud** require parameters that are provided when the `Compute class` is constructed.

To calculate the order parameter we call `compute`, which takes two arguments, a `tuple` *(box, points)* and a `dict`. We first focus on the first argument. The `box` is any object that can be coerced into a `freud.box.Box` as described in the previous section; in this case, we use a dictionary to specify a square (2-dimensional) box. The `points` must be anything that can be coerced into a 2-dimensional NumPy array of shape `(N, 3)` In general, the points may be provided as anything that can be interpreted as an $N \times 3$ list of positions; for more details on valid inputs here, see `numpy.asarray()`. Note that because the hexatic order parameter is designed for two-dimensional systems, the points must be provided of the form *[x, y, 0]* (i.e. the z-component must be 0). We'll go into more detail about the `(box, points)` tuple soon, but for now, it's sufficient to just think of it as specifying the system of points we want to work with.

Now let's return to the `neighbors` argument to `compute`, which is a dictionary is used to determine which particle neighbors to use. Many computations in **freud** (such as the hexatic order parameter) involve the bonds in the system (for example, the average length of bonds or the average number of bonds a given point has). However, the concept of a bond is sufficiently variable between different calculations; for instance, should points be considered bonded if they are within a certain distance of each other? Should every point be considered bonded to a fixed number of other points?

To accommodate this variability, **freud** offers a very general framework by which bonds can be specified, and we'll go into more details in the next section. In the example above, we've simply informed the `Hexatic` class that we want it to define bonds as pairs of particles that are less than 3 distance units apart. We then access the computed order parameter as `op.particle_order` (we use `np.absolute()` because the output is a complex number and we just want to see its magnitude).

Accessing Computed Properties

In general, `Compute classes` expose their calculations using properties. Any parameters to the `Compute` object (e.g. `k` in the above example) can typically be accessed as soon as the object is constructed:

```
op = freud.order.Hexatic(k=6)
op.k
```

Computed quantities can also be accessed in a similar manner, but only after the `compute` method is called. For example:

```
op = freud.order.Hexatic(k=6)

# This will raise an exception.
op.particle_order

op.compute(
    system=({'Lx': 5, 'Ly': 5, 'dimensions': 2}, positions),
    neighbors=dict(r_max=3)
)

# Now you can access this.
op.particle_order
```

Note: Most (but not all) of **freud**'s `Compute classes` are Python wrappers around high-performance implementations in C++. As a result, none of the data or the computations is actually stored in the Python object. Instead, the Python object just stores an instance of the C++ object that actually owns all its data, performs calculations, and returns computed quantities to the user. Python properties provide a nice way to hide this logic so that the Python code involves just a few lines.

Compute objects is that they can be used many times to calculate quantities, and the most recently calculated output can be accessed through the property. If you need to perform a series of calculations and save all the data, you can also easily do that:

```
# Recall that lists of length 2 automatically convert to 2D freud boxes.
box = [5, 5]

op = freud.order.Hexatic(k=6)

# Assuming that we have a list of Nx3 NumPy arrays that represents a
# simulation trajectory, we can loop over it and calculate the order
# parameter values in sequence.
trajectory  = ...  # Read trajectory file into a list of positions by frame.
hexatic_values = []
for points in trajectory:
    op.compute(system=(box, points), neighbors=dict(r_max=3))
    hexatic_values.append(op.particle_order)
```

To make using **freud** as simple as possible, all `Compute classes` are designed to return `self` when compute is called. This feature enables a very concise *method-chaining* idiom in **freud** where computed properties are accessed immediately:

```
particle_order = freud.order.Hexatic(k=6).compute(
    system=(box, points)).particle_order
```

7.4.3 Finding Neighbors

Now that you've been introduced to the basics of interacting with **freud**, let's dive into the central feature of **freud**: efficiently and flexibly finding neighbors in periodic systems.

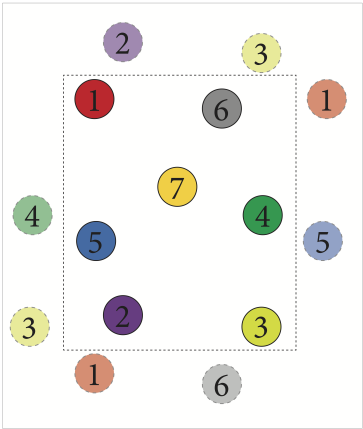
Problem Statement

Neighbor-Based Calculations

As discussed in *the previous section*, a central task in many of the computations in **freud** is finding particles' neighbors. These calculations typically only involve a limited subset of a particle's neighbors that are defined as characterizing its local environment. This requirement is analogous to the force calculations typically performed in molecular dynamics simulations, where a cutoff radius is specified beyond which pair forces are assumed to be small enough to neglect. Unlike in simulation, though, many analyses call for different specifications than simply selecting all points within a certain distance.

An important example is the calculation of order parameters, which can help characterize phase transitions. Such parameters can be highly sensitive to the precise way in which neighbors are selected. For instance, if a hard distance cutoff is imposed in finding neighbors for the hexatic order parameter, a particle may only be found to have five neighbors when it actually has six neighbors except the last particle is slightly outside the cutoff radius. To accomodate such differences in a flexible manner, **freud** allows users to specify neighbors in a variety of ways.

Finding Periodic Neighbors



Finding neighbors in periodic systems is significantly more challenging than in aperiodic systems. To illustrate the difference, consider the figure above, where the black dashed line indicates the boundaries of the system. If this system were aperiodic, the three nearest neighbors for point 1 would be points 5, 6, and 7. However, due to periodicity, point 2 is actually closer to point 1 than any of the others if you consider moving straight through the top (or equivalently, the bottom) boundary. Although many tools provide efficient implementations of algorithms for finding neighbors in aperiodic systems, they seldom generalize to periodic systems. Even more rare is the ability to work not just in *cubic* periodic systems, which are relatively tractable, but in arbitrary triclinic geometries as described in *Periodic Boundary Conditions*. This is precisely the type of calculation **freud** is designed for.

Neighbor Querying

To understand how `Compute` classes find neighbors in **freud**, it helps to start by learning about **freud**'s neighbor finding classes directly. Note that much more detail on this topic is available in the *Query API* topic guide; in this section we will restrict ourselves to a higher-level overview. For our demonstration, we will make use of the `freud.locality.AABBQuery` class, which implements one fast method for periodic neighbor finding. The primary mode of interfacing with this class (and other neighbor finding classes) is through the `query` interface.

```
import numpy as np
import freud

# As an example, we randomly generate 100 points in a 10x10x10 cubic box.
L = 10
num_points = 100

# We shift all points into the expected range for freud.
points = np.random.rand(num_points)*L - L/2
box = freud.box.Box.cube(L)
aq = freud.locality.AABBQuery(box, points)

# Now we generate a smaller sample of points for which we want to find
# neighbors based on the original set.
query_points = np.random.rand(num_points/10)*L - L/2
distances = []

# Here, we ask for the 4 nearest neighbors of each point in query_points.
for bond in aq.query(query_points, dict(num_neighbors=4)):
    # The returned bonds are tuples of the form
    # (query_point_index, point_index, distance). For instance, a bond
    # (1, 3, 0.2) would indicate that points[3] was one of the 4 nearest
    # neighbors for query_points[1], and that they are separated by a
    # distance of 0.2
    # (i.e. np.linalg.norm(query_points[1] - points[3]) == 2).
    distances.append(bond[2])

avg_distance = np.mean(distances)
```

Let's dig into this script a little bit. Our first step is creating a set of 100 points in a cubic box. Note that the shifting done in the code above could also be accomplished using the `Box.wrap` method like so: `box.wrap(np.random.rand(num_points)*L)`. The result would appear different, because if plotted without considering periodicity, the points would range from `-L/2` to `L/2` rather than from 0 to `L`. However, these two sets of points would be equivalent in a periodic system.

We then generate an additional set of `query_points` and ask for neighbors using the `query` method. This function accepts two arguments: a set of points, and a `dict` of **query arguments**. Query arguments are a central concept in **freud** and represent a complete specification of the set of neighbors to be found. In general, the most common forms of queries are those requesting either a fixed number of neighbors, as in the example above, or those requesting all neighbors within a specific distance. For example, if we wanted to rerun the above example but instead find all bonds of length less than or equal to 2, we would simply replace the for loop above with:

```
for bond in aq.query(query_points, dict(r_max=2)):
    distances.append(bond[2])
```

Query arguments constitute a powerful method for specifying a query request. Many query arguments may be combined for more specific purposes. A common use-case is finding all neighbors within a single set of points (i.e. setting `query_points = points` in the above example). In this situation, however, it is typically not useful for a point to find itself as a neighbor since it is trivially the closest point to itself and falls within any cutoff radius. To avoid this,

we can use the `exclude_ii` query argument:

```
query_points = points
for bond in aq.query(query_points, dict(num_neighbors=4, exclude_ii=True)):
    pass
```

The above example will find the 4 nearest neighbors to each point, excepting the point itself. A complete description of valid query arguments can be found in *Query API*.

Neighbor Lists

Query arguments provide a simple but powerful language with which to express neighbor finding logic. Used in the manner shown above, `query` can be used to express many calculations in a very natural, Pythonic way. By itself, though, the API shown above is somewhat restrictive because the output of `query` is a generator. If you aren't familiar with generators, the important thing to know is that they can be looped over, *but only once*. Unlike objects like lists, which you can loop over as many times as you like, once you've looped over a generator once, you can't start again from the beginning.

In the examples above, this wasn't a problem because we simply iterated over the bonds once for a single calculation. However, in many practical cases we may need to reuse the set of neighbors multiple times. A simple solution would be to simply to store the bonds into a list as we loop over them. However, because this is such a common use-case, **freud** provides its own containers for bonds: the `freud.locality.NeighborList`.

Queries can easily be used to generate `NeighborList` objects using their `toNeighborList` method:

```
query_result = aq.query(query_points, dict(num_neighbors=4, exclude_ii))
nlist = query_result.toNeighborList()
```

The resulting object provides a persistent container for bond data. Using `NeighborLists`, our original example might instead look like this:

```
import numpy as np
import freud

L = 10
num_points = 100

points = np.random.rand(num_points)*L - L/2
box = freud.box.Box.cube(L)
aq = freud.locality.AABBQuery(box, points)

query_points = np.random.rand(num_points/10)*L - L/2
distances = []

# Here, we ask for the 4 nearest neighbors of each point in query_points.
query_result = aq.query(query_points, dict(num_neighbors=4)):
nlist = query_result.toNeighborList()
for (i, j) in nlist:
    # Note that we have to wrap the bond vector before taking the norm;
    # this is the simplest way to compute distances in a periodic system.
    distances.append(np.linalg.norm(box.wrap(query_points[i] - points[j])))

avg_distance = np.mean(distances)
```

Note that in the above example we looped directly over the `nlist` and recomputed distances. However, the `query_result` contained information about distances: here's how we access that through the `nlist`:

```
assert np.all(nlist.distances == distances)
```

The indices are also accessible through properties, or through a NumPy-like slicing interface:

```
assert np.all(nlist.query_point_indices == nlist[:, 0])
assert np.all(nlist.point_indices == nlist[:, 1])
```

Note that the `query_points` are always in the first column, while the `points` are in the second column. `freud.locality.NeighborList` objects also store other properties; for instance, they may assign different weights to different bonds. This feature can be used by, for example, `freud.order.Steinhardt`, which is typically used for calculating Steinhardt order parameters, a standard tool for characterizing crystalline order. When provided appropriately weighted neighbors, however, the class instead computes Minkowski structure metrics, which are much more sensitive measures that can differentiate a wider array of crystal structures.

7.4.4 Pair Computations

Some computations in **freud** do not depend on bonds at all. For example, `freud.density.GaussianDensity` creates a "continuous equivalent" of a system of points by placing normal distributions at each point's location to smear out its position, then summing the value of these distributions at a set of fixed grid points. This calculation can be quite useful because it allows the application of various analysis tools like fast Fourier transforms, which require regular grids. For the purposes of this tutorial, however, the importance of this class is that it is an example of a calculation where neighbors are unimportant: the calculation is performed on a per-point basis only.

The much more common pattern in **freud**, though, is that calculations involve the local neighborhoods of points. To support efficient, flexible computations of such quantities, various `Compute classes` essentially expose the same API as the query interface demonstrated in the previous section. These `PairCompute classes` are designed to mirror the querying functionality of **freud** as closely as possible.

As an example, let's consider `freud.density.LocalDensity`, which calculates the density of points in the local neighborhood of each point. Adapting our code from the previous section, the simplest usage of this class would be as follows:

```
import numpy as np
import freud

L = 10
num_points = 100

points = np.random.rand(num_points)*L - L/2
box = freud.box.Box.cube(L)

# r_max specifies how far to search around each point for neighbors
r_max = 2

# For systems where the points represent, for instance, particles with a
# specific size, the diameter is used to add fractional volumes for
# neighbors that would be overlapping the sphere of radius r_max around
# each point.
diameter = 0.001

ld = freud.density.LocalDensity(r_max, diameter)
ld.compute(system=(box, points))

# Access the density.
ld.density
```

Using the same example system we've been working with so far, we've now calculated an estimate for the number of points in the neighborhood of each point. Since we already told the computation how far to search for neighbors based on `r_max`, all we had to do was pass a `tuple (box, points)` to compute indicate where the points were located.

Binary Systems

Imagine that instead of a single set of points, we actually had two different types of points and we were interested in finding the density of one set of points in the vicinity of the other. In that case, we could modify the above calculation as follows:

```
import numpy as np
import freud
L = 10
num_points = 100
points = np.random.rand(num_points)*L - L/2
query_points = np.random.rand(num_points/10)*L - L/2

r_max = 2
diameter = 0.001

ld = freud.density.LocalDensity(r_max, diameter)
ld.compute(system=(box, points), query_points=query_points)

# Access the density.
ld.density
```

The choice of names names here is suggestive of exactly what this calculation is now doing. Internally, `freud.density.LocalDensity` will search for all `points` that are within the cutoff distance `r_max` of every `query_point` (essentially using the query interface we introduced previously) and use that to calculate `ld.density`. Note that this means that `ld.density` now contains densities for every `query_point`, i.e. it is of length 10, not 100. Moreover, recall that one of the features of the querying API is the specification of whether or not to count particles as their own neighbors. `PairCompute classes` will attempt to make an intelligent determination of this for you; if you do not pass in a second set of `query_points`, they will assume that you are computing with a single set of points and automatically exclude self-neighbors, but otherwise all neighbors will be included.

So far, we have included all points within a fixed radius; however, one might instead wish to consider the density in some shell, such as the density between 1 and 2 distance units away. To address this need, you could simply adapt the call to `compute` above as follows:

```
ld.compute(system=(box, points), query_points=query_points,
           neighbors=dict(r_max=2, r_min=1))
```

The `neighbors` argument to `PairCompute` classes allows users to specify arbitary query arguments, making it possible to easily modify **freud** calculations on-the-fly. The `neighbors` argument is actually more general than query arguments you've seen so far: if query arguments are not precise enough to specify the exact set of neighbors you want to compute with, you can instead provide a `NeighborList` directly

```
ld.compute(system=(box, points), query_points=query_points,
           neighbors=nlist)
```

This feature allows users essentially arbitrary flexibility to specify the bonds that should be included in any bond-based computation. A common use-case for this is constructing a `NeighborList` using `freud.locality.Voronoi`; Voronoi constructions provide a powerful alternative method of defining neighbor relationships that can improve the accuracy and robustness of certain calculations in **freud**.

You may have noticed in the last example that all the arguments are specified using keyword arguments. As the previous examples have attempted to show, the `query_points` argument defines a second set of points to be used

when performing calculations on binary systems, while the `neighbors` argument is how users can specify which neighbors to consider in the calculation.

The `system` argument is what, to this point, we have been specifying as a `tuple (box, points)`. However, we don't have to use this tuple. Instead, we can pass in any `freud.locality.NeighborQuery`, the central class in **freud**'s querying infrastructure. In fact, you've already seen examples of `freud.locality.NeighborQuery`: the `freud.locality.AABBQuery` object that we originally used to find neighbors. There are also a number of other input types that can be converted via `freud.locality.NeighborQuery.from_system()`, see also *Reading Simulation Data for freud*. Since these objects all contain a `freud.box.Box` and a set of points, they can be directly passed to computations:

```
aq = freud.locality.AABBQuery(box, points)
ld.compute(system=aq, query_points=query_points, neighbors=nlist)
```

For more information on why you might want to use `freud.locality.NeighborQuery` objects instead of the tuples, see *Using freud Efficiently*. For now, just consider this to be a way in which you can simplify your calls to many **freud** computes in one script by storing `(box, points)` into another objects.

You've now covered the most important information needed to use **freud**! To recap, we've discussed how **freud** handles periodic boundary conditions, the structure and usage of `Compute classes`, and methods for finding and performing calculations with pairs of neighbors. For more detailed information on specific methods in **freud**, see the *Examples* page or look at the API documentation for specific modules.

7.5 Examples

Examples are provided as Jupyter notebooks in a separate freud-examples repository. These notebooks may be launched interactively on Binder or downloaded and run on your own system. Visualization of data is done via Matplotlib and Bokeh, unless otherwise noted.

7.5.1 Key concepts

There are a few critical concepts, algorithms, and data structures that are central to all of **freud**. The `freud.box.Box` class defines the concept of a periodic simulation box, and the `freud.locality` module defines methods for finding nearest neighbors of particles. Since both of these are used throughout **freud**, we recommend reading the *Tutorial* first, before delving into the workings of specific **freud** analysis modules.

freud.box.Box

In this notebook, we demonstrate the basic features of the `Box` class, including wrapping particles back into the box under periodic boundary conditions. For more information, see the introduction to Periodic Boundary Conditions and the `freud.box` documentation.

Creating a Box object

Boxes may be constructed explicitly using all arguments. Such construction is useful when performing *ad hoc* analyses involving custom boxes. In general, boxes are assumed to be 3D and orthorhombic unless otherwise specified.

[1]:
```
import freud.box

# All of the below examples are valid boxes.
box = freud.box.Box(Lx=5, Ly=6, Lz=7, xy=0.5, xz=0.6, yz=0.7, is2D=False)
box = freud.box.Box(1, 3, 2, 0.3, 0.9)
box = freud.box.Box(5, 6, 7)
box = freud.box.Box(5, 6, is2D=True)
box = freud.box.Box(5, 6, xy=0.5, is2D=True)
```

From another Box object

The simplest case is simply constructing one freud box from another.

Note that all forms of creating boxes aside from the explicit method above use methods defined within the Box class rather than attempting to overload the constructor itself.

[2]:
```
box = freud.box.Box(1, 2, 3)
box2 = freud.box.Box.from_box(box)
print("The original box: \n\t{}".format(box))
print("The copied box: \n\t{}\n".format(box2))

# Boxes are always copied by value, not by reference
box.Lx = 5
print("The original box is modified: \n\t{}".format(box))
print("The copied box is not: \n\t{}\n".format(box2))

# Note, however, that box assignment creates a new object that
# still points to the original box object, so modifications to
# one are visible on the other.
box3 = box2
print("The new copy: \n\t{}".format(box3))
box2.Lx = 2
print("The new copy after the original is modified: \n\t{}".format(box3))
print("The modified original box: \n\t{}".format(box2))
```

```
The original box:
        freud.box.Box(Lx=1.0, Ly=2.0, Lz=3.0, xy=0.0, xz=0.0, yz=0.0, is2D=False)
The copied box:
        freud.box.Box(Lx=1.0, Ly=2.0, Lz=3.0, xy=0.0, xz=0.0, yz=0.0, is2D=False)

The original box is modified:
        freud.box.Box(Lx=5.0, Ly=2.0, Lz=3.0, xy=0.0, xz=0.0, yz=0.0, is2D=False)
The copied box is not:
        freud.box.Box(Lx=1.0, Ly=2.0, Lz=3.0, xy=0.0, xz=0.0, yz=0.0, is2D=False)

The new copy:
        freud.box.Box(Lx=1.0, Ly=2.0, Lz=3.0, xy=0.0, xz=0.0, yz=0.0, is2D=False)
The new copy after the original is modified:
        freud.box.Box(Lx=2.0, Ly=2.0, Lz=3.0, xy=0.0, xz=0.0, yz=0.0, is2D=False)
The modified original box:
        freud.box.Box(Lx=2.0, Ly=2.0, Lz=3.0, xy=0.0, xz=0.0, yz=0.0, is2D=False)
```

From a matrix

A box can be constructed directly from the box matrix representation described above using the `Box.from_matrix` method.

[3]:
```
# Matrix representation. Note that the box vectors must represent
# a right-handed coordinate system! This translates to requiring
# that the matrix be upper triangular.
box = freud.box.Box.from_matrix([[1, 1, 0], [0, 1, 0.5], [0, 0, 0.5]])
print("This is a 3D box from a matrix: \n\t{}\n".format(box))

# 2D box
box = freud.box.Box.from_matrix([[1, 0, 0], [0, 1, 0], [0, 0, 0]])
print("This is a 2D box from a matrix: \n\t{}\n".format(box))

# Automatic matrix detection using from_box
box = freud.box.Box.from_box([[1, 1, 0], [0, 1, 0.5], [0, 0, 0.5]])
print("The box matrix was automatically detected: \n\t{}\n".format(box))

# Boxes can be numpy arrays as well
import numpy as np
box = freud.box.Box.from_box(np.array([[1, 1, 0], [0, 1, 0.5], [0, 0, 0.5]]))
print("Using a 3x3 numpy array: \n\t{}".format(box))
```

```
This is a 3D box from a matrix:
        freud.box.Box(Lx=1.0, Ly=1.0, Lz=0.5, xy=1.0, xz=0.0, yz=1.0, is2D=False)

This is a 2D box from a matrix:
        freud.box.Box(Lx=1.0, Ly=1.0, Lz=0.0, xy=0.0, xz=0.0, yz=0.0, is2D=True)

The box matrix was automatically detected:
        freud.box.Box(Lx=1.0, Ly=1.0, Lz=0.5, xy=1.0, xz=0.0, yz=1.0, is2D=False)

Using a 3x3 numpy array:
        freud.box.Box(Lx=1.0, Ly=1.0, Lz=0.5, xy=1.0, xz=0.0, yz=1.0, is2D=False)
```

From a namedtuple or dict

A box can be also be constructed from any object that provides an attribute for `Lx`, `Ly`, `Lz`, `xy`, `xz`, and `yz` (or some subset), such as a `namedtuple`. This method is suitable for passing in box objects constructed by some other program, for example.

[4]:
```
from collections import namedtuple
MyBox = namedtuple('mybox', ['Lx', 'Ly', 'Lz', 'xy', 'xz', 'yz'])

box = freud.box.Box.from_box(MyBox(Lx=5, Ly=3, Lz=2, xy=0, xz=0, yz=0))
print("Box from named tuple: \n\t{}\n".format(box))

box = freud.box.Box.from_box(MyBox(Lx=5, Ly=3, Lz=0, xy=0, xz=0, yz=0))
print("2D Box from named tuple: \n\t{}".format(box))
```

```
Box from named tuple:
        freud.box.Box(Lx=5.0, Ly=3.0, Lz=2.0, xy=0.0, xz=0.0, yz=0.0, is2D=False)

2D Box from named tuple:
        freud.box.Box(Lx=5.0, Ly=3.0, Lz=0.0, xy=0.0, xz=0.0, yz=0.0, is2D=True)
```

Similarly, construction is also possible using any object that supports key-value indexing, such as a dict.

[5]:
```
box = freud.box.Box.from_box(dict(Lx=5, Ly=3, Lz=2))
print("Box from dict: \n\t{}".format(box))
```

```
Box from dict:
        freud.box.Box(Lx=5.0, Ly=3.0, Lz=2.0, xy=0.0, xz=0.0, yz=0.0, is2D=False)
```

From a list

Finally, boxes can be constructed from any simple iterable that provides the elements in the correct order.

[6]:
```
box = freud.box.Box.from_box((5, 6, 7, 0.5, 0, 0.5))
print("Box from tuple: \n\t{}\n".format(box))

box = freud.box.Box.from_box([5, 6])
print("2D Box from list: \n\t{}".format(box))
```

```
Box from tuple:
        freud.box.Box(Lx=5.0, Ly=6.0, Lz=7.0, xy=0.5, xz=0.0, yz=0.5, is2D=False)

2D Box from list:
        freud.box.Box(Lx=5.0, Ly=6.0, Lz=0.0, xy=0.0, xz=0.0, yz=0.0, is2D=True)
```

Convenience APIs

We also provide convenience constructors for common geometries, namely square (2D) and cubic (3D) boxes.

[7]:
```
cube_box = freud.box.Box.cube(L=5)
print("Cubic Box: \n\t{}\n".format(cube_box))

square_box = freud.box.Box.square(L=5)
print("Square Box: \n\t{}".format(square_box))
```

```
Cubic Box:
        freud.box.Box(Lx=5.0, Ly=5.0, Lz=5.0, xy=0.0, xz=0.0, yz=0.0, is2D=False)

Square Box:
        freud.box.Box(Lx=5.0, Ly=5.0, Lz=0.0, xy=0.0, xz=0.0, yz=0.0, is2D=True)
```

Export

If you want to export or display the box, you can export box objects into their matrix or dictionary representations, which provide completely specified descriptions of the box.

[8]:
```
cube_box = freud.box.Box.cube(L=5)
cube_box.to_matrix()
```

[8]:
```
array([[5., 0., 0.],
       [0., 5., 0.],
       [0., 0., 5.]])
```

[9]:
```
cube_box.to_dict()
```

[9]:
```
{'Lx': 5.0,
 'Ly': 5.0,
 'Lz': 5.0,
 'xy': 0.0,
 'xz': 0.0,
 'yz': 0.0,
 'dimensions': 3}
```

Using boxes

Given a freud box object, you can query it for all its attributes.

[10]:
```
box = freud.box.Box.from_matrix([[10, 0, 0], [0, 10, 0], [0, 0, 10]])
print("L_x = {}, L_y = {}, L_z = {}, xy = {}, xz = {}, yz = {}".format(
    box.Lx, box.Ly, box.Lz, box.xy, box.xz, box.yz))

print("The length vector: {}".format(box.L))
print("The inverse length vector: ({:1.2f}, {:1.2f}, {:1.2f})".format(*[L for L in␣
↪box.L_inv]))
```

```
L_x = 10.0, L_y = 10.0, L_z = 10.0, xy = 0.0, xz = 0.0, yz = 0.0
The length vector: [10. 10. 10.]
The inverse length vector: (0.10, 0.10, 0.10)
```

Boxes also support converting between fractional and absolute coordinates.

Note that the origin in real coordinates is defined at the center of the box. This means the fractional coordinate range $[0, 1]$ maps onto $[-L/2, L/2]$, not $[0, L]$.

[11]:
```
# Convert from fractional to absolute coordinates.
print(box.make_absolute([[0, 0, 0], [0.5, 0.5, 0.5], [0.8, 0.3, 1]]))
print()

# Convert from fractional to absolute coordinates and back.
print(box.make_fractional(box.make_absolute([[0, 0, 0], [0.5, 0.5, 0.5], [0.8, 0.3,␣
↪1]])))
```

```
[[-5. -5. -5.]
 [ 0.  0.  0.]
 [ 3. -2.  5.]]

[[0.  0.  0. ]
 [0.5 0.5 0.5]
 [0.8 0.3 1. ]]
```

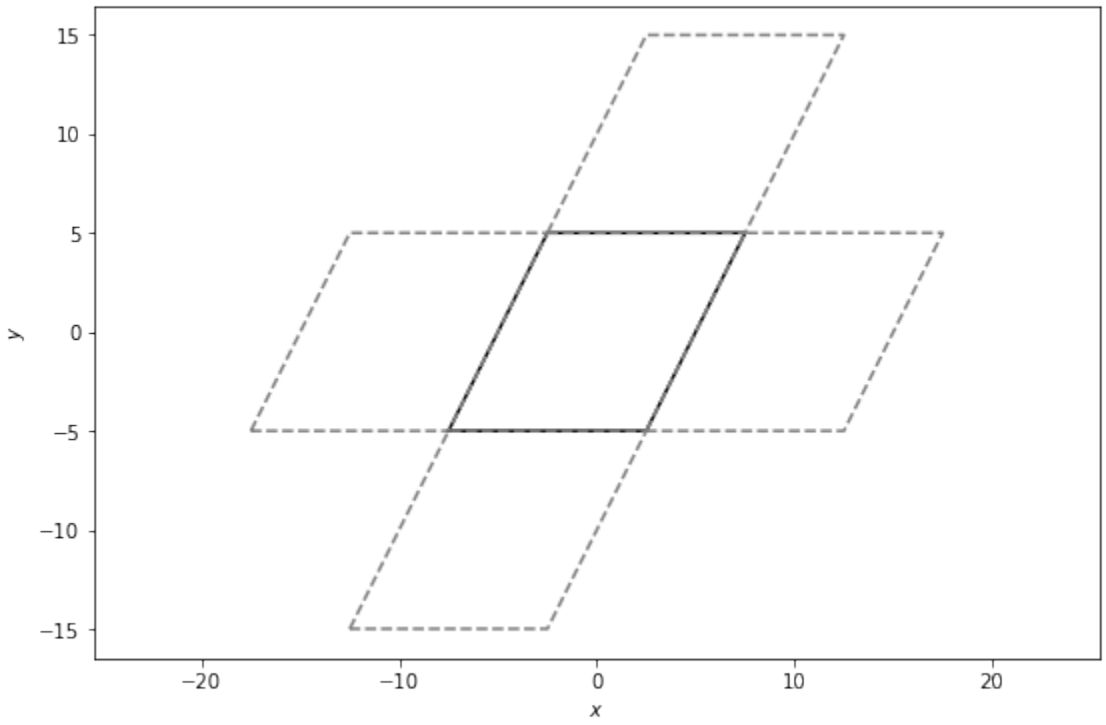
Finally (and most critically for enforcing periodicity), boxes support wrapping vectors from outside the box into the box. The concept of periodicity and box wrapping is most easily demonstrated visually.

[12]:
```
# Construct the box and get points for plotting
Lx = Ly = 10
xy = 0.5
box = freud.box.Box.from_matrix([[Lx, xy*Ly, 0], [0, Ly, 0], [0, 0, 0]])
box.plot()
```

[12]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2729d20518>
```

With periodic boundary conditions, what this actually represents is an infinite set of these boxes tiling space. For example, you can locally picture this box as surrounding by a set of identical boxes.

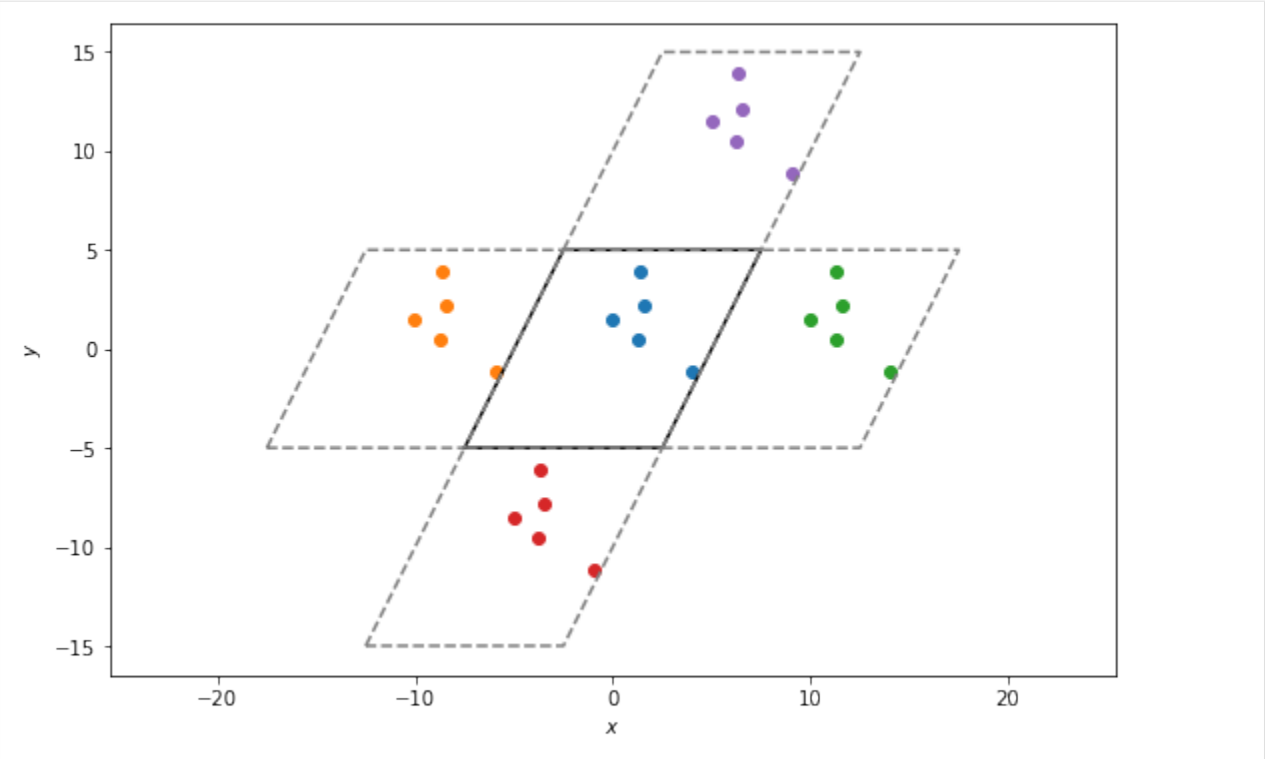
[13]:
```
%matplotlib inline
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(9, 6))
box.plot(ax=ax)
for image in [[-1, 0, 0], [1, 0, 0], [0, -1, 0], [0, 1, 0]]:
    box.plot(ax=ax, image=image, linestyle='dashed', color='gray')
plt.show()
```



Any particles in the original box will also therefore be seen as existing in all the neighboring boxes.

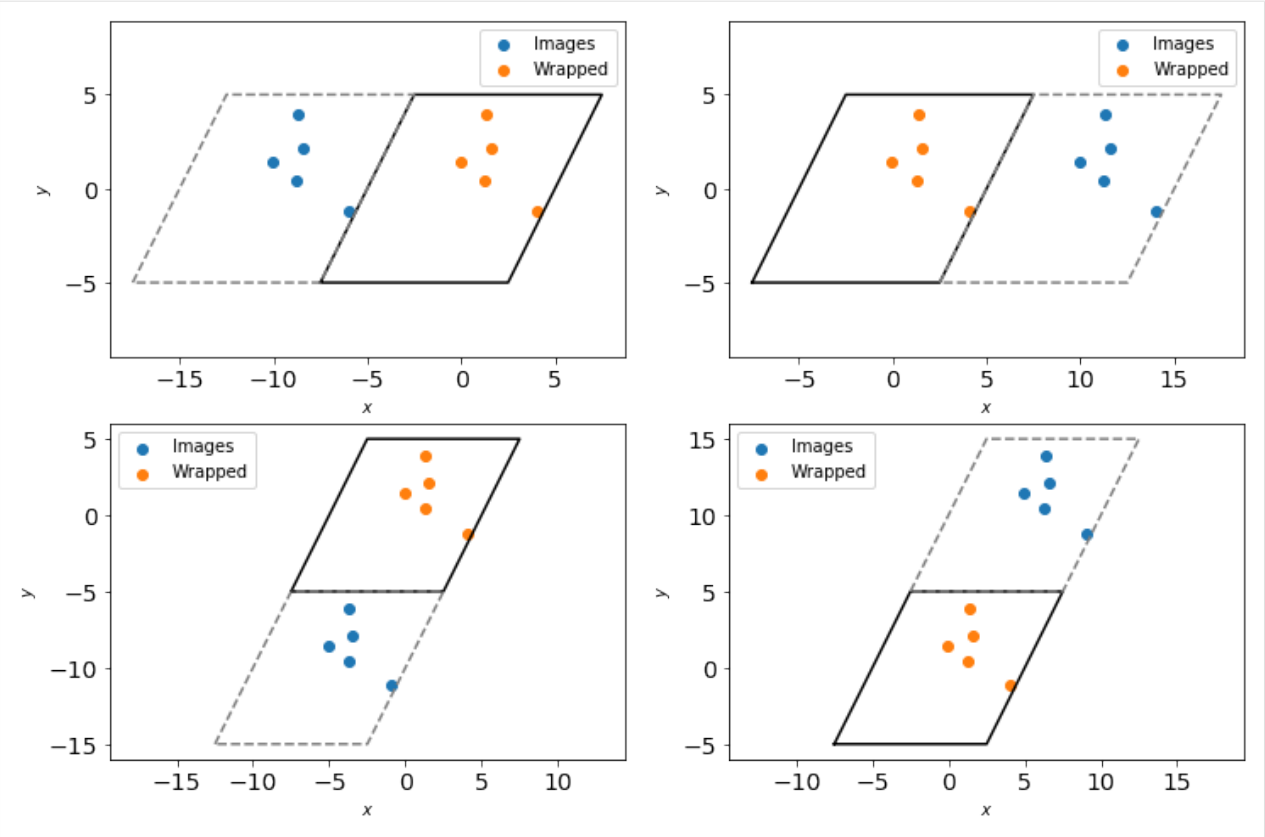
[14]:
```
np.random.seed(0)
fractional_coords = np.zeros((5, 3))
fractional_coords[:, :2] = np.random.rand(5, 2)
particles = box.make_absolute(fractional_coords)
```

[15]:
```
fig, ax = plt.subplots(figsize=(9, 6))

# Plot the points in the original box.
box.plot(ax=ax)
ax.scatter(particles[:, 0], particles[:, 1])

# Plot particles in each of the periodic boxes.
for image in [[-1, 0, 0], [1, 0, 0], [0, -1, 0], [0, 1, 0]]:
    box.plot(ax=ax, image=image, linestyle='dashed', color='gray')
    particle_images = box.unwrap(particles, image)
    ax.scatter(particle_images[:, 0], particle_images[:, 1])

plt.show()
```



Box wrapping takes points in the periodic images of a box, and brings them back into the original box. In this context, that means that if we apply wrap to each of the sets of particles plotted above, they should all overlap.

[16]:
```
fig, axes = plt.subplots(2, 2, figsize=(12, 8))
images = [[-1, 0, 0], [1, 0, 0], [0, -1, 0], [0, 1, 0]]

# Plot particles in each of the periodic boxes.
for ax, image in zip(axes.flatten(), images):
    box.plot(ax=ax)
    box.plot(ax=ax, image=image, linestyle='dashed', color='gray')
    particle_images = box.unwrap(particles, image)
    ax.scatter(particle_images[:, 0],
               particle_images[:, 1],
               label='Images')

    wrapped_particle_images = box.wrap(particle_images)
    ax.scatter(wrapped_particle_images[:, 0],
               wrapped_particle_images[:, 1],
               label='Wrapped')

    ax.tick_params(axis="both", which="both", labelsize=14)
    ax.legend()

plt.show()
```



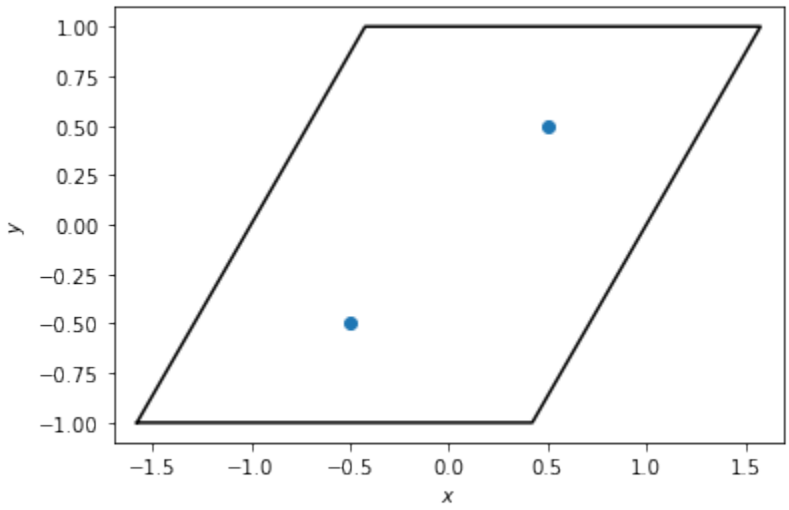
freud.locality.PeriodicBuffer: Unit Cell RDF

The `PeriodicBuffer` class is meant to replicate points beyond a single image while respecting box periodicity. This example demonstrates how we can use this to compute the radial distribution function from a sample crystal's unit cell.

[1]:
```
%matplotlib inline
import freud
import numpy as np
import matplotlib.pyplot as plt
```

Here, we create a box to represent the unit cell and put two points inside. We plot the box and points below.

[2]:
```
box = freud.box.Box(Lx=2, Ly=2, xy=np.sqrt(1/3), is2D=True)
points = np.array([[-0.5, -0.5, 0], [0.5, 0.5, 0]])
system = freud.AABBQuery(box, points)
system.plot(ax=plt.gca())
plt.show()
```



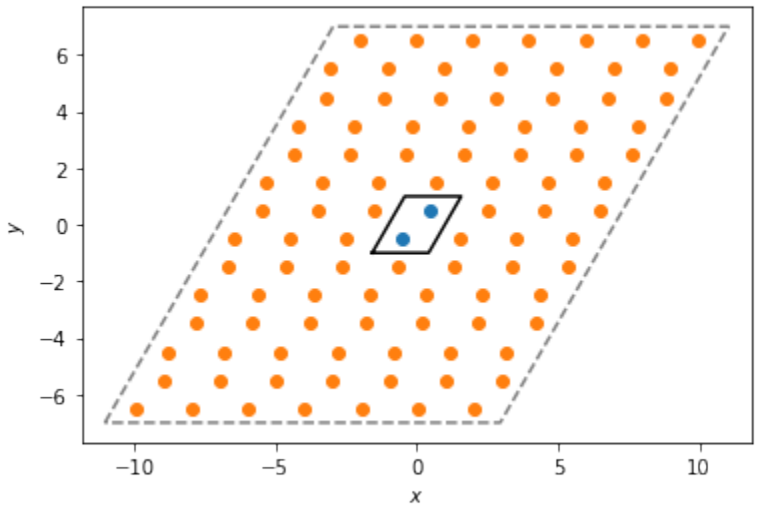
Next, we create a `PeriodicBuffer` instance and have it compute the "buffer" points that lie outside the first periodicity. These positions are stored in the `buffer_points` attribute. The corresponding `buffer_ids` array gives a mapping from the index of the buffer particle to the index of the particle it was replicated from, in the original array of `points`. Finally, the `buffer_box` attribute returns a larger box, expanded from the original box to contain the replicated points.

[3]:
```
pbuff = freud.locality.PeriodicBuffer()
pbuff.compute(system=(box, points), buffer=6, images=True)
print(pbuff.buffer_points[:10], '...')
```

```
[[ 0.65470022  1.5         0.        ]
 [ 1.80940032  3.5         0.        ]
 [ 2.96410179  5.5         0.        ]
 [-3.96410131 -6.5         0.        ]
 [-2.80940104 -4.49999952  0.        ]
 [-1.65470016 -2.50000048  0.        ]
 [ 1.50000024 -0.5         0.        ]
 [ 2.65470076  1.5         0.        ]
 [ 3.80940032  3.5         0.        ]
 [ 4.96410179  5.5         0.        ]] ...
```

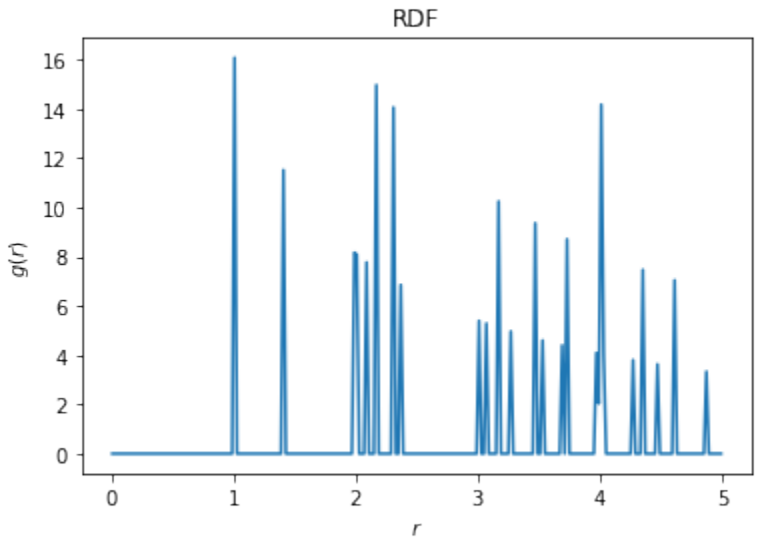
Below, we plot the original unit cell and the replicated buffer points and buffer box.

[4]:
```
system.plot(ax=plt.gca())
plt.scatter(pbuff.buffer_points[:, 0], pbuff.buffer_points[:, 1])
pbuff.buffer_box.plot(ax=plt.gca(), linestyle='dashed', color='gray')
plt.show()
```



Finally, we can plot the radial distribution function (RDF) of this replicated system, using a value of `r_max` that is larger than the size of the original box. This allows us to see the interaction of the original points with their replicated neighbors from the buffer.

[5]:
```
rdf = freud.density.RDF(bins=250, r_max=5)
rdf.compute(system=(pbuff.buffer_box, pbuff.buffer_points), query_points=points)
rdf.plot(ax=plt.gca())
plt.show()
```



freud.locality.Voronoi

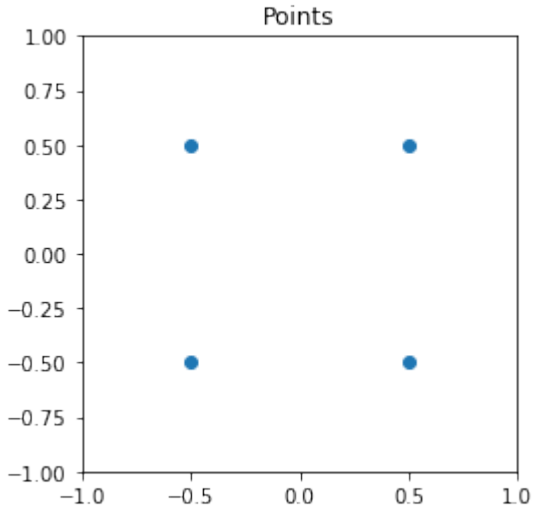
The `freud.locality.Voronoi` class uses voro++ to compute the Voronoi diagram of a set of points, **while respecting periodic boundary conditions** (which are not handled by `scipy.spatial.Voronoi`, documentation).

These examples are two-dimensional (with $z = 0$ for all particles) for simplicity, but the `Voronoi` class works for both 2D and 3D data.

[1]:
```
import numpy as np
import freud
import matplotlib
import matplotlib.pyplot as plt
```

First, we generate some sample points.

[2]:
```
points = np.array([
    [-0.5, -0.5, 0],
    [0.5, -0.5, 0],
    [-0.5, 0.5, 0],
    [0.5, 0.5, 0]])
plt.scatter(points[:, 0], points[:, 1])
plt.title('Points')
plt.xlim((-1, 1))
plt.ylim((-1, 1))
plt.gca().set_aspect('equal')
plt.show()
```



Now we create a box and a `Voronoi` compute object.

[3]:
```
L = 2
box = freud.box.Box.square(L)
voro = freud.locality.Voronoi()
```

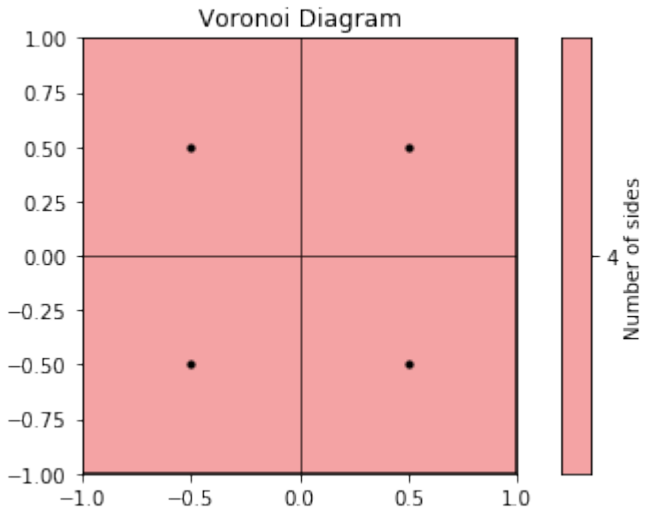
Next, we use the `compute` method to determine the Voronoi polytopes (cells) and the `polytopes` property to return their coordinates. Note that we use `freud`'s *method chaining* here, where a compute method returns the compute object.

[4]:
```
cells = voro.compute((box, points)).polytopes
print(cells)
```

```
[array([[-1., -1.,  0.],
       [ 0., -1.,  0.],
       [ 0.,  0.,  0.],
       [-1.,  0.,  0.]]), array([[ 0., -1.,  0.],
       [ 1., -1.,  0.],
       [ 1.,  0.,  0.],
       [ 0.,  0.,  0.]]), array([[-1.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  1.,  0.],
       [-1.,  1.,  0.]]), array([[0., 0., 0.],
       [1., 0., 0.],
       [1., 1., 0.],
       [0., 1., 0.]])]
```

The `Voronoi` class has built-in plotting methods for 2D systems.

[5]:
```
plt.figure()
ax = plt.gca()
voro.plot(ax=ax)
ax.scatter(points[:, 0], points[:, 1], s=10, c='k')
plt.show()
```



This also works for more complex cases, such as this hexagonal lattice.

[6]:
```
def hexagonal_lattice(rows=3, cols=3, noise=0, seed=None):
    if seed is not None:
        np.random.seed(seed)
    # Assemble a hexagonal lattice
    points = []
    for row in range(rows*2):
        for col in range(cols):
            x = (col + (0.5 * (row % 2)))*np.sqrt(3)
            y = row*0.5
            points.append((x, y, 0))
    points = np.asarray(points)
    points += np.random.multivariate_normal(mean=np.zeros(3), cov=np.eye(3)*noise,␣
↪size=points.shape[0])
    # Set z=0 again for all points after adding Gaussian noise
    points[:, 2] = 0
```

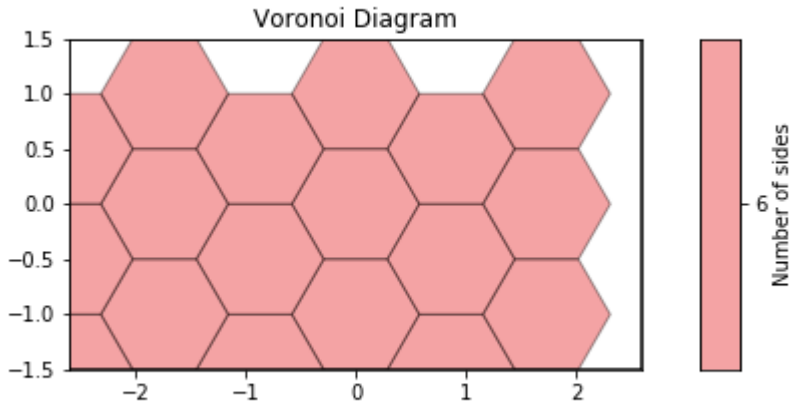
(continues on next page)

(continued from previous page)

```
    # Wrap the points into the box
    box = freud.box.Box(Lx=cols*np.sqrt(3), Ly=rows, is2D=True)
    points = box.wrap(points)
    return box, points
```

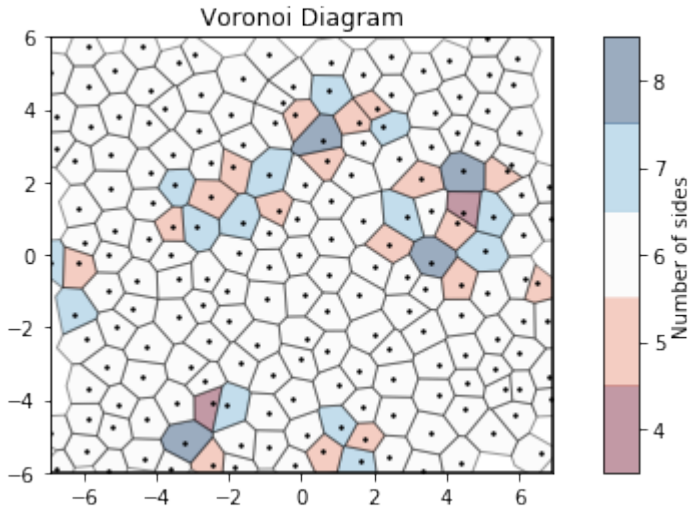
[7]:
```
# Compute the Voronoi diagram and plot
box, points = hexagonal_lattice()
voro = freud.locality.Voronoi()
voro.compute((box, points))
voro
```

[7]:



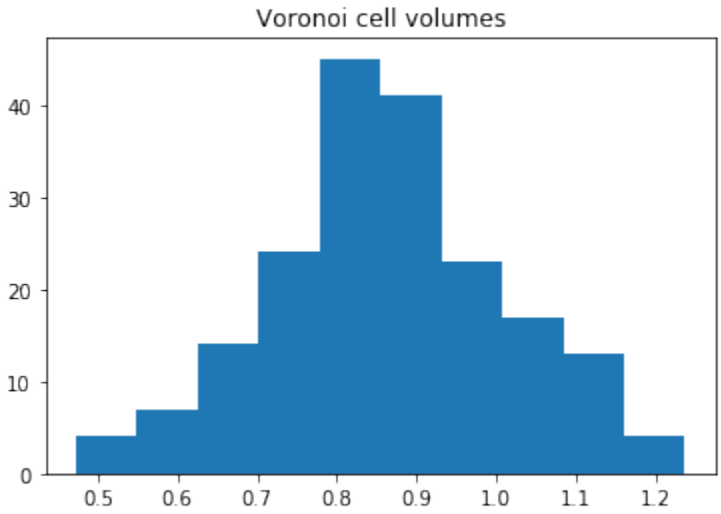
For noisy data, we see that the Voronoi diagram can change substantially. We perturb the positions with 2D Gaussian noise. Coloring by the number of sides of each Voronoi cell, we can see patterns in the defects: 5-gons and 7-gons tend to pair up.

[8]:
```
# Compute the Voronoi diagram
box, points = hexagonal_lattice(rows=12, cols=8, noise=0.03, seed=2)
voro = freud.locality.Voronoi()
voro.compute((box, points))

# Plot Voronoi with points and a custom cmap
plt.figure()
ax = plt.gca()
voro.plot(ax=ax, cmap='RdBu')
ax.scatter(points[:, 0], points[:, 1], s=2, c='k')
plt.show()
```



We can also compute the volumes of the Voronoi cells. Here, we plot them as a histogram:

[9]:
```
plt.hist(voro.volumes)
plt.title('Voronoi cell volumes')
plt.show()
```



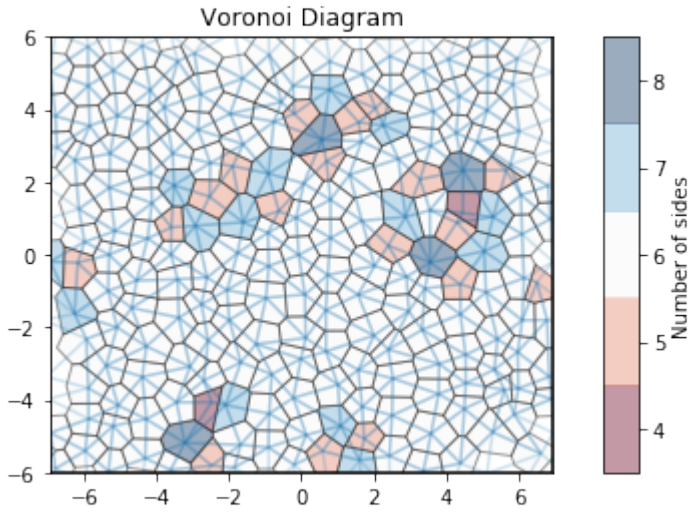
The `Voronoi` class also computes a `freud.locality.NeighborList`, where particles are neighbors if they share an edge in the Voronoi diagram. The `NeighborList` effectively represents the bonds in the Delaunay triangulation. The neighbors are **weighted** by the length (in 2D) or area (in 3D) between them. The neighbor weights are stored in `voro.nlist.weights`.

[10]:
```
nlist = voro.nlist
line_data = np.asarray([[points[i],
                         points[i] + box.wrap(points[j] - points[i])]
                        for i, j in nlist])[:, :, :2]
line_collection = matplotlib.collections.LineCollection(line_data, alpha=0.2)
plt.figure()
ax = plt.gca()
voro.plot(ax=ax, cmap='RdBu')
```

(continues on next page)

(continued from previous page)

```
ax.add_collection(line_collection)
plt.show()
```



7.5.2 Analysis Modules

These introductory examples showcase the functionality of specific modules in **freud**, showing how they can be used to perform specific types of analyses of simulations.

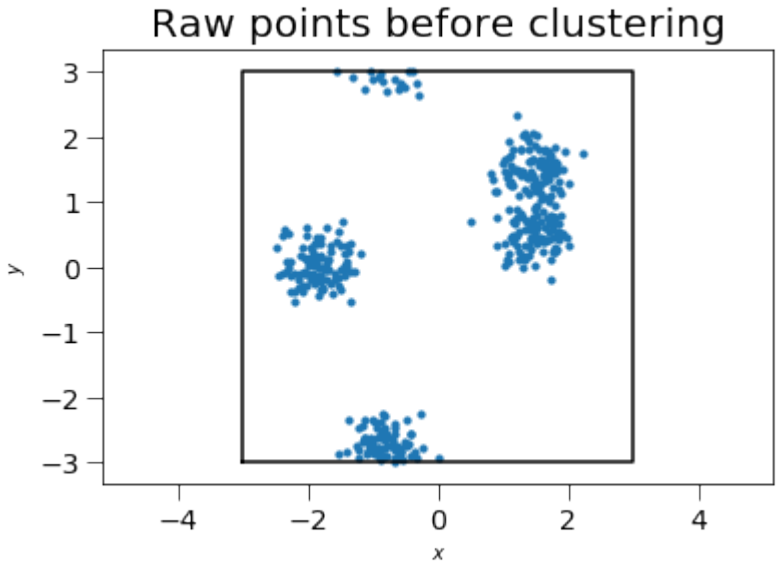
freud.cluster.Cluster and freud.cluster.ClusterProperties

The `freud.cluster` module determines clusters of points and computes cluster quantities like centers of mass, gyration tensors, and radii of gyration. The example below generates random points, and shows that they form clusters. This case is two-dimensional (with $z = 0$ for all particles) for simplicity, but the cluster module works for both 2D and 3D simulations.

[1]:
```
import numpy as np
import freud
import matplotlib.pyplot as plt
```

First, we generate a box and random points to cluster.

[2]:
```
box = freud.Box.square(L=6)
points = np.empty(shape=(0, 2))
for center_point in [(-1.8, 0), (1.5, 1.5), (-0.8, -2.8), (1.5, 0.5)]:
    points = np.concatenate(
        (points, np.random.multivariate_normal(mean=center_point, cov=0.08*np.eye(2),␣
↪size=(100,))))
points = np.hstack((points, np.zeros((points.shape[0], 1))))
points = box.wrap(points)
system = freud.AABBQuery(box, points)
system.plot(ax=plt.gca(), s=10)
plt.title('Raw points before clustering', fontsize=20)
plt.gca().tick_params(axis='both', which='both', labelsize=14, size=8)
plt.show()
```



Now we create a box and a cluster compute object.

[3]:
```
cl = freud.cluster.Cluster()
```

Next, we use the `computeClusters` method to determine clusters and the `clusterIdx` property to return their identities. Note that we use `freud`'s *method chaining* here, where a compute method returns the compute object.

[4]:
```
cl.compute(system, neighbors={'r_max': 1.0})
print(cl.cluster_idx)
```

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

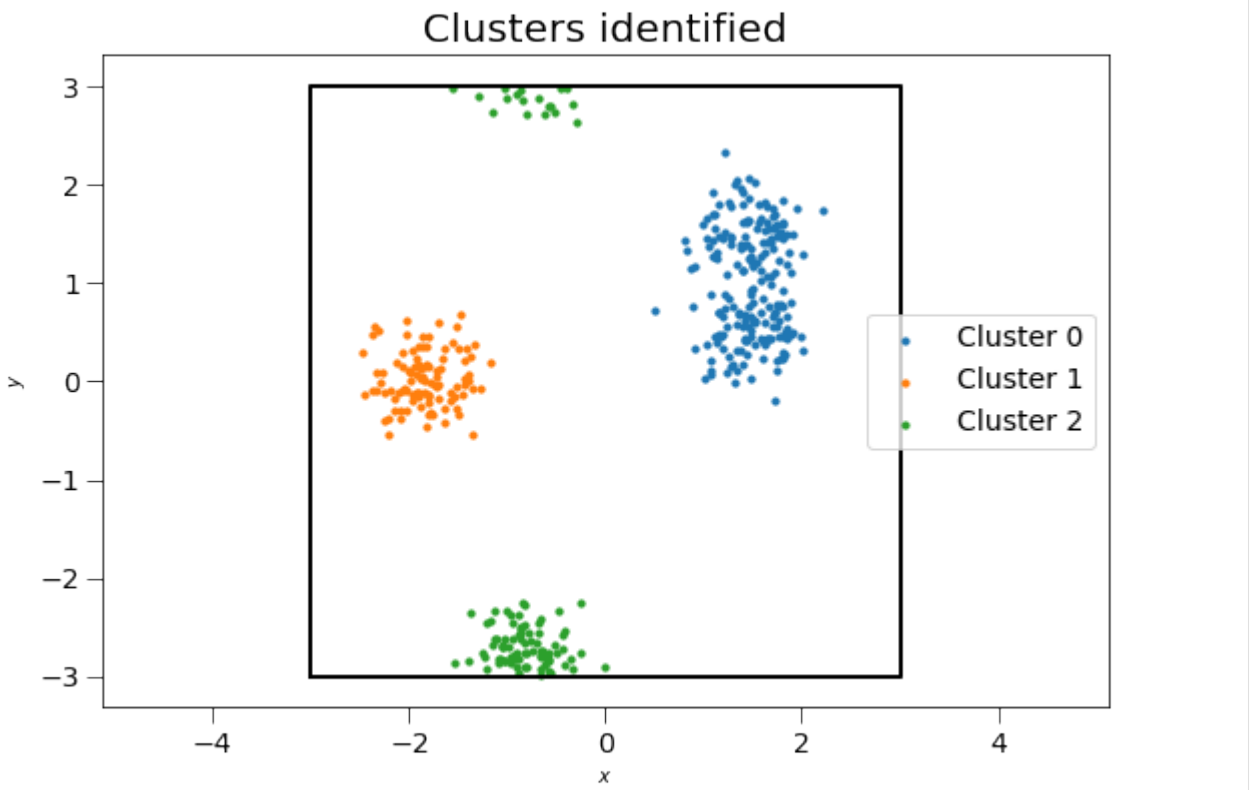
[5]:
```
fig, ax = plt.subplots(1, 1, figsize=(9, 6))
for cluster_id in range(cl.num_clusters):
    cluster_system = freud.AABBQuery(system.box, system.points[cl.cluster_
↪keys[cluster_id]])
    cluster_system.plot(ax=ax, s=10, label="Cluster {}".format(cluster_id))
    print("There are {} points in cluster {}.".format(len(cl.cluster_keys[cluster_
↪id]), cluster_id))

ax.set_title('Clusters identified', fontsize=20)
ax.legend(loc='best', fontsize=14)
ax.tick_params(axis='both', which='both', labelsize=14, size=8)
plt.show()
```

```
There are 200 points in cluster 0.
```

(continues on next page)

(continued from previous page)

```
There are 100 points in cluster 1.
There are 100 points in cluster 2.
```

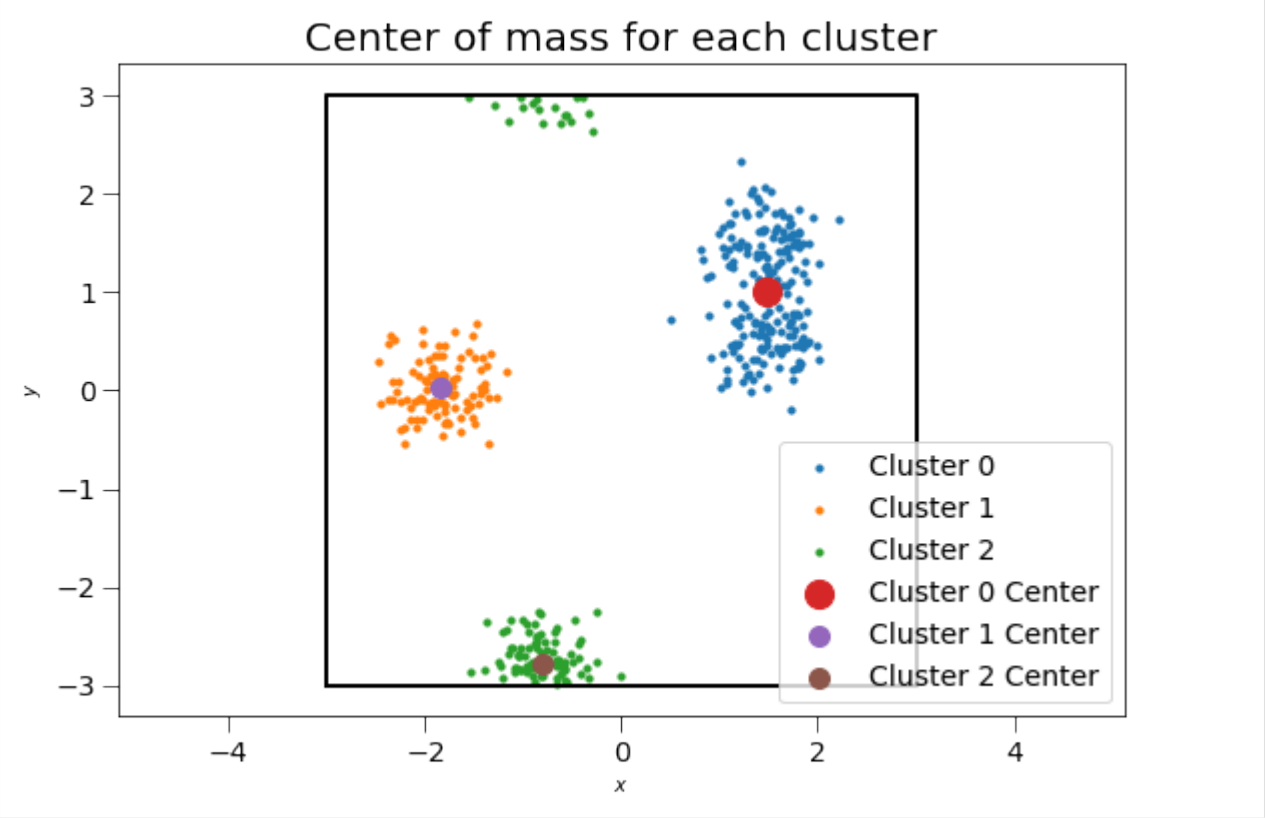


We may also compute the clusters' centers of mass and gyration tensor using the `ClusterProperties` class.

[6]:
```
clp = freud.cluster.ClusterProperties()
clp.compute(system, cl.cluster_idx);
```

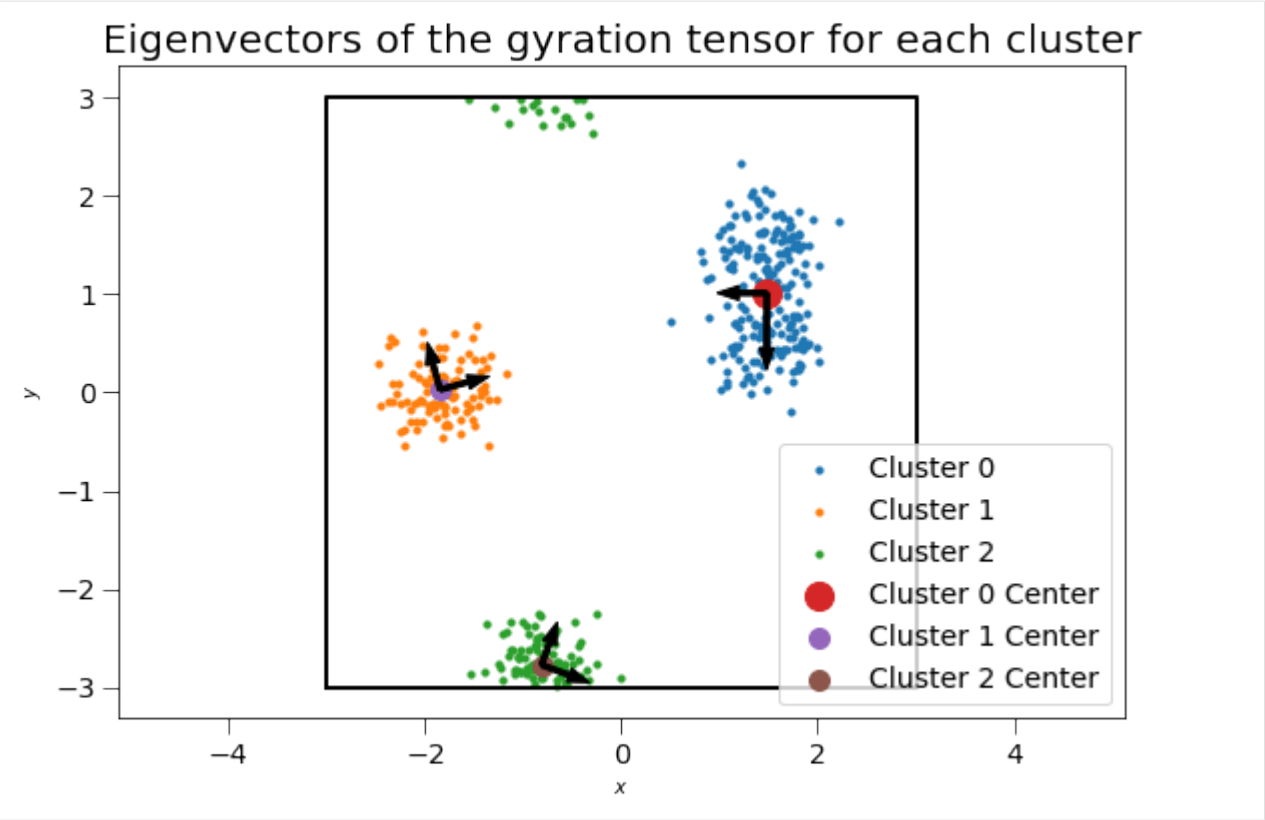
Plotting these clusters with their centers of mass, with size proportional to the number of clustered points:

[7]:
```
fig, ax = plt.subplots(1, 1, figsize=(9, 6))

for i in range(cl.num_clusters):
    cluster_system = freud.AABBQuery(system.box, system.points[cl.cluster_keys[i]])
    cluster_system.plot(ax=ax, s=10, label="Cluster {}".format(i))

for i, c in enumerate(clp.centers):
    ax.scatter(c[0], c[1], s=len(cl.cluster_keys[i]),
               label="Cluster {} Center".format(i))

plt.title('Center of mass for each cluster', fontsize=20)
plt.legend(loc='best', fontsize=14)
plt.gca().tick_params(axis='both', which='both', labelsize=14, size=8)
plt.gca().set_aspect('equal')
plt.show()
```



The 3x3 gyration tensors G can also be computed for each cluster. For this two-dimensional case, the z components of the gyration tensor are zero. The gyration tensor can be used to determine the principal axes of the cluster and radius of gyration along each principal axis. Here, we plot the gyration tensor's eigenvectors with length corresponding to the square root of the eigenvalues (the singular values).

[8]:
```
fig, ax = plt.subplots(1, 1, figsize=(9, 6))

for i in range(cl.num_clusters):
    cluster_system = freud.AABBQuery(system.box, system.points[cl.cluster_keys[i]])
    cluster_system.plot(ax=ax, s=10, label="Cluster {}".format(i))

for i, c in enumerate(clp.centers):
    ax.scatter(c[0], c[1], s=len(cl.cluster_keys[i]),
               label="Cluster {} Center".format(i))

for cluster_id in range(cl.num_clusters):
    com = clp.centers[cluster_id]
    G = clp.gyrations[cluster_id]
    evals, evecs = np.linalg.eig(G[:2, :2])
    arrows = np.sqrt(evals) * evecs
    for arrow in arrows.T:
        plt.arrow(com[0], com[1], arrow[0], arrow[1], width=0.05, color='k')

plt.title('Eigenvectors of the gyration tensor for each cluster', fontsize=20)
plt.legend(loc='best', fontsize=14)
ax.tick_params(axis='both', which='both', labelsize=14, size=8)
ax.set_aspect('equal')
plt.show()
```



freud.diffraction.DiffractionPattern

The `freud.diffraction.DiffractionPattern` class computes a diffraction pattern, which is a 2D image of the static structure factor $S(\vec{k})$ of a set of points.

[1]:
```
import freud
import matplotlib.pyplot as plt
import numpy as np
import rowan
```

First, we generate a sample system, a face-centered cubic crystal with some noise.

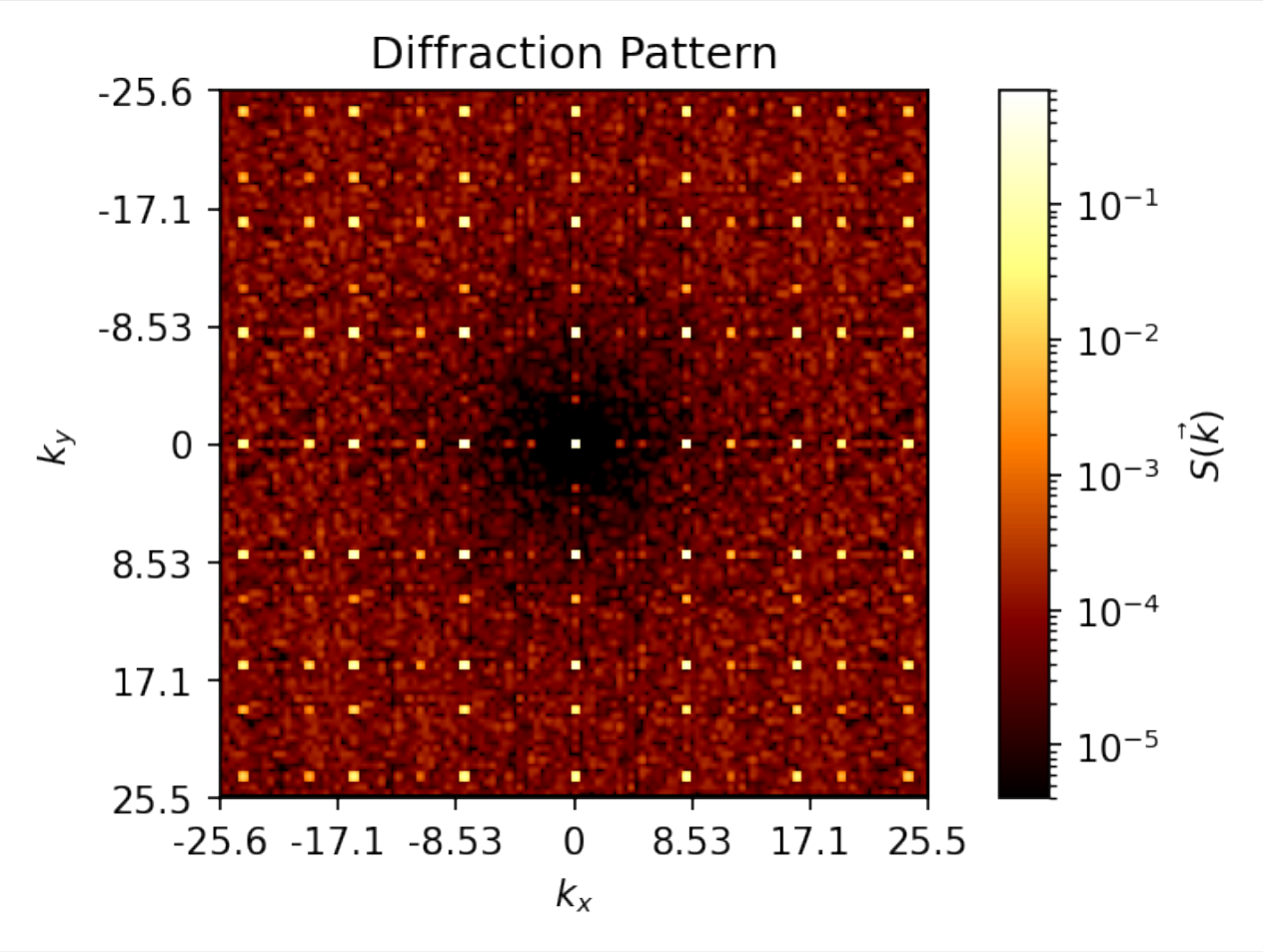
[2]:
```
box, points = freud.data.UnitCell.fcc().generate_system(num_replicas=10, sigma_
↪noise=0.02)
```

Now we create a `DiffractionPattern` compute object.

[3]:
```
dp = freud.diffraction.DiffractionPattern(grid_size=512, output_size=512)
```

Next, we use the `compute` method and plot the result. We use a view orientation with the identity quaternion `[1, 0, 0, 0]` so the view is aligned down the z-axis.

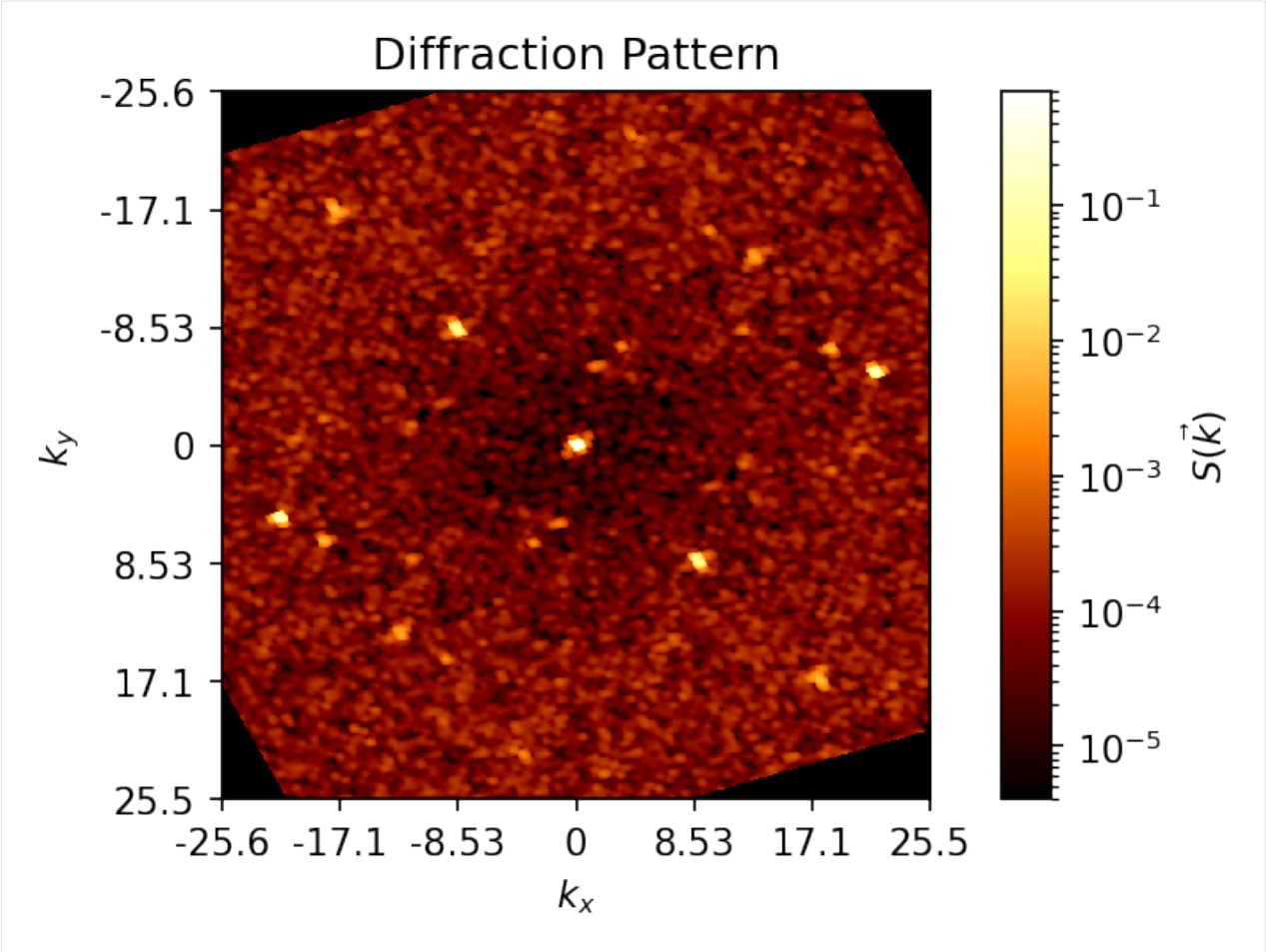
[4]:
```
fig, ax = plt.subplots(figsize=(4, 4), dpi=150)
dp.compute((box, points), view_orientation=[1, 0, 0, 0])
dp.plot(ax)
plt.show()
```



We can also use a random quaternion for the view orientation to see what the diffraction looks like from another axis.

[5]:
```
fig, ax = plt.subplots(figsize=(4, 4), dpi=150)
np.random.seed(0)
view_orientation = rowan.random.rand()
dp.compute((box, points), view_orientation=view_orientation)
print('Looking down the axis:', rowan.rotate(view_orientation, [0, 0, 1]))
dp.plot(ax)
plt.show()
```

```
Looking down the axis: [0.75707404 0.33639217 0.56007071]
```



The `DiffractionPattern` object also provides $\vec{k}$ vectors in the original 3D space and the magnitudes of k_x and k_y in the 2D projection along the view axis.

[6]:
```
print('Magnitudes of k_x and k_y along the plot axes:')
print(dp.k_values[:5], '...', dp.k_values[-5:])
```

```
Magnitudes of k_x and k_y along the plot axes:
[-25.6 -25.5 -25.4 -25.3 -25.2] ... [25.1 25.2 25.3 25.4 25.5]
```

[7]:
```
print('3D k-vectors corresponding to each pixel of the diffraction image:')
print('Array shape:', dp.k_vectors.shape)
print('Center value: k =', dp.k_vectors[dp.output_size//2, dp.output_size//2, :])
print('Top-left value: k =', dp.k_vectors[0, 0, :])
```

```
3D k-vectors corresponding to each pixel of the diffraction image:
Array shape: (512, 512, 3)
Center value: k = [0. 0. 0.]
Top-left value: k = [ 19.03669015 -29.77826902  -7.84723661]
```

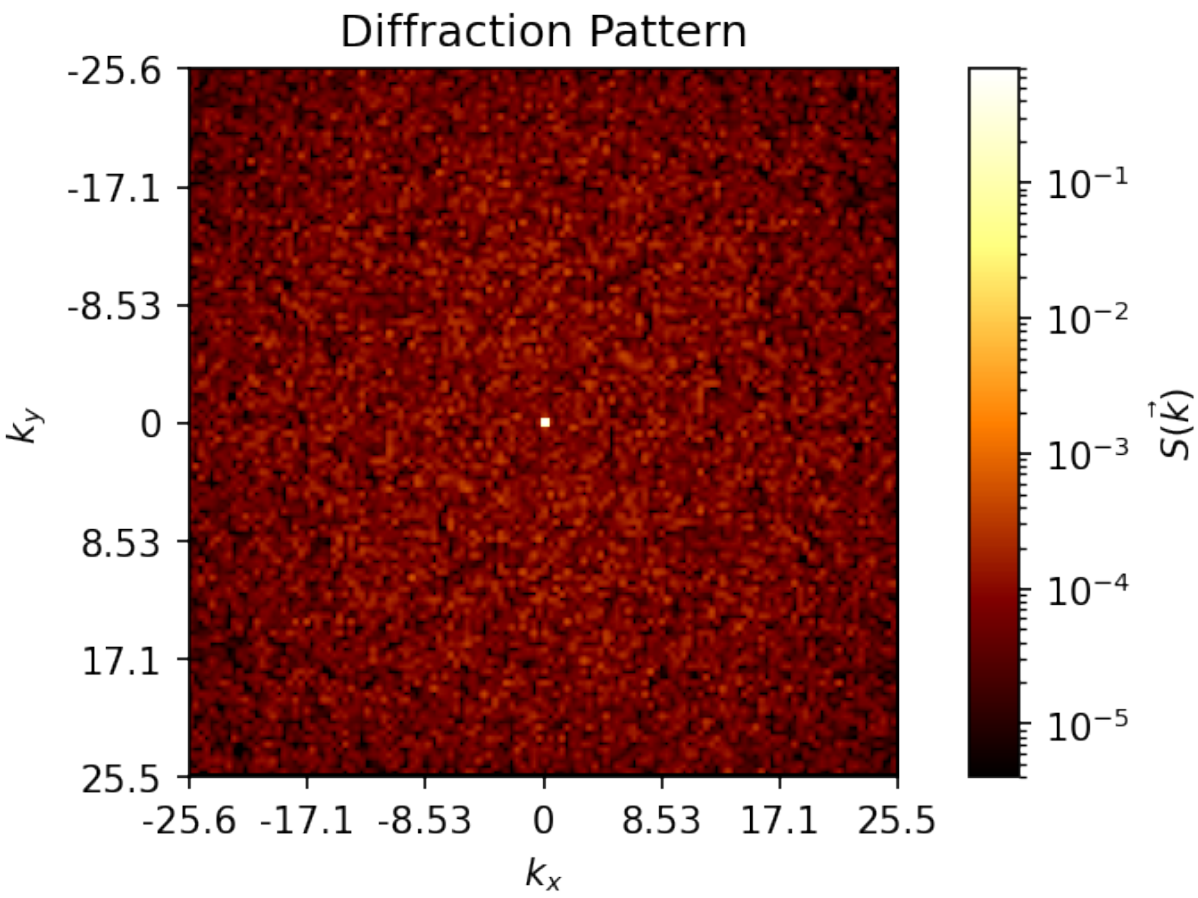
We can also measure the diffraction of a random system (note: this is an ideal gas, not a liquid-like system, because the particles have no volume exclusion or repulsion). Note that the peak at $\vec{k} = 0$ persists. The diffraction pattern returned by this class is normalized by dividing by N^2, so $S(\vec{k} = 0) = 1$ after normalization.

[8]:
```
box, points = freud.data.make_random_system(box_size=10, num_points=10000)
fig, ax = plt.subplots(figsize=(4, 4), dpi=150)
```

(continues on next page)

(continued from previous page)

```
dp.compute((box, points))
dp.plot(ax)
plt.show()
```



freud.density.CorrelationFunction

Orientational Ordering in 2D

The `freud.density` module is intended to compute a variety of quantities that relate spatial distributions of particles with other particles. This example shows how correlation functions can be used to measure orientational order in 2D.

[1]:
```
import numpy as np
import freud
import matplotlib.pyplot as plt
import matplotlib.cm
from matplotlib.colors import Normalize
```

This helper function will make plots of the data we generate in this example.

[2]:
```
def plot_data(title, points, angles, values, box, cf, s=200):
    cmap = matplotlib.cm.viridis
```

(continues on next page)

(continued from previous page)

```
    norm = Normalize(vmin=-np.pi/4, vmax=np.pi/4)
    plt.figure(figsize=(16, 6))
    plt.subplot(121)
    for point, angle, value in zip(points, angles, values):
        plt.scatter(point[0], point[1], marker=(4, 0, np.rad2deg(angle)+45),
                    edgecolor='k', c=[cmap(norm(angle))], s=s)
    plt.title(title)
    plt.gca().set_xlim([-box.Lx/2, box.Lx/2])
    plt.gca().set_ylim([-box.Ly/2, box.Ly/2])
    plt.gca().set_aspect('equal')
    sm = plt.cm.ScalarMappable(cmap='viridis', norm=norm)
    sm.set_array(angles)
    plt.colorbar(sm)
    plt.subplot(122)
    plt.title('Orientation Spatial Autocorrelation Function')
    cf.plot(ax=plt.gca())
    plt.xlabel(r'$r$')
    plt.ylabel(r'$C(r)$')
    plt.show()
```

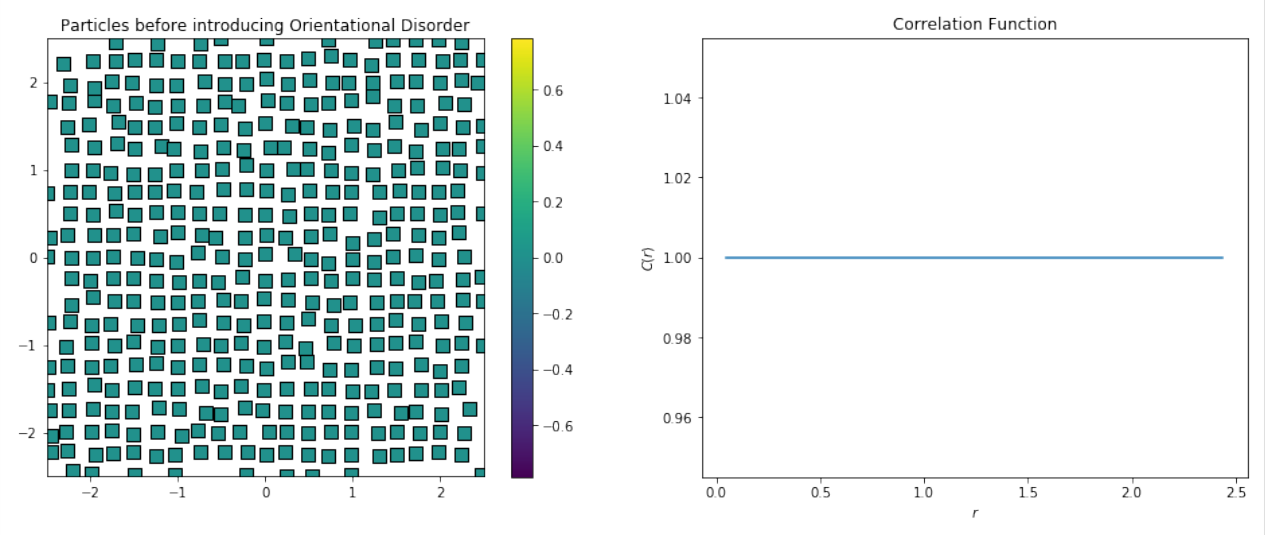
First, let's generate a 2D structure with perfect orientational order and slight positional disorder (the particles are not perfectly on a grid, but they are perfectly aligned). The color of the particles corresponds to their angle of rotation, so all the particles will be the same color to begin with.

We create a `freud.density.CorrelationFunction` object to compute the correlation functions. Given a particle orientation θ, we compute its complex orientation value (the quantity we are correlating) as $s = e^{4i\theta}$, to account for the fourfold symmetry of the particles. We will compute the correlation function $C(r) = \langle s_1^*(0) \cdot s_2(r) \rangle$ by taking the average over all particle pairs and binning the results into a histogram by the distance r between the particles.

When we compute the correlations between particles, the complex conjugate of the `values` array is used internally for the query points. This way, if θ_1 is close to θ_2, then we get $\left(e^{4i\theta_1}\right)^* \cdot \left(e^{4i\theta_2}\right) = e^{4i(\theta_2 - \theta_1)} \approx e^0 = 1$.

This system has perfect spatial correlation of the particle orientations, so we see $C(r) = 1$ for all values of r.

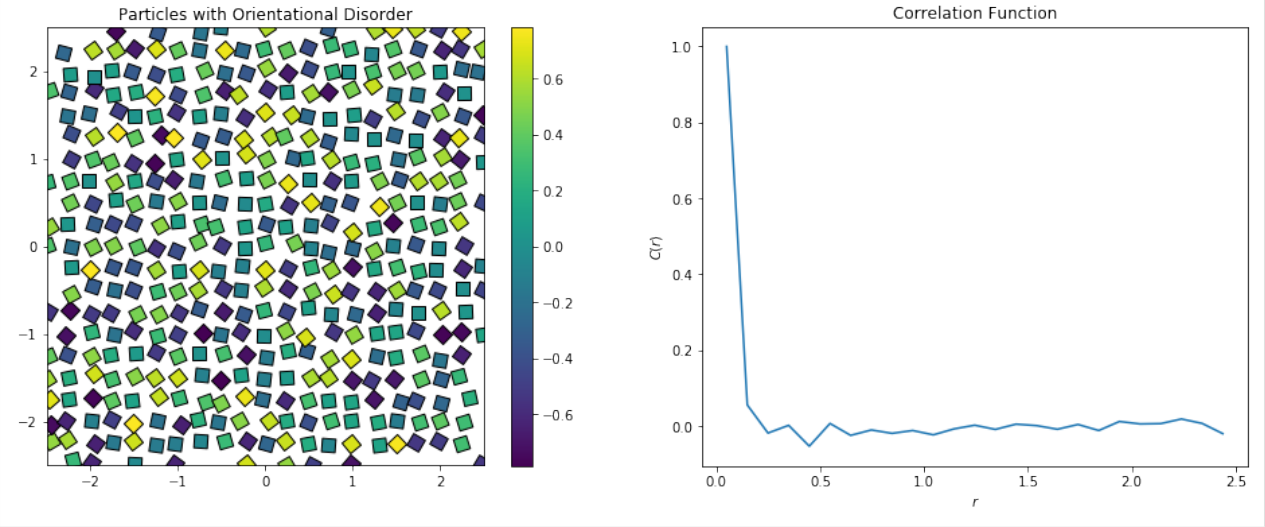
[3]:
```
def make_particles(L, repeats):
    uc = freud.data.UnitCell.square()
    return uc.generate_system(num_replicas=repeats, scale=L/repeats, sigma_noise=5e-
↪3*L)

# Make a small system
box, points = make_particles(L=5, repeats=20)

# All the particles begin with their orientation at 0
angles = np.zeros(len(points))
values = np.array(np.exp(angles * 4j))

# Create the CorrelationFunction compute object and compute the correlation function
cf = freud.density.CorrelationFunction(bins=25, r_max=box.Lx/2.01)
cf.compute(system=(box, points), values=values,
           query_points=points, query_values=values)


plot_data('Particles before introducing Orientational Disorder',
          points, angles, values, box, cf)
```



Now we will generate random angles from $-\frac{\pi}{4}$ to $\frac{\pi}{4}$, which orients our squares randomly. The four-fold symmetry of the squares means that the space of unique angles is restricted to a range of $\frac{\pi}{2}$. Again, we compute a complex value for each particle, $s = e^{4i\theta}$.

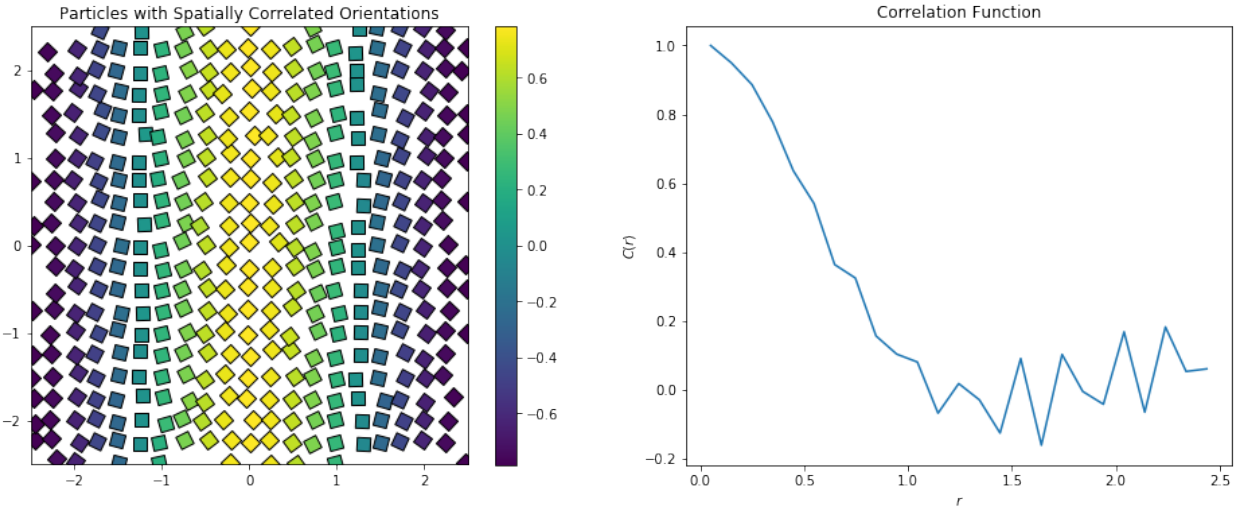
Because we have purely random orientations, we expect no spatial correlations in the plot above. As we see, $C(r) \approx 0$ for all r.

[4]:
```
# Change the angles to values randomly drawn from a uniform distribution
angles = np.random.uniform(-np.pi/4, np.pi/4, size=len(points))
values = np.exp(angles * 4j)

# Recompute the correlation functions
cf.compute(system=(box, points), values=values,
           query_points=points, query_values=values)

plot_data('Particles with Orientational Disorder',
          points, angles, values, box, cf)
```



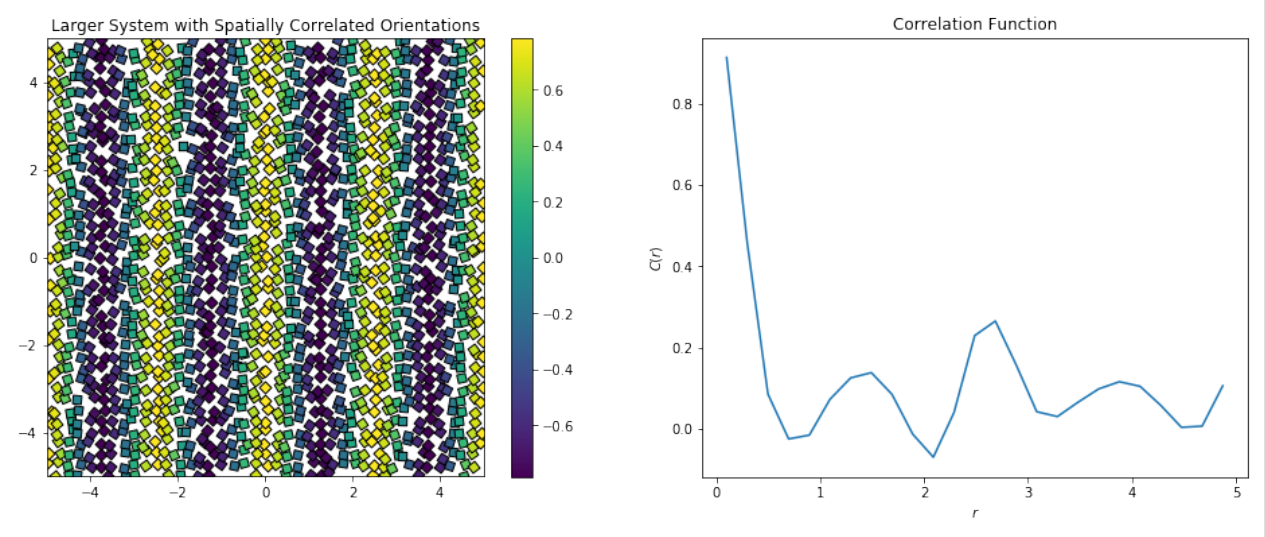
The plot below shows what happens when we intentionally introduce a correlation length by adding a spatial pattern to the particle orientations. At short distances, the correlation is very high. As r increases, the oppositely-aligned part of the pattern some distance away causes the correlation to drop.

[5]:
```
# Use angles that vary spatially in a pattern
angles = np.pi/4 * np.cos(2*np.pi*points[:, 0]/box.Lx)
values = np.exp(angles * 4j)

# Recompute the correlation functions
cf.compute(system=(box, points), values=values,
           query_points=points, query_values=values)

plot_data('Particles with Spatially Correlated Orientations',
          points, angles, values, box, cf)
```



In the larger system shown below, we see the spatial autocorrelation rise and fall with damping oscillations.

[6]:
```
# Make a large system
box, points = make_particles(L=10, repeats=40)

# Use angles that vary spatially in a pattern
angles = np.pi/4 * np.cos(8*np.pi*points[:, 0]/box.Lx)
values = np.exp(angles * 4j)

# Create a CorrelationFunction compute object
cf = freud.density.CorrelationFunction(bins=25, r_max=box.Lx/2.01)
cf.compute(system=(box, points), values=values,
           query_points=points, query_values=values)

plot_data('Larger System with Spatially Correlated Orientations',
          points, angles, values, box, cf, s=80)
```



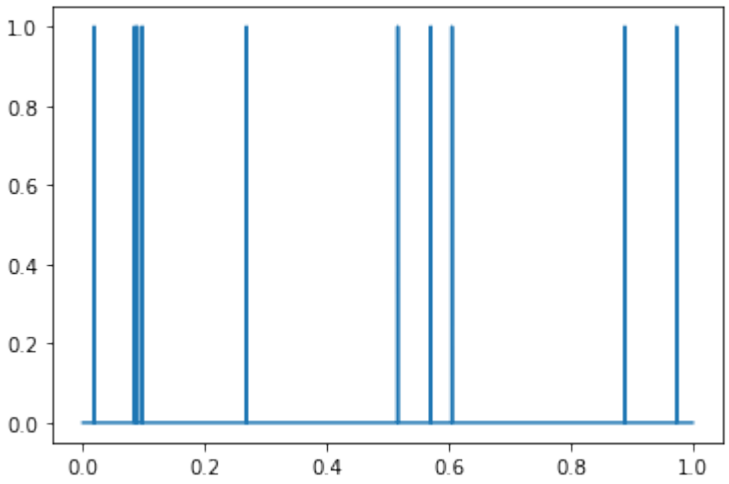
freud.density.GaussianDensity

The `freud.density` module is intended to compute a variety of quantities that relate spatial distributions of particles with other particles. In this notebook, we demonstrate `freud`'s Gaussian density calculation, which provides a way to interpolate particle configurations onto a regular grid in a meaningful way that can then be processed by other algorithms that require regularity, such as a Fast Fourier Transform.

[1]:
```
import numpy as np
from scipy import stats
import freud
import matplotlib.pyplot as plt
```

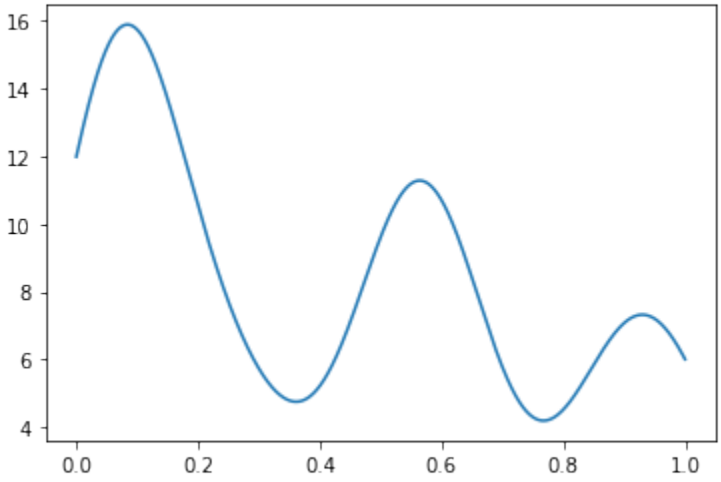
To illustrate the basic concept, consider a toy example: a simple set of point particles with unit mass on a line. For analytical purposes, the standard way to accomplish this would be using Dirac delta functions.

[2]:
```
n_p = 10000
np.random.seed(129)
x = np.linspace(0, 1, n_p)
y = np.zeros(n_p)
points = np.random.rand(10)
y[(points*n_p).astype('int')] = 1
plt.plot(x, y);
plt.show()
```



However, delta functions can be cumbersome to work with, so we might instead want to smooth out these particles. One option is to instead represent particles as Gaussians centered at the location of the points. In that case, the total particle density at any point in the interval $[0, 1]$ represented above would be based on the sum of the densities of those Gaussians at those points.

[3]:
```
# Note that we use a Gaussian with a small standard deviation
# to emphasize the differences on this small scale
dists = [stats.norm(loc=i, scale=0.1) for i in points]
y_gaussian = 0
for dist in dists:
    y_gaussian += dist.pdf(x)
plt.plot(x, y_gaussian);
plt.show()
```



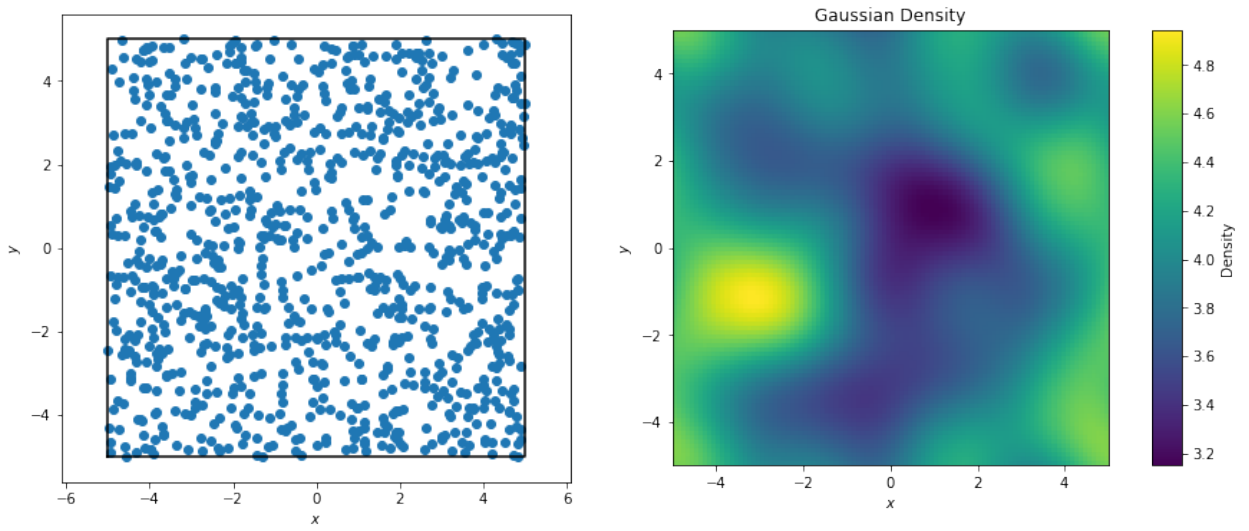
The goal of the GaussianDensity class is to perform the same interpolation for points on a 2D or 3D grid, accounting for Box periodicity.

[4]:
```
N = 1000  # Number of points
L = 10  # Box length

box, points = freud.data.make_random_system(L, N, is2D=True, seed=0)
```

(continues on next page)

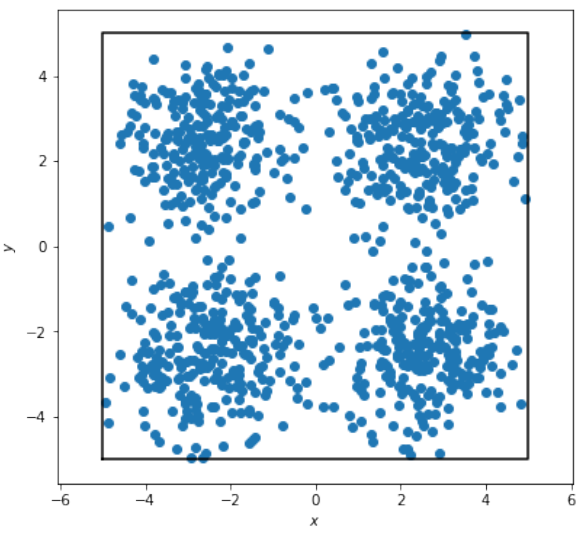
(continued from previous page)

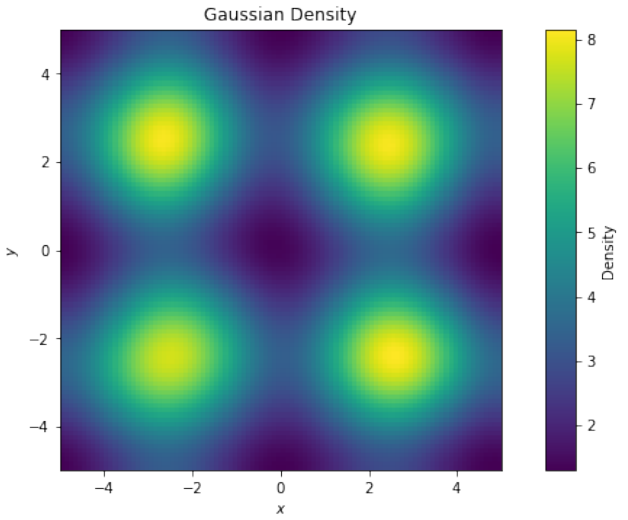
```
aq = freud.AABBQuery(box, points)
gd = freud.density.GaussianDensity(L*L, L/3, 1)
gd.compute(aq)

fig, axes = plt.subplots(1, 2, figsize=(14, 6))
aq.plot(ax=axes[0])
gd.plot(ax=axes[1])
plt.show()
```



The effects are much more striking if we explicitly construct our points to be centered at certain regions.

[5]:
```
N = 1000  # Number of points
L = 10  # Box length
box = freud.box.Box.square(L)
centers = np.array([[L/4, L/4, 0],
                    [-L/4, L/4, 0],
                    [L/4, -L/4, 0],
                    [-L/4, -L/4, 0]])

points = []
for center in centers:
    points.append(np.random.multivariate_normal(center, cov=np.diag([1, 1, 0]),␣
↪size=(int(N/4),)))
points = box.wrap(np.concatenate(points))
aq = freud.AABBQuery(box, points)

gd = freud.density.GaussianDensity(L*L, L/3, 1)
gd.compute(aq)

fig, axes = plt.subplots(1, 2, figsize=(14, 6))
aq.plot(ax=axes[0])
gd.plot(ax=axes[1])
plt.show()
```





freud.density.LocalDensity

The `freud.density` module is intended to compute a variety of quantities that relate spatial distributions of particles with other particles. In this notebook, we demonstrate `freud`'s local density calculation, which can be used to characterize the particle distributions in some systems. In this example, we consider a toy example of calculating the particle density in the vicinity of a set of other points. This can be visualized as, for example, billiard balls on a table with certain regions of the table being stickier than others. In practice, this method could be used for analyzing, *e.g*, binary systems to determine how densely one species packs close to the surface of the other.

[1]:
```
import numpy as np
import freud
import matplotlib.pyplot as plt
from matplotlib import patches
```

[2]:
```
# Define some helper plotting functions.
def add_patches(ax, points, radius=1, fill=False, color="#1f77b4", ls="solid",␣
↪lw=None):
    """Add set of points as patches with radius to the provided axis"""
    for pt in points:
        p = patches.Circle(pt, fill=fill, linestyle=ls, radius=radius,
                           facecolor=color, edgecolor=color, lw=lw)
        ax.add_patch(p)

def plot_lattice(box, points, radius=1, ls="solid", lw=None):
    """Helper function for plotting points on a lattice."""
    fig, ax = plt.subplots(1, 1, figsize=(9, 9))
    box.plot(ax=ax)
    add_patches(ax, points, radius, ls=ls, lw=lw)
    return fig, ax
```

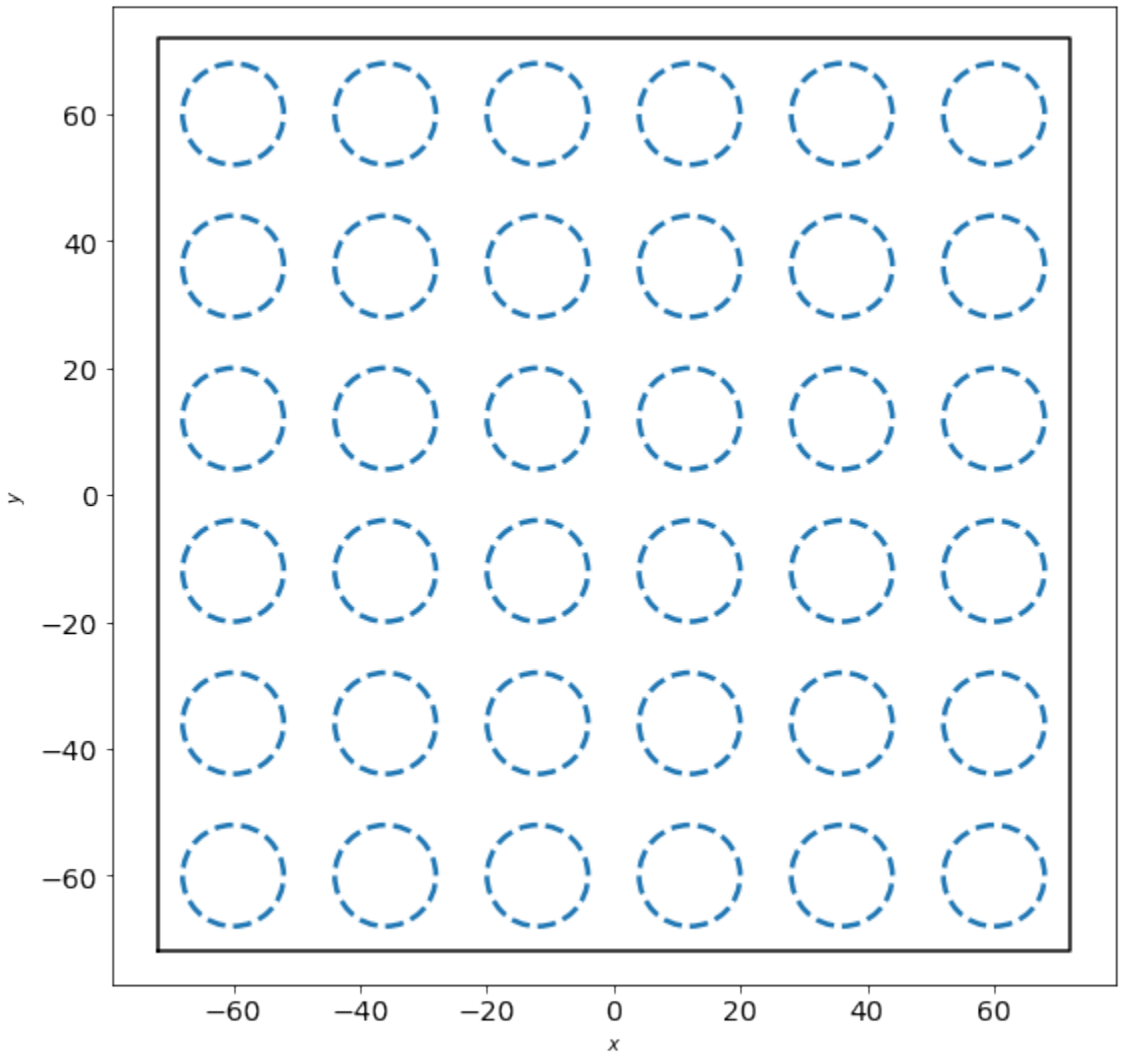
Let us consider a set of regions on a square lattice.

[3]:
```
area = 2
radius = np.sqrt(area/np.pi)
spot_area = area*100
```

(continues on next page)

(continued from previous page)

```
spot_radius = np.sqrt(spot_area/np.pi)
num = 6
scale = num*4
uc = freud.data.UnitCell(freud.Box.square(1), [[0.5, 0.5, 0]])
box, spot_centers = uc.generate_system(num, scale=scale)
fig, ax = plot_lattice(box, spot_centers, spot_radius, ls="dashed", lw=2.5)
plt.tick_params(axis="both", which="both", labelsize=14)
plt.show()
```



Now let's add a set of points to this box. Points are added by drawing from a normal distribution centered at each of the regions above. For demonstration, we will assume that each region has some relative "attractiveness," which is represented by the covariance in the normal distributions used to draw points. Specifically, as we go up and to the right, the covariance increases proportional to the distance from the lower right corner of the box.

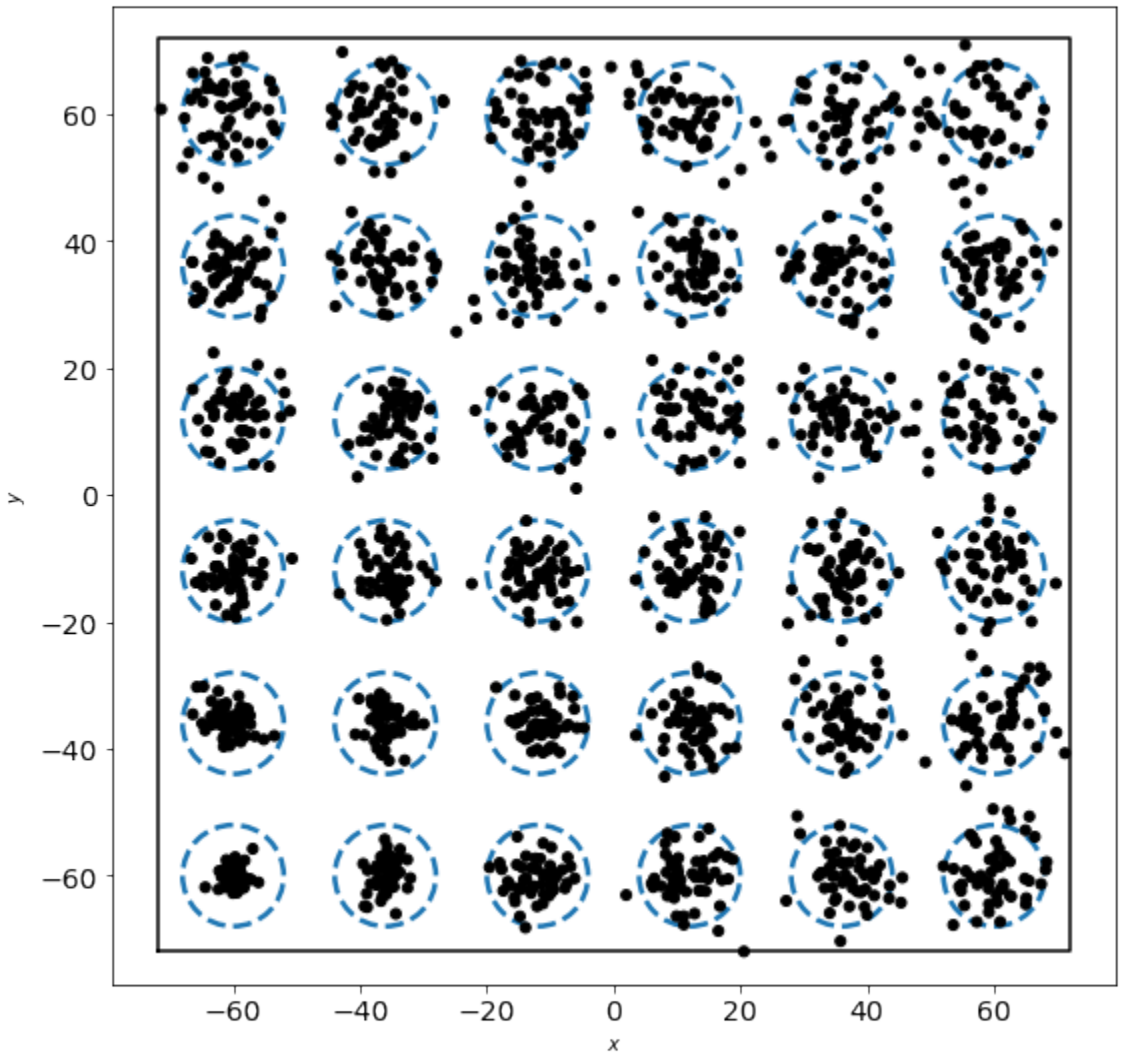
[4]:
```
points = []
fractional_distances_to_corner = np.linalg.norm(box.make_fractional(spot_centers),␣
↪axis=-1)
cov_basis = 20 * fractional_distances_to_corner
for i, p in enumerate(spot_centers):
```

(continues on next page)

(continued from previous page)

```
    np.random.seed(i)
    cov = cov_basis[i]*np.diag([1, 1, 0])
    points.append(
        np.random.multivariate_normal(p, cov, size=(50,)))
points = box.wrap(np.concatenate(points))
```

[5]:
```
fig, ax = plot_lattice(box, spot_centers, spot_radius, ls="dashed", lw=2.5)
plt.tick_params(axis="both", which="both", labelsize=14)
add_patches(ax, points, radius, True, 'k', lw=None)
plt.show()
```



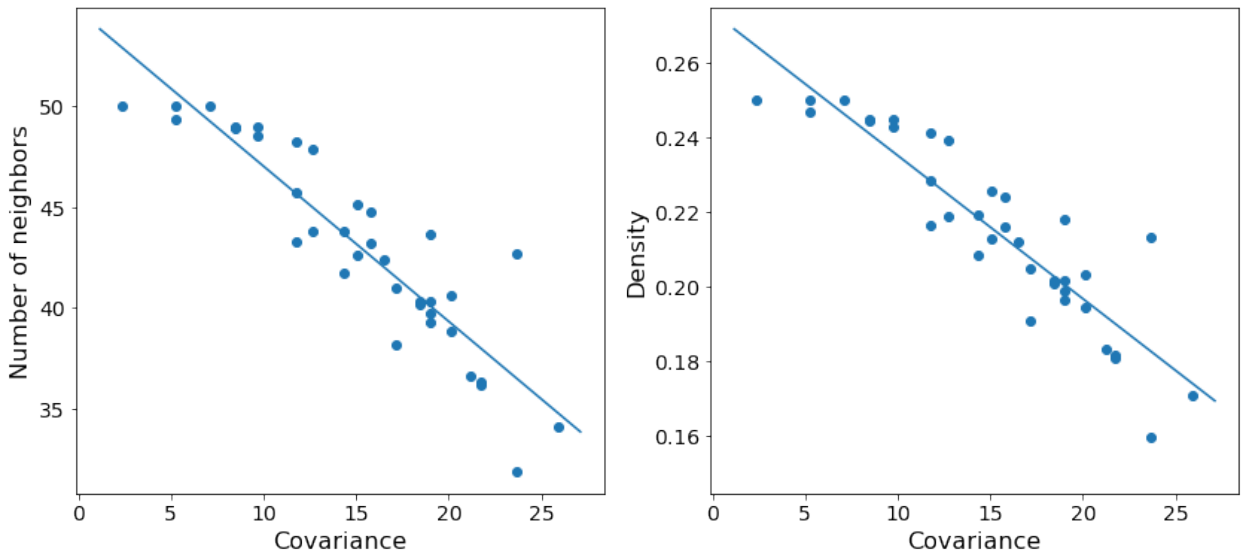
We see that the density decreases as we move up and to the right. In order to compute the actual densities, we can leverage the `LocalDensity` class. The class allows you to specify a set of query points around which the number of other points is computed. These other points can, but need not be, distinct from the query points. In our case, we want to use the blue regions as our query points with the small black dots as our data points.

When we construct the `LocalDensity` class, there are two arguments. The first is the radius from the query points within which particles should be included in the query point's counter. The second is the circumsphere diameter of the **data points**, not the query points. This distinction is critical for getting appropriate density values, since these values

are used to actually check cutoffs and calculate the density.

[6]:
```
density = freud.density.LocalDensity(spot_radius, radius)
density.compute(system=(box, points), query_points=spot_centers);
```

[7]:
```
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

for i, data in enumerate([density.num_neighbors, density.density]):
    poly = np.poly1d(np.polyfit(cov_basis, data, 1))
    axes[i].tick_params(axis="both", which="both", labelsize=14)
    axes[i].scatter(cov_basis, data)
    x = np.linspace(*axes[i].get_xlim(), 30)
    axes[i].plot(x, poly(x), label="Best fit")
    axes[i].set_xlabel("Covariance", fontsize=16)

axes[0].set_ylabel("Number of neighbors", fontsize=16);
axes[1].set_ylabel("Density", fontsize=16);
plt.show()
```



As expected, we see that increasing the variance in the number of points centered at a particular query point decreases the total density at that point. The trend is noisy since we are randomly sampling possible positions, but the general behavior is clear.

freud.density.RDF: Accumulating g(r) for a Fluid

The `freud.density` module is intended to compute a variety of quantities that relate spatial distributions of particles with other particles. This example demonstrates the calculation of the radial distribution function $g(r)$ for a fluid, averaged over multiple frames.

[1]:
```
import numpy as np
import freud
import matplotlib.pyplot as plt

data_path = "data/phi065"
box_data = np.load("{}/box_data.npy".format(data_path))
pos_data = np.load("{}/pos_data.npy".format(data_path))
```

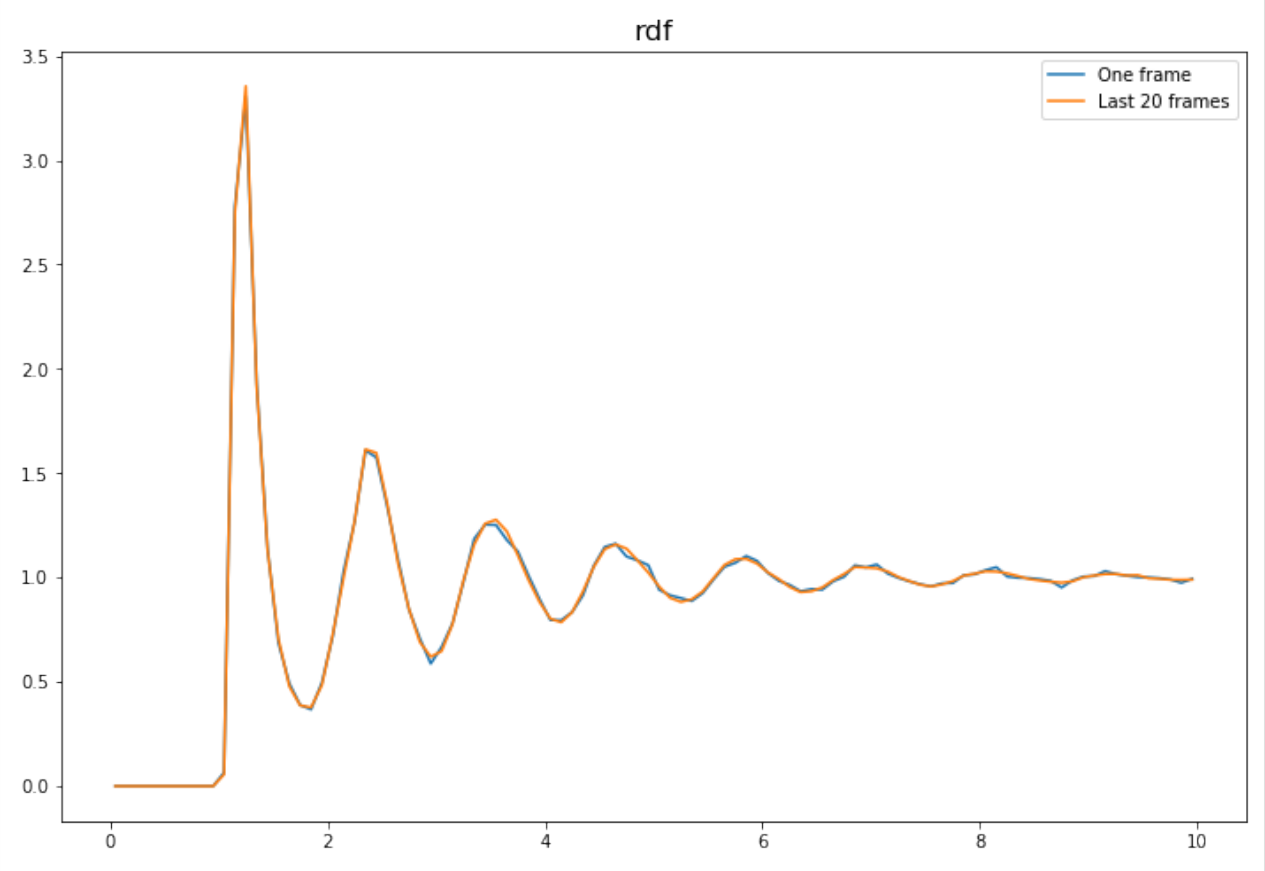
(continues on next page)

(continued from previous page)

```
def plot_rdf(box_arr, points_arr, prop, r_max=10, bins=100, label=None, ax=None):
    """Helper function for plotting RDFs."""
    if ax is None:
        fig, ax = plt.subplots(1, 1, figsize=(12, 8))
        ax.set_title(prop, fontsize=16)
    rdf = freud.density.RDF(bins, r_max)
    for box, points in zip(box_arr, points_arr):
        rdf.compute(system=(box, points), reset=False)
    if label is not None:
        ax.plot(rdf.bin_centers, getattr(rdf, prop), label=label)
        ax.legend()
    else:
        ax.plot(rdf.bin_centers, getattr(rdf, prop))
    return ax
```

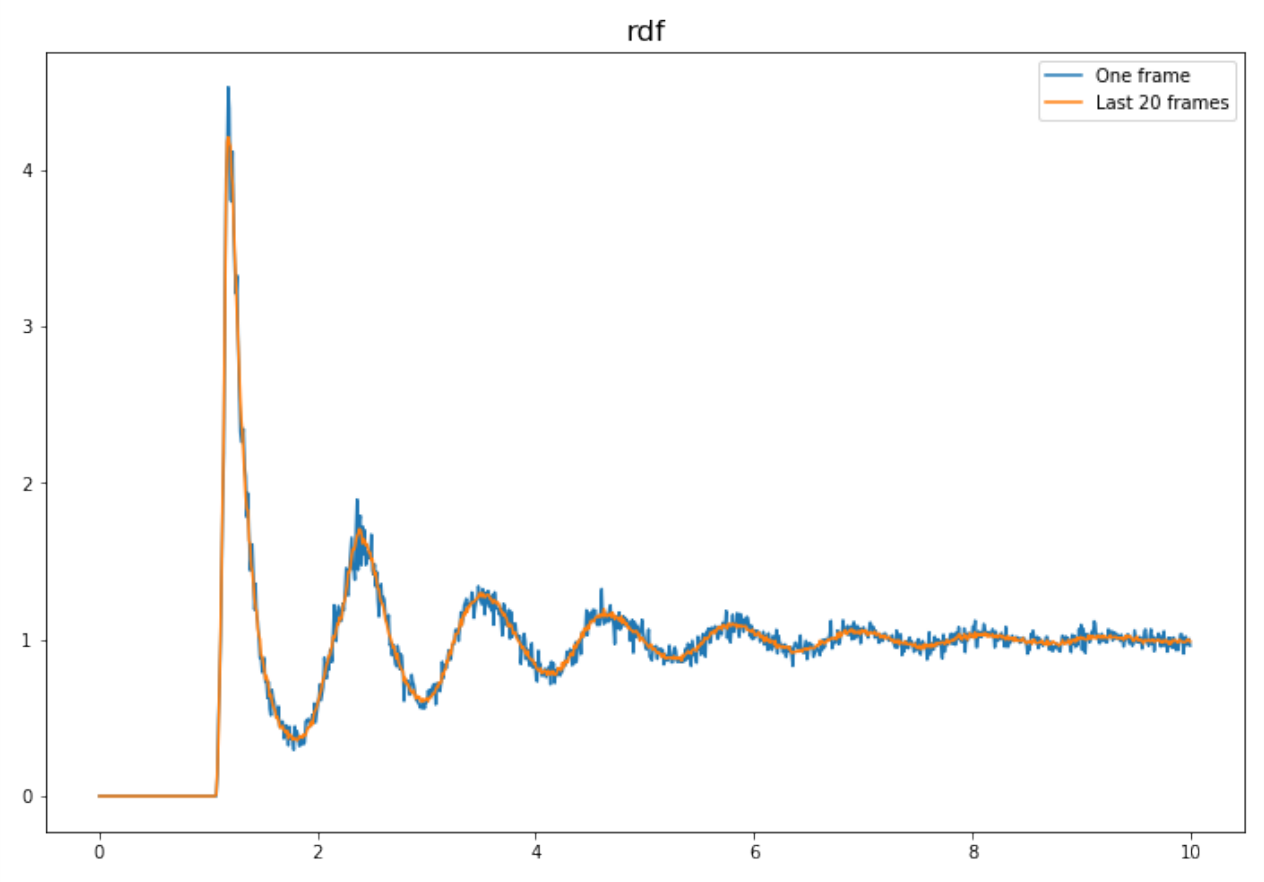
Here, we show the difference between the RDF of one frame and an accumulated (averaged) RDF from several frames. Including more frames makes the plot smoother.

[2]:
```
# Compute the RDF for the last frame
box_arr = [box_data[-1].tolist()]
pos_arr = [pos_data[-1]]
ax = plot_rdf(box_arr, pos_arr, 'rdf', label='One frame')

# Compute the RDF for the last 20 frames
box_arr = [box.tolist() for box in box_data[-20:]]
pos_arr = pos_data[-20:]
ax = plot_rdf(box_arr, pos_arr, 'rdf', label='Last 20 frames', ax=ax)

plt.show()
```



The difference between `accumulate` (which should be called on a series of frames) and `compute` (meant for a single frame) is more striking for smaller bin sizes, which are statistically noisier.

[3]:
```
# Compute the RDF for the last frame
box_arr = [box_data[-1].tolist()]
pos_arr = [pos_data[-1]]
ax = plot_rdf(box_arr, pos_arr, 'rdf', bins=1000, label='One frame')

# Compute the RDF for the last 20 frames
box_arr = [box.tolist() for box in box_data[-20:]]
pos_arr = pos_data[-20:]
ax = plot_rdf(box_arr, pos_arr, 'rdf', bins=1000, label='Last 20 frames', ax=ax)

plt.show()
```



freud.density.RDF: Choosing Bin Widths

The `freud.density` module is intended to compute a variety of quantities that relate spatial distributions of particles with other particles. This example demonstrates the calculation of the radial distribution function $g(r)$ using different bin sizes.

[1]:
```
import numpy as np
import freud
import matplotlib.pyplot as plt
```

[2]:
```
# Define some helper plotting functions.
def plot_rdf(box, points, prop, r_max=3.5, bins_array=[20, 75, 3000]):
    """Helper function for plotting RDFs."""
    fig, axes = plt.subplots(1, len(bins_array), figsize=(16, 3))
    for i, bins in enumerate(bins_array):
        rdf = freud.density.RDF(bins, r_max)
        rdf.compute(system=(box, points))
        axes[i].plot(rdf.bin_centers, getattr(rdf, prop))
        axes[i].set_title("Bin width: {:.3f}".format(r_max/bins), fontsize=16)
    plt.show()
```

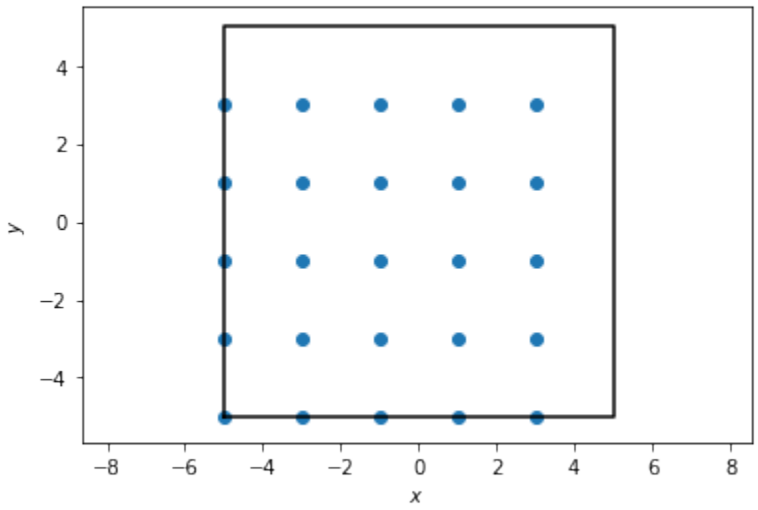
To start, we construct and visualize a set of points sitting on a simple square lattice.

[3]:
```
box, points = freud.data.UnitCell.square().generate_system(5, scale=2)
aq = freud.AABBQuery(box, points)
```

(continues on next page)

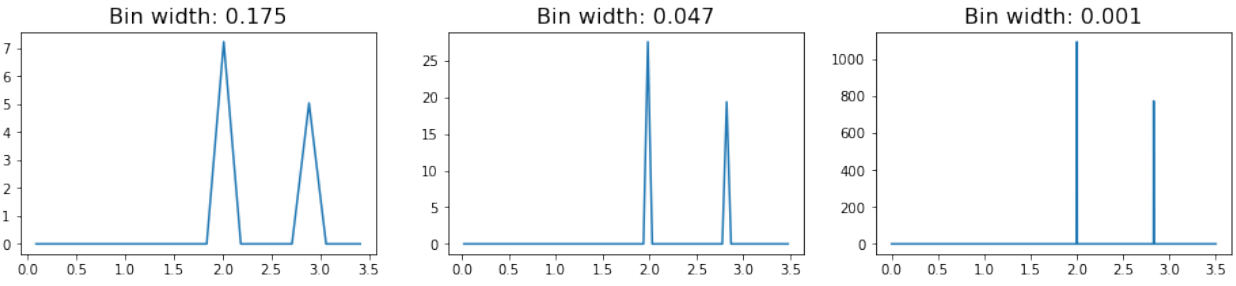
(continued from previous page)

```
aq.plot(ax=plt.gca())
plt.show()
```



If we try to compute the RDF directly from this, we will get something rather uninteresting since we have a perfect crystal. Indeed, we will observe that as we bin more and more finely, we approach the true behavior of the RDF for perfect crystals, which is a simple delta function.

[4]:
```
plot_rdf(box, points, 'rdf')
```



In these RDFs, we see two sharply defined peaks, with the first corresponding to the nearest neighbors on the lattice (which are all at a distance 2 from each other), and the second, smaller peak caused by the particles on the diagonal (which sit at distance $\sqrt{2^2 + 2^2} \approx 2.83$.

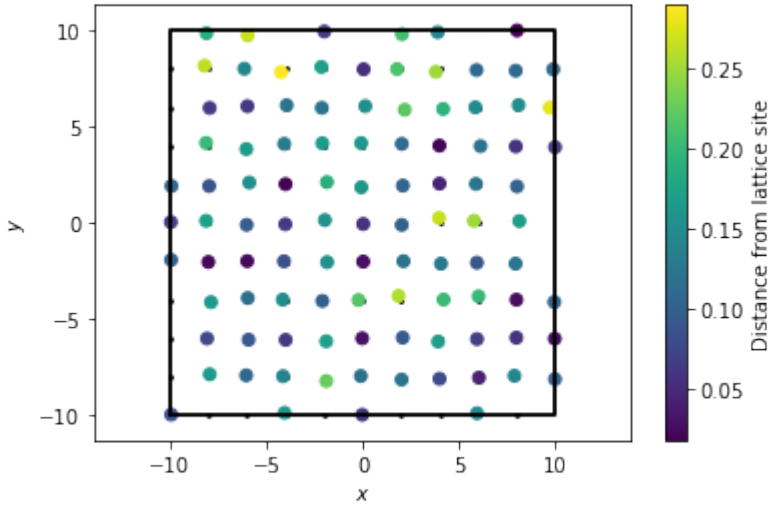
However, in more realistic systems, we expect that the lattice will not be perfectly formed. In this case, the RDF will exhibit more features. To demonstrate this fact, we reconstruct the square lattice of points from above, but we now introduce some noise into the system.

[5]:
```
box, clean_points = freud.data.UnitCell.square().generate_system(10, scale=2, sigma_
↪noise=0)
box, noisy_points = freud.data.UnitCell.square().generate_system(10, scale=2, sigma_
↪noise=0.1)
aq_clean = freud.AABBQuery(box, clean_points)
aq_clean.plot(ax=plt.gca(), c='k', s=3)
aq_noisy = freud.AABBQuery(box, noisy_points)
deviations = np.linalg.norm(box.wrap(noisy_points-clean_points), axis=-1)
_, sc = aq_noisy.plot(ax=plt.gca(), c=deviations)
cbar = plt.colorbar(sc)
```

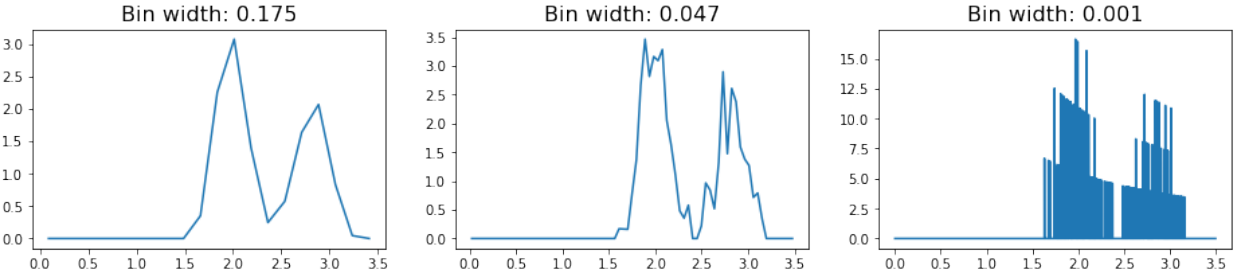
(continues on next page)

(continued from previous page)

```
cbar.set_label('Distance from lattice site')
plt.show()
```



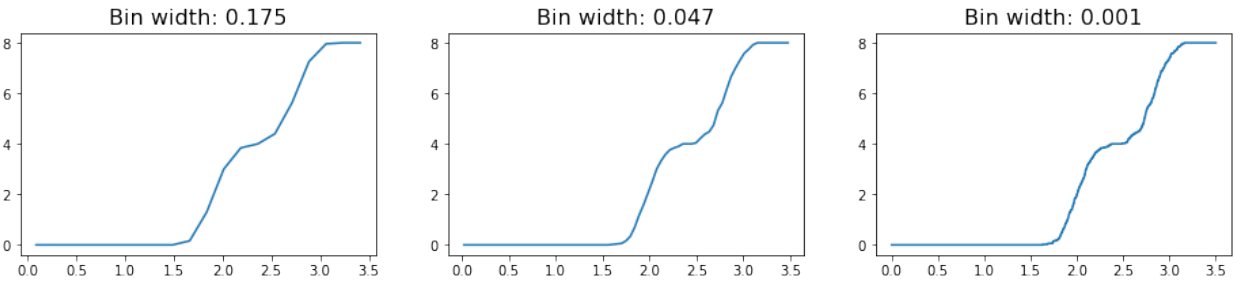
[6]:
```
plot_rdf(box, noisy_points, 'rdf')
```



In this RDF, we see the same rough features as we saw with the perfect lattice. However, the signal is much noisier, and in fact we see that increasing the number of bins essentially leads to overfitting of the data. As a result, we have to be careful with how we choose to bin our data when constructing the RDF object.

An alternative route for avoiding this problem can be using the cumulative RDF instead. The relationship between the cumulative RDF and the RDF is akin to that between a cumulative density and a probability density function, providing a measure of the total density of particles experienced up to some distance rather than the value at that distance. Just as a CDF can help avoid certain mistakes common to plotting a PDF, plotting the cumulative RDF may be helpful in some cases. Here, we see that decreasing the bin size slightly alters the features of the plot, but only in very minor way (*i.e.* decreasing the smoothness of the line due to small jitters).

[7]:
```
plot_rdf(box, noisy_points, 'n_r')
```



freud.environment.AngularSeparation

The `freud.environment` module analyzes the local environments of particles. The `freud.environment.AngularSeparation` class enables direct measurement of the relative orientations of particles.

[1]:
```
import freud
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['axes.titlepad'] = 20
from mpl_toolkits.mplot3d import Axes3D
import rowan  # for quaternion math, see rowan.readthedocs.io for more information.
```

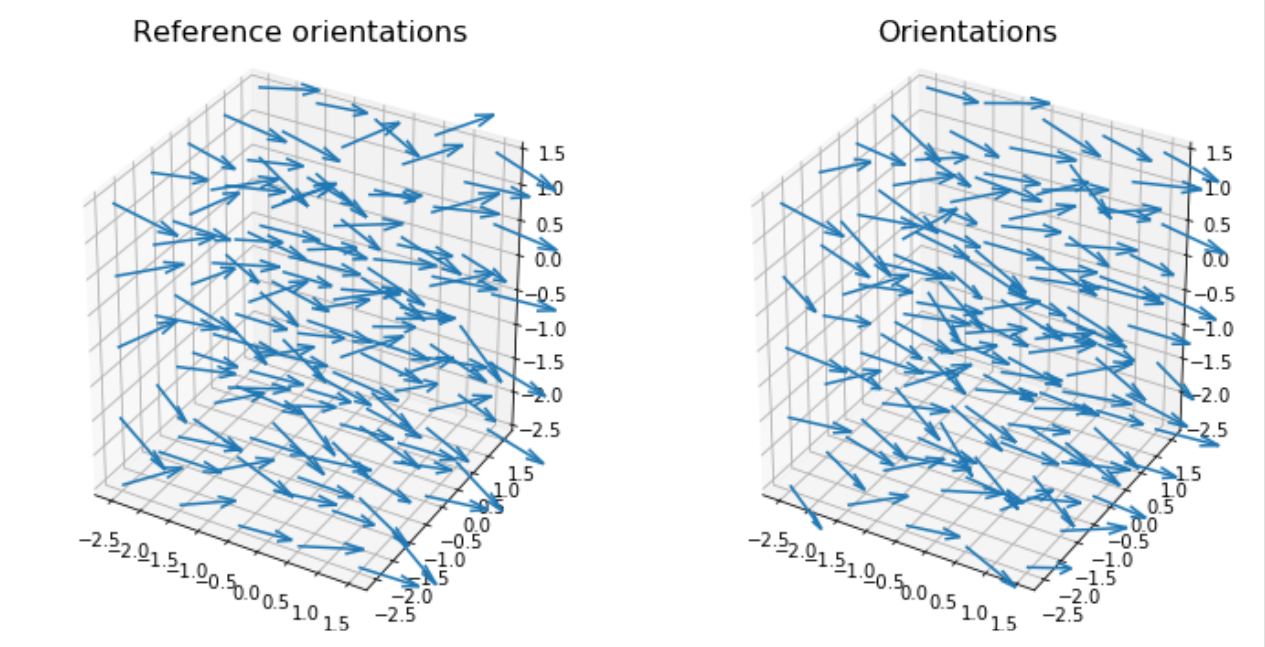
In order to work with orientations in freud, we need to do some math with quaternions. If you are unfamiliar with quaternions, you can read more about their definition and how they can be used to represent rotations. For the purpose of this tutorial, just consider them as 4D vectors, and know that the set of normalized (*i.e.* unit norm) 4D vectors can be used to represent rotations in 3D. In fact, there is a 1-1 mapping between normalized quaternions and 3x3 rotation matrices. Quaternions are more computationally convenient, however, because they only require storing 4 numbers rather than 9, and they can be much more easily chained together. The rowan library (rowan.readthedocs.io) defines many useful operations using quaternions, such as the rotations of vectors using quaternions instead of matrices.

Neighbor Angles

One usage of the AngularSeparation class is to compute angles between neighboring particles. To show how this works, we generate a simple configuration of particles with random orientations associated with each point.

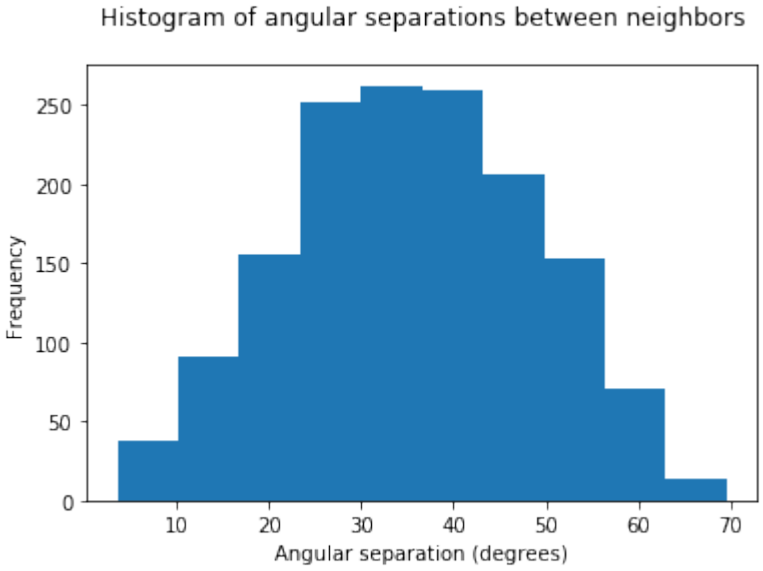
[2]:
```
uc = freud.data.UnitCell.sc()
box, positions = uc.generate_system(5)
N = len(positions)

# Generate random, correlated particle orientations by taking identity
# quaternions and slightly rotating them in a random direction
np.random.seed(0)
interpolate_amount = 0.2
identity_quats = np.array([[1, 0, 0, 0]] * N)
ref_orientations = rowan.interpolate.slerp(
    identity_quats, rowan.random.rand(N), interpolate_amount)
orientations = rowan.interpolate.slerp(
    identity_quats, rowan.random.rand(N), interpolate_amount)
```

[3]:
```
# To show orientations, we use arrows rotated by the quaternions.
ref_arrowheads = rowan.rotate(ref_orientations, np.array([1, 0, 0]))
arrowheads = rowan.rotate(orientations, np.array([1, 0, 0]))

fig = plt.figure(figsize=(12, 6))
ref_ax = fig.add_subplot(121, projection='3d')
ax = fig.add_subplot(122, projection='3d')
ref_ax.quiver3D(positions[:, 0], positions[:, 1], positions[:, 2],
                ref_arrowheads[:, 0], ref_arrowheads[:, 1], ref_arrowheads[:, 2])
ax.quiver3D(positions[:, 0], positions[:, 1], positions[:, 2],
            arrowheads[:, 0], arrowheads[:, 1], arrowheads[:, 2])
ref_ax.set_title("Reference orientations", fontsize=16)
ax.set_title("Orientations", fontsize=16)
plt.show()
```



We can now use the AngularSeparation class to compare the orientations in these two systems.

[4]:
```
# For simplicity, we'll assume that our "particles" are completely
# asymmetric, i.e. there are no rotations that map the particle
# back onto itself. If we had a regular polyhedron, then we would
# want to specify all the quaternions that rotate that polyhedron
# onto itself.
equiv_orientations = np.array([[1, 0, 0, 0]])
ang_sep = freud.environment.AngularSeparationNeighbor()
ang_sep.compute(system=(box, positions),
                orientations=orientations,
                query_points=positions,
                query_orientations=ref_orientations,
                equiv_orientations=equiv_orientations,
                neighbors={'num_neighbors': 12})

# Convert angles from radians to degrees and plot histogram
neighbor_angles = np.rad2deg(ang_sep.angles)
plt.hist(neighbor_angles)
plt.title('Histogram of angular separations between neighbors')
plt.xlabel('Angular separation (degrees)')
plt.ylabel('Frequency')
plt.show()
```

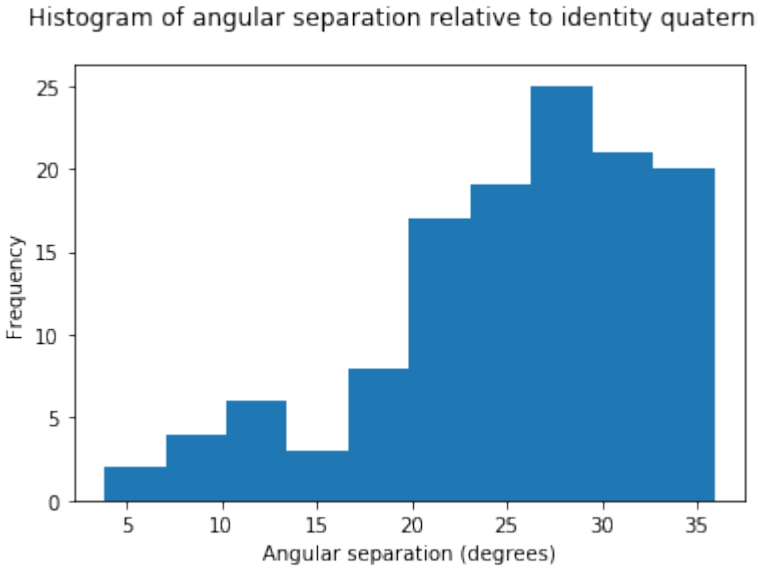


Global Angles

Alternatively, the AngularSeparationGlobal class can also be used to compute the orientation of all points in the system relative to some global set of orientations. In this case, we simply provide a set of global quaternions that we want to consider. For simplicity, let's consider 180° rotations about each of the coordinate axes, which have very simple quaternion representations.

[5]:
```
global_orientations = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0,␣
↪1]])
ang_sep = freud.environment.AngularSeparationGlobal()
ang_sep.compute(global_orientations, ref_orientations, equiv_orientations)
global_angles = np.rad2deg(ang_sep.angles)
```

[6]:
```
plt.hist(global_angles[:, 0])
plt.title('Histogram of angular separation relative to identity quaternion')
plt.xlabel('Angular separation (degrees)')
plt.ylabel('Frequency')
plt.show()
```



As a simple check, we can ensure that for the identity quaternion $(1, 0, 0, 0)$, which performs a $0°$ rotation, the angles between the reference orientations and that quaternion are equal to the original angles of rotation of those quaternions (*i.e.* how much those orientations were already rotated relative to the identity).

[7]:
```
ref_axes, ref_angles = rowan.to_axis_angle(ref_orientations)
np.allclose(global_angles[:, 0], np.rad2deg(ref_angles), rtol=1e-4)
```

[7]:
```
True
```

freud.environment.BondOrder

Computing Bond Order Diagrams

The `freud.environment` module analyzes the local environments of particles. In this example, the `freud.environment.BondOrder` class is used to plot the bond order diagram (BOD) of a system of particles.

[1]:
```
import numpy as np
import freud
import matplotlib.pyplot as plt
import matplotlib
from mpl_toolkits.mplot3d import Axes3D
```

Setup

Our sample data will be taken from an face-centered cubic (FCC) structure. The array of points is rather large, so that the plots are smooth. Smaller systems may need to gather data from multiple frames in order to smooth the resulting array's statistics, by computing multiple times with `reset=False`.

[2]:
```
uc = freud.data.UnitCell.fcc()
box, points = uc.generate_system(40, sigma_noise=0.05)
```

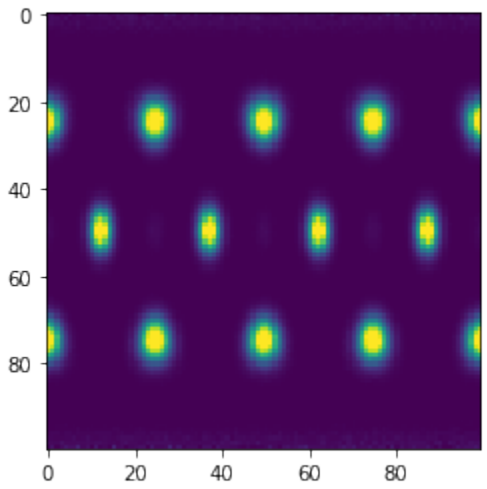
Now we create a `BondOrder` compute object and create some arrays useful for plotting.

[3]:
```
n_bins_theta = 100
n_bins_phi = 100
bod = freud.environment.BondOrder((n_bins_theta, n_bins_phi))

phi = np.linspace(0, np.pi, n_bins_phi)
theta = np.linspace(0, 2*np.pi, n_bins_theta)
phi, theta = np.meshgrid(phi, theta)
x = np.sin(phi) * np.cos(theta)
y = np.sin(phi) * np.sin(theta)
z = np.cos(phi)
```

Computing the Bond Order Diagram

Next, we use the `compute` method and the `bond_order` property to return the array.

[4]:
```
bod_array = bod.compute(system=(box, points), neighbors={'num_neighbors': 12}).bond_
↪order
# Clean up polar bins for plotting
bod_array = np.clip(bod_array, 0, np.percentile(bod_array, 99))
plt.imshow(bod_array.T)
plt.show()
```



Plotting on a sphere

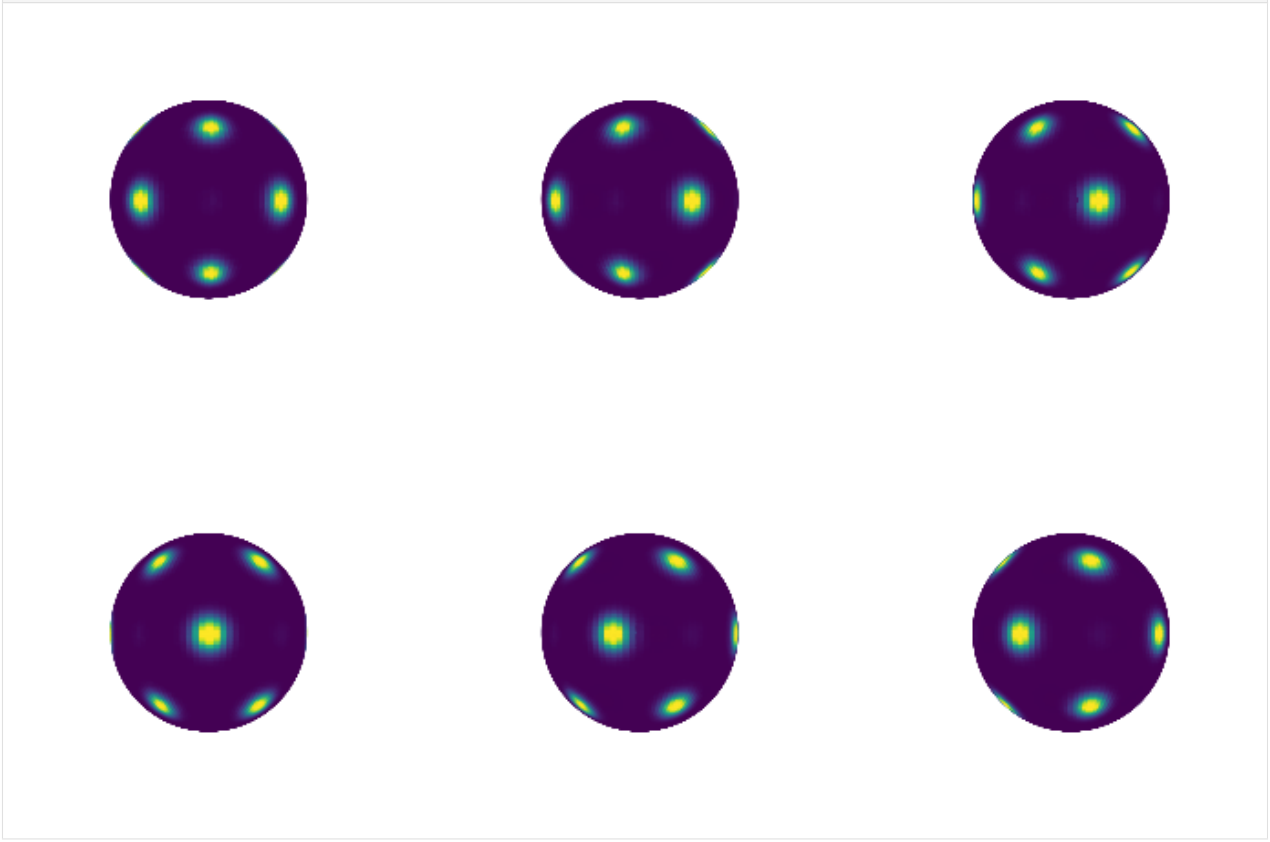
This code shows the bond order diagram on a sphere as the sphere is rotated. The code takes a few seconds to run, so be patient.

[5]:
```
fig = plt.figure(figsize=(12, 8))
for plot_num in range(6):
    ax = fig.add_subplot(231 + plot_num, projection='3d')
    ax.plot_surface(x, y, z, rstride=1, cstride=1, shade=False,
                    facecolors=matplotlib.cm.viridis(bod_array / np.max(bod_array)))
    ax.set_xlim(-1, 1)
    ax.set_ylim(-1, 1)
    ax.set_zlim(-1, 1)
    ax.set_axis_off()
```

(continues on next page)

(continued from previous page)

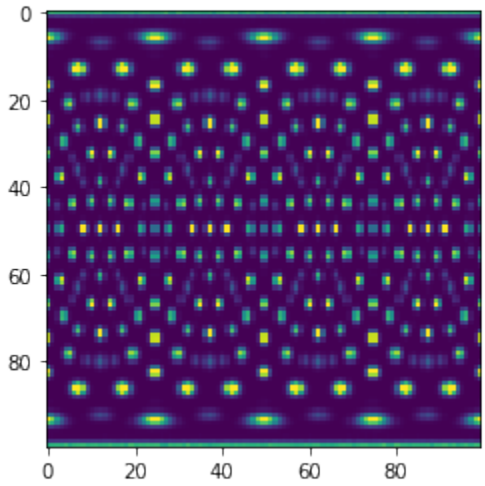
```
    # View angles in degrees
    view_angle = 0, plot_num*15
    ax.view_init(*view_angle)
plt.show()
```



Using Custom Neighbors

We can also use a custom neighbor query to determine bonds. For example, we can filter for a range of bond lengths. Below, we only consider neighbors between $r_{min} = 2.5$ and $r_{max} = 3$ and plot the resulting bond order diagram.

[6]:
```
bod_array = bod.compute(system=(box, points), neighbors={'r_max': 3.0, 'r_min': 2.5}).
↪bond_order
# Clean up polar bins for plotting
bod_array = np.clip(bod_array, 0, np.percentile(bod_array, 99))
plt.imshow(bod_array.T)
plt.show()
```



freud.environment.EnvironmentCluster

The `freud.environment.EnvironmentCluster` class finds and clusters local environments, as determined by the vectors pointing to neighbor particles. Neighbors can be defined by a cutoff distance or a number of nearest-neighbors, and the resulting `freud.locality.NeighborList` is used to enumerate a set of vectors, defining an "environment." These environments are compared with the environments of neighboring particles to form spatial clusters, which usually correspond to grains, droplets, or crystalline domains of a system. `EnvironmentCluster` has several parameters that alter its behavior, please see the documentation or helper functions below for descriptions of these parameters.

In this example, we cluster the local environments of hexagons. Clusters with 5 or fewer particles are colored dark gray.

[1]:
```
import numpy as np
import freud
from collections import Counter
import matplotlib.pyplot as plt

def get_cluster_arr(system, num_neighbors, threshold,
                    registration=False, global_search=False):
    """Computes clusters of particles' local environments.

    Args:
        system:
            Any object that is a valid argument to
            :class:`freud.locality.NeighborQuery.from_system`.
        num_neighbors (int):
            Number of neighbors to consider in every particle's local environment.
        threshold (float):
            Maximum magnitude of the vector difference between two vectors,
            below which we call them matching.
        global_search (bool):
            If True, do an exhaustive search wherein the environments of
            every single pair of particles in the simulation are compared.
            If False, only compare the environments of neighboring particles.
        registration (bool):
            Controls whether we first use brute force registration to
```

(continues on next page)

(continued from previous page)

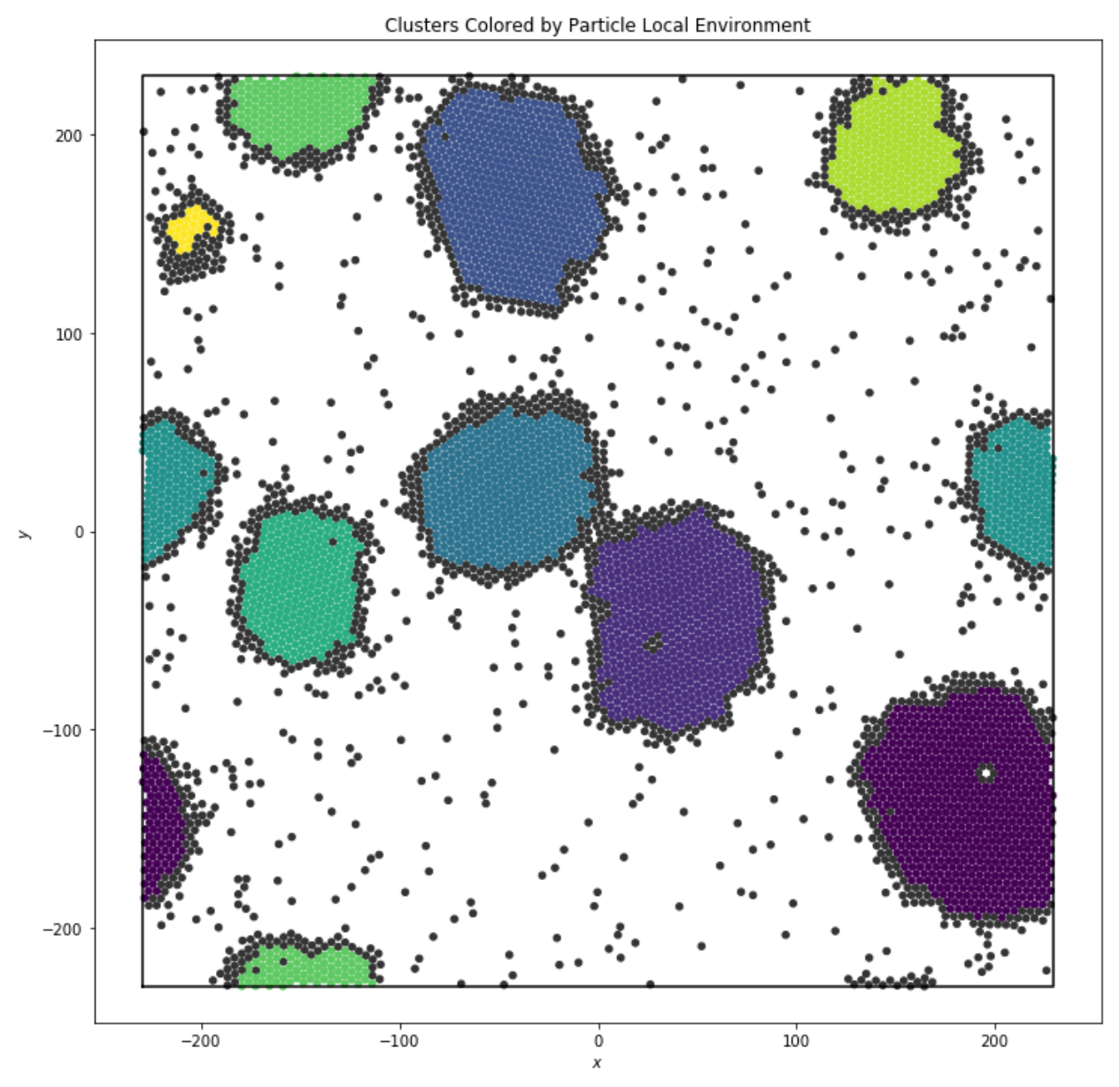
```
            orient the second set of vectors such that it minimizes the
            RMSD between the two sets.

    Returns:
        tuple(np.ndarray, dict): array of cluster indices for every particle
        and a dictionary mapping from cluster_index keys to vector_array)
        pairs giving all vectors associated with each environment.
    """
    # Perform the env-matching calcuation
    neighbors = {'num_neighbors': num_neighbors}
    match = freud.environment.EnvironmentCluster()
    match.compute(system, threshold, neighbors=neighbors,
                  registration=registration, global_search=global_search)
    return match.cluster_idx, match.cluster_environments

def color_by_clust(cluster_index_arr, no_color_thresh=1,
                   no_color='#333333', cmap=plt.get_cmap('viridis')):
    """Takes a cluster_index_array for every particle and returns a
    dictionary of (cluster index, hexcolor) color pairs.

    Args:
        cluster_index_arr (numpy.ndarray):
            The array of cluster indices, one per particle.
        no_color_thresh (int):
            Clusters with this number of particles or fewer will be
            colored with no_color.
        no_color (color):
            What we color particles whose cluster size is below no_color_thresh.
        cmap (color map):
            The color map we use to color all particles whose
            cluster size is above no_color_thresh.
    """
    # Count to find most common clusters
    cluster_counts = Counter(cluster_index_arr)
    # Re-label the cluster indices by size
    color_count = 0
    color_dict = {cluster[0]: counter for cluster, counter in
                  zip(cluster_counts.most_common(),
                      range(len(cluster_counts)))}

    # Don't show colors for clusters below the threshold
    for cluster_id in cluster_counts:
        if cluster_counts[cluster_id] <= no_color_thresh:
            color_dict[cluster_id] = -1
    OP_arr = np.linspace(0.0, 1.0, max(color_dict.values())+1)

    # Get hex colors for all clusters of size greater than no_color_thresh
    for old_cluster_index, new_cluster_index in color_dict.items():
        if new_cluster_index == -1:
            color_dict[old_cluster_index] = no_color
        else:
            color_dict[old_cluster_index] = cmap(OP_arr[new_cluster_index])

    return color_dict
```

We load the simulation data and call the analysis functions defined above. Notice that we use 6 nearest neighbors, since our system is made of hexagons that tend to cluster with 6 neighbors.

[2]:
```
ex_data = np.load('data/MatchEnv_Hexagons.npz')
box = ex_data['box']
positions = ex_data['positions']
orientations = ex_data['orientations']
aq = freud.AABBQuery(box, positions)

cluster_index_arr, cluster_envs = get_cluster_arr(
    aq, num_neighbors=6, threshold=0.2,
    registration=False, global_search=False)
color_dict = color_by_clust(cluster_index_arr, no_color_thresh=5)
colors = [color_dict[i] for i in cluster_index_arr]
```

Below, we plot the resulting clusters. The colors correspond to the cluster size.

[3]:
```
plt.figure(figsize=(12, 12), facecolor='white')
aq.plot(ax=plt.gca(), c=colors, s=20)
plt.title('Clusters Colored by Particle Local Environment')
plt.show()
```



freud.environment.LocalDescriptors: Steinhardt Order Parameters from Scratch

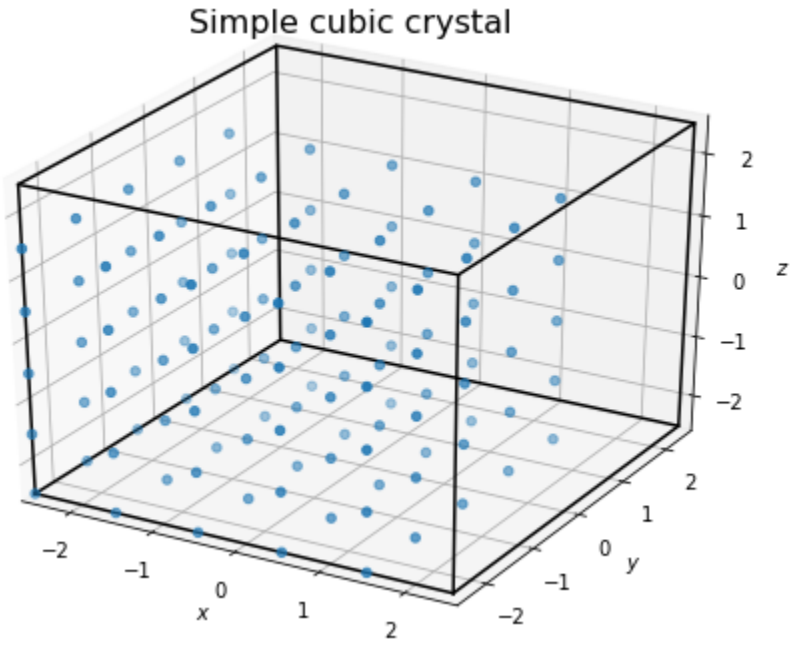
The `freud.environment` module analyzes the local environments of particles. The `freud.environment.LocalDescriptors` class is a useful tool for analyzing identifying crystal structures in a rotationally invariant manner using local particle environments. The primary purpose of this class is to compute spherical harmonics between neighboring particles in a way that orients particles correctly relative to their local environment, ensuring that global orientational shifts do not change the output.

[1]:
```
import freud
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

Computing Spherical Harmonics

To demonstrate the basic application of the class, let's compute the spherical harmonics between neighboring particles. For simplicity, we consider points on a simple cubic lattice.

[2]:
```
uc = freud.data.UnitCell.sc()
box, points = uc.generate_system(5)
system = freud.AABBQuery(box, points)
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
system.plot(ax=ax)
ax.set_title("Simple cubic crystal", fontsize=16)
plt.show()
```



Now, let's use the class to compute an array of spherical harmonics for the system. The harmonics are computed for each bond, where a bond is defined by a pair of particles that are determined to lie within each others' nearest neighbor shells based on a standard neighbor list search. The number of bonds and spherical harmonics to calculate is configurable.

[3]:
```
num_neighbors = 6
l_max = 12

nlist = system.query(points, {'num_neighbors': num_neighbors, 'exclude_ii': True}).
↪toNeighborList()
ld = freud.environment.LocalDescriptors(l_max, mode='global')
ld.compute(system, neighbors=nlist);
```

Accessing the Data

The resulting spherical harmonic array has a shape corresponding to the number of neighbors. We can now extract the spherical harmonics corresponding to a particular (l, m) pair using the ordering used by the `LocalDescriptors` class: increasing values of l, and for each l, the nonnegative m values followed by the negative values.

[4]:
```
sph_raw = np.mean(ld.sph, axis=0)
count = 0
sph = np.zeros((l_max+1, l_max+1), dtype=np.complex128)
for l in range(l_max+1):
    for m in range(l+1):
        sph[l, m] = sph_raw[count]
        count += 1
    for m in range(-l, 0):
        sph[l, m] = sph_raw[count]
        count += 1
```

Using Spherical Harmonics to Compute Steinhardt Order Parameters

The raw per bond spherical harmonics are not typically useful quantities on their own. However, they can be used to perform sophisticated crystal structure analyses with different methods; for example, the pythia library uses machine learning to find patterns in the spherical harmonics computed by this class. In this notebook, we'll use the quantities for a more classical application: the computation of Steinhardt order parameters. The order parameters q_l provide a rotationally invariant measure of the system that can for some structures, provide a unique identifying fingerprint. They are a particularly useful measure for various simple cubic structures such as structures with underlying simple cubic, BCC, or FCC lattices. The `freud` library actually provides additional classes to efficiently calculate these order parameters directly, but they also provide a reasonable demonstration here.

For more information on Steinhardt order parameters, see the original paper or the `freud.order.Steinhardt` documentation.

[5]:
```
def get_ql(num_particles, descriptors, nlist, weighted=False):
    """Given a set of points and a LocalDescriptors object (and the
    underlying NeighborList), compute the per-particle Steinhardt ql
    order parameter for all :math:`l` values up to the maximum quantum
    number used in the computation of the descriptors."""
    qbar_lm = np.zeros((num_particles, descriptors.sph.shape[1]),
                       dtype=np.complex128)
    for i in range(num_particles):
        indices = nlist.query_point_indices == i
        Ylms = descriptors.sph[indices, :]
        if weighted:
            weights = nlist.weights[indices, np.newaxis]
            weights /= np.sum(weights)
            num_neighbors = 1
        else:
            weights = np.ones_like(Ylms)
            num_neighbors = descriptors.sph.shape[0]/num_particles
        qbar_lm[i, :] = np.sum(Ylms * weights, axis=0)/num_neighbors

    ql = np.zeros((qbar_lm.shape[0], descriptors.l_max+1))
    for i in range(ql.shape[0]):
        for l in range(ql.shape[1]):
            for k in range(l**2, (l+1)**2):
                ql[i, l] += np.absolute(qbar_lm[i, k])**2
```

(continues on next page)

(continued from previous page)

```
            ql[i, l] = np.sqrt(4*np.pi/(2*l + 1) * ql[i, l])

    return ql

ld_ql = get_ql(len(points), ld, nlist)
```

Since `freud` provides the ability to calculate these parameter as well, we can directly check that our answers are correct. *Note: More information on the ``Steinhardt`` class can be found in the documentation or in the ``Steinhardt`` example.*

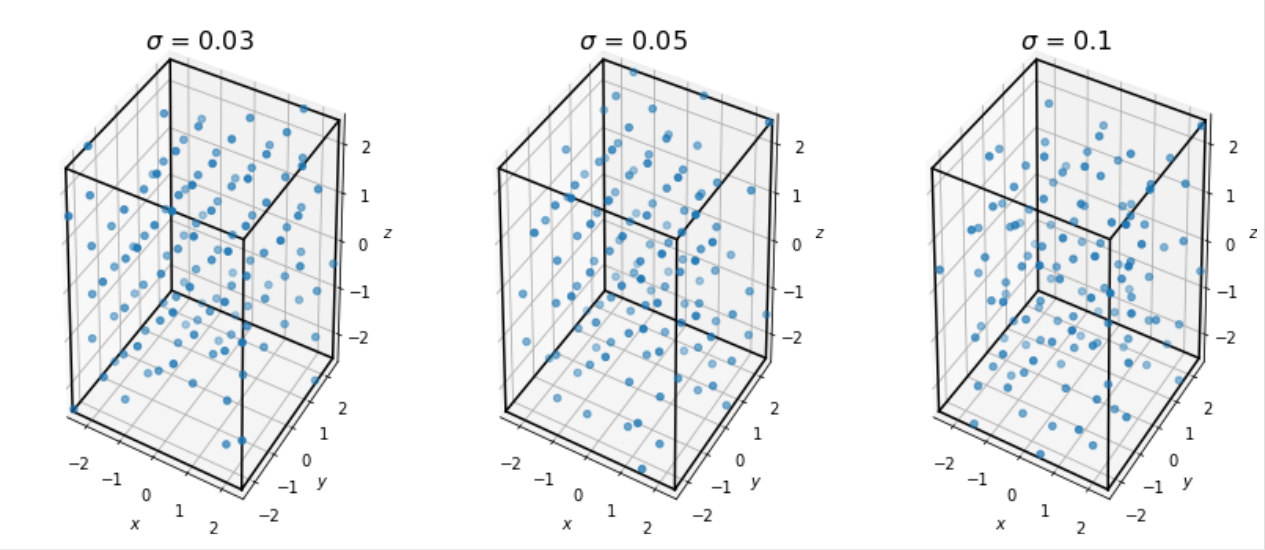
[6]:
```
L = 6
steinhardt = freud.order.Steinhardt(l=L)
steinhardt.compute(system, neighbors=nlist)
if np.allclose(steinhardt.ql, ld_ql[:, L]):
    print("Our manual calculation matches the Steinhardt class!")
```

```
Our manual calculation matches the Steinhardt class!
```

For a brief demonstration of why the Steinhardt order parameters can be useful, let's look at the result of thermalizing our points and recomputing this measure.

[7]:
```
sigmas = [0.03, 0.05, 0.1]
systems = []
nlists = []
for sigma in sigmas:
    box, points = uc.generate_system(5, sigma_noise=sigma)
    system = freud.AABBQuery(box, points)
    systems.append(system)
    nlists.append(
        system.query(
            points, {'num_neighbors': num_neighbors, 'exclude_ii': True}
        ).toNeighborList()
    )
```

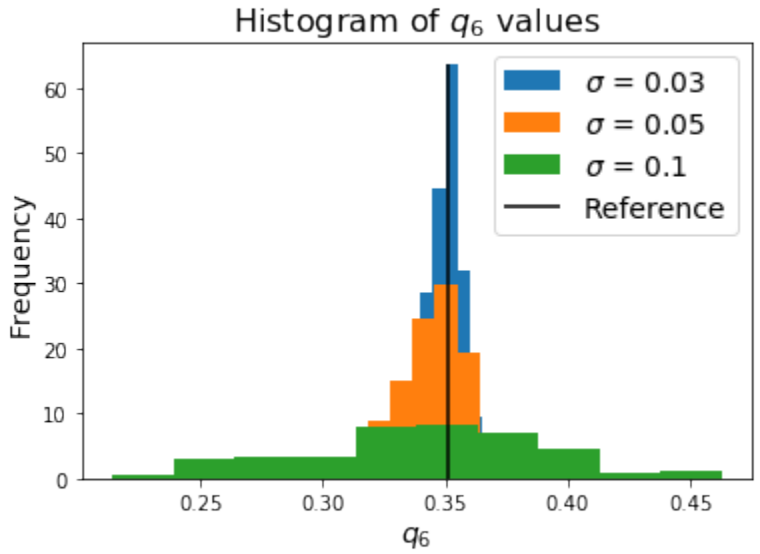
[8]:
```
fig = plt.figure(figsize=(14, 6))
axes = []
for i, v in enumerate(sigmas):
    ax = fig.add_subplot("1{}{}".format(len(sigmas), i+1), projection='3d')
    systems[i].plot(ax=ax)
    ax.set_title("$\sigma$ = {}".format(v), fontsize=16);
plt.show()
```



If we recompute the Steinhardt OP for each of these data sets, we see that adding noise has the effect of smoothing the order parameter such that the peak we observed for the perfect crystal is no longer observable.

[9]:
```
ld_qls = []
for i, sigma in enumerate(sigmas):
    ld = freud.environment.LocalDescriptors(l_max, mode='global')
    ld.compute(systems[i], neighbors=nlists[i])
    ld_qls.append(get_ql(len(systems[i].points), ld, nlists[i]))
```

[10]:
```
fig, ax = plt.subplots()
for i, ld_ql in enumerate(ld_qls):
    lim_out = ax.hist(ld_ql[:, L], label="$\sigma$ = {}".format(sigmas[i]),␣
↪density=True)
    if i == 0:
        # Can choose any element, all are identical in the reference case
        ax.vlines(ld_ql[:, L][0], 0, np.max(lim_out[0]), label='Reference')
ax.set_title("Histogram of $q_{L}$ values".format(L=L), fontsize=16)
ax.set_ylabel("Frequency", fontsize=14)
ax.set_xlabel("$q_{L}$".format(L=L), fontsize=14)
ax.legend(fontsize=14)
plt.show()
```



This type of identification process is what the LocalDescriptors data outputs may be used for. In the case of Steinhardt OPs, it provides a simple fingerprint for comparing thermalized systems to a known ideal structure to measure their similarity.

For reference, we can also check these values against the `Steinhardt` class again.

[11]:
```
for i, (system, nlist) in enumerate(zip(systems, nlists)):
    steinhardt = freud.order.Steinhardt(l=L)
    steinhardt.compute(system, nlist)
    if np.allclose(steinhardt.particle_order, ld_qls[i][:, L]):
        print("Our manual calculation matches the Steinhardt class!")
```

```
Our manual calculation matches the Steinhardt class!
Our manual calculation matches the Steinhardt class!
Our manual calculation matches the Steinhardt class!
```

freud.interface.Interface

Locating Particles on Interfacial Boundaries

The `freud.interface` module compares the distances between two sets of points to determine the interfacial particles.

[1]:
```
import freud
import numpy as np
import matplotlib.pyplot as plt
```

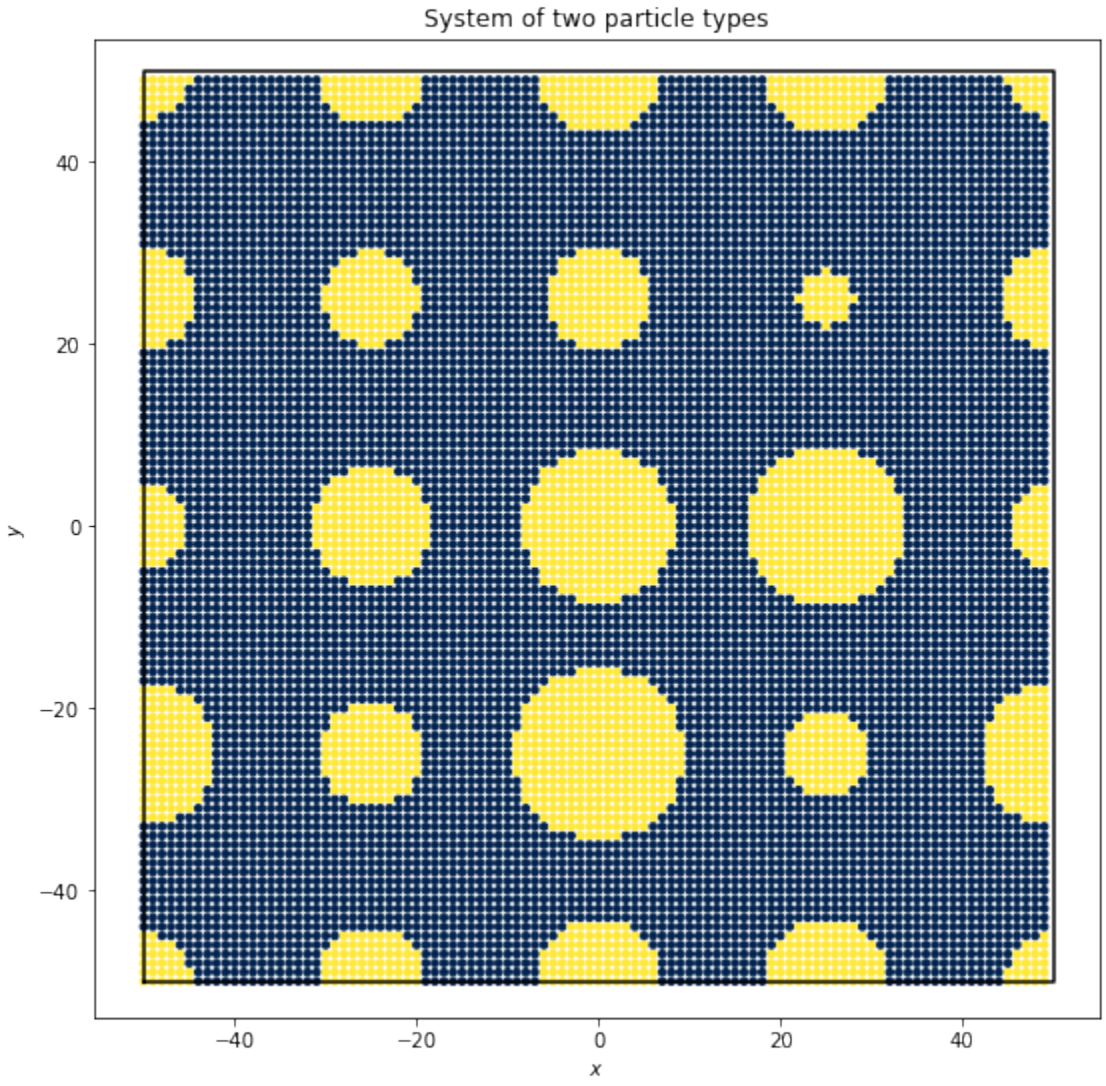
To make a pretend data set, we create a large number of **blue (-1)** particles on a square grid. Then we place grain centers on a larger grid and draw grain radii from a normal distribution. We color the particles **yellow (+1)** if their distance from a grain center is less than the grain radius.

[2]:
```
np.random.seed(0)
system_size = 100
num_grains = 4
uc = freud.data.UnitCell.square()
```

(continues on next page)

(continued from previous page)

```
box, points = uc.generate_system(num_replicas=system_size, scale=1)
_, centroids = uc.generate_system(num_replicas=num_grains, scale=system_size/num_
↪grains)
system = freud.AABBQuery(box, points)
values = np.array([-1 for p in points])
grain_radii = np.abs(np.random.normal(size=num_grains**2, loc=5, scale=2))
for center, radius in zip(centroids, grain_radii):
    for i, j, dist in system.query(center, {'r_max': radius}):
        values[j] = 1
plt.figure(figsize=(9, 9))
system.plot(ax=plt.gca(), c=values, cmap='cividis', s=12)
plt.title('System of two particle types')
plt.show()
```



This system is **phase-separated** because the yellow particles are generally near one another, and so are the blue particles.

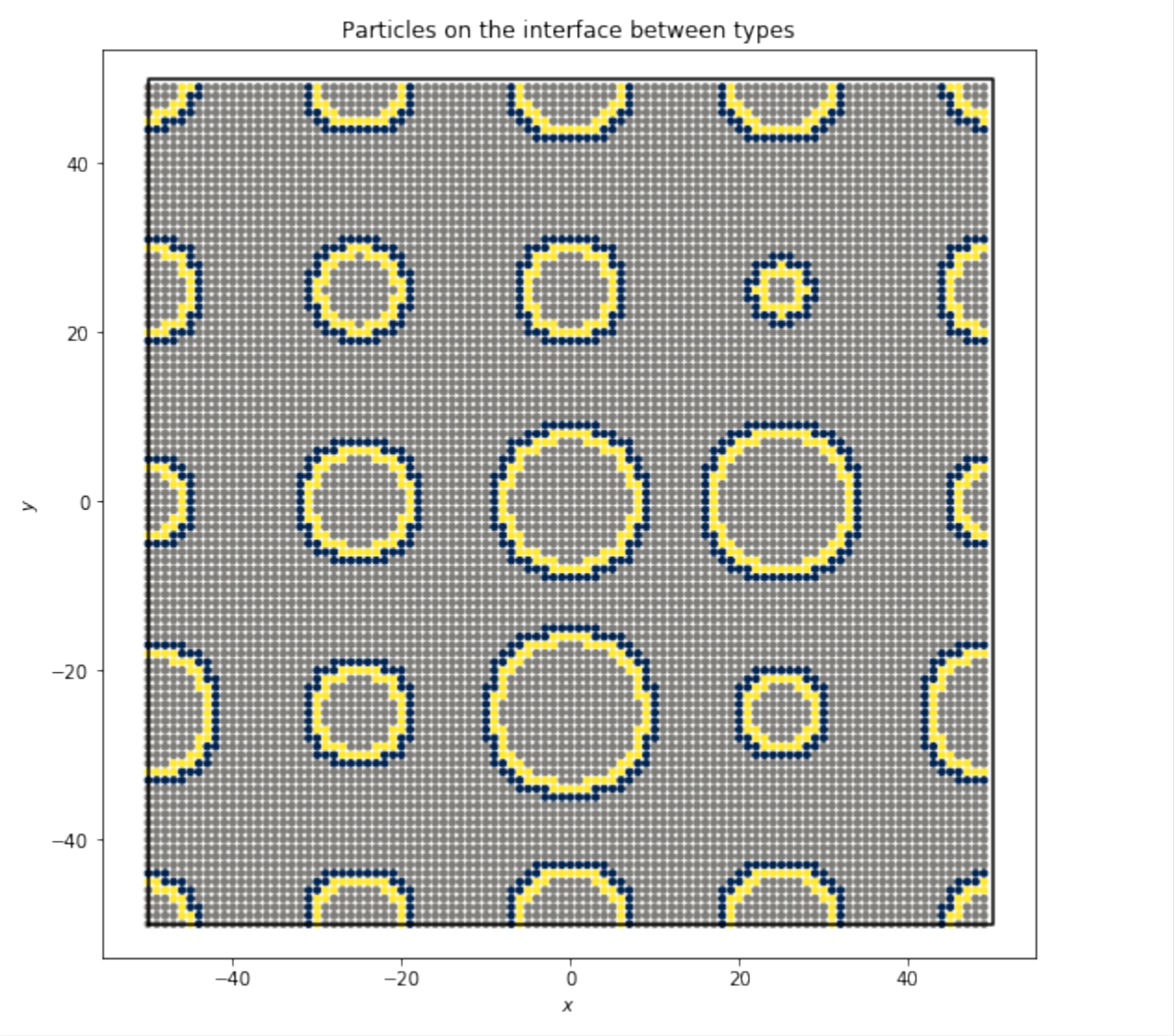
We can use `freud.interface.InterfaceMeasure` to label the particles on either side of the yellow-blue boundary. The class can tell us how many points are on either side of the interface:

[3]:
```
iface = freud.interface.Interface()
iface.compute((box, points[values > 0]), points[values < 0], neighbors={'r_max': 1.5})

print('There are {} query points (blue) on the interface.'.format(iface.query_point_
↪count))
print('There are {} points (yellow) on the interface.'.format(iface.point_count))
```

```
There are 856 query points (blue) on the interface.
There are 724 points (yellow) on the interface.
```

Now we can plot the particles on the interface. We color the outside of the interface dark blue and the inside of the interface yellow.

[4]:
```
plt.figure(figsize=(9, 9))

interface_values = np.zeros(len(points))
interface_values[np.where(values < 0)[0][iface.query_point_ids]] = -1
interface_values[np.where(values > 0)[0][iface.point_ids]] = 1

system.plot(ax=plt.gca(), c=interface_values, cmap='cividis', s=12)
plt.title('Particles on the interface between types')
plt.show()
```



freud.order.Hexatic: Hard Hexagons

Hexatic Order Parameter

The hexatic order parameter measures how closely the local environment around a particle resembles perfect k-atic symmetry, *e.g.* how closely the environment resembles hexagonal/hexatic symmetry for $k = 6$. The order parameter is given by:

$$\psi_k\left(i\right) = \frac{1}{n}\sum_j^n e^{ki\theta_{ij}}$$

where θ_{ij} is the angle between the vector $\vec{r}_{ij}$ and $(1, 0)$.

The pseudocode is given below:

```
for each particle i:
    neighbors = nearestNeighbors(i, n):
```

(continues on next page)

(continued from previous page)

```
    for each particle j in neighbors:
        r_ij = position[j] - position[i]
        theta_ij = arctan2(r_ij.y, r_ij.x)
        psi_array[i] += exp(complex(0,k*theta_ij))
```

The data sets used in this example are a system of hard hexagons, simulated in the NVT thermodynamic ensemble in HOOMD-blue, for a dense fluid of hexagons at packing fraction $\phi = 0.65$ and solids at packing fractions $\phi = 0.75, 0.85$.

[1]:
```
import numpy as np
import freud
from bokeh.io import output_notebook
from bokeh.plotting import figure, show
# The following line imports this set of utility functions:
# https://github.com/glotzerlab/freud-examples/blob/master/util.py
import util
output_notebook()
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

[2]:
```
def plot_hex_order_param(data_path, title):
    # Create hexatic object
    hex_order = freud.order.Hexatic(k=6)

    # Load the data
    box_data = np.load("{}/box_data.npy".format(data_path))
    pos_data = np.load("{}/pos_data.npy".format(data_path))
    quat_data = np.load("{}/quat_data.npy".format(data_path))

    # Grab data from last frame
    box = box_data[-1].tolist()
    points = pos_data[-1]
    quats = quat_data[-1]
    angles = 2*np.arctan2(quats[:, 3], quats[:, 0])

    # Compute hexatic order for 6 nearest neighbors
    hex_order.compute(system=(box, points), neighbors={'num_neighbors': 6})
    psi_k = hex_order.particle_order
    avg_psi_k = np.mean(psi_k)

    # Create hexagon vertices
    verts = util.make_polygon(sides=6, radius=0.6204)
    # Create array of transformed positions
    patches = util.local_to_global(verts, points[:, :2], angles)
    # Create an array of angles relative to the average
    relative_angles = np.angle(psi_k) - np.angle(avg_psi_k)
    # Plot in bokeh
    p = figure(title=title)
    p.patches(xs=patches[:, :, 0].tolist(), ys=patches[:, :, 1].tolist(),
              fill_color=[util.cubeellipse(x) for x in relative_angles],
              line_color="black")
    util.default_bokeh(p)
    show(p)
```

[3]:
```
plot_hex_order_param('data/phi065', 'Hexatic Order Parameter, 0.65 density')
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

As the density increases to $\phi = 0.75$, the shapes are forced to align more closely so that they may tile space effectively.

[4]:
```
plot_hex_order_param('data/phi075', 'Hexatic Order Parameter, 0.75 density')
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

As the density increases to $\phi = 0.85$, the alignment becomes even stronger and defects are no longer visible.

[5]:
```
plot_hex_order_param('data/phi085', 'Hexatic Order Parameter, 0.85 density')
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

freud.order.Hexatic: 2D Minkowski Structure Metrics

This demonstrates a variant of the hexatic order parameter ψ_k that weighs each neighbor bond according to its corresponding side length in a Voronoi diagram of the system. This variant, called a Minkowski Structure Metric, is invariant under rotation, translation, and scaling. We denote the 2D Minkowski Structure Metric (the Voronoi-weighted form of the hexatic order parameter) as ψ'_k.

See also: - https://morphometry.org/theory/anisotropy-analysis-by-imt/ - https://aip.scitation.org/doi/10.1063/1.4774084

[1]:
```
import numpy as np
import freud
from mpl_toolkits.axes_grid1.axes_divider import make_axes_locatable
from matplotlib.colorbar import Colorbar
```

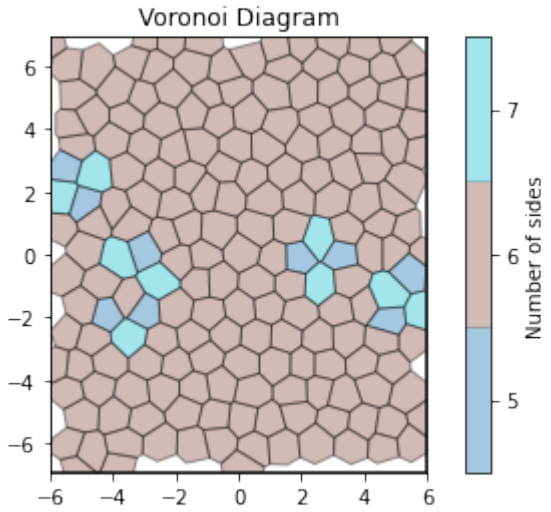
[2]:
```
def show_minkowski_structure_metrics(system):
    voro = freud.locality.Voronoi()
    voro.compute(system)
    voro.plot()
    for k in [0, 1, 2, 3, 4, 5, 6, 7, 8]:
        psi = freud.order.Hexatic(k=k, weighted=True)
        psi.compute(system, neighbors=voro.nlist)
        order = np.absolute(psi.particle_order)

        ax = voro.plot()
        patches = ax.collections[0]
        patches.set_array(order)
        patches.set_cmap('viridis')
        patches.set_clim(0, 1)
        patches.set_alpha(0.7)
        # Remove old colorbar coloring by number of sides
        ax.figure.delaxes(ax.figure.axes[-1])
```

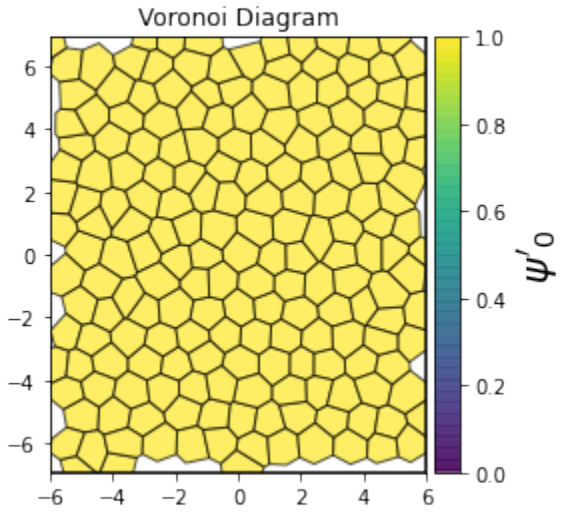
(continues on next page)

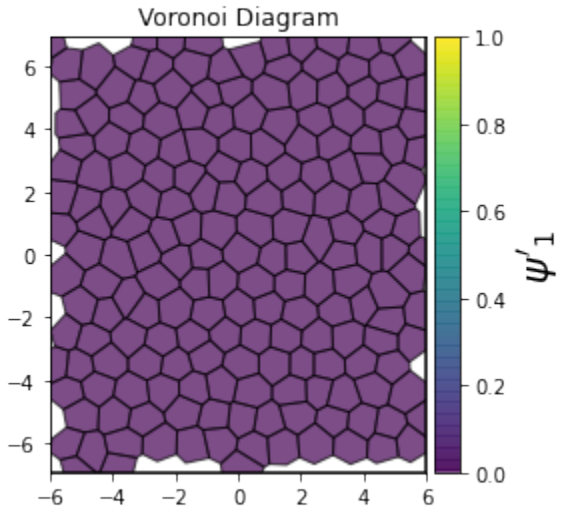
(continued from previous page)

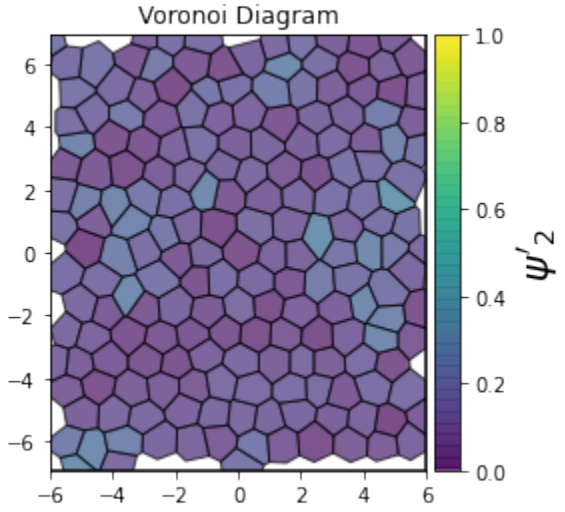
```
        ax_divider = make_axes_locatable(ax)
        # Add a new colorbar to the right of the main axes.
        cax = ax_divider.append_axes("right", size="7%", pad="2%")
        cbar = Colorbar(cax, patches)
        cbar.set_label("$\psi'_{k}$".format(k=k), size=20)
        ax
```

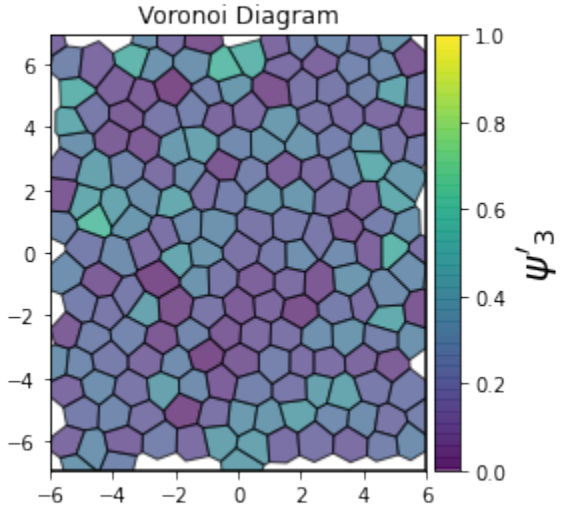
[3]:
```
unit_cell = freud.data.UnitCell.hex()
system = unit_cell.generate_system(num_replicas=[12, 8, 1], sigma_noise=0.15)
show_minkowski_structure_metrics(system)
```

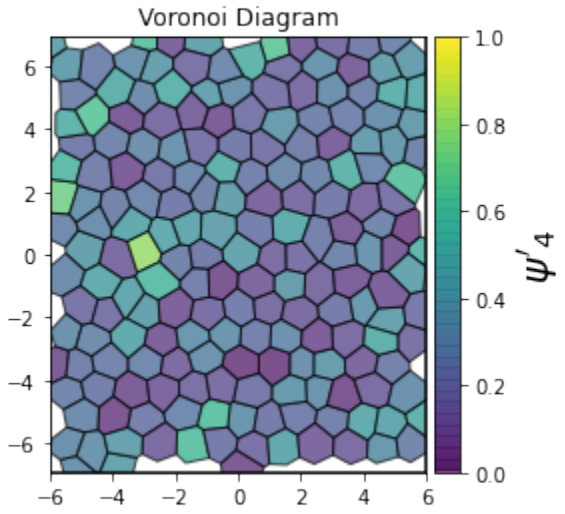


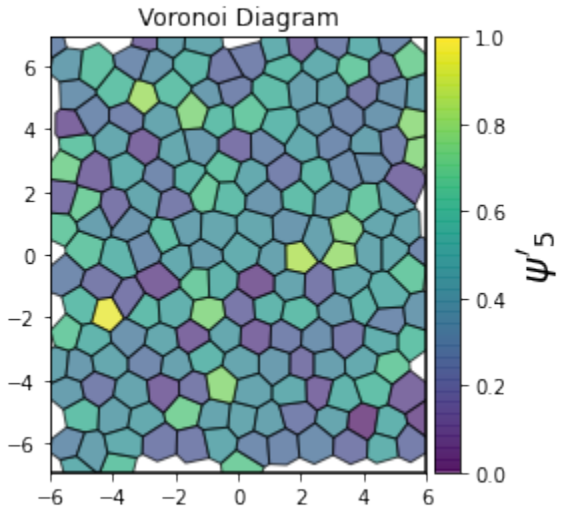


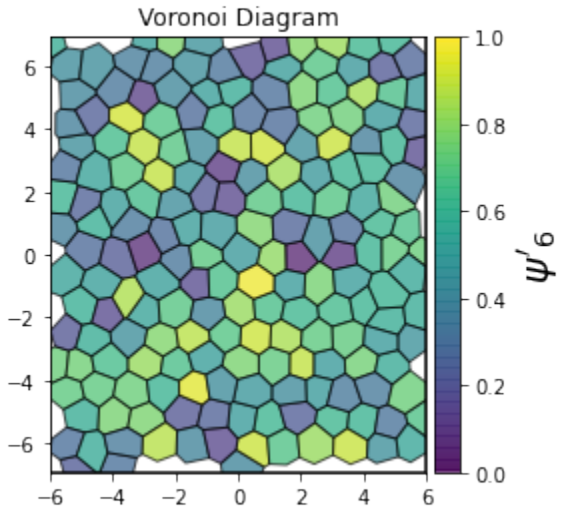

Voronoi Diagram
ψ'_1

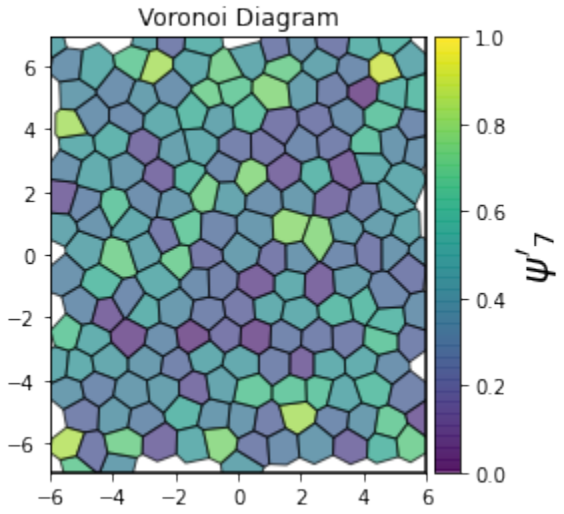

Voronoi Diagram
ψ'_2

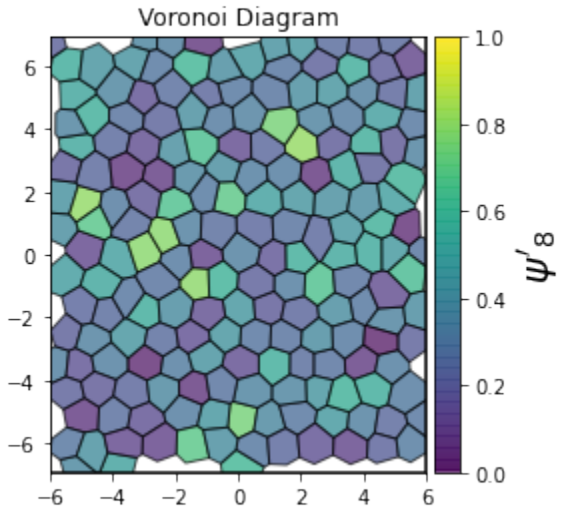

Voronoi Diagram
ψ'_3


Voronoi Diagram
ψ'_4

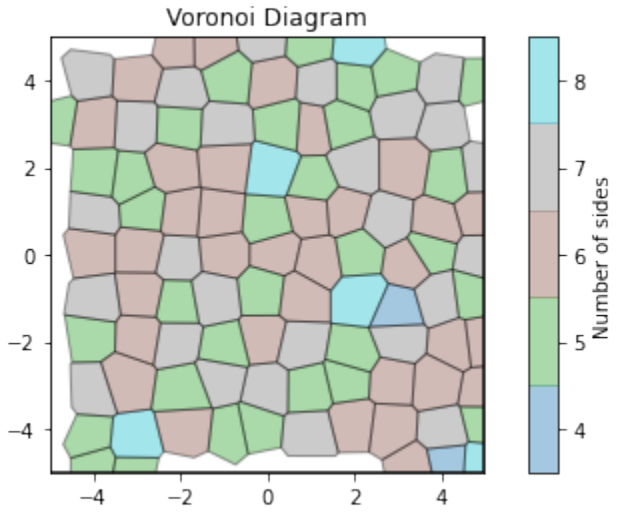

Voronoi Diagram
ψ'_5

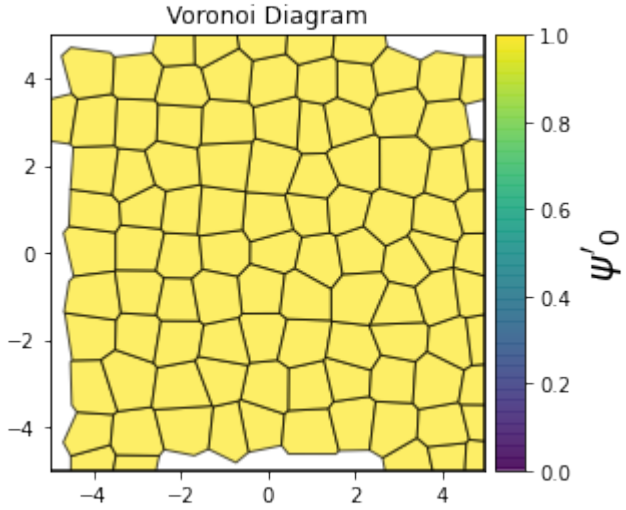

Voronoi Diagram
ψ'_6

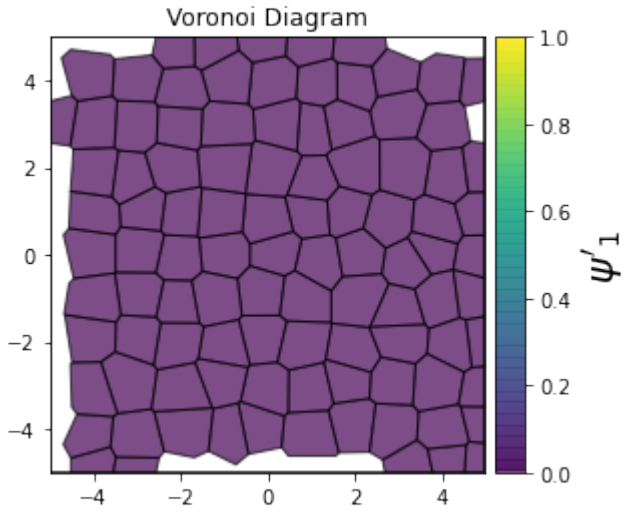


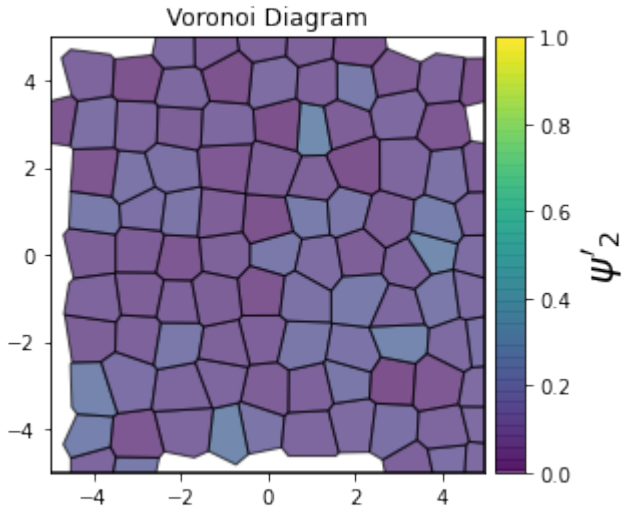


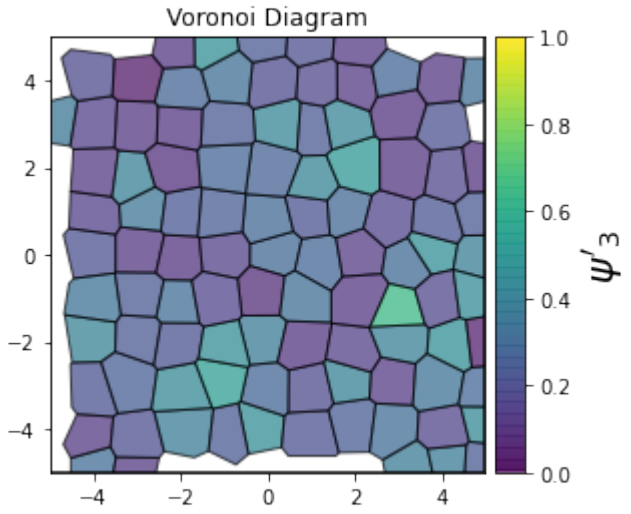
[4]:
```
unit_cell = freud.data.UnitCell.square()
system = unit_cell.generate_system(num_replicas=10, sigma_noise=0.15)
show_minkowski_structure_metrics(system)
```

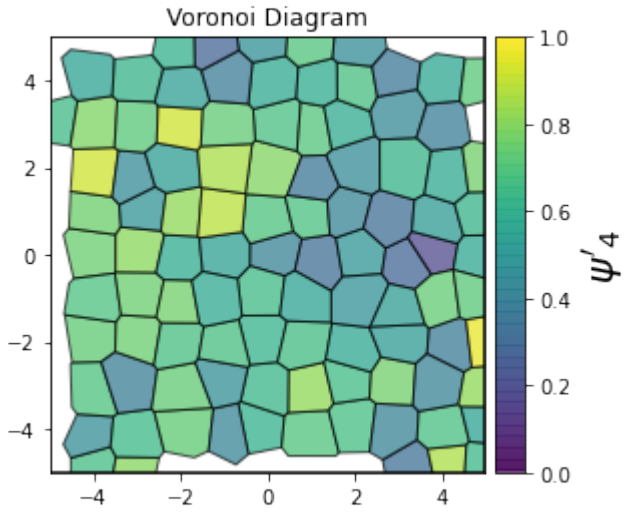

Voronoi Diagram
Number of sides
4
5
6
7
8

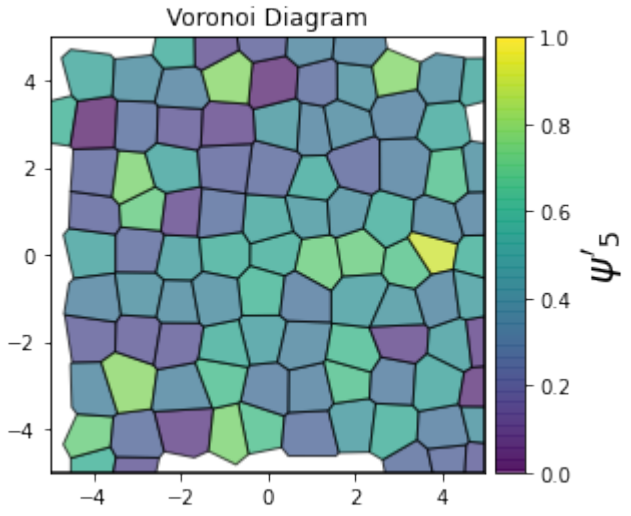

Voronoi Diagram
ψ'_0
0.0
0.2
0.4
0.6
0.8
1.0

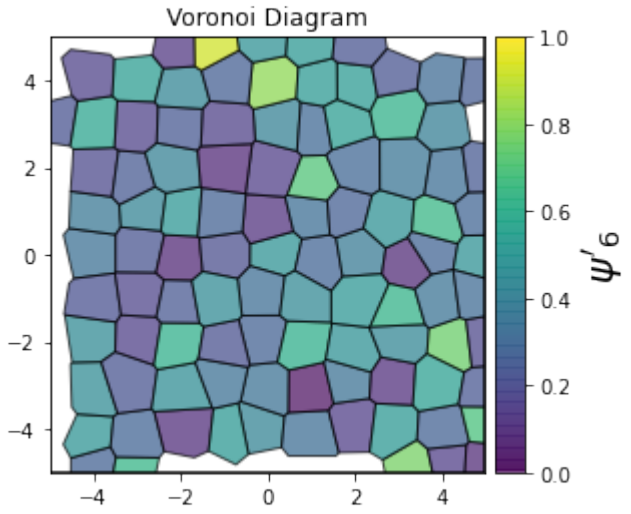

Voronoi Diagram
ψ'_1
0.0
0.2
0.4
0.6
0.8
1.0

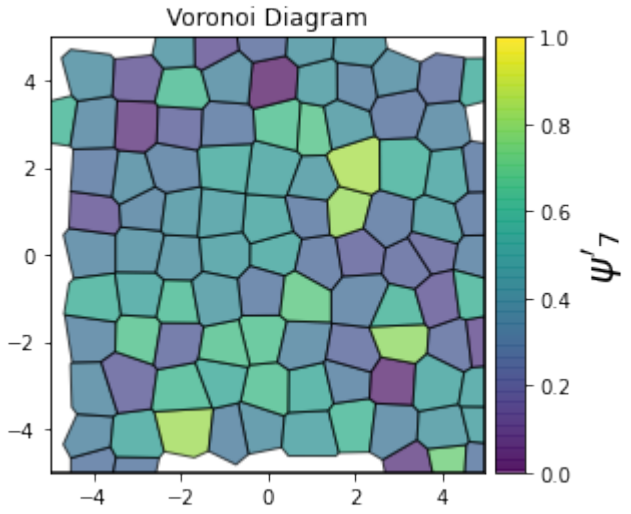

Voronoi Diagram
4
2
0
−2
−4
−4
−2
0
2
4
1.0
0.8
0.6
0.4
0.2
0.0
ψ'_2

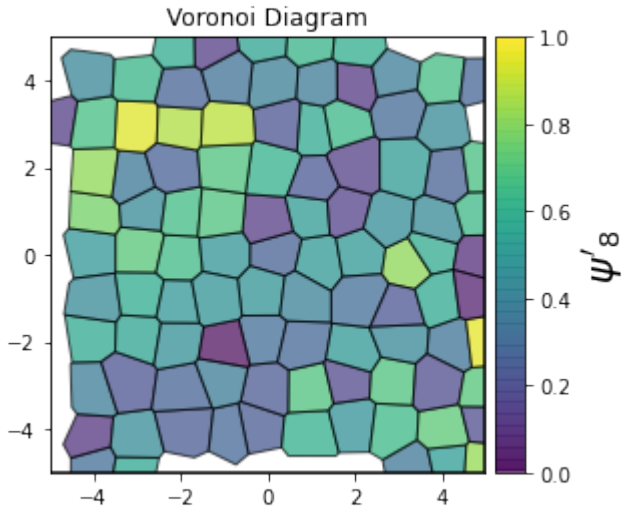

Voronoi Diagram
4
2
0
−2
−4
−4
−2
0
2
4
1.0
0.8
0.6
0.4
0.2
0.0
ψ'_3


Voronoi Diagram
4
2
0
−2
−4
−4
−2
0
2
4
1.0
0.8
0.6
0.4
0.2
0.0
ψ'_4


Voronoi Diagram
4
2
0
−2
−4
−4
−2
0
2
4
1.0
0.8
0.6
0.4
0.2
0.0
ψ'_5


Voronoi Diagram
4
2
0
−2
−4
−4
−2
0
2
4
1.0
0.8
0.6
0.4
0.2
0.0
ψ'_6


Voronoi Diagram
4
2
0
−2
−4
−4
−2
0
2
4
1.0
0.8
0.6
0.4
0.2
0.0
ψ'_7



freud.order.Nematic

Nematic Order Parameter

The `freud.order` module provids the tools to calculate various order parameters that can be used to identify phase transitions. This notebook demonstrates the nematic order parameter, which can be used to identify systems with strong orientational ordering but no translational ordering. For this example, we'll start with a set of random positions in a 3D system, each with a fixed, assigned orientation. Then, we will show how deviations from these orientations are exhibited in the order parameter.

[1]:
```
import freud
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import rowan  # for quaternion math, see rowan.readthedocs.io for more information.
```

In order to work with orientations in freud, we need to do some math with quaternions. If you are unfamiliar with quaternions, you can read more about their definition and how they can be used to represent rotations. For the purpose of this tutorial, just consider them as 4D vectors, and know that the set of normalized (*i.e.* unit norm) 4D vectors can be used to represent rotations in 3D. In fact, there is a 1-1 mapping between normalize quaternions and 3x3 rotation matrices. Quaternions are more computationally convenient, however, because they only require storing 4 numbers rather than 9, and they can be much more easily chained together. For our purposes, you can largely ignore the contents of the next cell, other than to note that this is how we perform rotations of vectors using quaternions instead of matrices.

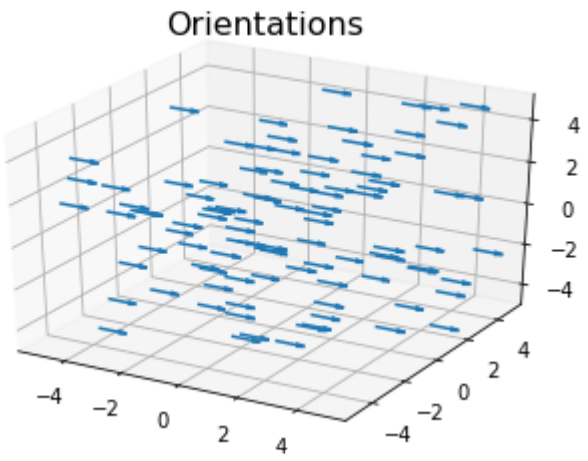
[2]:
```
# Random positions are fine for this. Order is measured
# in terms of similarity of orientations, not positions.
L = 10
N = 100
box, points = freud.data.make_random_system(L, N, seed=0)
orientations = np.array([[1, 0, 0, 0]] * N)
```

[3]:
```
# To show orientations, we use arrows rotated by the quaternions.
arrowheads = rowan.rotate(orientations, [1, 0, 0])
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.quiver3D(points[:, 0], points[:, 1], points[:, 2],
            arrowheads[:, 0], arrowheads[:, 1], arrowheads[:, 2])
ax.set_title("Orientations", fontsize=16);
```



The nematic order parameter provides a measure of how much of the system is aligned with respect to some provided reference vector. As a result, we can now compute the order parameter for a few simple cases. Since our original system is oriented along the x-axis, we can immediately test for that, as well as orientation along any of the other coordinate axes.

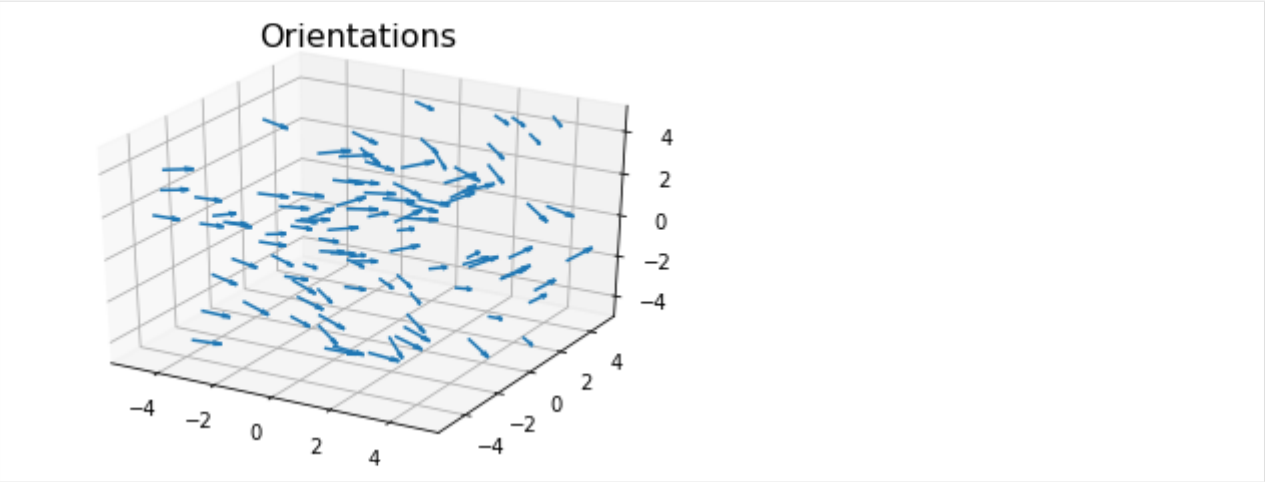
[4]:
```
nop = freud.order.Nematic([1, 0, 0])
nop.compute(orientations)
print("The value of the order parameter is {}.".format(nop.order))
```

```
The value of the order parameter is 1.0.
```

In general, the nematic order parameter is defined as the eigenvalue corresponding to the largest eigenvector of the nematic tensor, which is also computed by this class and provides an average over the orientations of all particles in the system. As a result, we can also look at the intermediate results of our calculation and see how they are related. To do so, let's consider a more interesting system with random orientations.

[5]:
```
# We rotate identity quaternions slightly, in a random direction
np.random.seed(0)
interpolate_amount = 0.3
identity_quats = np.array([[1, 0, 0, 0]] * N)
orientations = rowan.interpolate.slerp(
    identity_quats, rowan.random.rand(N), interpolate_amount)
```

[6]:
```
# To show orientations, we use arrows rotated by the quaternions.
arrowheads = rowan.rotate(orientations, [1, 0, 0])

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.quiver3D(points[:, 0], points[:, 1], points[:, 2],
            arrowheads[:, 0], arrowheads[:, 1], arrowheads[:, 2])
ax.set_title("Orientations", fontsize=16);
```



First, we see that for this nontrivial system the order parameter now depends on the choice of director.

[7]:
```
axes = [[1, 0, 0], [0, 1, 0], [0, 0, 1], [1, 1, 0], [1, 0, 1], [0, 1, 1], [1, 1, 1]]
for ax in axes:
    nop = freud.order.Nematic(ax)
    nop.compute(orientations)
    print("For axis {}, the value of the order parameter is {:0.3f}.".format(ax, nop.
↪order))
```

```
For axis [1, 0, 0], the value of the order parameter is 0.600.
For axis [0, 1, 0], the value of the order parameter is 0.586.
For axis [0, 0, 1], the value of the order parameter is 0.587.
For axis [1, 1, 0], the value of the order parameter is 0.591.
For axis [1, 0, 1], the value of the order parameter is 0.589.
For axis [0, 1, 1], the value of the order parameter is 0.573.
For axis [1, 1, 1], the value of the order parameter is 0.578.
```

Furthermore, increasing the amount of variance in the orientations depresses the value of the order parameter even further.

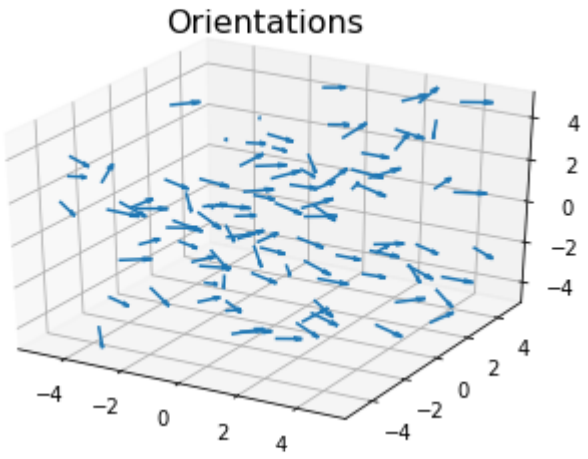
[8]:
```
interpolate_amount = 0.4
orientations = rowan.interpolate.slerp(
    identity_quats, rowan.random.rand(N), interpolate_amount)

arrowheads = rowan.rotate(orientations, [1, 0, 0])
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.quiver3D(points[:, 0], points[:, 1], points[:, 2],
            arrowheads[:, 0], arrowheads[:, 1], arrowheads[:, 2])
ax.set_title("Orientations", fontsize=16);

axes = [[1, 0, 0], [0, 1, 0], [0, 0, 1], [1, 1, 0], [1, 0, 1], [0, 1, 1], [1, 1, 1]]
for ax in axes:
    nop = freud.order.Nematic(ax)
    nop.compute(orientations)
    print("For axis {}, the value of the order parameter is {:0.3f}.".format(ax, nop.
↪order))
```

```
For axis [1, 0, 0], the value of the order parameter is 0.451.
For axis [0, 1, 0], the value of the order parameter is 0.351.
```

(continues on next page)

(continued from previous page)

```
For axis [0, 0, 1], the value of the order parameter is 0.342.
For axis [1, 1, 0], the value of the order parameter is 0.374.
For axis [1, 0, 1], the value of the order parameter is 0.391.
For axis [0, 1, 1], the value of the order parameter is 0.316.
For axis [1, 1, 1], the value of the order parameter is 0.344.
```



Finally, we can look at the per-particle quantities and build them up to get the actual value of the order parameter.

[9]:
```
# The per-particle values averaged give the nematic tensor
print(np.allclose(np.mean(nop.particle_tensor, axis=0), nop.nematic_tensor))
print("The nematic tensor:")
print(nop.nematic_tensor)

eig = np.linalg.eig(nop.nematic_tensor)
print("The eigenvalues of the nematic tensor:")
print(eig[0])
print("The eigenvectors of the nematic tensor:")
print(eig[1])

# The largest eigenvalue
print("The largest eigenvalue, {:0.3f}, is equal to the order parameter {:0.3f}.".
↪format(
    np.max(eig[0]), nop.order))
```

```
True
The nematic tensor:
[[ 0.0115407   0.21569438  0.14729623]
 [ 0.21569438  0.02040018  0.14309749]
 [ 0.14729623  0.14309748 -0.03194092]]
The eigenvalues of the nematic tensor:
[ 0.34387365 -0.20013455 -0.14373913]
The eigenvectors of the nematic tensor:
[[ 0.6173224   0.73592573 -0.27807635]
 [ 0.6237324  -0.6732561  -0.3970945 ]
 [ 0.47944868 -0.07169023  0.87463677]]
The largest eigenvalue, 0.344, is equal to the order parameter 0.344.
```

freud.order.Steinhardt

Steinhardt Order Parameters

The `freud.order` module provids the tools to calculate various order parameters that can be used to identify phase transitions. In the context of crystalline systems, some of the best known order parameters are the Steinhardt order parameters q_l and w_l. These order parameters are mathematically defined according to certain rotationally invariant combinations of spherical harmonics calculated between particles and their nearest neighbors, so they provide information about local particle environments. As a result, considering distributions of these order parameters across a system can help characterize the overall system's ordering. The primary utility of these order parameters arises from the fact that they often exhibit certain characteristic values for specific crystal structures.

In this notebook, we will use the order parameters to identify certain basic structures: BCC, FCC, and simple cubic. FCC, BCC, and simple cubic structures each exhibit characteristic values of q_l for some l value, meaning that in a perfect crystal all the particles in one of these structures will have the same value of q_l. As a result, we can use these characteristic q_l values to determine whether a disordered fluid is beginning to crystallize into one structure or another. The l values correspond to the l quantum number used in defining the underlying spherical harmonics; for example, the q_4 order parameter would provide a measure of 4-fold ordering.

[1]:
```
import freud
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
# Try to plot using KDE if available, otherwise revert to histogram
try:
    from sklearn.neighbors.kde import KernelDensity
    kde = True
except:
    kde = False

np.random.seed(1)
```

We first construct ideal crystals and then extract the characteristic value of q_l for each of these structures. In this case, we know that simple cubic has a coordination number of 6, BCC has 8, and FCC has 12, so we are looking for the values of q_6, q_8, and q_{12}, respectively. Therefore, we can also enforce that we require 6, 8, and 12 nearest neighbors to be included in the calculation, respectively.

[2]:
```
L = 6
sc = freud.data.UnitCell.sc()
sc_system = sc.generate_system(5)
ql = freud.order.Steinhardt(L)
ql_sc = ql.compute(sc_system, neighbors={'num_neighbors': L}).particle_order
mean_sc = np.mean(ql_sc)
print("The Q{} values computed for simple cubic are {:.3f} +/- {:.3e}".format(
    L, mean_sc, np.std(ql_sc)))

L = 8
bcc = freud.data.UnitCell.bcc()
bcc_system = bcc.generate_system(5, sigma_noise=0)
ql = freud.order.Steinhardt(L)
ql.compute(bcc_system, neighbors={'num_neighbors': L})
ql_bcc = ql.particle_order
mean_bcc = np.mean(ql_bcc)
print("The Q{} values computed for bcc are {:.3f} +/- {:.3e}".format(
    L, mean_bcc, np.std(ql_bcc)))
```

(continues on next page)

(continued from previous page)

```
L = 12
fcc = freud.data.UnitCell.fcc()
fcc_system = fcc.generate_system(5)
ql = freud.order.Steinhardt(L)
ql_fcc = ql.compute(fcc_system, neighbors={'num_neighbors': L}).particle_order
mean_fcc = np.mean(ql_fcc)
print("The Q{} values computed for fcc are {:.3f} +/- {:.3e}".format(
    L, mean_fcc, np.std(ql_fcc)))
```

```
The Q6 values computed for simple cubic are 0.354 +/- 3.938e-08
The Q8 values computed for bcc are 0.213 +/- 1.137e-12
The Q12 values computed for fcc are 0.600 +/- 1.155e-12
```

Given that the per-particle order parameter values are essentially identical to within machine precision, we can be confident that we have found the characteristic value of q_l for each of these systems. We can now compare these values to the values of q_l in thermalized systems to determine the extent to which they are exhibiting the ordering expected of one of these perfect crystals.

[3]:
```
def make_noisy_replicas(unitcell, sigmas):
    """Given a unit cell, return a noisy system."""
    systems = []
    for sigma in sigmas:
        systems.append(unitcell.generate_system(5, sigma_noise=sigma))
    return systems
```

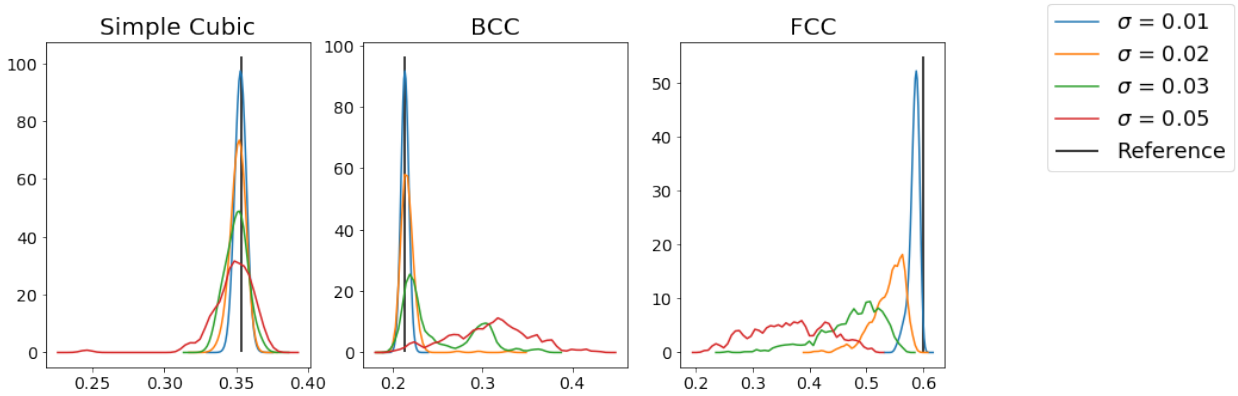
[4]:
```
sigmas = [0.01, 0.02, 0.03, 0.05]
sc_systems = make_noisy_replicas(sc, sigmas)
bcc_systems = make_noisy_replicas(bcc, sigmas)
fcc_systems = make_noisy_replicas(fcc, sigmas)
```

[5]:
```
fig, axes = plt.subplots(1, 3, figsize=(16, 5))

# Zip up the data that will be needed for each structure type.
zip_obj = zip([sc_systems, bcc_systems, fcc_systems], [mean_sc, mean_bcc, mean_fcc],
              [6, 8, 12], ["Simple Cubic", "BCC", "FCC"])

for i, (systems, ref_val, L, title) in enumerate(zip_obj):
    ax = axes[i]
    for j, (system, sigma) in enumerate(zip(systems, sigmas)):
        ql = freud.order.Steinhardt(L)
        ql.compute(system, neighbors={'num_neighbors': L})
        if not kde:
            ax.hist(ql.particle_order, label="$\sigma$ = {}".format(sigma),␣
↪density=True)
        else:
            padding = 0.02
            N = 50
            bins = np.linspace(np.min(ql.particle_order)-padding,
                               np.max(ql.particle_order)+padding, N)

            kde = KernelDensity(bandwidth=0.004)
            kde.fit(ql.particle_order[:, np.newaxis])
            ql = np.exp(kde.score_samples(bins[:, np.newaxis]))

            ax.plot(bins, ql, label="$\sigma$ = {}".format(sigma))
        ax.set_title(title, fontsize=20)
```

(continues on next page)

(continued from previous page)

```
        ax.tick_params(axis='both', which='both', labelsize=14)
        if j == 0:
            # Can choose any element, all are identical in the reference case
            ax.vlines(ref_val, 0, np.max(ax.get_ylim()[1]), label='Reference')
fig.legend(*ax.get_legend_handles_labels(), fontsize=18)  # Only have one legend
fig.subplots_adjust(right=0.78)
```



From this figure, we can see that for each type of structure, increasing the amount of noise makes the distribution of the order parameter values less peaked at the expected reference value. As a result, we can use this method to identify specific structures. Choosing the appropriate parameterization for the order parameter (which quantum number l to use, how to choose neighbors, etc.) can be very important.

In addition to the q_l parameters demonstrated here, this class can also compute the third-order invariant w_l. The w_l may be better at identifying some structures, so some experimentation and reference to the appropriate literature can be useful (as a starting point, see Steinhardt, Nelson, and Ronchetti (1983)).

By setting `average=True` in the constructor, the `Steinhardt` class will perform an additional level of averaging over the second neighbor shells of particles, to accumulate more information on particle environments (see Lechner and Dellago (2008)). To get a sense for the best method for analyzing a specific system, the best course of action is try out different parameters or to consult the literature to see how these have been used in the past.

freud.pmft.PMFTXY

The PMFT returns the potential energy associated with finding a particle pair in a given spatial (positional and orientational) configuration. The PMFT is computed in the same manner as the RDF. The basic algorithm is described below:

```
for each particle i:
    for each particle j:
        v_ij = position[j] - position[i]
        bin_x, bin_y = convert_to_bin(v_ij)
        pcf_array[bin_y][bin_x]++
```

`freud` uses spatial data structures and parallelism to optimize this algorithm.

The data sets used in this example are a system of hard hexagons, simulated in the NVT thermodynamic ensemble in HOOMD-blue, for a dense fluid of hexagons at packing fraction $\phi = 0.65$ and solids at packing fractions $\phi = 0.75, 0.85$.

[1]:
```
import freud
freud.set_num_threads(1)
```

(continues on next page)

(continued from previous page)

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
# The following line imports this set of utility functions:
# https://github.com/glotzerlab/freud-examples/blob/master/util.py
import util
from scipy.ndimage.filters import gaussian_filter

%matplotlib inline
matplotlib.rcParams.update({'font.size': 20,
                            'axes.titlesize': 20,
                            'axes.labelsize': 20,
                            'xtick.labelsize': 16,
                            'ytick.labelsize': 16,
                            'savefig.pad_inches': 0.025,
                            'lines.linewidth': 2})
```

[2]:
```
def plot_pmft(data_path, phi):
    # Create the pmft object
    pmft = freud.pmft.PMFTXY(x_max=3.0, y_max=3.0, bins=300)

    # Load the data
    box_data = np.load("{}/box_data.npy".format(data_path))
    pos_data = np.load("{}/pos_data.npy".format(data_path))
    quat_data = np.load("{}/quat_data.npy".format(data_path))
    n_frames = pos_data.shape[0]

    for i in range(1, n_frames):
        # Read box, position data
        box = box_data[i].tolist()
        points = pos_data[i]
        quats = quat_data[i]
        angles = 2*np.arctan2(quats[:, 3], quats[:, 0]) % (2 * np.pi)
        pmft.compute(system=(box, points), query_orientations=angles, reset=False)

    # Get the value of the PMFT histogram bins
    pmft_arr = pmft.pmft.T

    # Do some simple post-processing for plotting purposes
    pmft_arr[np.isinf(pmft_arr)] = np.nan
    dx = (2.0 * 3.0) / pmft.nbins[0]
    dy = (2.0 * 3.0) / pmft.nbins[1]
    nan_arr = np.where(np.isnan(pmft_arr))
    for i in range(pmft.nbins[0]):
        x = -3.0 + dx * i
        for j in range(pmft.nbins[1]):
            y = -3.0 + dy * j
            if ((x*x + y*y < 1.5) and (np.isnan(pmft_arr[j, i]))):
                pmft_arr[j, i] = 10.0
    w = int(2.0 * pmft.nbins[0] / (2.0 * 3.0))
    center = int(pmft.nbins[0] / 2)

    # Get the center of the histogram bins
    pmft_smooth = gaussian_filter(pmft_arr, 1)
    pmft_image = np.copy(pmft_smooth)
    pmft_image[nan_arr] = np.nan
```

(continues on next page)

(continued from previous page)

```
    pmft_smooth = pmft_smooth[center-w:center+w, center-w:center+w]
    pmft_image = pmft_image[center-w:center+w, center-w:center+w]
    x, y = pmft.bin_centers
    reduced_x = x[center-w:center+w]
    reduced_y = y[center-w:center+w]

    # Plot figures
    f = plt.figure(figsize=(12, 5), facecolor='white')
    values = [-2, -1, 0, 2]
    norm = matplotlib.colors.Normalize(vmin=-2.5, vmax=3.0)
    n_values = [norm(i) for i in values]
    colors = matplotlib.cm.viridis(n_values)
    colors = colors[:, :3]
    verts = util.make_polygon(sides=6, radius=0.6204)
    lims = (-2, 2)
    ax0 = f.add_subplot(1, 2, 1)
    ax1 = f.add_subplot(1, 2, 2)
    for ax in (ax0, ax1):
        ax.contour(reduced_x, reduced_y, pmft_smooth,
                   [9, 10], colors='black')
        ax.contourf(reduced_x, reduced_y, pmft_smooth,
                    [9, 10], hatches='X', colors='none')
        ax.plot(verts[:, 0], verts[:, 1], color='black', marker=',')
        ax.fill(verts[:, 0], verts[:, 1], color='black')
        ax.set_aspect('equal')
        ax.set_xlim(lims)
        ax.set_ylim(lims)
        ax.xaxis.set_ticks([i for i in range(lims[0], lims[1]+1)])
        ax.yaxis.set_ticks([i for i in range(lims[0], lims[1]+1)])
        ax.set_xlabel(r'$x$')
        ax.set_ylabel(r'$y$')

    ax0.set_title('PMFT Heat Map, $\phi = {}$'.format(phi))
    im = ax0.imshow(np.flipud(pmft_image),
                    extent=[lims[0], lims[1], lims[0], lims[1]],
                    interpolation='nearest', cmap='viridis',
                    vmin=-2.5, vmax=3.0)
    ax1.set_title('PMFT Contour Plot, $\phi = {}$'.format(phi))
    ax1.contour(reduced_x, reduced_y, pmft_smooth,
            [-2, -1, 0, 2], colors=colors)

    f.subplots_adjust(right=0.85)
    cbar_ax = f.add_axes([0.88, 0.1, 0.02, 0.8])
    f.colorbar(im, cax=cbar_ax)
    plt.show()
```

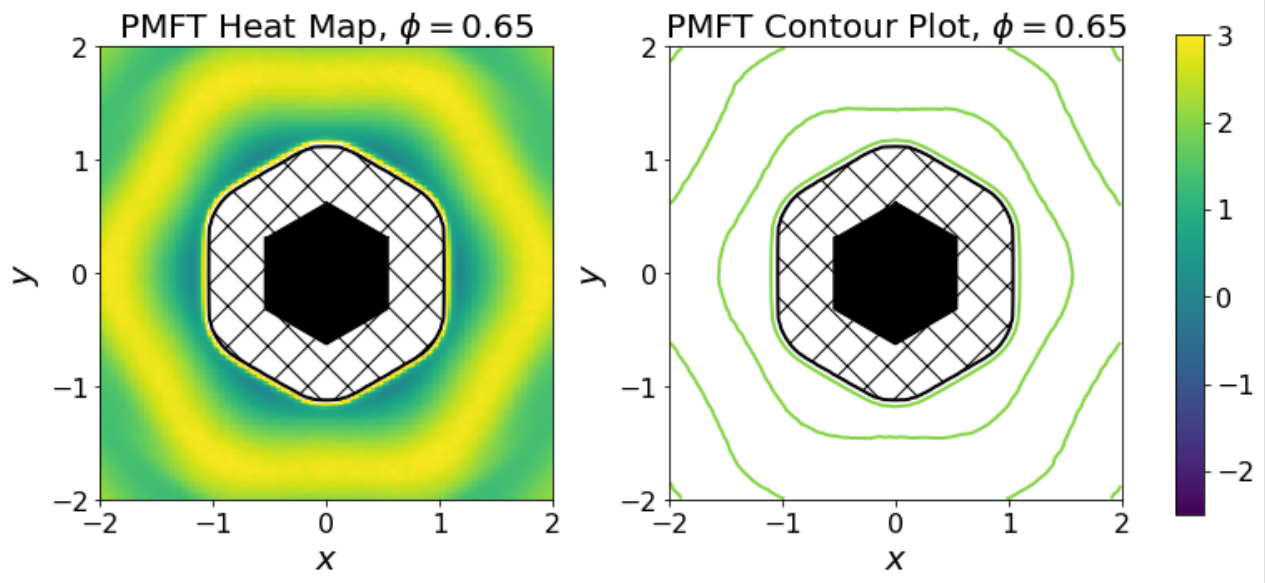
65% density

The plot below shows the PMFT of hexagons at 65% density. The hexagons tend to be close to one another, in the darker regions (the lower values of the potential of mean force and torque).

The hatched region near the black hexagon in the center is a region where no data were collected: the hexagons are hard shapes and cannot overlap, so there is an excluded region of space close to the hexagon.

The ring around the hexagon where the PMFT rises and then falls corresponds to the minimum of the radial distribution function – particles tend to not occupy that region, preferring instead to be at close range (in the first neighbor shell) or further away (in the second neighbor shell).

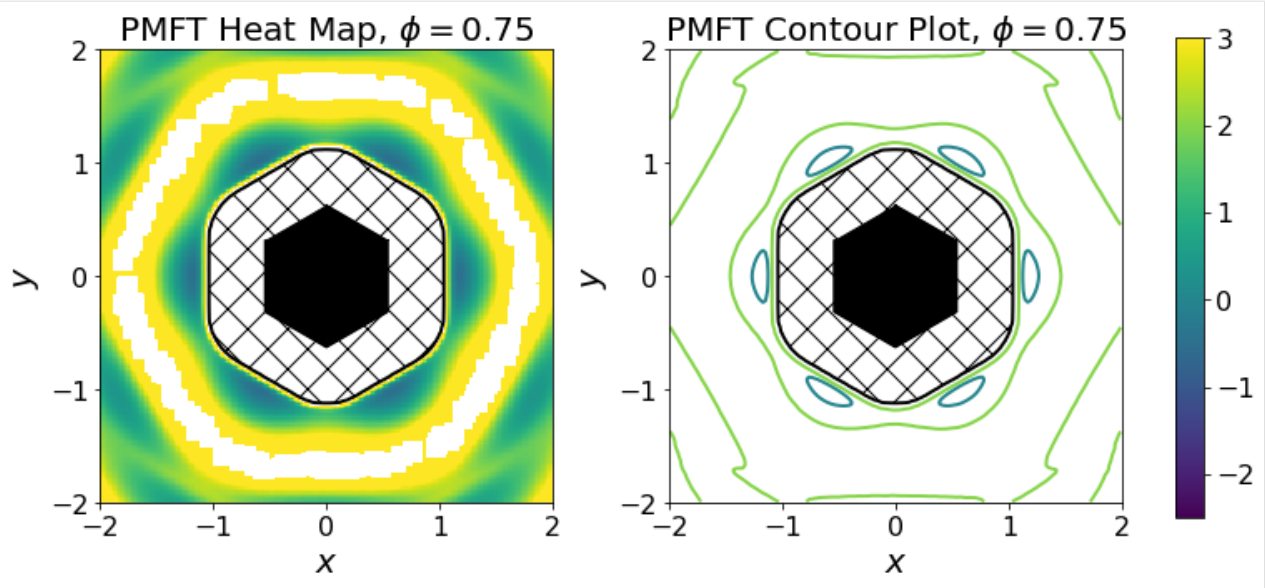
[3]:
```
plot_pmft('data/phi065', 0.65)
```



75% density

As the system density is increased to 75%, the propensity for hexagons to occupy the six sites on the faces of their neighbors increases, as seen by the deeper (darker) wells of the PMFT. Conversely, the shapes strongly dislike occupying the yellow regions, and no particle pairs occupied the white region (so there is no data).

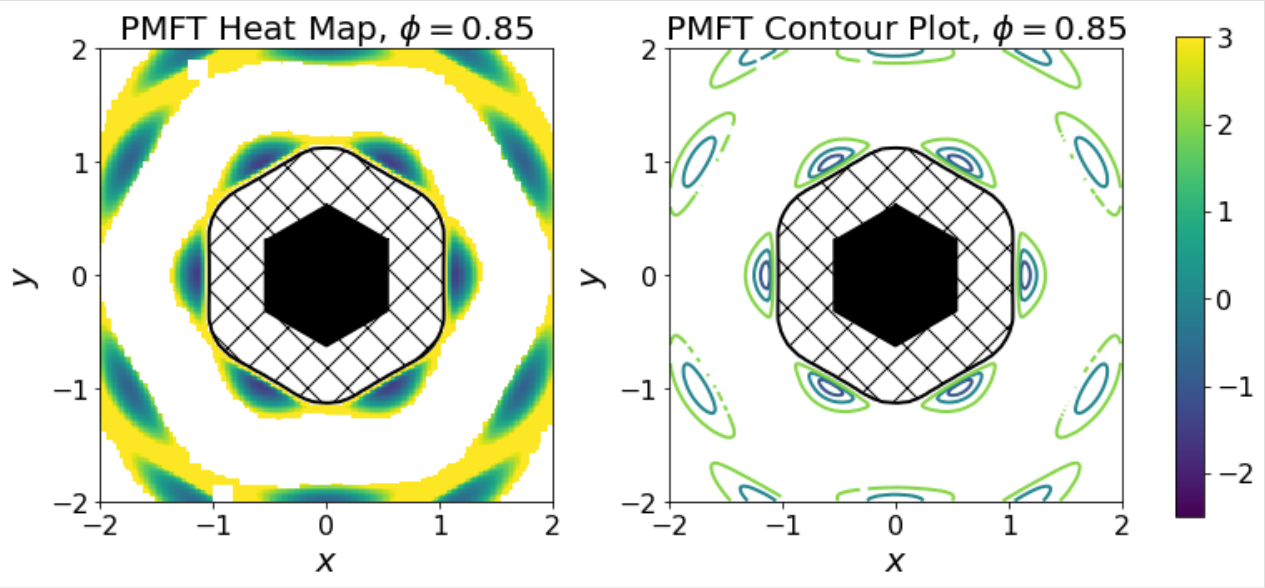
[4]:
```
plot_pmft('data/phi075', 0.75)
```



85% density

Finally, at 85% density, there is a large region where no neighbors can be found, and hexagons strictly occupy sites near those of the perfect hexagonal lattice, at the first- and second-neighbor shells. The wells are deeper and much more spatially confined that those of the systems at lower densities.

[5]:
```
plot_pmft('data/phi085', 0.85)
```



freud.pmft.PMFTXYZ: Shifting Example

This notebook shows how to use the shifting option on PMFTXYZ to get high resolution views of PMFT features that are not centered.

[1]:
```
import numpy as np
import freud
import matplotlib.pyplot as plt
```

First we load in our data. The particles used here are implemented with a simple Weeks-Chandler-Andersen isotropic pair potential, so particle orientation is not meaningful.

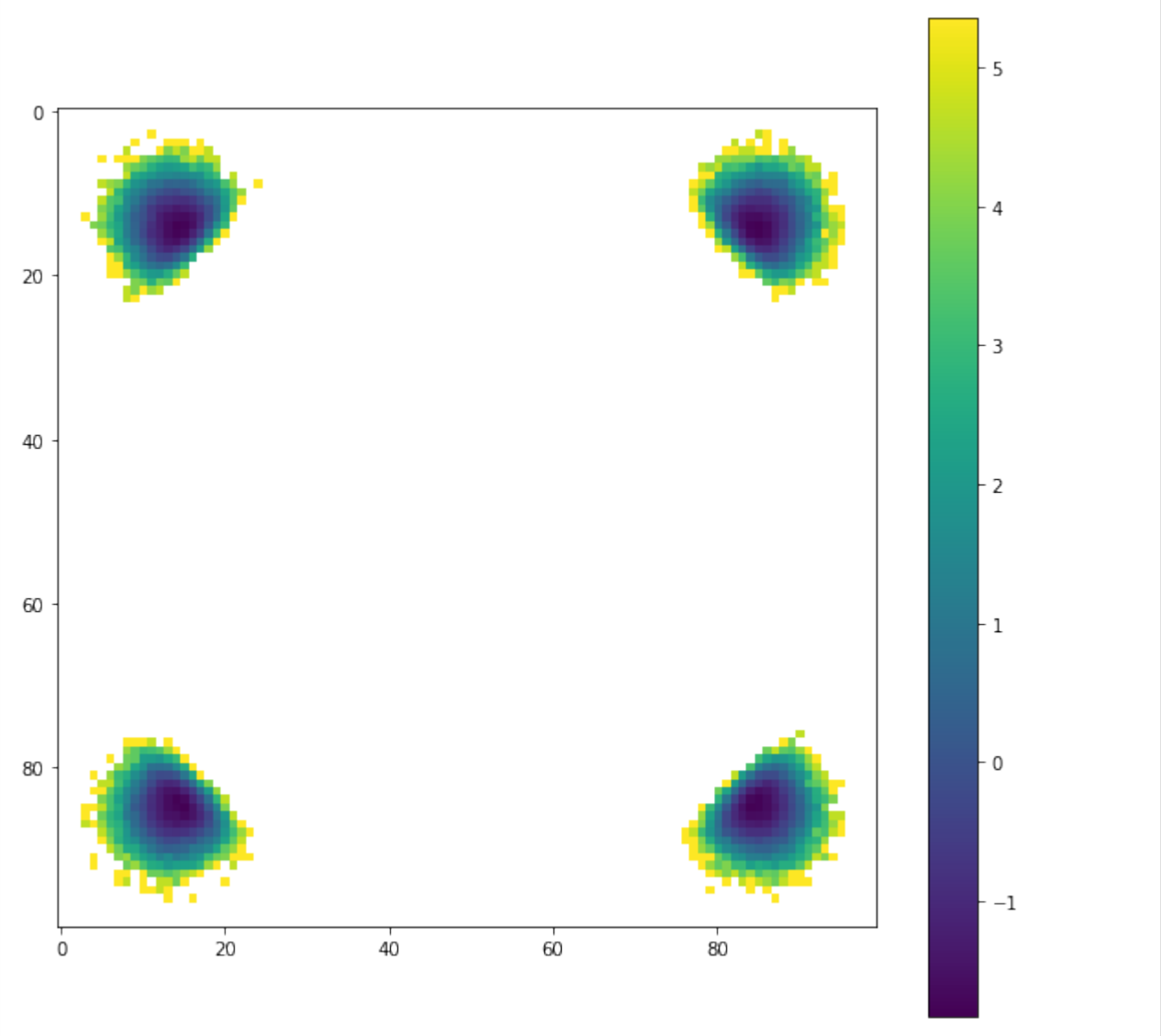
[2]:
```
pos_data = np.load('data/XYZ/positions.npy')
box_data = np.load('data/XYZ/boxes.npy')
```

We calculate the PMFT the same way as shown in other examples first

[3]:
```
window = 2**(1/6) # The size of the pmft calculation

bins = 100
pmft = freud.pmft.PMFTXYZ(x_max=window, y_max=window, z_max=window, bins=bins)

# This data is for isotropic particles, so we will just make some unit quaternions
# to use as the orientations
quats = np.zeros((pos_data.shape[1],4)).astype(np.float32)
quats[:,0] = 1

for i in range(10, pos_data.shape[0]):
    box = box_data[i]
    points = pos_data[i]
    pmft.compute((box, points), quats, reset=False)

unshifted = pmft.pmft

x, y, z = pmft.bin_centers
```

When we plot a centered slice of the XYZ pmft, we see that a number of wells are present at some distance from the origin

[4]:
```
%matplotlib inline

plt.figure(figsize=(10,10))
plt.imshow(unshifted[int(bins/2),:,:])
plt.colorbar()
plt.show()
```



If we want a closer look at the details of those wells, then we could increase the PMFT resolution. But this will increase the computational cost by a lot, and we are wasting a big percentage of the pixels.

This use case is why the shiftvec argument was implemented. Now we will do the same calculation, but we will use a much smaller window centered on on of the wells.

To do this we need to pass a vector into the PMFTXYZ construction. The window will be centered on this vector.

[5]:
```
shiftvec = [0.82, 0.82, 0]

window = 2**(1/6)/6 # Smaller window for the shifted case

bins = 50

pmft = freud.pmft.PMFTXYZ(x_max=window, y_max=window, z_max=window,
                          bins=50, shiftvec=shiftvec)

# This data is for isotropic particles, so we will just make some unit quaternions
# to use as the orientations
```

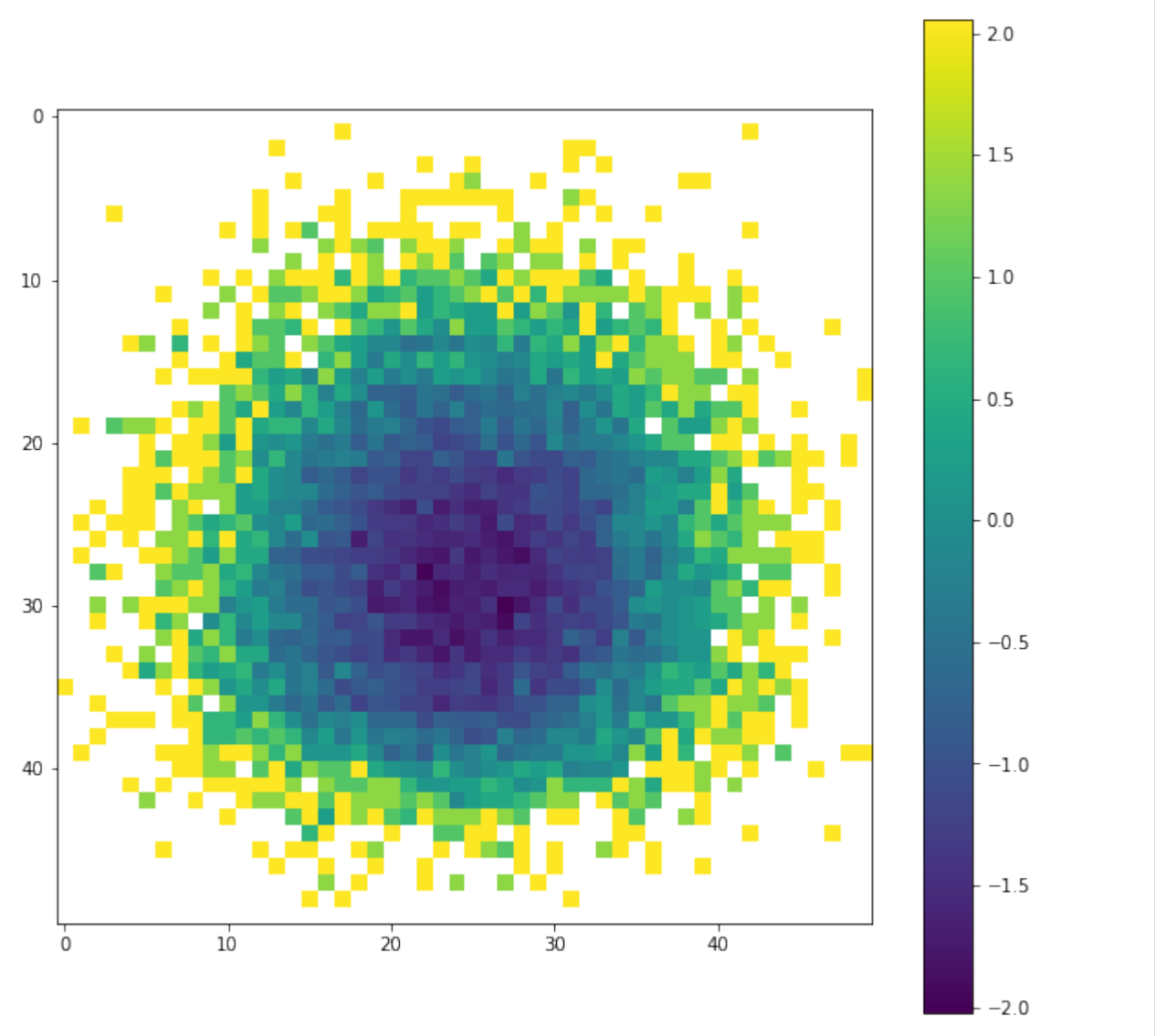
(continues on next page)

(continued from previous page)

```
orientations = np.array([[1, 0, 0, 0]] * pos_data.shape[1])

for i in range(10, pos_data.shape[0]):
    box = box_data[i]
    points = pos_data[i]
    pmft.compute(system=(box, points), query_orientations=orientations, reset=False)

shifted = pmft.pmft

x, y, z = pmft.bin_centers
```

Now the PMFT is a high resolution close up of one of the bonding wells. Note that as you increase the sampling resolution, you need to increase your number of samples because there is less averaging in each bin

[6]:
```
%matplotlib inline
plt.figure(figsize=(10, 10))
plt.imshow(shifted[int(bins/2), :, :])
plt.colorbar()
plt.show()
```



7.5.3 Example Analyses

The examples below go into greater detail about specific applications of **freud** and use cases that its analysis methods enable, such as user-defined analyses, machine learning, and data visualization.

Implementing Common Neighbor Analysis as a custom method

Researchers commonly wish to implement their own custom analysis methods for particle simulations. Here, we show an example of how to write Common Neighbor Analysis (Honeycutt and Andersen, J. Phys. Chem. 91, 4950) as a custom method using `freud` and the NetworkX package.

NetworkX can be installed with `pip install networkx`.

First, we generate random points and determine which points share neighbors.

[1]:
```
import freud
import numpy as np
from collections import defaultdict
```

[2]:
```
# Use a face-centered cubic (fcc) system
box, points = freud.data.UnitCell.fcc().generate_system(4)
aq = freud.AABBQuery(box, points)
nl = aq.query(points, {'num_neighbors': 12, 'exclude_ii': True}).toNeighborList()
```

[3]:
```
# Get all sets of common neighbors.
common_neighbors = defaultdict(list)
for i, p in enumerate(points):
    for j in nl.point_indices[nl.query_point_indices == i]:
        for k in nl.point_indices[nl.query_point_indices == j]:
            if i != k:
                common_neighbors[(i, k)].append(j)
```

Next, we use NetworkX to build graphs of common neighbors and compute the Common Neighbor Analysis signatures.

[4]:
```
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 identifiers for a CNA diagram:
    # The first integer is 1 if the particles are bonded, otherwise 2
    # The second integer is the number of shared neighbors
    # The third integer is the number of bonds among shared neighbors
    # The fourth integer is an index, just to ensure uniqueness of diagrams
    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 add
    # the new graph immediately.
    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)
    else:
        diagrams[key].append(g)
        idx = diagrams[key].index(g)
```

(continues on next page)

(continued from previous page)

```
    cna_signature = key + (idx,)
    particle_counts[a].update([cna_signature])
```

Looking at the counts of common neighbor signatures, we see that the first particle of the fcc structure has 12 bonds with signature $(1, 4, 2, 0)$ as we expect.

[5]:
```
particle_counts[0]
```

[5]:
```
Counter({(1, 4, 2, 0): 12,
         (2, 4, 4, 0): 6,
         (2, 1, 0, 0): 12,
         (2, 2, 1, 0): 24})
```

Analyzing simulation data from HOOMD-blue at runtime

The following script shows how to use `freud` to compute the radial distribution function $g(r)$ on data generated by the molecular dynamics simulation engine HOOMD-blue *during a simulation run*.

Generally, most users will want to run analyses as post-processing steps, on the saved frames of a particle trajectory file. However, it is possible to use analysis callbacks in HOOMD-blue to compute and log quantities at runtime, too. By using analysis methods at runtime, it is possible to stop a simulation early or change the simulation parameters dynamically according to the analysis results.

HOOMD-blue can be installed with `conda install -c conda-forge hoomd`.

The simulation script runs a Monte Carlo simulation of spheres, with outputs parsed with `numpy.genfromtxt`.

[1]:
```
%matplotlib inline
import hoomd
from hoomd import hpmc
import freud
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
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)
w6 = freud.order.Steinhardt(l=6, wl=True)

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

def calc_W6(timestep):
    hoomd.util.quiet_status()
    snap = system.take_snapshot()
    hoomd.util.unquiet_status()
    w6.compute(system=snap, neighbors={'num_neighbors': 12})
    return np.mean(w6.particle_order)
```

(continues on next page)

(continued from previous page)

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

logger = hoomd.analyze.log(filename='output.log',
                           quantities=['w6'],
                           period=100,
                           header_prefix='#',
                           overwrite=True)

logger.register_callback('w6', calc_W6)

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)')
```

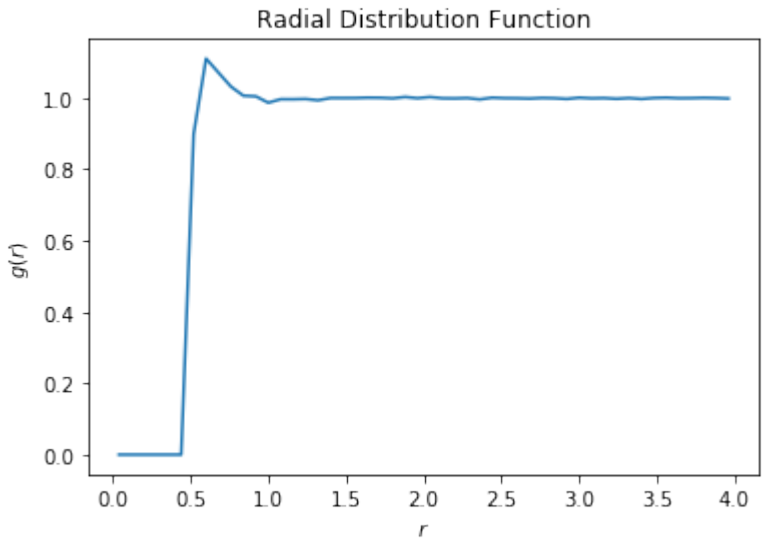
```
HOOMD-blue v2.7.0-77-g568406147 DOUBLE HPMC_MIXED MPI TBB SSE SSE2 SSE3 SSE4_1 SSE4_2␣
↪AVX AVX2
Compiled: 10/28/2019
Copyright (c) 2009-2019 The Regents of the University of Michigan.
-----
You are using HOOMD-blue. Please cite the following:
* J A Anderson, C D Lorenz, and A Travesset. "General purpose molecular dynamics
  simulations fully implemented on graphics processing units", Journal of
  Computational Physics 227 (2008) 5342--5359
* J Glaser, T D Nguyen, J A Anderson, P Liu, F Spiga, J A Millan, D C Morse, and
  S C Glotzer. "Strong scaling of general-purpose molecular dynamics simulations
  on GPUs", Computer Physics Communications 192 (2015) 97--107
-----
-----
You are using HPMC. Please cite the following:
* J A Anderson, M E Irrgang, and S C Glotzer. "Scalable Metropolis Monte Carlo
  for simulation of hard shapes", Computer Physics Communications 204 (2016) 21
  --30
-----
HOOMD-blue is running on the CPU
notice(2): Group "all" created containing 1000 particles
** starting run **
Time 00:00:10 | Step 3878 / 10000 | TPS 387.761 | ETA 00:00:15
Time 00:00:20 | Step 7808 / 10000 | TPS 392.99 | ETA 00:00:05
Time 00:00:25 | Step 10000 / 10000 | TPS 398.521 | ETA 00:00:00
Average TPS: 392.122
---------
notice(2): -- HPMC stats:
notice(2): Average translate acceptance: 0.933106
notice(2): Trial moves per second:        1.56844e+06
notice(2): Overlap checks per second:     4.07539e+07
notice(2): Overlap checks per trial move: 25.9838
notice(2): Number of overlap errors:      0
** run complete **
** starting run **
Time 00:00:35 | Step 13501 / 20000 | TPS 349.776 | ETA 00:00:18
Time 00:00:45 | Step 17001 / 20000 | TPS 349.699 | ETA 00:00:08
Time 00:00:54 | Step 20000 / 20000 | TPS 352.224 | ETA 00:00:00
Average TPS: 350.471
```

(continues on next page)

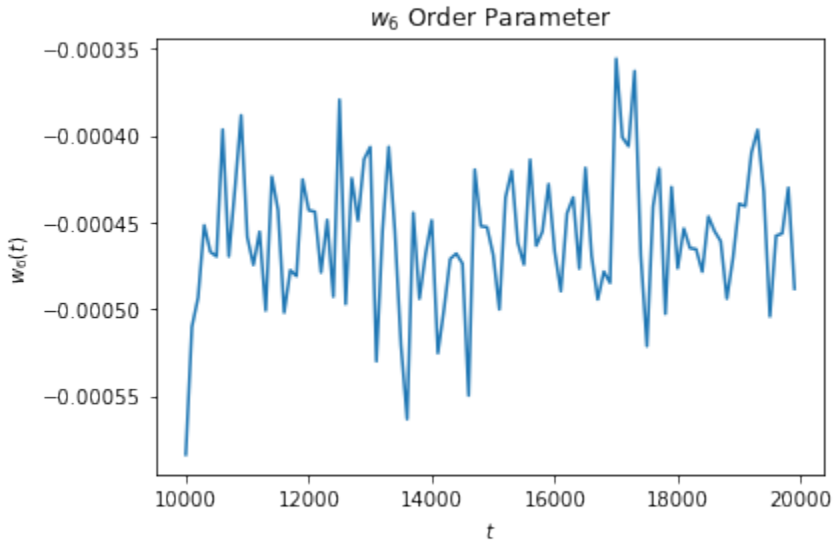
(continued from previous page)

```
---------
notice(2): -- HPMC stats:
notice(2): Average translate acceptance: 0.932846
notice(2): Trial moves per second:        1.40185e+06
notice(2): Overlap checks per second:     3.63552e+07
notice(2): Overlap checks per trial move: 25.9338
notice(2): Number of overlap errors:      0
** run complete **
```

[3]:
```
rdf_data = np.genfromtxt('rdf.csv', delimiter=',')
plt.plot(rdf_data[:, 0], rdf_data[:, 1])
plt.title('Radial Distribution Function')
plt.xlabel('$r$')
plt.ylabel('$g(r)$')
plt.show()
```



[4]:
```
w6_data = np.genfromtxt('output.log')
plt.plot(w6_data[:, 0], w6_data[:, 1])
plt.title('$w_6$ Order Parameter')
plt.xlabel('$t$')
plt.ylabel('$w_6(t)$')
plt.show()
```



Analyzing GROMACS data with freud and MDTraj: Computing an RDF for Water

In this notebook, we demonstrate how `freud` could be used to compute the RDF of the output of an atomistic simulation, namely the simulation of TIP4P water. In the process, we show how the subsetting functionality of such tools can be leveraged to feed data into `freud`. We use this example to also demonstrate how this functionality can be replicated with pure NumPy and explain why this usage pattern is sufficient for common use-cases of `freud`. The simulation data is read with MDTraj and the results are compared for the same RDF calculation with `freud` and MDTraj.

Simulating water

To run this notebook, we have generated data of a simulation of TIP4P using GROMACS. All of the scripts used to generate this data are provided in this repository, and for convenience the final output files are also saved.

[1]:
```
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')

mdtraj_rdf = mdtraj.compute_rdf(
    traj, oxygen_pairs, (r_min, r_max), n_bins=bins)

# 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
```

(continues on next page)

(continued from previous page)

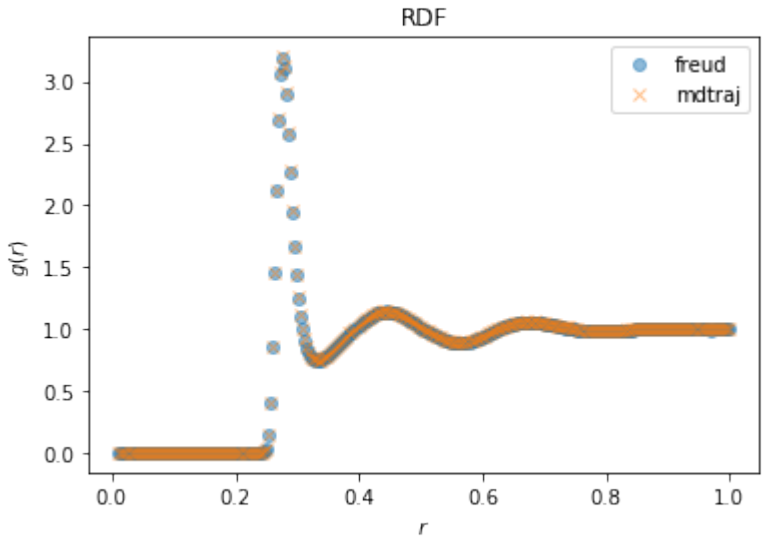
```
# selectors require the user to define the nature of the selection,
# but can be more precisely tailored to a specific system.
oxygen_indices = [atom.index for atom in traj.top.atoms
                  if atom.name == 'O']

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)
```

[2]:
```
from matplotlib import pyplot as plt
```

[3]:
```
%matplotlib inline
```

[4]:
```
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('RDF')
ax.legend()
```

[4]:
```
<matplotlib.legend.Legend at 0x7fb10d1898e0>
```



Analyzing data from LAMMPS

The following script shows how to use `freud` to compute the radial distribution function $g(r)$ on data generated by the molecular dynamics simulation engine LAMMPS. The input script runs a Lennard-Jones system, which is then parsed with `numpy.genfromtxt`.

The input script is below. Note that we must dump images with `ix iy iz`, so that the mean squared displacement can be calculated correctly.

[1]:
```
!cat lj.in
```

```
#  From http://utkstair.org/clausius/docs/mse614/text/examples.html
#  define units
units       lj

#  specify periodic boundary conditions
boundary p p p

#  define atom_style
#  full covers everything
atom_style  full

#  define simulation volume
#  If I want N = 512 atoms
#  and I want a density of rho = 0.5 atoms/lj-sigma^3
#  Then I can determine the size of a cube by
#  size = (N/rho)^(1/3)
variable side      equal 10
region      boxid block 0.0 ${side} 0.0 ${side} 0.0 ${side}
create_box  1 boxid

#  specify initial positions of atoms
#  sc = simple cubic
#  0.5 = density in lj units
lattice     sc 0.50

#  place atoms of type 1 in boxid
create_atoms    1 box

#   define mass of atom type 1
mass        1 1.0

#  specify initial velocity of atoms
#  group = all
#  reduced temperature is T = 1.0 = lj-eps/kb
#  seed for random number generator
#  distribution is gaussian (e.g. Maxwell-Boltzmann)
velocity    all create 1.0 87287 dist gaussian

#  specify interaction potential
#  pairwise interaction via the Lennard-Jones potential with a cut-off at 2.5 lj-sigma
pair_style  lj/cut 2.5

#  specify parameters between atoms of type 1 with an atom of type 1
#  epsilon = 1.0, sigma = 1.0, cutoff = 2.5
pair_coeff  1 1 1.0 1.0 2.5

# add long-range tail correction
pair_modify tail yes

#  specify parameters for neighbor list
#  rnbr = rcut + 0.3
neighbor    0.3 bin

#  specify thermodynamic properties to be output
#  pe = potential energy
#  ke = kinetic energy
#  etotal = pe + ke
```

(continues on next page)

(continued from previous page)

```
#  temp = temperature
#  press = pressure
#  density = number density
#  output every thousand steps
#  norm = normalize by # of atoms (yes or no)
thermo_style custom step pe ke etotal temp press density

# report instantaneous thermo values every 100 steps
thermo 100

# normalize thermo properties by number of atoms (yes or no)
thermo_modify norm no

#  specify ensemble
#  fixid = 1
#  atoms = all
#  ensemble = nve or nvt
fix     1 all nve


timestep 0.005

# run 1000 steps in the NVE ensemble
# (this equilibrates positions)
run 1000

#  stop fix with given fixid
#  fixid = 1
unfix 1

#  specify ensemble
#  fixid = 2
#  atoms = all
#  ensemble = nvt
#  temp = temperature
#  initial temperature = 1.0
#  final temperature = 1.0
#  thermostat controller gain = 0.1 (units of time, bigger is less tight control)
fix     2 all nvt temp 1.0 1.0 0.1

# run 1000 steps in the NVT ensemble
# (this equilibrates thermostat)
run     1000

#   save configurations
#   dumpid = 1
#   all atoms
#   atomic symbol is Ar
#   save positions every 100 steps
#   filename = output.xyz
#
dump    2       all custom 100 output_custom.xyz x y z ix iy iz

# run 1000 more steps in the NVT ensemble
# (this is data production, from which configurations are saved)
run     8000
```

Next, we run LAMMPS to generate the output file. LAMMPS can be installed with `conda install -c conda-forge lammps`.

[2]:
```
!lmp_serial -in lj.in
```

```
LAMMPS (5 Jun 2019)
Created orthogonal box = (0 0 0) to (10 10 10)
  1 by 1 by 1 MPI processor grid
Lattice spacing in x,y,z = 1.25992 1.25992 1.25992
Created 512 atoms
  create_atoms CPU = 0.00122023 secs
Neighbor list info ...
  update every 1 steps, delay 10 steps, check yes
  max neighbors/atom: 2000, page size: 100000
  master list distance cutoff = 2.8
  ghost atom cutoff = 2.8
  binsize = 1.4, bins = 8 8 8
  1 neighbor lists, perpetual/occasional/extra = 1 0 0
  (1) pair lj/cut, perpetual
      attributes: half, newton on
      pair build: half/bin/newton
      stencil: half/bin/3d/newton
      bin: standard
Setting up Verlet run ...
  Unit style    : lj
  Current step  : 0
  Time step     : 0.005
Per MPI rank memory allocation (min/avg/max) = 6.109 | 6.109 | 6.109 Mbytes
Step PotEng KinEng TotEng Temp Press Density
       0   -1804.3284        766.5   -1037.8284            1   -2.1872025        0.512
     100   -1834.8127    774.55302   -1060.2596    1.0105062  -0.32671112        0.512
     200   -1852.2773    789.53605   -1062.7413    1.0300536  -0.30953463        0.512
     300   -1857.4621    795.78772   -1061.6744    1.0382097  -0.22960441        0.512
     400    -1864.766    801.81089   -1062.9551    1.0460677  -0.24901206        0.512
     500   -1860.0198    796.65657   -1063.3633    1.0393432  -0.14280039        0.512
     600   -1859.1835    796.96259    -1062.221    1.0397425   -0.2828161        0.512
     700   -1848.9874    786.01864   -1062.9688    1.0254646  -0.34512435        0.512
     800   -1821.7263    759.86418   -1061.8622    0.9913427   -0.1766353        0.512
     900   -1840.7256    777.68022   -1063.0453    1.0145861    -0.318844        0.512
    1000   -1862.6606    799.32963   -1063.3309    1.0428306  -0.25224674        0.512
Loop time of 0.197457 on 1 procs for 1000 steps with 512 atoms

Performance: 2187817.275 tau/day, 5064.392 timesteps/s
99.5% CPU use with 1 MPI tasks x no OpenMP threads

MPI task timing breakdown:
Section |  min time  |  avg time  |  max time  |%varavg| %total
---------------------------------------------------------------
Pair    | 0.13402    | 0.13402    | 0.13402    |   0.0 | 67.87
Bond    | 8.2254e-05 | 8.2254e-05 | 8.2254e-05 |   0.0 |  0.04
Neigh   | 0.049226   | 0.049226   | 0.049226   |   0.0 | 24.93
Comm    | 0.0084078  | 0.0084078  | 0.0084078  |   0.0 |  4.26
Output  | 0.00015664 | 0.00015664 | 0.00015664 |   0.0 |  0.08
Modify  | 0.0042217  | 0.0042217  | 0.0042217  |   0.0 |  2.14
Other   |            | 0.001339   |            |       |  0.68

Nlocal:    512 ave 512 max 512 min
Histogram: 1 0 0 0 0 0 0 0 0 0
```

(continues on next page)

(continued from previous page)

```
Nghost:    1447 ave 1447 max 1447 min
Histogram: 1 0 0 0 0 0 0 0 0 0
Neighs:    12018 ave 12018 max 12018 min
Histogram: 1 0 0 0 0 0 0 0 0 0

Total # of neighbors = 12018
Ave neighs/atom = 23.4727
Ave special neighs/atom = 0
Neighbor list builds = 100
Dangerous builds = 100
Setting up Verlet run ...
  Unit style    : lj
  Current step  : 1000
  Time step     : 0.005
Per MPI rank memory allocation (min/avg/max) = 6.109 | 6.109 | 6.109 Mbytes
Step PotEng KinEng TotEng Temp Press Density
    1000   -1862.6606    799.32963   -1063.3309    1.0428306  -0.25224674        0.512
    1100   -1853.4242    819.28434   -1034.1399    1.0688641  -0.16446166        0.512
    1200   -1840.5875    793.33971   -1047.2477    1.0350159  -0.21578932        0.512
    1300   -1838.9016     796.0771   -1042.8245    1.0385872  -0.19354995        0.512
    1400   -1848.5392     752.5312    -1096.008   0.98177587  -0.22928676        0.512
    1500   -1856.8763    746.44097   -1110.4353   0.97383035  -0.18936813        0.512
    1600   -1869.5931    732.08398   -1137.5091   0.95509978   -0.2751998        0.512
    1700   -1887.7451    761.66169   -1126.0834   0.99368779  -0.35301947        0.512
    1800   -1882.9325    729.51153    -1153.421   0.95174368  -0.33872437        0.512
    1900   -1867.9452    763.40829   -1104.5369   0.99596646  -0.30614623        0.512
    2000   -1874.4475     752.8181   -1121.6294   0.98215017  -0.30908533        0.512
Loop time of 0.199068 on 1 procs for 1000 steps with 512 atoms

Performance: 2170111.968 tau/day, 5023.407 timesteps/s
99.7% CPU use with 1 MPI tasks x no OpenMP threads

MPI task timing breakdown:
Section |  min time  |  avg time  |  max time  |%varavg| %total
---------------------------------------------------------------
Pair    | 0.13415    | 0.13415    | 0.13415    |   0.0 | 67.39
Bond    | 6.5804e-05 | 6.5804e-05 | 6.5804e-05 |   0.0 |  0.03
Neigh   | 0.049349   | 0.049349   | 0.049349   |   0.0 | 24.79
Comm    | 0.0079701  | 0.0079701  | 0.0079701  |   0.0 |  4.00
Output  | 0.00013518 | 0.00013518 | 0.00013518 |   0.0 |  0.07
Modify  | 0.0060918  | 0.0060918  | 0.0060918  |   0.0 |  3.06
Other   |            | 0.001308   |            |       |  0.66

Nlocal:    512 ave 512 max 512 min
Histogram: 1 0 0 0 0 0 0 0 0 0
Nghost:    1464 ave 1464 max 1464 min
Histogram: 1 0 0 0 0 0 0 0 0 0
Neighs:    11895 ave 11895 max 11895 min
Histogram: 1 0 0 0 0 0 0 0 0 0

Total # of neighbors = 11895
Ave neighs/atom = 23.2324
Ave special neighs/atom = 0
Neighbor list builds = 100
Dangerous builds = 100
Setting up Verlet run ...
  Unit style    : lj
```

(continues on next page)

(continued from previous page)

```
  Current step  : 2000
  Time step     : 0.005
Per MPI rank memory allocation (min/avg/max) = 7.383 | 7.383 | 7.383 Mbytes
Step PotEng KinEng TotEng Temp Press Density
    2000   -1874.4475     752.8181   -1121.6294   0.98215017  -0.30908533         0.512
    2100   -1858.3201     763.1433   -1095.1768   0.99562074  -0.25351893         0.512
    2200   -1866.9213    770.43352   -1096.4878    1.0051318  -0.27646217         0.512
    2300   -1879.7957    721.28174    -1158.514   0.94100683  -0.31881659         0.512
    2400   -1886.0524    740.29981   -1145.7526   0.96581841  -0.36988824         0.512
    2500   -1862.4955    731.77932   -1130.7162   0.95470231  -0.23656666         0.512
    2600    -1847.542    748.14185   -1099.4002   0.97604938  -0.22297358         0.512
    2700   -1863.1603    715.01181   -1148.1485   0.93282689  -0.27535839         0.512
    2800   -1858.9263    711.64082   -1147.2855   0.92842899  -0.31272288         0.512
    2900   -1862.0527     788.4678   -1073.5849    1.0286599  -0.20135611         0.512
    3000   -1848.1516    797.66227   -1050.4894    1.0406553  -0.27353978         0.512
    3100   -1883.8621    793.05475   -1090.8073    1.0346442  -0.29972206         0.512
    3200   -1890.4065    791.32467   -1099.0819     1.032387  -0.35642545         0.512
    3300   -1859.2997    745.34089   -1113.9588   0.97239516  -0.26722308         0.512
    3400   -1869.8929    762.57135   -1107.3216   0.99487457  -0.14226646         0.512
    3500   -1879.6557    732.72846   -1146.9273   0.95594058  -0.21775981         0.512
    3600   -1899.0227    766.18046   -1132.8422   0.99958312   -0.2798366         0.512
    3700   -1872.6895    817.06218   -1055.6273     1.065965  -0.23193326         0.512
    3800   -1891.1356    802.56843   -1088.5672     1.047056  -0.23387156         0.512
    3900    -1840.088    753.28729   -1086.8007   0.98276228  -0.21465531         0.512
    4000   -1882.7617    803.22857   -1079.5332    1.0479172  -0.31896543         0.512
    4100   -1873.9061    787.05281   -1086.8533    1.0268138  -0.26608644         0.512
    4200   -1871.6627    832.59728   -1039.0655    1.0862326  -0.29040189         0.512
    4300   -1865.3725    819.61212   -1045.7603    1.0692917  -0.22592305         0.512
    4400   -1875.5306    806.71297   -1068.8176    1.0524631  -0.31604788         0.512
    4500    -1857.109    828.16158   -1028.9474    1.0804456   -0.2464398         0.512
    4600   -1857.8912     729.7257   -1128.1655    0.9520231  -0.31385004         0.512
    4700    -1842.205    734.17836   -1108.0267   0.95783217  -0.27130372         0.512
    4800   -1864.7696    776.14641   -1088.6232     1.012585  -0.31668109         0.512
    4900   -1858.1103    793.41913   -1064.6911    1.0351195  -0.16583366         0.512
    5000   -1867.7818    815.23276   -1052.5491    1.0635783  -0.28680645         0.512
    5100   -1838.0477      725.412   -1112.6357    0.9463953  -0.28647867         0.512
    5200   -1810.7731     731.9772   -1078.7959   0.95496047  -0.16033508         0.512
    5300   -1837.5311    749.48424   -1088.0469   0.97780071  -0.20281441         0.512
    5400   -1873.1094    764.60064   -1108.5088   0.99752204  -0.41358648         0.512
    5500   -1888.9361    748.61774   -1140.3184   0.97667025  -0.36938658         0.512
    5600   -1869.9513    762.05258   -1107.8988   0.99419776   -0.4223791         0.512
    5700    -1858.339    746.55871   -1111.7803   0.97398396  -0.42269281         0.512
    5800   -1863.2613    749.34951   -1113.9118   0.97762493  -0.38710722         0.512
    5900   -1873.7293    773.93107   -1099.7982    1.0096948  -0.26021895         0.512
    6000    -1873.456    787.00426   -1086.4518    1.0267505  -0.22677264         0.512
    6100   -1856.3965    789.71834   -1066.6782    1.0302914  -0.23662444         0.512
    6200   -1868.1487    781.09973   -1087.0489    1.0190473  -0.13471937         0.512
    6300   -1873.9941    740.70637   -1133.2877   0.96634882  -0.26089329         0.512
    6400   -1879.5293    758.83006   -1120.6993   0.98999355  -0.40717493         0.512
    6500    -1873.208    730.21233   -1142.9956   0.95265797  -0.33679524         0.512
    6600    -1893.088    738.17171   -1154.9163   0.96304202  -0.34898503         0.512
    6700   -1854.9994    735.97428   -1119.0252   0.96017518  -0.28228204         0.512
    6800   -1841.9759    797.06384   -1044.9121    1.0398745  -0.19145452         0.512
    6900   -1850.4935    786.14747   -1064.3461    1.0256327  -0.29327665         0.512
    7000   -1845.6749    797.15417   -1048.5207    1.0399924  -0.45867335         0.512
    7100     -1831.03    827.34343   -1003.6866    1.0793782    -0.179498         0.512
    7200   -1888.1042    749.22706   -1138.8771   0.97746518  -0.53010406         0.512
```

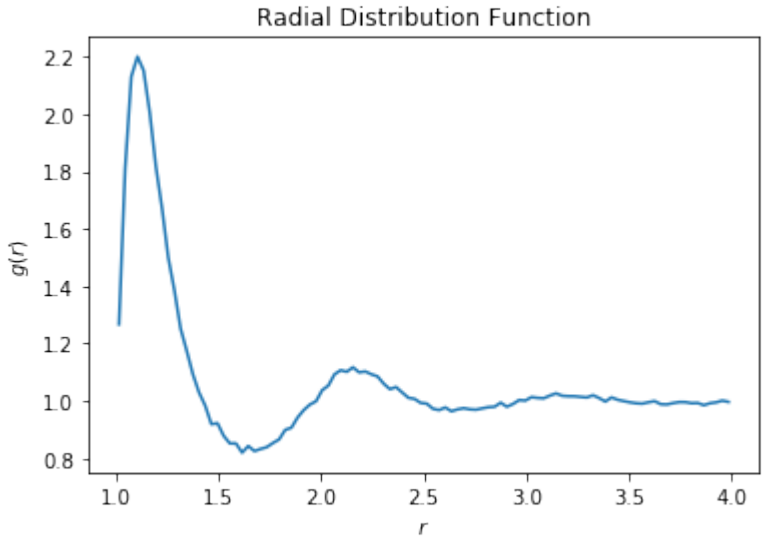
(continues on next page)

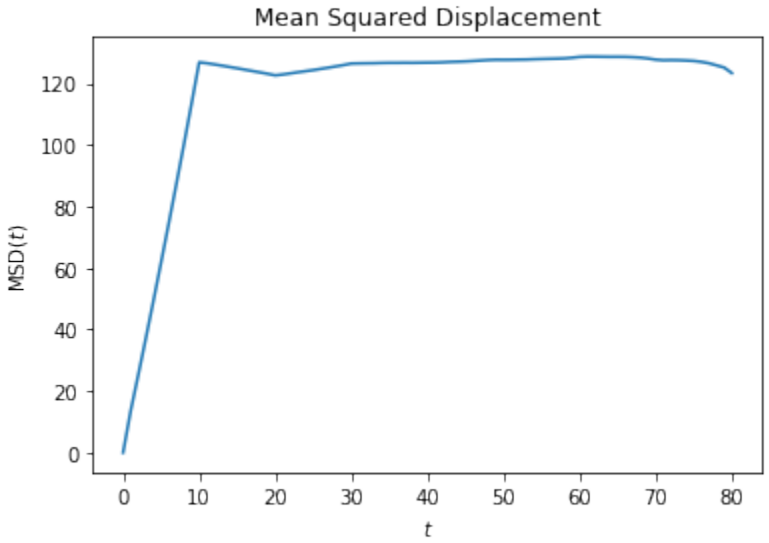
(continued from previous page)

```
    7300   -1859.9233     754.0352   -1105.8881   0.98373803  -0.39545192         0.512
    7400   -1851.9183    787.60897   -1064.3093    1.0275394  -0.37094061         0.512
    7500   -1848.0739    759.73299   -1088.3409   0.99117155  -0.34780329         0.512
    7600   -1853.6532    764.84642   -1088.8067   0.99784269 -0.098590718         0.512
    7700   -1876.6886    756.38707   -1120.3016   0.98680636  -0.17912577         0.512
    7800   -1857.6403    719.20424   -1138.4361   0.93829647  -0.32247855         0.512
    7900   -1891.2369    707.44358   -1183.7933   0.92295314  -0.44928961         0.512
    8000   -1930.5545    747.85472   -1182.6997   0.97567478   -0.2607688         0.512
    8100   -1931.3403    744.07929    -1187.261   0.97074924  -0.36763161         0.512
    8200   -1920.9036     757.0399   -1163.8637   0.98765806  -0.29103201         0.512
    8300   -1904.5561    747.57535   -1156.9807    0.9753103  -0.38464012         0.512
    8400   -1844.7405    820.31281   -1024.4277    1.0702059 -0.044405706         0.512
    8500   -1860.3078    809.13555   -1051.1723    1.0556237 -0.018849627         0.512
    8600   -1841.1531    776.85955   -1064.2935    1.0135154 -0.080192818         0.512
    8700   -1860.6583      785.807   -1074.8513    1.0251885  -0.29734141         0.512
    8800   -1841.0455    779.78036   -1061.2651     1.017326  -0.11420405         0.512
    8900   -1887.3837    878.92659   -1008.4571    1.1466753  -0.34666733         0.512
    9000   -1879.4834    767.25891   -1112.2245    1.0009901   -0.3331713         0.512
    9100   -1900.1999    818.54475   -1081.6552    1.0678992  -0.19458572         0.512
    9200   -1882.1203    794.90843   -1087.2118    1.0370625  -0.25879106         0.512
    9300   -1893.5664    783.13068   -1110.4357    1.0216969  -0.25735285         0.512
    9400   -1893.5147    756.00962   -1137.5051   0.98631392  -0.26461519         0.512
    9500   -1908.8115    742.60538   -1166.2061   0.96882633   -0.4468834         0.512
    9600   -1887.0565    762.24949    -1124.807   0.99445465  -0.36695082         0.512
    9700   -1878.5858    771.53563   -1107.0502    1.0065696   -0.2300855         0.512
    9800   -1848.4047    752.27373   -1096.1309   0.98143997  -0.28729274         0.512
    9900    -1865.561    731.41466   -1134.1464   0.95422656   -0.3874617         0.512
   10000   -1887.2808    787.80237   -1099.4784    1.0277917  -0.26779032         0.512
Loop time of 1.63759 on 1 procs for 8000 steps with 512 atoms

Performance: 2110423.670 tau/day, 4885.240 timesteps/s
99.4% CPU use with 1 MPI tasks x no OpenMP threads

MPI task timing breakdown:
Section |  min time  |  avg time  |  max time  |%varavg| %total
---------------------------------------------------------------
Pair    | 1.0823     | 1.0823     | 1.0823     |   0.0 | 66.09
Bond    | 0.00054955 | 0.00054955 | 0.00054955 |   0.0 |  0.03
Neigh   | 0.39492    | 0.39492    | 0.39492    |   0.0 | 24.12
Comm    | 0.064503   | 0.064503   | 0.064503   |   0.0 |  3.94
Output  | 0.035598   | 0.035598   | 0.035598   |   0.0 |  2.17
Modify  | 0.049172   | 0.049172   | 0.049172   |   0.0 |  3.00
Other   |            | 0.01058    |            |       |  0.65

Nlocal:    512 ave 512 max 512 min
Histogram: 1 0 0 0 0 0 0 0 0 0
Nghost:    1398 ave 1398 max 1398 min
Histogram: 1 0 0 0 0 0 0 0 0 0
Neighs:    12036 ave 12036 max 12036 min
Histogram: 1 0 0 0 0 0 0 0 0 0

Total # of neighbors = 12036
Ave neighs/atom = 23.5078
Ave special neighs/atom = 0
Neighbor list builds = 800
Dangerous builds = 800
Total wall time: 0:00:02
```

[3]:
```
%matplotlib inline

import freud
from matplotlib import pyplot as plt
import numpy as np
import warnings
```

[4]:
```
with warnings.catch_warnings():
    warnings.simplefilter('ignore')
    # 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:])

# Plot the RDF
plt.plot(rdf.bin_centers, rdf.rdf)
plt.title('Radial Distribution Function')
plt.xlabel('$r$')
plt.ylabel('$g(r)$')
plt.show()

# Plot the MSD
plt.plot(msd.msd)
plt.title('Mean Squared Displacement')
plt.xlabel('$t$')
plt.ylabel('MSD$(t)$')
plt.show()
```





Using Machine Learning for Structural Identification

This notebook provides a demonstration of how a simple set of descriptors computed by freud can be coupled with machine learning for structural identification. The set of descriptors used here are not enough to identify complex crystal structures, but this notebook provides an introduction. For a more powerful set of descriptors, see the paper Machine learning for crystal identification and discovery (Spellings 2018) and the library pythia, both of which use freud for their computations.

[1]:
```
import freud
import matplotlib.pyplot as plt
import matplotlib.cm
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings("ignore")
```

We generate sample body-centered cubic, face-centered cubic, and simple cubic structures. Each structure has at least 4000 particles.

[2]:
```
N = 4000
noise = 0.04
structures = {}
n = round((N/2)**(1/3))
structures['bcc'] = freud.data.UnitCell.bcc().generate_system(n, sigma_noise=noise)
n = round((N/4)**(1/3))
structures['fcc'] = freud.data.UnitCell.fcc().generate_system(n, sigma_noise=noise)
n = round((N/1)**(1/3))
structures['sc'] = freud.data.UnitCell.sc().generate_system(n, sigma_noise=noise)
for name, (box, positions) in structures.items():
    print(name, 'has', len(positions), 'particles.')
```

```
bcc has 4394 particles.
fcc has 4000 particles.
sc has 4096 particles.
```

Next, we compute the Steinhardt order parameters q_l for $l \in \{4, 6, 8, 10, 12\}$.

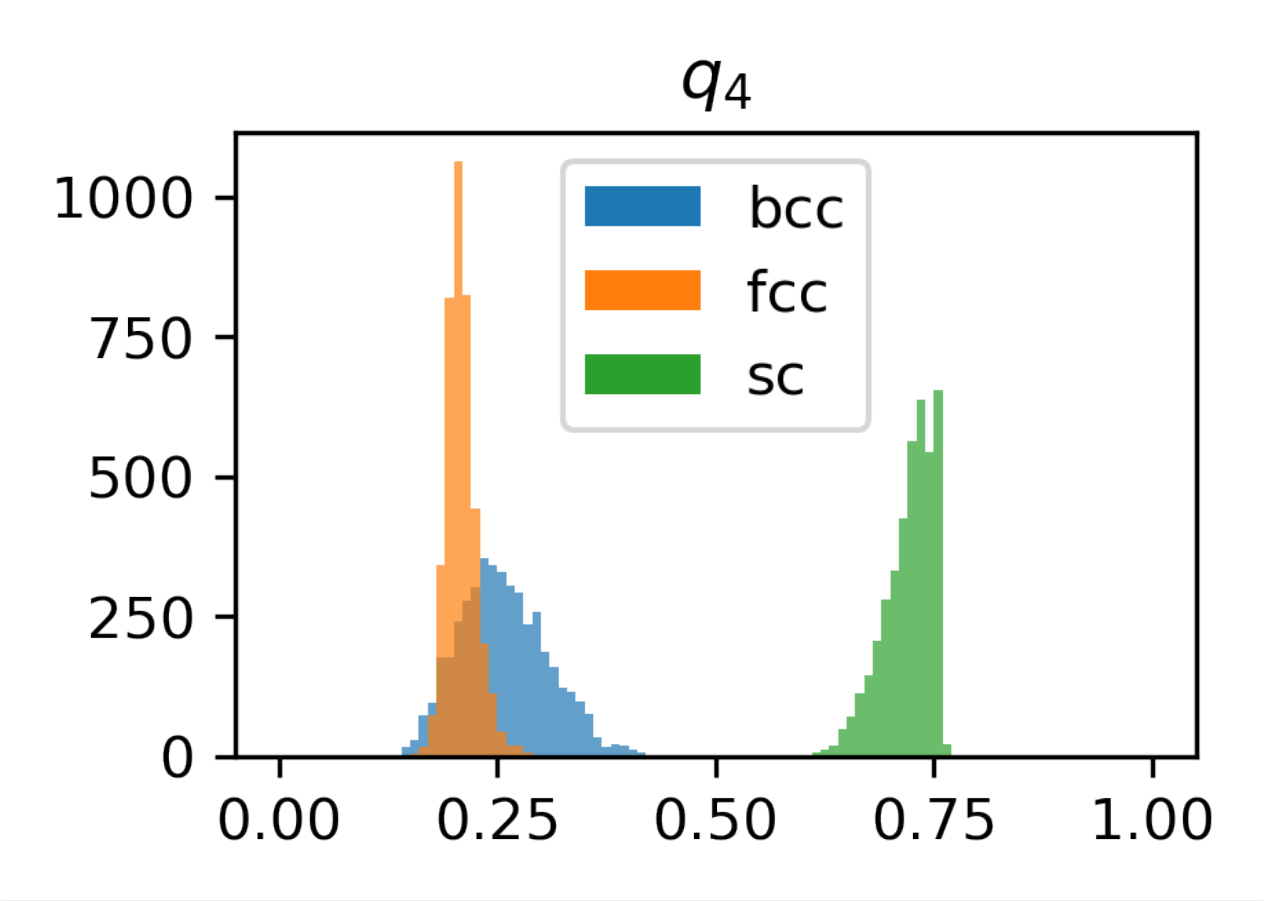
We use the Voronoi neighbor list, removing neighbors whose Voronoi facets are small.

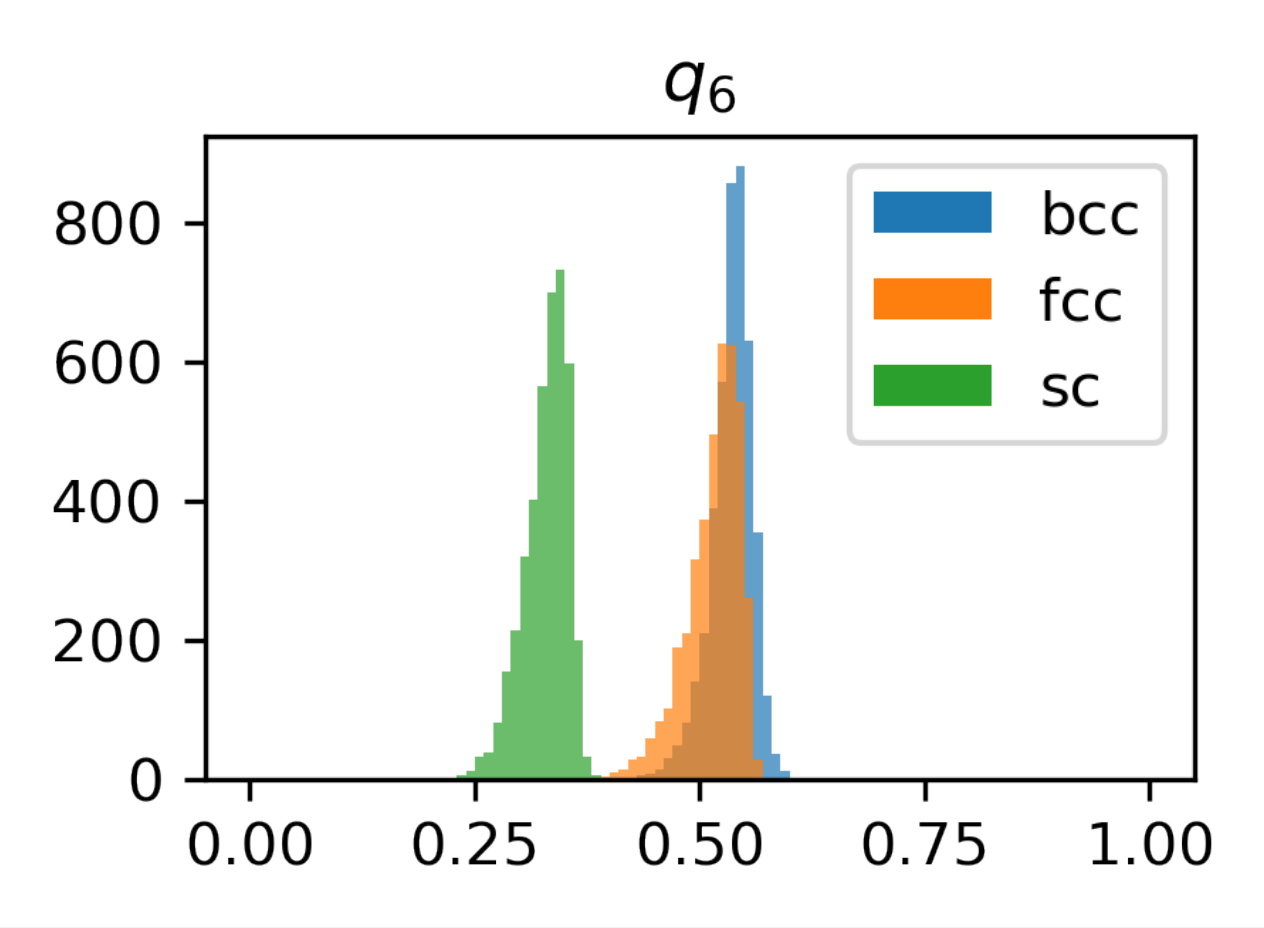
[3]:
```
def get_features(box, positions, structure):
    voro = freud.locality.Voronoi()
    voro.compute(system=(box, positions))
    nlist = voro.nlist.copy()
    nlist.filter(nlist.weights > 0.1)
    features = {}
    for l in [4, 6, 8, 10, 12]:
        ql = freud.order.Steinhardt(l=l, weighted=True)
        ql.compute(system=(box, positions), neighbors=nlist)
        features['q{}'.format(l)] = ql.particle_order

    return features
```

[4]:
```
structure_features = {}
for name, (box, positions) in structures.items():
    structure_features[name] = get_features(box, positions, name)
```

Here, we plot a histogram of the q_4 and q_6 values for each structure.

[5]:
```
for l in [4, 6]:
    plt.figure(figsize=(3, 2), dpi=300)
    for name in structures.keys():
        plt.hist(structure_features[name]['q{}'.format(l)], range=(0, 1), bins=100,␣
↪label=name, alpha=0.7)
    plt.title(r'$q_{{{l}}}$'.format(l=l))
    plt.legend()
    for lh in plt.legend().legendHandles:
        lh.set_alpha(1)
    plt.show()
```


q_4
1000
750
500
250
0
0.00
0.25
0.50
0.75
1.00
bcc
fcc
sc



Next, we will train a Support Vector Machine to predict particles' structures based on these Steinhardt Q_l descriptors. We build `pandas` data frames to hold the structure features, encoding the structure as an integer. We use `train_test_split` to train on part of the data and test the model on a separate part of the data.

[6]:
```
structure_dfs = {}
for i, structure in enumerate(structure_features):
    df = pd.DataFrame.from_dict(structure_features[structure])
    df['class'] = i
    structure_dfs[structure] = df
```

[7]:
```
df = pd.concat(structure_dfs.values()).reset_index(drop=True)
```

[8]:
```
from sklearn.preprocessing import normalize
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
```

[9]:
```
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)
```

(continues on next page)

(continued from previous page)

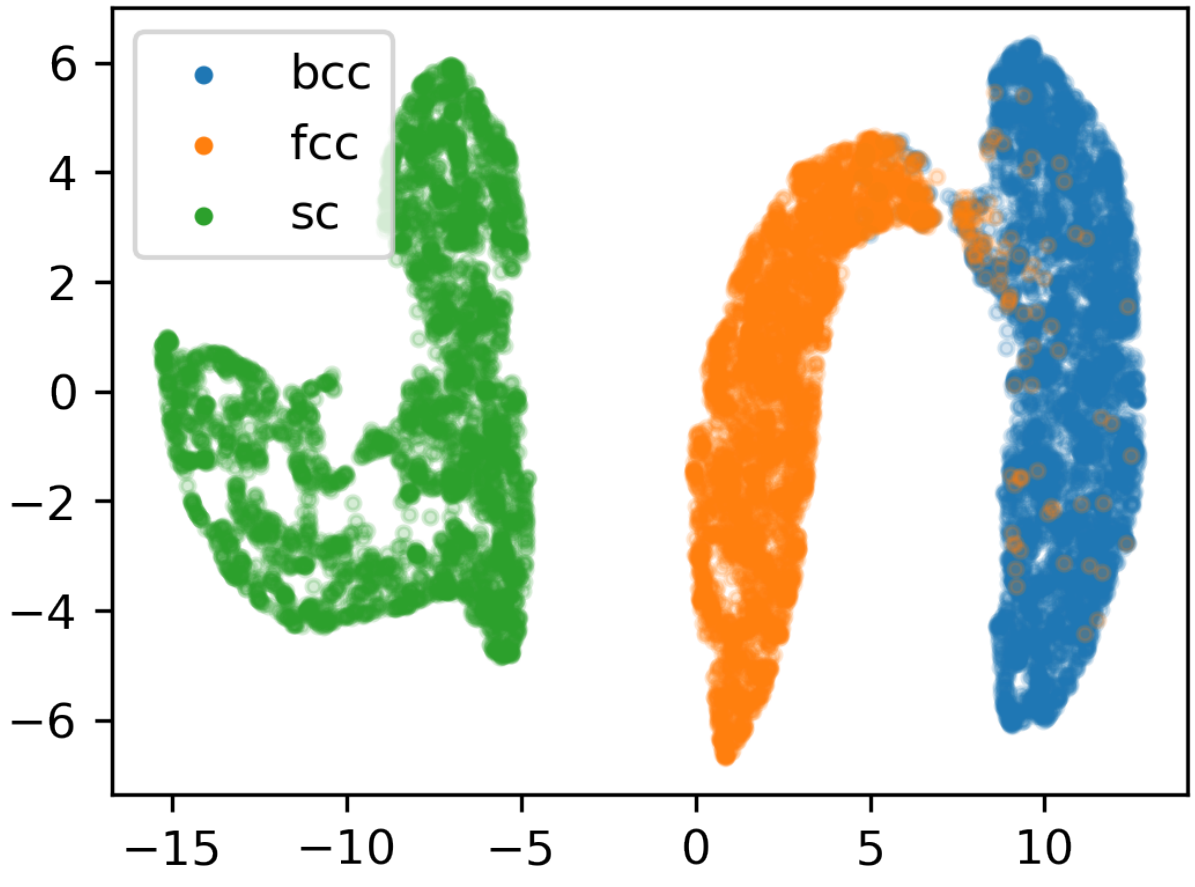
```
print('Score:', svm.score(X_test, y_test))
```

```
Score: 0.9868995633187773
```

Finally, we use the Uniform Manifold Approximation and Projection method (McInnes 2018, GitHub repo) to project the high-dimensional descriptors into a two-dimensional plot. Notice that some bcc particles overlap with fcc particles. This can be expected from the noise that was added to the structures. The particles that were incorrectly classified by the SVM above are probably located in this overlapping region.

[10]:
```
from umap import UMAP
umap = UMAP(random_state=42)

X_reduced = umap.fit_transform(X)
```

[11]:
```
plt.figure(figsize=(4, 3), dpi=300)
for i in range(max(y) + 1):
    indices = np.where(y == i)[0]
    plt.scatter(X_reduced[indices, 0], X_reduced[indices, 1],
                color=matplotlib.cm.tab10(i), s=8, alpha=0.2,
                label=list(structure_features.keys())[i])
plt.legend()
for lh in plt.legend().legendHandles:
    lh.set_alpha(1)
plt.show()
```



Calculating Strain via Voxelization

This notebook shows how to use `freud`'s neighbor finding to create a voxelized version of a system.

In brief, we are going to create a set of points that define the centers of our voxels, then assign all particles to one of these voxels. Then we sum up some property of the particles amongst all particles in a bin.

At the end we want to have a sampling of some particle property in our system on a regular grid (as a NumPy array).

[1]:
```
%matplotlib inline
import freud
import numpy as np
import matplotlib.pyplot as plt
import re
from scipy.sparse import csr_matrix, csc_matrix
```

This uses data from some text files that were output from the visualization software OVITO (https://ovito.org/)

The files have a header with box information, and then a list of particle info. These files have 10 fields per particle:

```
(ID#, position(x,y,z), strains(xx,yy,zz,yz,xz,xy))
```

The goal is to turn this into an $(N_x, N_y, N_z, 3, 3)$ NumPy array, where N_x, N_y, N_z are the number of bins in each dimension, and each of those bins has an averaged 3x3 strain array.

First we read in the box info from our text files and construct an average box. We need this so we can make our bin centers

[2]:
```
framefiles = ['data/strain_data/frame{f}'.format(f=f) for f in [100, 110, 120, 130]]

# read all the boxes, so we can make the grid points for voxelizing
boxes = []
for f in framefiles:
    ff = open(f, 'r')
    _ = ff.readline()
    header = ff.readline()

    match = re.match('^Lattice=".*"', header)
    boxstring = match.group(0)
    boxes.append(np.array(str.split(boxstring[9:-1]), dtype=np.float).reshape((3,3)).
↪T)
    ff.close()

# find the average box
ave_box = np.array(boxes).mean(axis=0)
```

Now we make the bin centers using `np.meshgrid`, but append and combine the X, Y, and Z coordinates into an array of shape $(N_x N_y N_z, 3)$ to pass to `freud`.

[3]:
```
res = (60, 10, 45) # The number of bins (in x,y,z)
xx = np.linspace(-ave_box[0,0]/2,ave_box[0,0]/2,num=res[0])
yy = np.linspace(-ave_box[1,1]/2,ave_box[1,1]/2,num=res[1])
zz = np.linspace(-ave_box[2,2]/2,ave_box[2,2]/2,num=res[2])
XX, YY, ZZ = np.meshgrid(xx,yy,zz)

XYZ = np.append(np.append(XX.flatten().reshape((-1,1)),
                          YY.flatten().reshape((-1,1)), axis=1),
                ZZ.flatten().reshape((-1,1)), axis=1).astype(np.float32)
```

Now we iterate over our files and compute the first nearest neighbor (among the bin centers) of the particles, so we know which bin to average them in.

It is important to use scipy's `csr_matrix` for this process when the number of particles is large. These files contain >80,000 particles, and without the sparse matrix, the dot product to determine grid totals would be extremely slow.

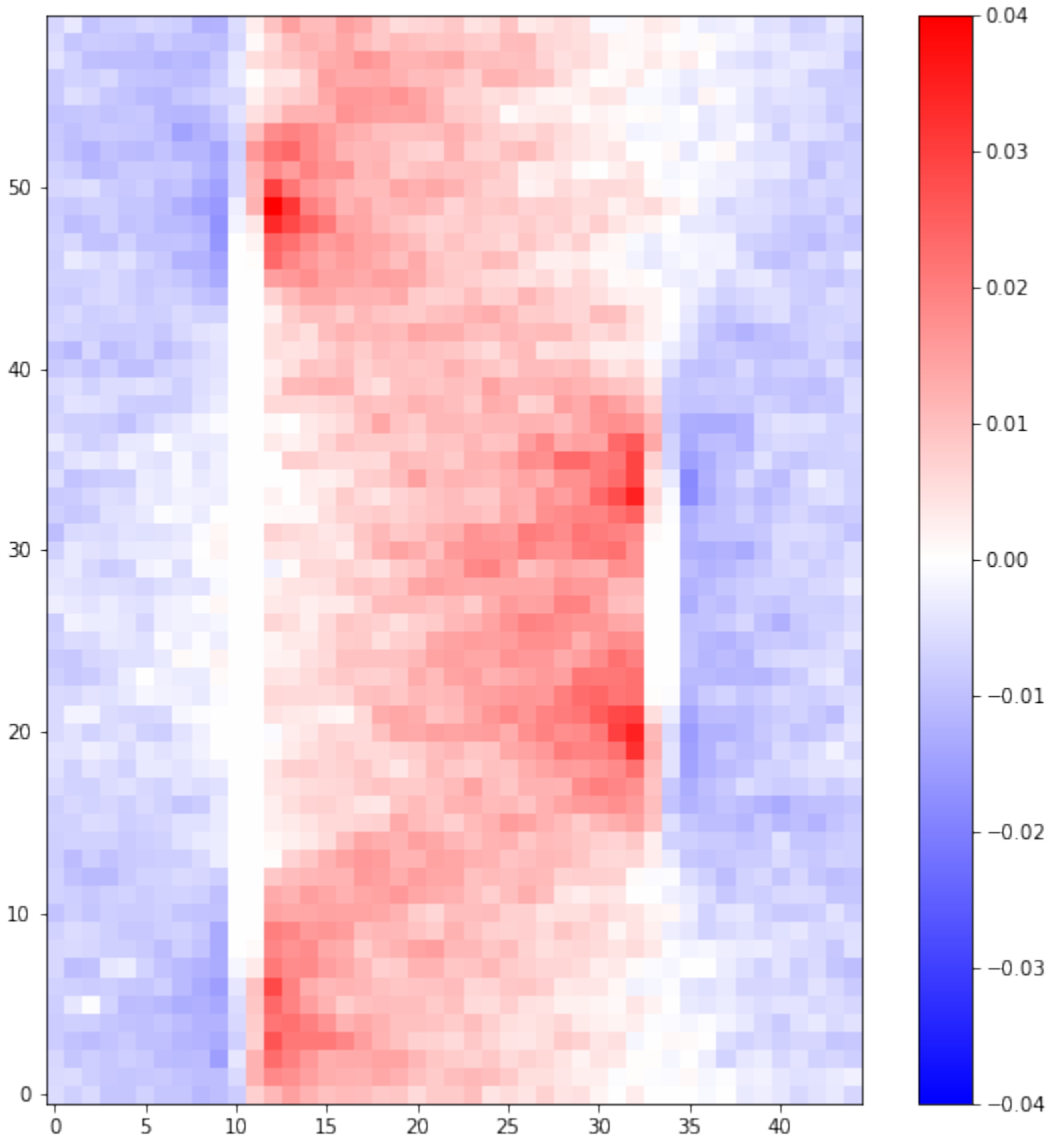
[4]:
```
master_strains = np.zeros((XYZ.shape[0], 6)) # matrix to sum into

for i in range(len(framefiles)):
    data = np.loadtxt(framefiles[i], skiprows=2).astype(np.float32)

    box = freud.box.Box(Lx=boxes[i][0, 0],
                        Ly=boxes[i][1, 1],
                        Lz=boxes[i][2, 2],
                        yz=boxes[i][1, 2],
                        xz=boxes[i][0, 2],
                        xy=boxes[i][0, 1])
    nlist = freud.AABBQuery(box, XYZ).query(
        data[:,1:4], {'num_neighbors': 1}).toNeighborList()
    neighbors = nlist.point_indices

    sprse = csr_matrix((np.ones(len(neighbors)), (neighbors, np.
↪arange(len(neighbors)))),
                       shape=(XYZ.shape[0], len(neighbors)))

    # strain data
    sdata = data[:, 4:]
    binned = np.zeros((XYZ.shape[0], 6))
    # number of particles in each bin
    grid_totals = sprse.dot(np.ones(len(neighbors)))
    grid_totals[grid_totals==0] = 1 # get rid of division errors

    for j in range(6):
        binned[:,j] = sprse.dot(sdata[:, j]) / grid_totals

    master_strains = master_strains + binned

master_strains = master_strains/len(framefiles) # divide by number of frames
```

Now we pack up the resulting array into the shape we want it to be: $(N_x, N_y, N_z, 3, 3)$

[5]:
```
final_matrix = np.zeros((res[1],res[0],res[2],3,3))

# this mapping turns 6 strain values into a symmetric (3,3) matrix
voigt_map = {0:(0,0), 1:(1,1), 2:(2,2), 3:(1,2), 4:(0,2), 5:(0,1)}

for i in range(6):
    v = voigt_map[i]
    final_matrix[:,:,:,v[0],v[1]] = master_strains[:,i].reshape(res[1],res[0],res[2])
    if v[0]!=v[1]:
        final_matrix[:,:,:,v[1],v[0]] = master_strains[:,i].reshape(res[1],res[0],
↪res[2])
```

Since we are only using four frames, the distribution is not very well sampled. But we can get a clue that a distinct distribution of strain is emerging if we average along the first axis of the matrix (this particular system should not vary in that direction)

[6]:
```
plt.figure(figsize=(10,10))
plt.imshow(final_matrix[:,:,:,0,0].mean(axis=0),
           origin='lower', cmap=plt.cm.bwr,
           vmin=-0.04, vmax=0.04, interpolation='none')
plt.colorbar()
plt.show()
```



Visualizing analyses with fresnel

In this notebook, we simulate a system of tetrahedra, color particles according to their local density, and path-trace the resulting image with fresnel.

The cell below runs a short HOOMD-blue simulation of tetrahedra using Hard Particle Monte Carlo (HPMC).

[1]:
```
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 our tetrahedra and configure the HPMC integrator
mc = hoomd.hpmc.integrate.convex_polyhedron(seed=42)
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()
```

```
HOOMD-blue v2.6.0-151-gea140cffb DOUBLE HPMC_MIXED MPI TBB SSE SSE2 SSE3 SSE4_1 SSE4_
↪2 AVX AVX2
Compiled: 09/25/2019
Copyright (c) 2009-2019 The Regents of the University of Michigan.
-----
You are using HOOMD-blue. Please cite the following:
* J A Anderson, C D Lorenz, and A Travesset. "General purpose molecular dynamics
  simulations fully implemented on graphics processing units", Journal of
  Computational Physics 227 (2008) 5342--5359
* J Glaser, T D Nguyen, J A Anderson, P Liu, F Spiga, J A Millan, D C Morse, and
  S C Glotzer. "Strong scaling of general-purpose molecular dynamics simulations
  on GPUs", Computer Physics Communications 192 (2015) 97--107
-----
-----
You are using HPMC. Please cite the following:
* J A Anderson, M E Irrgang, and S C Glotzer. "Scalable Metropolis Monte Carlo
  for simulation of hard shapes", Computer Physics Communications 204 (2016) 21
  --30
-----
HOOMD-blue is running on the CPU
notice(2): Group "all" created containing 512 particles
** starting run **
Time 00:00:10 | Step 2094 / 5000 | TPS 209.394 | ETA 00:00:13
Time 00:00:20 | Step 4238 / 5000 | TPS 214.352 | ETA 00:00:03
Time 00:00:23 | Step 5000 / 5000 | TPS 213.528 | ETA 00:00:00
Average TPS: 212.118
---------
notice(2): -- HPMC stats:
notice(2): Average translate acceptance: 0.749166
notice(2): Average rotate acceptance:    0.867601
notice(2): Trial moves per second:        434403
```

(continues on next page)

(continued from previous page)

```
notice(2): Overlap checks per second:     3.12535e+07
notice(2): Overlap checks per trial move: 71.946
notice(2): Number of overlap errors:      0
** run complete **
```

Now we import the modules needed for analysis and visualization.

[2]:
```
import fresnel
import freud
import matplotlib.cm
from matplotlib.colors import Normalize
import numpy as np
device = fresnel.Device()
```

Next, we'll set up the arrays needed for the scene and its geometry. This includes the analysis used for coloring particles.

[3]:
```
poly_info = fresnel.util.convex_polyhedron_from_vertices(vertices)
positions = snap.particles.position
orientations = snap.particles.orientation
box = freud.Box.from_box(snap.box)
ld = freud.density.LocalDensity(3.0, 1.0)
ld.compute(system=snap)
colors = matplotlib.cm.viridis(Normalize()(ld.density))
box_points = np.asarray([
    box.make_absolute(
        [[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.make_absolute(
        [[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]])])
```

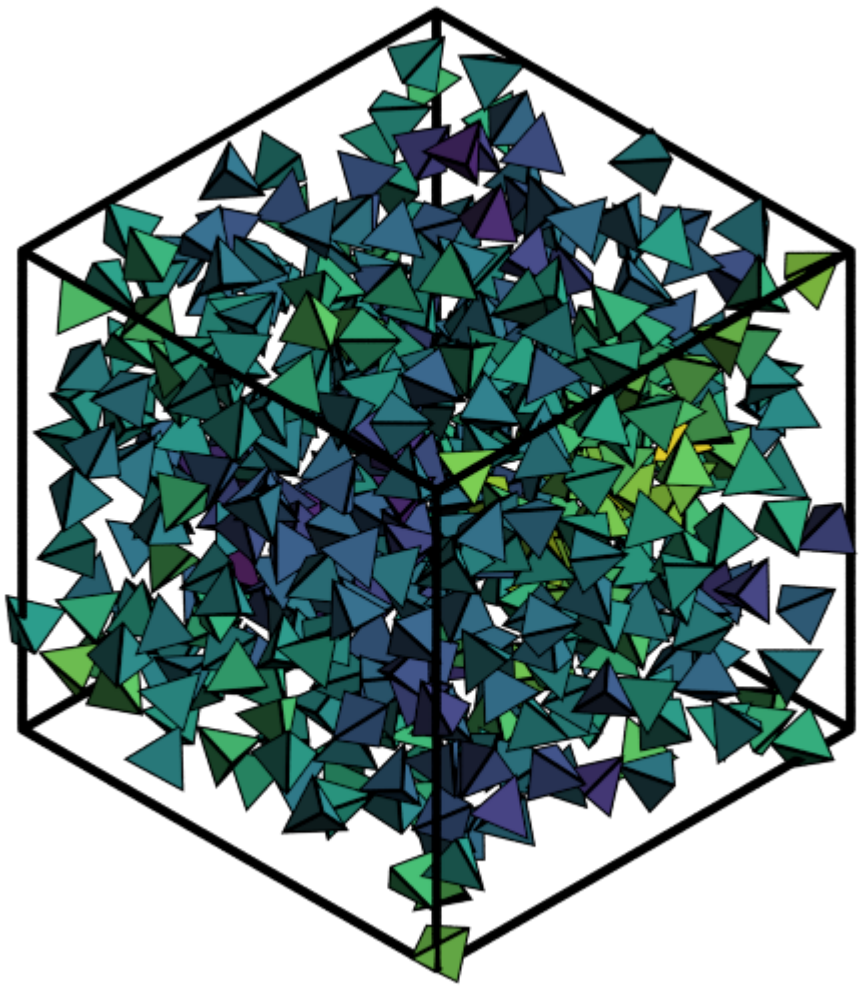
This cell creates the scene and geometry objects to be rendered by `fresnel`.

[4]:
```
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)
```

First, we preview the scene. (This doesn't use path tracing, and is much faster.)

[5]:
```
fresnel.preview(scene, aa_level=3, w=600, h=600)
```

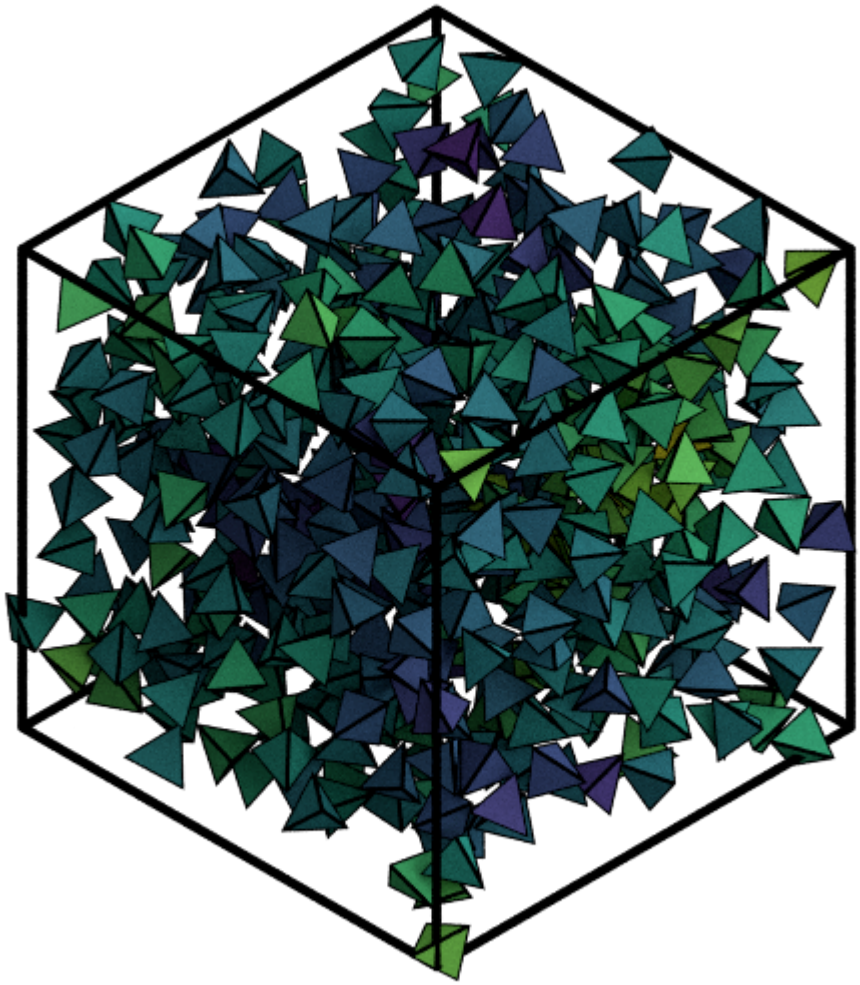
[5]:



Finally, we use path tracing for a high quality image. The number of light samples can be increased to reduce path tracing noise.

[6]:
```
fresnel.pathtrace(scene, light_samples=16, w=600, h=600)
```

[6]:



Visualization with plato

In this notebook, we run a Lennard-Jones simulation, color particles according to their local density computed with `freud`, and display the results with plato. Note that plato has multiple backends – see the plato documentation for information about each backend and the features it supports.

[1]:
```
import hoomd
import hoomd.md
hoomd.context.initialize('')

# Silence the HOOMD output
hoomd.util.quiet_status()
hoomd.option.set_notice_level(0)
```

(continues on next page)

(continued from previous page)

```
# Create a 10x10x10 simple cubic lattice of particles with type name A
system = hoomd.init.create_lattice(unitcell=hoomd.lattice.sc(a=1.5, type_name='A'),␣
↪n=10)

# Specify Lennard-Jones interactions between particle pairs
nl = hoomd.md.nlist.cell()
lj = hoomd.md.pair.lj(r_cut=3.0, nlist=nl)
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)

# Integrate at constant temperature
hoomd.md.integrate.mode_standard(dt=0.005)
integrator = hoomd.md.integrate.nvt(group=hoomd.group.all(), kT=0.01, tau=0.5)
integrator.randomize_velocities(seed=42)

# Run for 10,000 time steps
hoomd.run(10e3)
snap = system.take_snapshot()
```

```
HOOMD-blue v2.6.0-151-gea140cffb DOUBLE HPMC_MIXED MPI TBB SSE SSE2 SSE3 SSE4_1 SSE4_
↪2 AVX AVX2
Compiled: 09/25/2019
Copyright (c) 2009-2019 The Regents of the University of Michigan.
-----
You are using HOOMD-blue. Please cite the following:
* J A Anderson, C D Lorenz, and A Travesset. "General purpose molecular dynamics
  simulations fully implemented on graphics processing units", Journal of
  Computational Physics 227 (2008) 5342--5359
* J Glaser, T D Nguyen, J A Anderson, P Liu, F Spiga, J A Millan, D C Morse, and
  S C Glotzer. "Strong scaling of general-purpose molecular dynamics simulations
  on GPUs", Computer Physics Communications 192 (2015) 97--107
-----
HOOMD-blue is running on the CPU
```

Now we import the modules needed for visualization.

[2]:
```
import freud
import matplotlib.cm
from matplotlib.colors import Normalize
import numpy as np
import plato
# For interactive scenes, use:
import plato.draw.pythreejs as draw
# For static scenes, use:
#import plato.draw.fresnel as draw
```

This code sets up the plato Scene object with the particles and colors computed above.

[3]:
```
positions = snap.particles.position
box = freud.Box.from_box(snap.box)
ld = freud.density.LocalDensity(3.0, 1.0)
ld.compute(system=snap)
colors = matplotlib.cm.viridis(Normalize()(ld.density))
radii = np.ones(len(positions)) * 0.5
box_prim = draw.Box.from_box(box, width=0.2)
sphere_prim = draw.Spheres(
    positions=snap.particles.position,
    radii=radii,
```

(continues on next page)

(continued from previous page)

```
    colors=colors,
    vertex_count=32)
scene = draw.Scene((sphere_prim, box_prim), zoom=1.5)
```

Click and drag the 3D scene below - it's interactive!

[4]:
```
scene.show()
```

```
Renderer(camera=OrthographicCamera(bottom=-15.0,␣
↪children=(DirectionalLight(intensity=0.692820323027551, posit...
```

Visualizing 3D Voronoi and Voxelization

The plato-draw package allows for visualizing particle data in 2D and 3D using a variety of backend libraries. Here, we show a 3D Voronoi diagram drawn using fresnel and pythreejs. We use rowan to generate the view rotation.

To install dependencies:

- `conda install -c conda-forge fresnel`
- `pip install plato-draw rowan`

[1]:
```
import freud
import matplotlib.cm
import numpy as np
import rowan
import plato.draw.fresnel
backend = plato.draw.fresnel
# For interactive scenes:
# import plato.draw.pythreejs
# backend = plato.draw.pythreejs
```

[2]:
```
def plot_crystal(box, positions, colors=None, radii=None, backend=None,
                 polytopes=[], polytope_colors=None):
    if backend is None:
        backend = plato.draw.fresnel
    if colors is None:
        colors = np.array([[0.5, 0.5, 0.5, 1]] * len(positions))
    if radii is None:
        radii = np.array([0.5] * len(positions))
    sphere_prim = backend.Spheres(positions=positions, colors=colors, radii=radii)
    box_prim = backend.Box.from_box(box, width=0.1)
    if polytope_colors is None:
        polytope_colors = colors * np.array([1, 1, 1, 0.4])
    polytope_prims = []
    for p, c in zip(polytopes, polytope_colors):
        p_prim = backend.ConvexPolyhedra(
            positions=[[0, 0, 0]], colors=c, vertices=p, outline=0)
        polytope_prims.append(p_prim)
    rotation = rowan.multiply(
        rowan.from_axis_angle([1, 0, 0], np.pi/10),
        rowan.from_axis_angle([0, 1, 0], -np.pi/10))
    scene = backend.Scene([sphere_prim, box_prim, *polytope_prims],
                          zoom=3, rotation=rotation)
    if backend is not plato.draw.fresnel:
        scene.enable('directional_light')
```

(continues on next page)

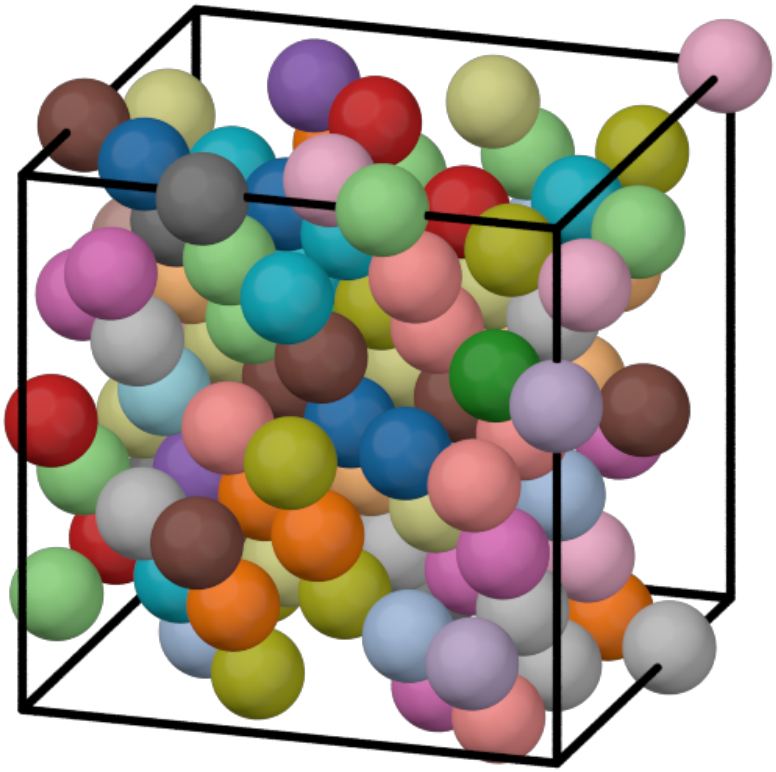
(continued from previous page)

```
    else:
        scene.enable('antialiasing')
    scene.show()
```

We generate an fcc structure and add Gaussian noise to the positions. Colors are assigned randomly.

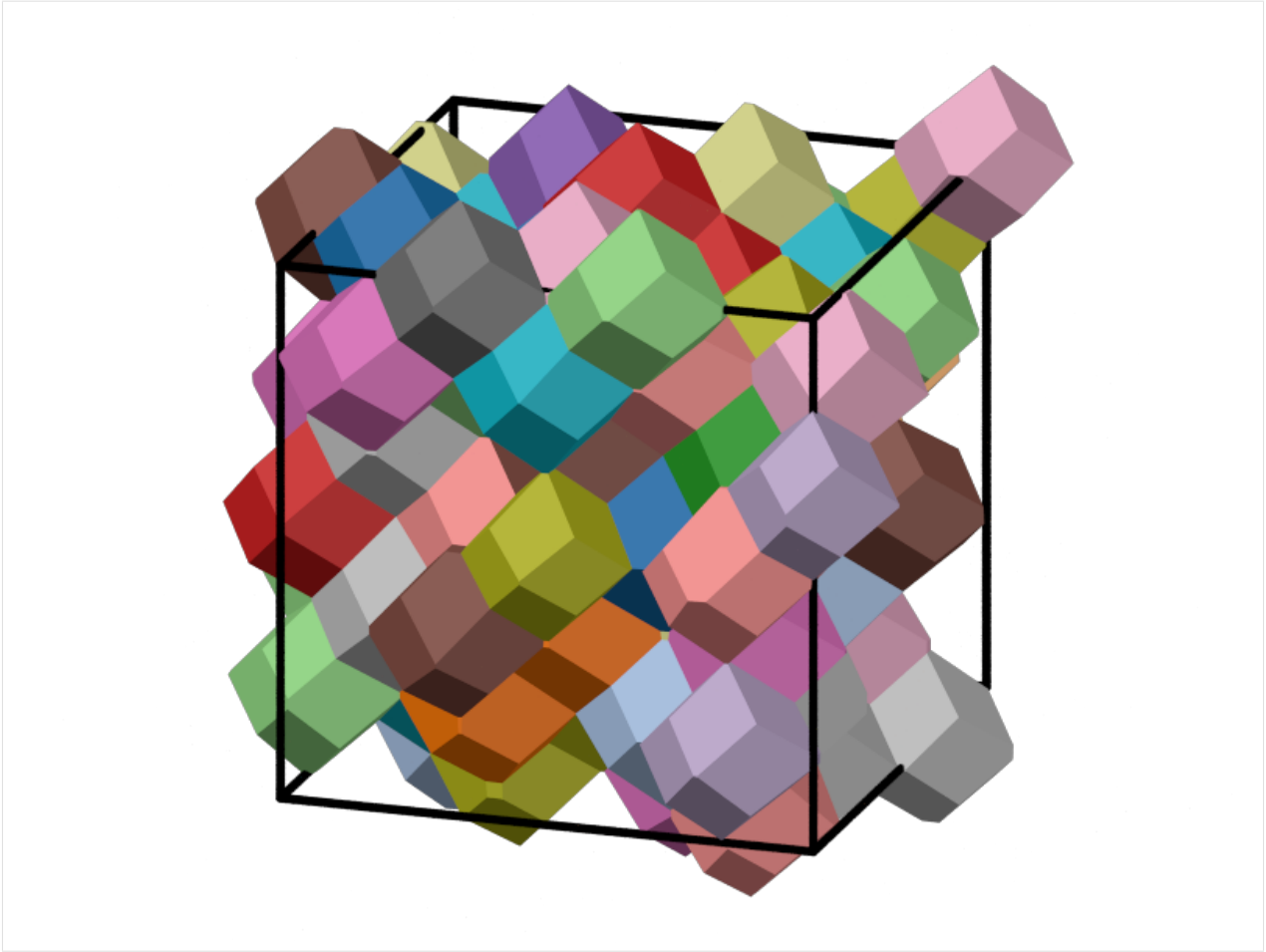
[3]:
```
np.random.seed(12)
box, positions = freud.data.UnitCell.fcc().generate_system(3, scale=2, sigma_noise=0.
↪05)
cmap = matplotlib.cm.get_cmap('tab20')
colors = cmap(np.random.rand(len(positions)))
```

[4]:
```
plot_crystal(box, positions, colors, backend=backend)
```



We make a Voronoi tesselation of the system and plot it in 3D. The Voronoi cells are approximately rhombic dodecahedra, which tesselate 3D space in a face-centered cubic lattice.

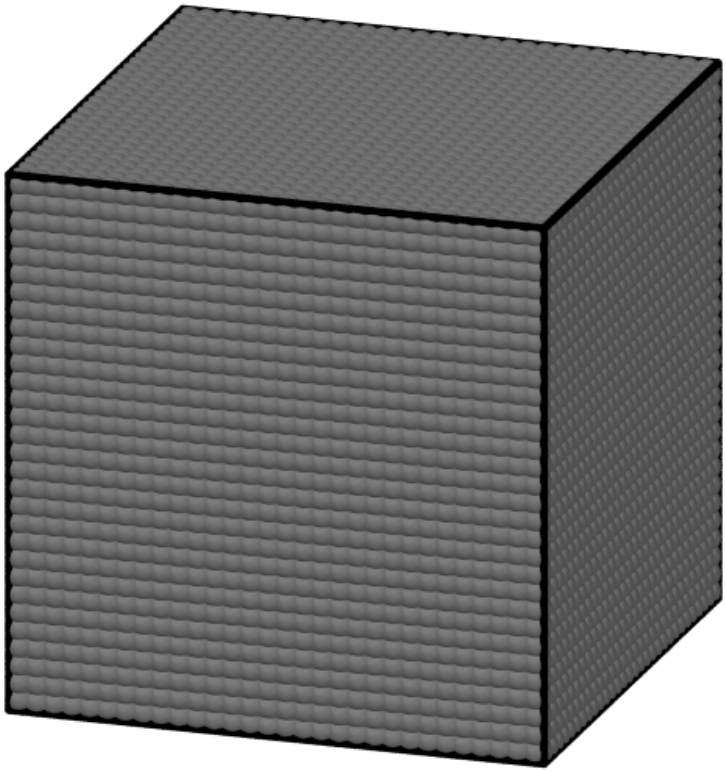
[5]:
```
voro = freud.locality.Voronoi()
voro.compute(system=(box, positions))
plot_crystal(box, positions, colors=colors,
             backend=backend, polytopes=voro.polytopes)
```



We generate a voxelization of this space by creating a dense lattice of points on a simple cubic lattice.

[6]:
```
def make_cubic_grid(box, voxels_per_side):
    v_space = np.linspace(0, 1, voxels_per_side+1)
    v_space = (v_space[:-1] + v_space[1:])/2  # gets centers of the voxels
    return np.array([box.make_absolute([x, y, z])
                     for x in v_space for y in v_space for z in v_space])
```

[7]:
```
voxels_per_side = 30
cubic_grid = make_cubic_grid(box, voxels_per_side)

# Make the spheres overlap just a bit
radii = np.ones(len(cubic_grid)) * 0.8 * np.max(box.L) / voxels_per_side

plot_crystal(box, cubic_grid, radii=radii, backend=backend)
```

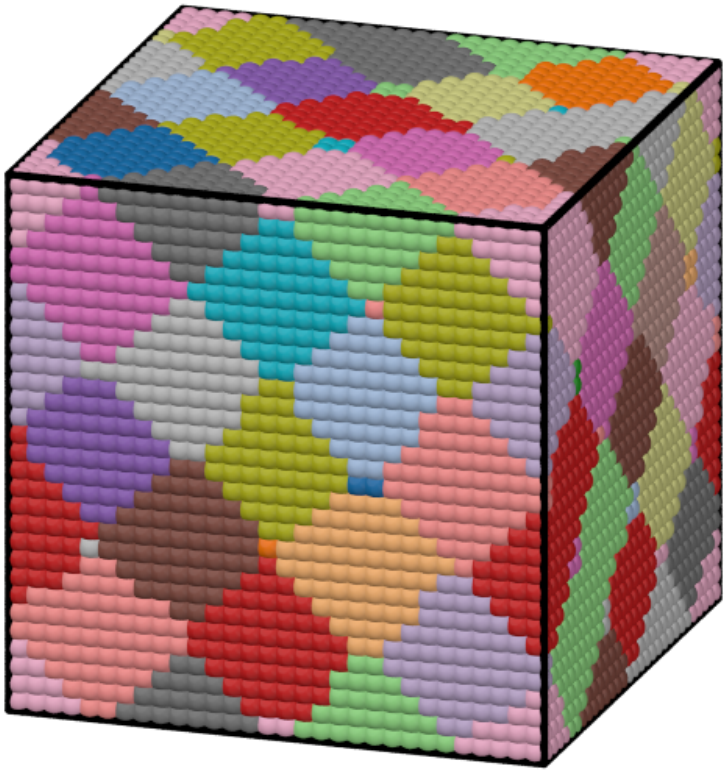


We color the voxels by their first nearest neighbor. This is mathematically equivalent to being inside the corresponding Voronoi cell. Here, we get the neighbor indices (this can be used to separate the Voronoi cells into voxels).

[8]:
```
aq = freud.AABBQuery(box, positions)
voxel_neighbors = -np.ones(len(cubic_grid), dtype=np.int)
for i, j, distance in aq.query(cubic_grid, {'num_neighbors': 1}):
    voxel_neighbors[i] = j
```

Next, we use these indices to color and draw the voxelization.

[9]:
```
voxel_colors = np.array([colors[i] for i in voxel_neighbors])
plot_crystal(box, cubic_grid, colors=voxel_colors,
             radii=radii, backend=backend)
```



7.5.4 Benchmarks

Performance is a central consideration for **freud**. Below are some benchmarks comparing **freud** to other tools offering similar analysis methods.

Benchmarking Neighbor Finding against scipy

The neighbor finding algorithms in freud are highly efficient and rely on parallelized C++ code. Below, we show a benchmark of freud's `AABBQuery` algorithm against the `scipy.spatial.cKDTree`. This benchmark was run on an Intel(R) Xeon(R) i3-8100B CPU @ 3.60GHz.

[1]:
```
import freud
import scipy.spatial
import numpy as np
import matplotlib.pyplot as plt
import timeit
from tqdm.notebook import tqdm
```

[2]:
```
def make_scaled_system(N, Nneigh=12):
    L = (4 / 3 * np.pi * N / Nneigh)**(1/3)
    return freud.data.make_random_system(L, N)
```

(continues on next page)

(continued from previous page)

```
box, points = make_scaled_system(1000)
```

Timing Functions

[3]:
```
def time_statement(stmt, repeat=5, number=100, **kwargs):
    timer = timeit.Timer(stmt=stmt, globals=kwargs)
    times = timer.repeat(repeat, number)
    return np.mean(times), np.std(times)
```

[4]:
```
def time_scipy_cKDTree(box, points):
    shifted_points = points + np.asarray(box.L)/2
    # SciPy only supports cubic boxes
    assert box.Lx == box.Ly == box.Lz
    assert box.xy == box.xz == box.yz == 0
    return time_statement("kdtree = scipy.spatial.cKDTree(points, boxsize=L);"
                          "kdtree.query_ball_tree(kdtree, r=rcut)",
                          scipy=scipy, points=shifted_points, L=box.Lx, rcut=1.0)
```

[5]:
```
def time_freud_AABBQuery(box, points):
    return time_statement("aq = freud.locality.AABBQuery(box, points);"
                          "aq.query(points, {'r_max': r_max, 'exclude_ii': False}).
↪toNeighborList()",
                          freud=freud, box=box, points=points, r_max=1.0)
```

[6]:
```
# Test timing functions
kd_t = time_scipy_cKDTree(box, points)
print(kd_t)
abq_t = time_freud_AABBQuery(box, points)
print(abq_t)
```

```
(0.6436181232333184, 0.008598492136056879)
(0.09153120275586843, 0.00780408130095089)
```

Perform Measurements

[7]:
```
def measure_runtime_scaling_N(Ns, r_max=1.0):
    result_times = []
    for N in tqdm(Ns):
        box, points = make_scaled_system(N)
        result_times.append((
            time_scipy_cKDTree(box, points),
            time_freud_AABBQuery(box, points)))
    return np.asarray(result_times)
```

[8]:
```
def plot_result_times(result_times, Ns):
    fig, ax = plt.subplots(figsize=(6, 4), dpi=200)
    ax.plot(Ns, result_times[:, 0, 0]/100, 'o',
            linestyle='-', markersize=5,
            label="scipy v{} cKDTree".format(scipy.__version__))
```

(continues on next page)

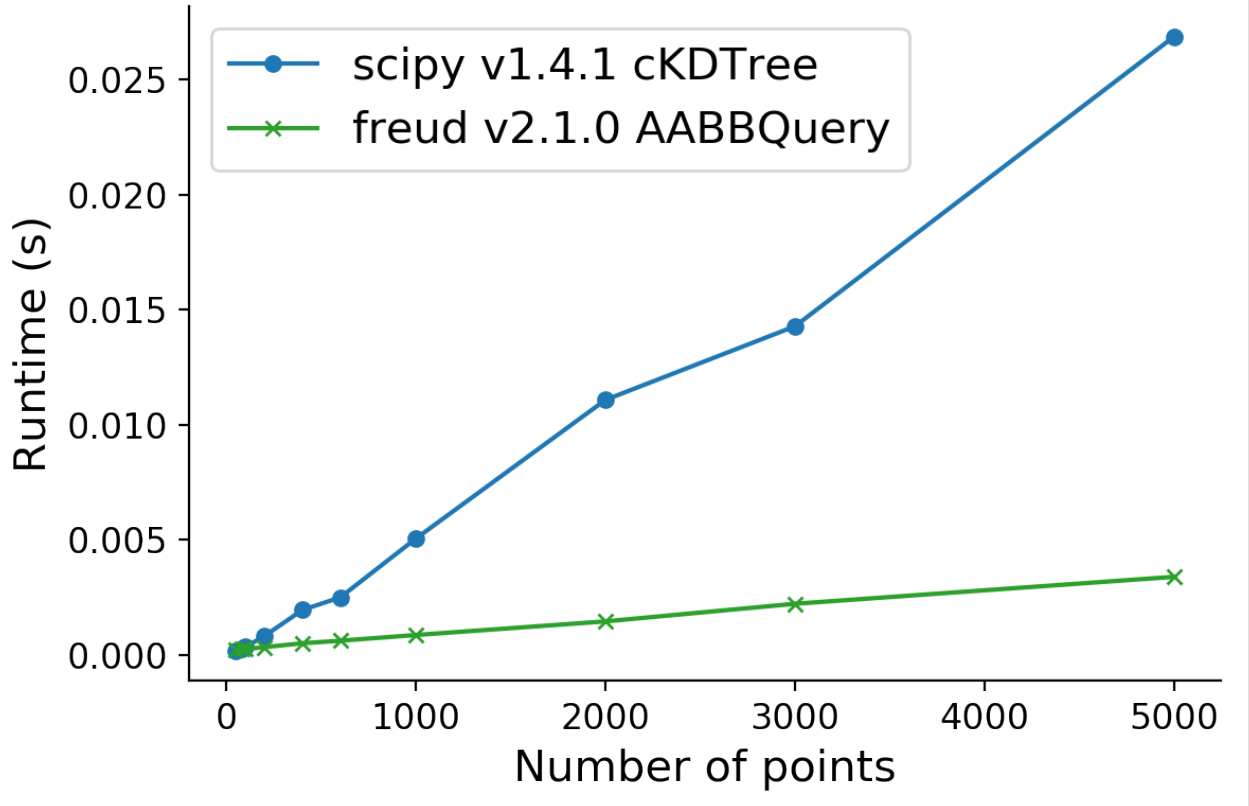
(continued from previous page)

```
    ax.plot(Ns, result_times[:, 1, 0]/100, 'x',
            linestyle='-', markersize=5, c='#2ca02c',
            label="freud v{} AABBQuery".format(freud.__version__))
    ax.set_xlabel(r'Number of points', fontsize=15)
    ax.set_ylabel(r'Runtime (s)', fontsize=15)
    ax.legend(fontsize=15)

    ax.spines["top"].set_visible(False)
    ax.spines["right"].set_visible(False)
    ax.tick_params(axis='both', which='both', labelsize=12)
    fig.tight_layout()

    return fig, ax
```

[9]:
```
# Use geometrically-spaced values of N, rounded to one significant figure
Ns = list(sorted(set(map(
    lambda x: int(round(x, -int(np.floor(np.log10(np.abs(x)))))),
    np.exp(np.linspace(np.log(50), np.log(5000), 10))))))
```

[10]:
```
result_times = measure_runtime_scaling_N(Ns)
fig, ax = plot_result_times(result_times, Ns)
```

```
HBox(children=(FloatProgress(value=0.0, max=10.0), HTML(value='')))
```



Benchmarking RDF against MDAnalysis

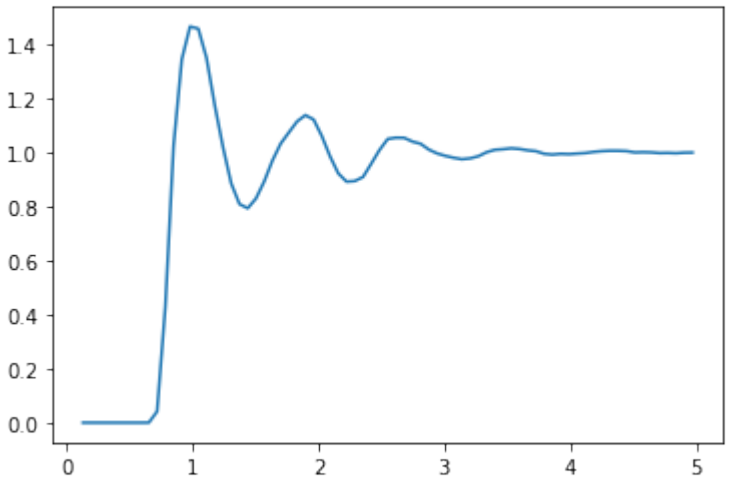
The algorithms in freud are highly efficient and rely on parallelized C++ code. Below, we show a benchmark of `freud.density.RDF` against `MDAnalysis.analysis.rdf`. This benchmark was run on an Intel(R) Core(TM) i3-8100B CPU @ 3.60GHz.

[1]:
```
import freud
import gsd
import MDAnalysis
import MDAnalysis.analysis.rdf
import multiprocessing as mp
import numpy as np
import matplotlib.pyplot as plt
import timeit
from tqdm import tqdm
```

[2]:
```
trajectory_filename = 'data/rdf_benchmark.gsd'
r_max = 5
r_min = 0.1
nbins = 75
```

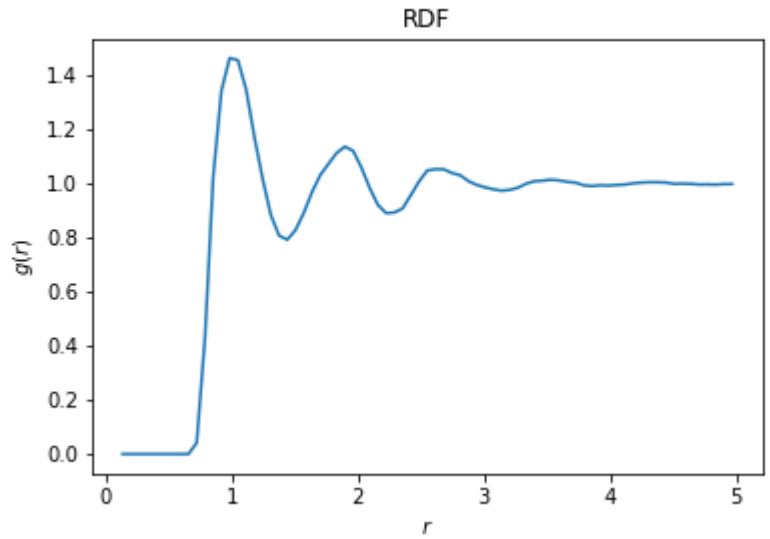
[3]:
```
trajectory = MDAnalysis.coordinates.GSD.GSDReader(trajectory_filename)
topology = MDAnalysis.core.topology.Topology(n_atoms=trajectory[0].n_atoms)
u = MDAnalysis.as_Universe(topology, trajectory_filename)

rdf = MDAnalysis.analysis.rdf.InterRDF(g1=u.atoms, g2=u.atoms,
                                       nbins=nbins,
                                       range=(r_min, r_max)).run()
```

[4]:
```
plt.plot(rdf.bins, rdf.rdf)
plt.show()
```



[5]:
```
freud_rdf = freud.density.RDF(bins=nbins, r_max=r_max, r_min=r_min)
for frame in trajectory:
    freud_rdf.compute(system=frame, reset=False)
freud_rdf
```

[5]:



Timing Functions

[6]:
```
def time_statement(stmt, repeat=3, number=1, **kwargs):
    timer = timeit.Timer(stmt=stmt, globals=kwargs)
    times = timer.repeat(repeat, number)
    return np.mean(times), np.std(times)
```

[7]:
```
def time_mdanalysis_rdf(trajectory_filename, r_max, r_min, nbins):
    trajectory = MDAnalysis.coordinates.GSD.GSDReader(trajectory_filename)
    frame = trajectory[0]
    topology = MDAnalysis.core.topology.Topology(n_atoms=frame.n_atoms)
    u = MDAnalysis.as_Universe(topology, trajectory_filename)
    code = """rdf = MDAnalysis.analysis.rdf.InterRDF(g1=u.atoms, g2=u.atoms,␣
↪nbins=nbins, range=(r_min, r_max)).run()"""
    return time_statement(code, MDAnalysis=MDAnalysis, u=u, r_max=r_max, r_min=r_min,␣
↪nbins=nbins)
```

[8]:
```
def time_freud_rdf(trajectory_filename, r_max, r_min, nbins):
    trajectory = MDAnalysis.coordinates.GSD.GSDReader(trajectory_filename)
    code = """
rdf = freud.density.RDF(bins=nbins, r_max=r_max, r_min=r_min)
for frame in trajectory:
    rdf.compute(system=frame, reset=False)"""
    return time_statement(code, freud=freud, trajectory=trajectory, r_max=r_max, r_
↪min=r_min, nbins=nbins)
```

[9]:
```
# Test timing functions
params = dict(
    trajectory_filename=trajectory_filename,
    r_max=r_max,
    r_min=r_min,
    nbins=nbins)
```

(continues on next page)

(continued from previous page)

```
def system_size(trajectory_filename, **kwargs):
    with gsd.hoomd.open(params['trajectory_filename'], 'rb') as trajectory:
        return {'frames': len(trajectory),
                'particles': len(trajectory[0].particles.position)}

print(system_size(**params))
mdanalysis_rdf_runtime = time_mdanalysis_rdf(**params)
print('MDAnalysis:', mdanalysis_rdf_runtime)
freud_rdf_runtime = time_freud_rdf(**params)
print('freud:', freud_rdf_runtime)
```

```
{'frames': 5, 'particles': 15625}
MDAnalysis: (18.00504128, 0.054983033593944214)
freud: (2.8556541516666747, 0.05481114115556424)
```

Perform Measurements

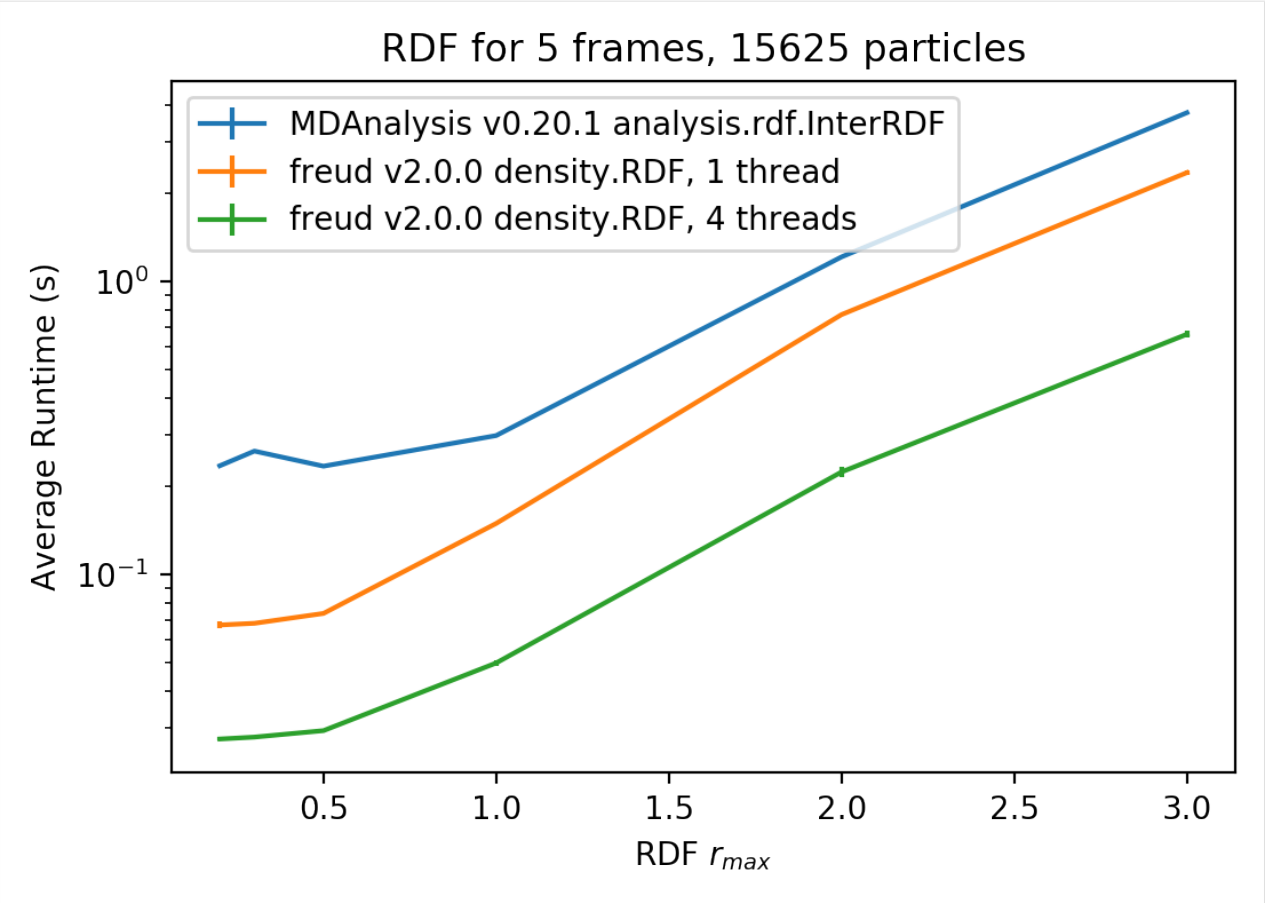
[10]:
```
def measure_runtime_scaling_r_max(r_maxes, **params):
    result_times = []
    for r_max in tqdm(r_maxes):
        params.update(dict(r_max=r_max))
        freud.parallel.set_num_threads(1)
        freud_single = time_freud_rdf(**params)
        freud.parallel.set_num_threads(0)
        result_times.append((time_mdanalysis_rdf(**params), freud_single, time_freud_
↪rdf(**params)))
    return np.asarray(result_times)
```

[11]:
```
def plot_result_times(result_times, r_maxes, frames, particles):
    plt.figure(figsize=(6, 4), dpi=200)
    plt.errorbar(r_maxes, result_times[:, 0, 0], result_times[:, 0, 1],
                 label="MDAnalysis v{} analysis.rdf.InterRDF".format(MDAnalysis.__
↪version__))
    plt.errorbar(r_maxes, result_times[:, 1, 0], result_times[:, 1, 1],
                 label="freud v{} density.RDF, 1 thread".format(freud.__version__))
    plt.errorbar(r_maxes, result_times[:, 2, 0], result_times[:, 2, 1],
                 label="freud v{} density.RDF, {} threads".format(freud.__version__,␣
↪mp.cpu_count()))
    plt.title(r'RDF for {} frames, {} particles'.format(frames, particles))
    plt.xlabel(r'RDF $r_{{max}}$')
    plt.ylabel(r'Average Runtime (s)')
    plt.yscale('log')
    plt.legend()
    plt.show()
```

[12]:
```
r_maxes = [0.2, 0.3, 0.5, 1, 2, 3]
```

[13]:
```
result_times = measure_runtime_scaling_r_max(r_maxes, **params)
plot_result_times(result_times, r_maxes, **system_size(params['trajectory_filename']))
```

```
100%|| 6/6 [00:31<00:00,  5.28s/it]
```



[14]:
```
print('Speedup, parallel freud / serial freud: {:.3f}x'.format(np.average(result_
↪times[:, 1, 0] / result_times[:, 2, 0])))
print('Speedup, parallel freud / MDAnalysis: {:.3f}x'.format(np.average(result_times[:
↪, 0, 0] / result_times[:, 2, 0])))
print('Speedup, serial freud / MDAnalysis: {:.3f}x'.format(np.average(result_times[:,␣
↪0, 0] / result_times[:, 1, 0])))
```

```
Speedup, parallel freud / serial freud: 2.900x
Speedup, parallel freud / MDAnalysis: 7.182x
Speedup, serial freud / MDAnalysis: 2.619x
```

7.6 Query API

This page provides a thorough review of how neighbor finding is structured in **freud**. It assumes knowledge at the level of the *Finding Neighbors* level of the tutorial; if you're not familiar with using the `query` method with query arguments to find neighbors of points, please familiarize yourself with that section of the tutorial.

The central interface for neighbor finding is the `freud.locality.NeighborQuery` family of classes, which provide methods for dynamically finding neighbors given a `freud.box.Box`. The `freud.locality.NeighborQuery` class defines an abstract interface for neighbor finding that is implemented by its subclasses, namely the `freud.locality.LinkCell` and `freud.locality.AABBQuery` classes. These classes represent specific data structures used to accelerate neighbor finding. These two different methods have different performance characteristics, but in most cases `freud.locality.AABBQuery` performs at least as well as, if not

better than, `freud.locality.LinkCell` and is entirely parameter free, so it is the default method of choice used internally in **freud**'s `PairCompute` classes.

In general, these data structures operate by constructing them using one set of points, after which they can be queried to efficiently find the neighbors of arbitrary other points using `freud.locality.NeighborQuery.query()`.

7.6.1 Query Arguments

The table below describes the set of valid query arguments.

Query Argument	Definition	Data type	Legal Values	Valid for
mode	The type of query to perform (distance cutoff or number of neighbors)	str	'none', 'ball', 'nearest'	`freud.locality.AABBQuery`, `freud.locality.LinkCell`
r_max	Maximum distance to find neighbors	float	r_max > 0	`freud.locality.AABBQuery`, `freud.locality.LinkCell`
r_min	Minimum distance to find neighbors	float	0 <= r_min < r_max	`freud.locality.AABBQuery`, `freud.locality.LinkCell`
num_neighbors	Number of neighbors	int	num_neighbors > 0	`freud.locality.AABBQuery`, `freud.locality.LinkCell`
exclude_ii	Whether or not to include neighbors with the same index in the array	bool	True/False	`freud.locality.AABBQuery`, `freud.locality.LinkCell`
r_guess	Initial search distance for sequence of ball queries	float	r_guess > 0	`freud.locality.AABBQuery`
scale	Scale factor for r_guess when not enough neighbors are found	float	scale > 1	`freud.locality.AABBQuery`

7.6.2 Query Modes

Ball Query (Distance Cutoff)

A ball query finds all particles within a specified radial distance of the provided query points. This query is executed when `mode='ball'`. As described in the table above, this mode can be coupled with filters for a minimum distance (`r_min`) and/or self-exclusion (`exclude_ii`).

Nearest Neighbors Query (Fixed Number of Neighbors)

A nearest neighbor query (sometimes called k-nearest neighbors) finds a desired number of neighbor points for each query point, ordered by distance to the query point. This query is executed when `mode='nearest'`. As described in the table above, this mode can be coupled with filters for a maximum distance (`r_max`), minimum distance (`r_min`), and/or self-exclusion (`exclude_ii`).

Mode Deduction

The `mode` query argument specifies the type of query that is being performed, and it therefore governs how other arguments are interpreted. In most cases, however, the query mode can be deduced from the set of query arguments. Specifically, any query with the `num_neighbors` key set is assumed to be a query with `mode='nearest'`. One of `num_neighbors` or `r_max` must always be specified to form a valid set of query arguments. Specifying the `mode` key explicitly will ensure that querying behavior is consistent if additional query modes are added to **freud**.

7.6.3 Query Results

Although they don't typically need to be operated on directly, it can be useful to know a little about the objects returned by queries. The `freud.locality.NeighborQueryResult` stores the `query_points` passed to a `query` and returns neighbors for them one at a time (like any Python `iterator`). The primary goal of the result class is to support easy iteration and conversion to more persistent formats. Since it is an iterator, you can use any typical Python approach to consuming it, including passing it to `list` to build a list of the neighbors. For a more **freud**-friendly approach, you can use the `toNeighborList` method to convert the object into a **freud** `freud.locality.NeighborList`. Under the hood, the underlying C++ classes loop through candidate points and identifying neighbors for each `query_point`; this is the same process that occurs when `Compute classes` employ `NeighborQuery` objects for finding neighbors on-the-fly, but in that case it all happens on the C++ side.

7.6.4 Custom NeighborLists

Thus far, we've mostly discussed `NeighborLists` as a way to persist neighbor information beyond a single query. In *Using freud Efficiently*, more guidance is provided on how you can use these objects to speed up certain uses of **freud**. However, these objects are also extremely useful because they provide a *completely customizable* way to specify neighbors to **freud**. Of particular note here is the `freud.locality.NeighborList.from_arrays()` factory function that allows you to make `freud.locality.NeighborList` objects by directly specifying the `(i, j)` pairs that should be in the list. This kind of explicit construction of the list enables custom analyses that would otherwise be impossible. For example, consider a molecular dynamics simulation in which particles only interact via extremely short-ranged patches on their surface, and that particles should only be considered bonded if their patches are actually interacting, irrespective of how close together the particles themselves are. This type of neighbor interaction cannot be captured by any normal querying mode, but could be constructed by the user and then fed to **freud** for downstream analysis.

7.6.5 Nearest Neighbor Asymmetry

There is one important but easily overlooked detail associated with using query arguments with mode `'nearest'`. Consider a simple example of three points on the x-axis located at -1, 0, and 2 (and assume the box is of dimensions $(100, 100, 100)$, i.e. sufficiently large that periodicity plays no role):

```
box = [100, 100, 100]
points = [[-1, 0, 0], [0, 0, 0], [2, 0, 0]]
query_args = dict(mode='nearest', num_neighbors=1, exclude_ii=True)
list(freud.locality.AABBQuery(box, points).query(points, query_args))
# Output: [(0, 1, 1), (1, 0, 1), (2, 1, 2)]
```

Evidently, the calculation is not symmetric. This feature of nearest neighbor queries can have unexpected side effects if a `PairCompute` is performed using distinct `points` and `query_points` and the two are interchanged. In such cases, users should always keep in mind that **freud** promises that every `query_point` will end up with `num_neighbors` points (assuming no hard cutoff `r_max` is imposed and enough points are present in the system). However, it is possible (and indeed likely) that any given `point` will have more or fewer than that many neighbors.

This distinction can be particularly tricky for calculations that depend on vector directionality: **freud** imposes the convention that bond vectors always point from `query_point` to `point`, so users of calculations like PMFTs where directionality is important should keep this in mind.

7.7 Using freud Efficiently

The **freud** library is designed to be both fast and easy-to-use. In many cases, the library's performance is good enough that users don't need to worry about their usage patterns. However, in highly performance-critical applications (such as real-time visualization or on-the-fly calculations mid-simulation), uses can benefit from knowing the best ways to make use of **freud**. This page provides some guidance on this topic.

7.7.1 Reusing Locality Information

Perhaps the most powerful method users have at their disposal for speeding up calculations is proper reuse of the data structures in `freud.locality`. As one example, consider using **freud** to calculate multiple neighbor-based quantities for the same set of data points. It is important to recognize that internally, each time such a calculation is performed using a `(box, points)` `tuple`, the compute class is internally rebuilding a neighbor-finding accelerator such a `freud.locality.AABBQuery` object and then using it to find neighbors:

```
# Behind the scenes, freud is essentially running
# freud.locality.AABBQuery(box, points).query(points, dict(r_max=5, exclude_ii=True))
# and feeding the result to the RDF calculation.
rdf = freud.density.RDF(bins=50, r_max=5)
rdf.compute(system=(box, points))
```

If users anticipate performing many such calculations on the same system of points, they can amortize the cost of rebuilding the `AABBQuery` object by constructing it once and then passing it into multiple computations:

```
# Now, let's instead reuse the object for a pair of calculations:
nq = freud.locality.AABBQuery(box=box, points=points)
rdf = freud.density.RDF(bins=50, r_max=5)
rdf.compute(system=nq)

r_max = 4
orientations = np.array([[1, 0, 0, 0]] * num_points)
pmft = freud.pmft.PMFTXYZ(r_max, r_max, r_max, bins=100)
pmft.compute(system=nq, orientations=orientations)
```

This reuse can significantly improve performance in e.g. visualization contexts where users may wish to calculate a `bond order diagram` and an `RDF` at each frame, perhaps for integration with a visualization toolkit like OVITO.

A slightly different use-case would be the calculation of multiple quantities based on *exactly the same set of neighbors*. If the user in fact expects to perform computations with the exact same pairs of neighbors (for example, to compute `freud.order.Steinhardt` for multiple l values), then the user can further speed up the calculation by precomputing the entire `freud.NeighborList` and storing it for future use.

```
r_max = 3
nq = freud.locality.AABBQuery(box=box, points=points)
nlist = nq.query(points, dict(r_max=r_max))
q6_arrays = []
for l in range(3, 6):
    ql = freud.order.Steinhardt(l=l)
    q6_arrays.append(ql.compute((box, points), neighbors=nlist).particle_order)
```

Notably, if the user calls a compute method with `compute(system=(box, points))`, unlike in the examples above **freud will not construct** a `freud.locality.NeighborQuery` internally because the full set of neighbors is completely specified by the `NeighborList`. In all these cases, **freud** does the minimal work possible to find neighbors, so judicious use of these data structures can substantially accelerate your code.

7.7.2 Proper Data Inputs

Minor speedups may also be gained from passing properly structured data to **freud**. The package was originally designed for analyzing particle simulation trajectories, which are typically stored in single-precision binary formats. As a result, the **freud** library also operates in single precision and therefore converts all inputs to single-precision. However, NumPy will typically work in double precision by default, so depending on how data is streamed to **freud**, the package may be performing numerous data copies in order to ensure that all its data is in single-precision. To avoid this problem, make sure to specify the appropriate data types (numpy.float32) when constructing your NumPy arrays.

7.8 Reading Simulation Data for freud

The **freud** package is designed for maximum flexibility by making minimal assumptions about its data. However, users accustomed to the more restrictive patterns of most other tools may find this flexibility confusing. In particular, knowing how to provide data from specific simulation sources can be a significant source of confusion. This page is intended to describe how various types of data may be converted into a form suitable for **freud**.

To simplify the examples below, we will assume in all cases that the user wishes to compute a `radial distribution function` over all frames in the trajectory and that the following code has already been run:

```
import freud
rdf = freud.density.RDF(bins=50, r_max=5)
```

7.8.1 Native Integrations

The **freud** library offers interoperability with several popular tools for particle simulations, analysis, and visualization. Below is a list of file formats and tools that are directly supported as "system-like" objects (see `freud.locality.NeighborQuery.from_system`). Such system-like objects are data containers that store information about a periodic box and particle positions. Other attributes, such as particle orientations, are not included automatically in the system representation and must be loaded as separate NumPy arrays.

GSD Trajectories

Using the GSD Python API, GSD files can be easily integrated with **freud** as shown in the *Quickstart Guide*. This format is natively supported by HOOMD-blue. Note: the GSD format can also be read by *MDAnalysis* and *garnett*. Here, we provide an example that reads data from a GSD file.

```
import gsd.hoomd
traj = gsd.hoomd.open('trajectory.gsd', 'rb')

for frame in traj:
    rdf.compute(system=frame, reset=False)
```

MDAnalysis Readers

The MDAnalysis package can read many popular trajectory formats, including common output formats from CHARMM, NAMD, LAMMPS, Gromacs, Tinker, AMBER, GAMESS, HOOMD-blue, and more.

DCD files are among the most familiar simulation outputs due to their longevity. Here, we provide an example that reads data from a DCD file.

```
import MDAnalysis
reader = MDAnalysis.coordinates.DCD.DCDReader('trajectory.dcd')

for frame in reader:
    rdf.compute(system=frame, reset=False)
```

MDTraj Readers

The MDTraj package can read many popular trajectory formats, including common output formats from AMBER, MSMBuilder2, Protein Data Bank files, OpenMM, Tinker, Gromacs, LAMMPS, HOOMD-blue, and more.

To use data read with MDTraj in freud, a system-like object must be manually constructed because it does not have a "frame-like" object containing information about the periodic box and particle positions (both quantities are provided as arrays over the whole trajectory). Here, we provide an example of how to construct a system:

```
import mdtraj
traj = mdtraj.load_xtc('output/prd.xtc', top='output/prd.gro')

for system in zip(np.asarray(traj.unitcell_vectors), traj.xyz):
    rdf.compute(system=system, reset=False)
```

garnett Trajectories

The garnett package can read several trajectory formats that have historically been supported by the HOOMD-blue simulation engine, as well as other common types such as DCD and CIF. The **garnett** package will auto-detect supported file formats by the file extension. Here, we provide an example that reads data from a POS file.

```
import garnett

with garnett.read('trajectory.pos') as traj:
    for frame in traj:
        rdf.compute(system=frame, reset=False)
```

OVITO Modifiers

The OVITO Open Visualization Tool supports user-written Python modifiers. The **freud** package can be installed alongside OVITO to enable user-written Python script modifiers that leverage analyses from **freud**. Below is an example modifier that creates a user particle property in the OVITO pipeline for Steinhardt bond order parameters.

```
import freud

def modify(frame, data):
    ql = freud.order.Steinhardt(l=6)
    ql.compute(system=data, neighbors={'num_neighbors': 6})
    data.create_user_particle_property(
```

(continues on next page)

(continued from previous page)

```
        name='ql', data_type=float, data=ql.ql)
    print('Created ql property for {} particles.'.format(data.particles.count))
```

HOOMD-blue Snapshots

HOOMD-blue supports analyzers, callback functions that can perform analysis. Below is an example demonstrating how to use an anlyzer to log the Steinhardt bond order parameter q_6 during the simulation run.

```
import hoomd
from hoomd import md
import freud

hoomd.context.initialize()

# Create a 10x10x10 simple cubic lattice of particles with type name A
system = hoomd.init.create_lattice(
    unitcell=hoomd.lattice.sc(a=2.0, type_name='A'), n=10)

# Specify Lennard-Jones interactions between particle pairs
nl = md.nlist.cell()
lj = md.pair.lj(r_cut=3.0, nlist=nl)
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)

# Integrate at constant temperature
md.integrate.mode_standard(dt=0.005)
hoomd.md.integrate.langevin(group=hoomd.group.all(), kT=1.2, seed=4)

# Create a Steinhardt object to analyze snapshots
ql = freud.order.Steinhardt(l=6)

def compute_q6(timestep):
    snap = system.take_snapshot()
    ql.compute(system=snap, neighbors={'num_neighbors': 6})
    return ql.order

# Register a logger that computes q6 and saves to a file
ql_logger = hoomd.analyze.log(filename='ql.dat', quantities=['q6'], period=100)
ql_logger.register_callback('q6', compute_q6)

# Run for 10,000 time steps
hoomd.run(10e3)
```

7.8.2 Reading Text Files

Typically, it is best to use one of the natively supported data readers described above; however it is sometimes necessary to parse trajectory information directly from a text file. One example of a plain text format is the XYZ file format, which can be generated and used by many tools for particle simulation and analysis, including LAMMPS and VMD. Note that various readers do exist for XYZ files, including MDAnalysis, but in this example we read the file manually to demonstrate how to read these inputs as plain text. Though they are easy to parse, XYZ files usually contain no information about the system box, so this must already be known by the user. Assuming knowledge of the box used in the simulation, a LAMMPS XYZ file could be used as follows:

```
N = int(np.genfromtxt('trajectory.xyz', max_rows=1))
traj = np.genfromtxt(
    'trajectory.xyz', skip_header=2,
    invalid_raise=False)[:, 1:4].reshape(-1, N, 3)
box = freud.box.Box.cube(L=20)

for frame_positions in traj:
    rdf.compute(system=(box, frame_positions), reset=False)
```

The first line is the number of particles, so we read this line and use it to determine how to reshape the contents of the rest of the file into a NumPy array.

7.8.3 Other External Readers

For many trajectory formats, high-quality readers already exist. However sometimes these readers' data structures must be converted into a format understood by **freud**. Below, we show an example that converts the MDAnalysis box dimensions from a matrix into a `freud.box.Box`. Note that *MDAnalysis inputs* are natively supported by **freud** without this extra step. For other formats not supported by a reader listed above, a similar process can be followed.

```
import MDAnalysis
reader = MDAnalysis.coordinates.DCD.DCDReader('trajectory.dcd')

for frame in reader:
    box = freud.box.Box.from_matrix(frame.triclinic_dimensions)
    rdf.compute(system=(box, frame.positions), reset=False)
```

7.9 Box Module

Overview

—

Details

7.10 Cluster Module

Overview

—

Details

7.11 Data Module

Overview

Details

7.12 Density Module

Overview

Details

7.13 Diffraction Module

Overview

Details

7.14 Environment Module

Overview

Details

7.15 Interface Module

Overview

Details

7.16 Locality Module

Overview

Details

7.17 MSD Module

Overview

Details

7.18 Order Module

Overview

Details

7.19 Parallel Module

Overview

Details

7.20 PMFT Module

Overview

Details

7.21 Development Guide

Contributions to **freud** are highly encouraged. The pages below offer information about how the project is structured, the goals of the **freud** library, and how to contribute new modules.

7.21.1 Design Principles

Vision

The **freud** library is designed to be a powerful and flexible library for the analysis of simulation output. To support a variety of analysis routines, **freud** places few restrictions on its components. The primary requirement for an analysis routine in **freud** is that it should be substantially computationally intensive so as to require coding up in C++: **all freud code should be composed of fast C++ routines operating on systems of particles in periodic boxes.** To remain easy-to-use, all C++ modules should be wrapped in Python code so they can be easily accessed from Python scripts or through a Python interpreter.

In order to achieve this goal, **freud** takes the following viewpoints:

- **freud** works directly with *NumPy <http://www.numpy.org/>_* arrays to retain maximum flexibility. Integrations with other tools should be performed via the common data representations of NumPy arrays.
- For ease of maintenance, **freud** uses Git for version control; GitHub for code hosting and issue tracking; and the PEP 8 standard for code, stressing explicitly written code which is easy to read.
- To ensure correctness, **freud** employs unit testing using the Python unittest framework. In addition, **freud** utilizes CircleCI for continuous integration to ensure that all of its code works correctly and that any changes or new features do not break existing functionality.

Language choices

The **freud** library is written in two languages: Python and C++. C++ allows for powerful, fast code execution while Python allows for easy, flexible use. Intel Threading Building Blocks parallelism provides further power to C++ code. The C++ code is wrapped with Cython, allowing for user interaction in Python. NumPy provides the basic data structures in **freud**, which are commonly used in other Python plotting libraries and packages.

Unit Tests

All modules should include a set of unit tests which test the correct behavior of the module. These tests should be simple and short, testing a single function each, and completing as quickly as possible (ideally < 10 sec, but times up to a minute are acceptable if justified).

Benchmarks

Modules can be benchmarked in the following way. The following code is an example benchmark for the `freud.density.RDF` module.

```
import numpy as np
import freud
from benchmark import Benchmark
from benchmarker import run_benchmarks


class BenchmarkDensityRDF(Benchmark):
    def __init__(self, r_max, bins, r_min):
        self.r_max = r_max
        self.bins = bins
        self.r_min = r_min

    def bench_setup(self, N):
        self.box_size = self.r_max*3.1
        np.random.seed(0)
        self.points = np.random.random_sample((N, 3)).astype(np.float32) \
            * self.box_size - self.box_size/2
        self.rdf = freud.density.RDF(self.bins, self.r_max, r_min=self.r_min)
        self.box = freud.box.Box.cube(self.box_size)

    def bench_run(self, N):
        self.rdf.compute((self.box, self.points), reset=False)
        self.rdf.compute((self.box, self.points))


def run():
    Ns = [1000, 10000]
    r_max = 10.0
    bins = 10
    r_min = 0
    number = 100
    name = 'freud.density.RDF'
    classobj = BenchmarkDensityRDF

    return run_benchmarks(name, Ns, number, classobj,
                          r_max=r_max, bins=bins, r_min=r_min)


if __name__ == '__main__':
    run()
```

in a file `benchmark_density_RDF.py` in the `benchmarks` directory. More examples can be found in the `benchmarks` directory. The runtime of `BenchmarkDensityRDF.bench_run` will be timed for `number` of times on the input sizes of `Ns`. Its runtime with respect to the number of threads will also be measured. Benchmarks are run as a part of continuous integration, with performance comparisons between the current commit and the master branch.

Make Execution Explicit

While it is tempting to make your code do things "automatically", such as have a calculate method find all `_calc` methods in a class, call them, and add their returns to a dictionary to return to the user, it is preferred in **freud** to execute code explicitly. This helps avoid issues with debugging and undocumented behavior:

```
# this is bad
class SomeFreudClass(object):
    def __init__(self, **kwargs):
        for key in kwargs.keys:
            setattr(self, key, kwargs[key])

# this is good
class SomeOtherFreudClass(object):
    def __init__(self, x=None, y=None):
        self.x = x
        self.y = y
```

Code Duplication

When possible, code should not be duplicated. However, being explicit is more important. In **freud** this translates to many of the inner loops of functions being very similar:

```
// somewhere deep in function_a
for (int i = 0; i < n; i++)
    {
    vec3[float] pos_i = position[i];
    for (int j = 0; j < n; j++)
        {
        pos_j = = position[j];
        // more calls here
        }
    }

// somewhere deep in function_b
for (int i = 0; i < n; i++)
    {
    vec3[float] pos_i = position[i];
    for (int j = 0; j < n; j++)
        {
        pos_j = = position[j];
        // more calls here
        }
    }
```

While it *might* be possible to figure out a way to create a base C++ class all such classes inherit from, run through positions, call a calculation, and return, this would be rather complicated. Additionally, any changes to the internals of the code, and may result in performance penalties, difficulty in debugging, etc. As before, being explicit is better.

However, if you have a class which has a number of methods, each of which requires the calling of a function, this function should be written as its own method (instead of being copy-pasted into each method) as is typical in object-oriented programming.

Python vs. Cython vs. C++

The **freud** library is meant to leverage the power of C++ code imbued with parallel processing power from TBB with the ease of writing Python code. The bulk of your calculations should take place in C++, as shown in the snippet below:

```
# this is bad
def badHeavyLiftingInPython(positions):
    # check that positions are fine
    for i, pos_i in enumerate(positions):
        for j, pos_j in enumerate(positions):
            if i != j:
                r_ij = pos_j - pos_i
                # ...
                computed_array[i] += some_val
    return computed_array

# this is good
def goodHeavyLiftingInCPlusPlus(positions):
    # check that positions are fine
    cplusplus_heavy_function(computed_array, positions, len(pos))
    return computed_array
```

In the C++ code, implement the heavy lifting function called above from Python:

```
void cplusplus_heavy_function(float* computed_array,
                              float* positions,
                              int n)
    {
    for (int i = 0; i < n; i++)
        {
        for (int j = 0; j < n; j++)
            {
            if (i != j)
                {
                r_ij = pos_j - pos_i;
                // ...
                computed_array[i] += some_val;
                }
            }
        }
    }
```

Some functions may be necessary to write at the Python level due to a Python library not having an equivalent C++ library, complexity of coding, etc. In this case, the code should be written in Cython and a *reasonable* attempt to optimize the code should be made.

7.21.2 Contributing to freud

Code Conventions

Python

Python (and Cython) code in **freud** should follow PEP 8.

During continuous integration (CI), all Python and Cython code in **freud** is tested with flake8 to ensure PEP 8 compliance. Additionally, all CMake code is tested using cmakelang's cmake-format. It is strongly recommended to set up a pre-commit hook to ensure code is compliant before pushing to the repository:

```
pip install -r requirements-precommit.txt
pre-commit install
```

To manually run pre-commit for all the files present in the repository, run the following command:

```
pre-commit run --all-files --show-diff-on-failure
```

Documentation is written in reStructuredText and generated using Sphinx. It should be written according to the Google Python Style Guide. A few specific notes:

- The shapes of NumPy arrays should be documented as part of the type in the following manner:

  ```
  points ((:math:`N_{points}`, 3) :class:`numpy.ndarray`):
  ```

- Optional arguments should be documented as such within the type after the actual type, and the default value should be included within the description:

  ```
  box (:class:`freud.box.Box`, optional): Simulation box (Default value = None).
  ```

C++

C++ code should follow the result of running `clang-format-6.0` with the style specified in the file `.clang-format`. Please refer to Clang Format 6 for details.

When in doubt, run `clang-format -style=file FILE_WITH_YOUR_CODE` in the top directory of the **freud** repository. If installing `clang-format` is not a viable option, the `check-style` step of continuous integration (CI) contains the information on the correctness of the style.

Doxygen docstrings should be used for classes, functions, etc.

Code Organization

The code in **freud** is a mix of Python, Cython, and C++. From a user's perspective, methods in **freud** correspond to `Compute` classes, which are contained in Python modules that group methods by topic. To keep modules well-organized, **freud** implements the following structure:

- All C++ code is stored in the `cpp` folder at the root of the repository, with subdirectories corresponding to each module (e.g. `cpp/locality`).
- Python code is stored in the `freud` folder at the root of the repository.
- C++ code is exposed to Python using Cython code contained in pxd files with the following convention: `freud/_MODULENAME.pxd` (note the preceding underscore).

- The core Cython code for modules is contained in `freud/MODULENAME.pyx` (no underscore).
- Generated Cython C++ code (e.g. `freud/MODULENAME.cxx`) should not be committed during development. These files are generated using Cython when building from source, and are unnecessary when installing compiled binaries.
- If a Cython module contains code that must be imported into other Cython modules (such as the `freud.box.Box` class), the `pyx` file must be accompanied by a `pxd` file with the same name: `freud/MODULENAME.pxd` (distinguished from `pxd` files used to expose C++ code by the lack of a preceding underscore). For more information on how `pxd` files work, see the Cython documentation.
- All tests in **freud** are based on the Python standard `unittest` library and are contained in the `tests` folder. Test files are named by the convention `tests/test_MODULENAME_CLASSNAME.py`.
- Benchmarks for **freud** are contained in the `benchmarks` directory and are named analogously to tests: `benchmarks/benchmark_MODULENAME_CLASSNAME.py`.

Benchmarks

Benchmarking in **freud** is performed by running the `benchmarks/benchmarker.py` script. This script finds all benchmarks (using the above naming convention) and attempts to run them. Each benchmark is defined by extending the `Benchmark` class defined in `benchmarks/benchmark.py`, which provides the standard benchmarking utilities used in **freud**. Subclasses just need to define a few methods to parameterize the benchmark, construct the **freud** object being benchmarked, and then call the relevant compute method. Rather than describing this process in detail, we consider the benchmark for the `freud.density.RDF` module as an example.

```
import numpy as np
import freud
from benchmark import Benchmark
from benchmarker import run_benchmarks


class BenchmarkDensityRDF(Benchmark):
    def __init__(self, r_max, bins, r_min):
        self.r_max = r_max
        self.bins = bins
        self.r_min = r_min

    def bench_setup(self, N):
        self.box_size = self.r_max*3.1
        np.random.seed(0)
        self.points = np.random.random_sample((N, 3)).astype(np.float32) \
            * self.box_size - self.box_size/2
        self.rdf = freud.density.RDF(self.bins, self.r_max, r_min=self.r_min)
        self.box = freud.box.Box.cube(self.box_size)

    def bench_run(self, N):
        self.rdf.compute((self.box, self.points), reset=False)
        self.rdf.compute((self.box, self.points))


def run():
    Ns = [1000, 10000]
    r_max = 10.0
    bins = 10
    r_min = 0
    number = 100
```

(continues on next page)

(continued from previous page)

```
    name = 'freud.density.RDF'
    classobj = BenchmarkDensityRDF

    return run_benchmarks(name, Ns, number, classobj,
                          r_max=r_max, bins=bins, r_min=r_min)


if __name__ == '__main__':
    run()
```

The `__init__` method defines basic parameters of the run, the `bench_setup` method is called to build up the `RDF` object, and the `bench_run` is used to time and call `compute`. More examples can be found in the `benchmarks` directory. The runtime of `BenchmarkDensityRDF.bench_run` will be timed for `number` of times on the input sizes of `Ns`. Its runtime with respect to the number of threads will also be measured. Benchmarks are run as a part of continuous integration, with performance comparisons between the current commit and the master branch.

Steps for Adding New Code

Once you've determined to add new code to **freud**, the first step is to create a new branch off of `master`. The process of adding code differs based on whether or not you are editing an existing module in **freud**. Adding new methods to an existing module in **freud** requires creating the new C++ files in the `cpp` directory, modifying the corresponding `_MODULENAME.pxd` file in the `freud` directory, and creating a wrapper class in `freud/MODULENAME.pyx`. If the new methods belong in a new module, you must create the corresponding `cpp` directory and the `pxd` and `pyx` files accordingly.

In order for code to compile, it must be added to the relevant `CMakeLists.txt` file. New C++ files for existing modules must be added to the corresponding `cpp/MODULENAME/CMakeLists.txt` file. For new modules, a `cpp/NEWMODULENAME/CMakeLists.txt` file must be created, and in addition the new module must be added to the `cpp/CMakeLists.txt` file in the form of both an `add_subdirectory` command and addition to the `libfreud` library in the form of an additional source in the `add_library` command. Similarly, new Cython modules must be added to the appropriate list in the `freud/CMakeLists.txt` file depending on whether or not there is C++ code associated with the module. Finally, you will need to import the new module in `freud/__init__.py` by adding `from . import MODULENAME` so that your module is usable as `freud.MODULENAME`.

Once the code is added, appropriate tests should be added to the `tests` folder. Test files are named by the convention `tests/test_MODULENAME_CLASSNAME.py`. The final step is updating documentation, which is contained in `rst` files named with the convention `doc/source/modules/MODULENAME.rst`. If you have added a class to an existing module, all you have to do is add that same class to the `autosummary` section of the corresponding `rst` file. If you have created a new module, you will have to create the corresponding `rst` file with the summary section listing classes and functions in the module followed by a more detailed description of all classes. All classes and functions should be documented inline in the code, which allows automatic generation of the detailed section using the `automodule` directive (see any of the module `rst` files for an example). Finally, the new file needs to be added to `doc/source/index.rst` in the `API` section.

7.21.3 Special Topics

Some of the most central components have a high level of abstraction. This abstraction has multiple advantages: it dramatically simplifies the process of implementing new code, it reduces duplication throughout the code base, and ensures that bug fixes and optimization can occur along a single path for the entire module. However, this abstraction comes at the cost of significant complexity. This documentation should help orient developers familiarizing themselves with these topics by providing high-level overviews of how these parts of the code are structured and how the pieces fit together.

Memory Management

Memory handling in **freud** is a somewhat intricate topic. Most **freud** developers do not need to be aware of such details; however, certain practices must be followed to ensure that the expected behavior is achieved. This page provides an overview of how data should be handled in **freud** and how module developers should use **freud**'s core classes to ensure proper memory management. A thorough description of the process is also provided for developers who need to understand the internal logic for further development.

Note: This page specifically deals with modules primarily written in C++. These concepts do not apply to pure Python/Cython modules.

Problem Statement

The standard structure for **freud** modules involves a core implementation in a C++ class wrapped in a Cython class that owns a pointer to the C++ object. Python `compute` methods call through to C++ `compute` methods, which perform the calculation and populate class member arrays that are then accessed via properties of the owning Cython class. These classes are designed to be reusable, i.e. `compute` may be called many times on the same object with different data, and the accessed properties will return the most current data. Users have a reasonable expectation that if the accessed property is saved to another variable it will remain unchanged by future calls to `compute` or if the originating C++ object is destructed, but a naive implementation that ensures this invariant would involve reallocating memory on every call to compute, an unnecessarily expensive operation. Ultimately, what we want is a method that performs the minimal number of memory allocations while allowing users to operate transparently on outputs without worrying about whether the data will be invalidated by future operations.

ManagedArray

The **freud** `ManagedArray` template class provides a solution to this problem for arbitrary types of numerical data. Proper usage of the class can be summarized by the following steps:

1. Declaring `ManagedArray` class members in C++.
2. Calling the `prepare` method in every `compute`.
3. Making the array accessible via a getter method that **returns a const reference**.
4. Calling `make_managed_numpy_array` in Cython and returning the output as a property.

Plenty of examples of following this pattern can be found throughout the codebase, but for clarity we provide a complete description with examples below. If you are interested in more details on the internals of `ManagedArray` and how it actually works, you can skip to *Explaining ManagedArrays*.

Using ManagedArrays

We'll use `freud.cluster.Cluster` to illustrate how the four steps above may be implemented. This class takes in a set of points and assigns each of them to clusters, which are store in the C++ array `m_cluster_idx`.

Step 1 is simple: we note that `m_cluster_idx` is a member variable of type `ManagedArray<unsigned int>`. For step 2, we look at the first few lines of `Cluster::compute`, where we see a call to `m_cluster_idx.prepare`. This method encapsulates the core logic of `ManagedArray`, namely the intelligent reallocation of memory whenever other code is still accessing it. This means that, if a user saves the corresponding Python property `freud.cluster.Cluster.cluster_idx` to a local variable in a script and then calls `freud.cluster.Cluster.compute`, the saved variable will still reference the original data, and the new data may be accessed again using `freud.cluster.Cluster.cluster_idx`.

Step 3 for the cluster indices is accomplished in the following code block:

```
//! Get a reference to the cluster ids.
const util::ManagedArray<unsigned int> &getClusterIdx() const
{
    return m_cluster_idx;
}
```

The return type of this method is crucial: all such methods must return const references to the members.

The final step is accomplished on the Cython side. Here is how the cluster indices are exposed in `freud.cluster.Cluster`:

```
@_Compute._computed_property
def cluster_idx(self):
    """:math:`N_{points}` :class:`numpy.ndarray`: The cluster index for
    each point."""
    return freud.util.make_managed_numpy_array(
        &self.thisptr.getClusterIdx(),
        freud.util.arr_type_t.UNSIGNED_INT)
```

Essentially all the core logic is abstracted away from the user through the `freud.data.make_managed_numpy_array()`, which creates a NumPy array that is a view on an existing `ManagedArray`. This NumPy array will, in effect, take ownership of the data in the event that the user keeps a reference to it and requests a recomputation. Note the signature of this function: the first argument must be **a pointer to the ManagedArray** (which is why we had to return it by reference), and the second argument indicates the type of the data (the possible types can be found in `freud/util.pxd`). There is one other point to note that is not covered by the above example; if the template type of the `ManagedArray` is not a scalar, you also need to provide a third argument indicating the size of this vector. The most common use-case is for methods that return an object of type `ManagedArray<vec3<float>>`: in this case, we would call `make_managed_numpy_array(&GETTER_FUNC, freud.util.arr_type_t.FLOAT, 3)`.

Indexing ManagedArrays

With respect to indexing, the `ManagedArray` class behaves like any standard array-like container and can be accessed using e.g. `m_cluster_idx[index]`. In addition, because many calculations in **freud** output multidimensional information, `ManagedArray` also supports multidimensional indexing using `operator()`. For example, setting the element at second row and third column of a 2D `ManagedArray array` to one can be done using `array(1, 2) = 1` (indices beginning from 0). Therefore, `ManagedArray` objects can be used easily inside the core C++ calculations in **freud**.

Explaining ManagedArrays

We now provide a more detailed accounting of how the `ManagedArray` class actually works. Consider the following block of code:

```
rdf = freud.density.RDF(bins=100, r_max=3)

rdf.compute(system=(box1, points1))
rdf1 = rdf.rdf

rdf2.compute(system=(box2, points2))
rdf2 = rdf.rdf
```

We require that `rdf1` and `rdf2` be distinct arrays that are only equal if the results of computing the RDF are actually equivalent for the two systems, and we want to achieve this with the minimal number of memory allocations. In this case, that means there are two required allocations; returning copies would double that.

To achieve this goal, `ManagedArray` objects store a pointer to a pointer. Multiple `ManagedArray` objects can point to the same data array, and the pointers are all shared pointers to automate deletion of arrays when no references remain. The key using the class properly is the `prepare` method, which checks the reference count to determine whether it's safe to simply zero out the existing memory or if it needs to allocate a new array. In the above example, when `compute` is called a second time the `rdf1` object in Python still refers to the computed data, so `prepare` will detect that there are multiple (two) shared pointers pointing to the data and choose to reallocate the class's `ManagedArray` storing the RDF. By calling `prepare` at the top of every `compute` method, developers ensure that the array used for the rest of the method has been properly zeroed out, and they do not need to worry about whether reallocation is needed (including cases where array sizes change).

To ensure that all references to data are properly handled, some additional logic is required on the Python side as well. The Cython `make_managed_numpy_array` instantiates a `_ManagedArrayContainer` class, which is essentially just a container for a `ManagedArray` that points to the same data as the `ManagedArray` provided as an argument to the function. This link is what increments the underlying shared pointer reference counter. The `make_managed_numpy_array` uses the fact that a `_ManagedArrayContainer` can be transparently converted to a NumPy array that points to the container; as a result, no data copies are made, but all NumPy arrays effectively share ownership of the data along with the originating C++ class. If any such arrays remain in scope for future calls to `compute`, `prepare` will recognize this and reallocate memory as needed.

Neighbor Finding

Neighbor finding is central to many methods in **freud**. The purpose of this page is to introduce the various C++ classes and functions involved in the neighbor finding process. This page focuses on aspects of the neighbor finding utilities in **freud** that are important for developers. As such, it assumes knowledge on the level of *Finding Neighbors* and *Pair Computations*, so please familiarize yourself with the contents of those pages before proceeding.

There are two primary use-cases for the neighbor finding code in **freud**. One is to directly expose this functionality to the user, via the `NeighborQuery` abstract class and its subclasses. The second it to enable looping over nearest neighbors (as defined by arbitrary query arguments or a precomputed `NeighborList`) inside of compute methods defined in C++. To support both of these use-cases, **freud** defines how to find neighbors inside iterator classes, which can be naturally looped over in either case. In this page, we first discuss these iterators and how they are structured with respect to the `locality::NeighborQuery` C++ class. We then discuss the utility functions built around this class to enable easier C++ computations, at which point we also discuss how `NeighborList` objects fit into this framework.

Per-Point Iterators and the NeighborQuery class

The lowest-level unit of the neighbor finding infrastructure in **freud** is the `locality::NeighborPerPointIterator`. This class defines an abstract interface for all neighbor iteration in **freud**, an interface essentially composed of the `next` and `end` methods. Given an instance of this class, these two methods provide the means for client code to get the next neighbor in the iteration until there are no further neighbors. Calls to `next` produces instances of `locality::NeighborBond`, a simple data class that contains the core information defining a bond (a pair of points, a distance, and any useful ancillary information).

The rationale for making per-point iteration the central element is twofold. The first is conceptual: all logic for finding neighbors is naturally reducible to a set of conditions on the neighbors of each query point. The second is more practical: since finding neighbors for each point must be sequential in many cases (such as nearest neighbor queries), per-point iteration is the smallest logical unit that can be parallelized.

Instances of `locality::NeighborPerPointIterator` should not be constructed directly; instead, they should be constructed via the `locality::NeighborQuery::querySingle` method. The `locality::NeighborQuery` class is an abstract data type that defines an interface for any method implemented for finding neighbors. All subclasses must implement the `querySingle` method, which should return a subclass of `locality::NeighborPerPointIterator`. For instance, the `locality::AABBQuery` class implements `querySingle`, which returns an instance of the `locality::AABBIterator` subclass. In general, different `NeighborQuery` subclasses will need to implement separate per-point iterators for each query mode; for `locality::AABBQuery`, these are the `locality::AABBQueryIterator` and the `locality::AABBQueryBallIterator`, which encode the logic for nearest neighbor and ball queries, respectively.

Although the `querySingle` method is what subclasses should implement, the primary interface to `NeighborQuery` subclasses is the `query` method, which accepts an arbitrary set of query points and query arguments and simply generates a `locality::NeighborQueryIterator` object. The `locality::NeighborQueryIterator` class is an intermediary that allows lazy generation of neighbors. It essentially functions as the container for a set of points, query points, and query arguments; once iteration of this object begins, it produces `NeighborPerPointIterator` objects on demand. This mode of operation also enables the generator approach to looping over neighbors in Python, since iterating in Python corresponds directly to calling `next` on the underlying `NeighborQueryIterator`.

There is one conceptual complexity associated with this class that is important to understand. Since all of the logic for finding neighbors is contained in the per-point iterator classes, the `NeighborQueryIterator` actually retains a reference to the constructing `NeighborQuery` object so that it can call `querySingle` for each point. This bidirectionally linked structure enables encapsulation of the neighbor finding logic while also supporting easily parallelization (via parallel calls to `querySingle`). Additionally, this structure makes it natural to generate `NeighborList` objects.

NeighborLists

The `NeighborList` class represents a static list of neighbor pairs. The results of any query can be converted to a `NeighborList` by calling the `toNeighborList` method of the `NeighborQueryIterator`, another reason why the iterator class logic is separated from the `NeighborQuery` object: it allows generation of a `NeighborList` from – and more generally, independent operation on – the result of a query. The `NeighborList` is simply implemented as a collection of raw arrays, one of which holds pairs of neighbors. The others hold any additional information associated with each bond, such as distances or weights.

By definition, the bonds in a `NeighborList` are stored in the form `(query_point, point)` (i.e. this is how the underlying array is indexed) and ordered by `query_point`. This ordering makes the structure amenable to a fast binary search algorithm. Looping over neighbors of a given `query_point` is then simply a matter of finding the first index in the list where that `query_point` appears and then iterating until the `query_point` index in the list no longer matches the one under consideration.

Computing With Neighbors

One of the most common operations in **freud** is performing some computation over all neighbor-bonds in the system. Users have multiple ways of specifying neighbors (using query arguments or by a `NeighborList`), so **freud** provides some utility functions to abstract the process of looping over neighbors. These functions are defined in `locality/NeighborComputeFunctional.h`; the two most important ones are `loopOverNeighbors` and `loopOverNeighborsIterator`. Compute functions that perform neighbor computations typically accept a `NeighborQuery`, a `QueryArgs`, and a `NeighborList` object. These objects can then be passed to either of the utility functions, which loop over the `NeighborList` if it was provided (if no `NeighborList` is provided by the Python user, a `NULL` pointer is passed through), and if not, perform a query on the `NeighborQuery` object using the provided `QueryArgs` to generate the required neighbors. The actual computation should be encapsulated as a lambda function that is passed as an argument to these utilities.

The distinction between the two utility functions lies in the signature of the accepted lambda functions, which enables a slightly different form of computation. The default `loopOverNeighbors` function does exactly what is described above, namely it calls the provided compute function for every single bond. However, some computations require some additional code to be executed for each `query_point`, such as some sort of normalization. To enable this mode of operation, the `loopOverNeighborsIterator` method instead requires a lambda function that accepts two arguments, the `query_point` index and a `NeighborPerPointIterator`. This way, the client code can loop over the neighbors of a given `query_point` and perform the needed computation, then execute additional code (which may optionally depend on the index of the `query_point`, e.g. to update a specific array index).

Default Systems

There is one important implementation detail to note. The user is permitted to simply provide a set of points rather than a `NeighborQuery` object on the Python side (i.e. any valid argument to `from_system()`), but we need a natural way to mirror this in C++, ideally without too many method overloads. To implement this, we provide the `RawPoints` C++ class and its Python `_RawPoints` mirror, which is essentially a plain container for a box and a set of query points. This object inherits from `NeighborQuery`, allowing it to be passed directly into the C++ compute methods.

However, neighbor computations still need to know how to find neighbors. In this case, they must construct a `NeighborQuery` object capable of neighbor finding and then use the provided query arguments to find neighbors. To enable this calculation, the `RawPoints` class implements a query method that simply constructs an `AABBQuery` internally and queries it for neighbors.

Default NeighborLists

Some compute methods are actually computations that produce quantities per bond. One example is the `SolidLiquid` order parameter, which computes an order parameter value for each bond. The `NeighborComputeFunctional.h` file implements a `makeDefaultNList` function that supports this calculation by creating a `NeighborList` object from whatever inputs are provided on demand.

Histograms

Histograms are a common type of calculation implemented in **freud** because custom histograms are hard to compute efficiently in pure Python. The C++ `Histogram` class support weighted N-dimensional histograms with different spacings in each dimension. The key to this flexibility is the `Axis` class, which defines the range spacing along a single axis; an N-dimensional `Histogram` is composed of a sequence of N `Axis` objects. Binning values into the histogram is performed by binning along each axis. The standard `RegularAxis` subclass of `Axis` defines an evenly spaced axis with bin centers defined as the center of each bin; additional subclasses may be defined to add different spacing if desired.

Multithreading is achieved through the `ThreadLocalHistogram` class, which is a simple wrapper around the `Histogram` that creates an equivalent histogram on each thread. The standard pattern for parallel histogramming is to generate a `ThreadLocalHistogram` and add data into it, then call the `Histogram::reduceOverThreads` method to accumulate these data into a single histogram. In case any additional post-processing is required per bin, it can also be executed in parallel by providing it as a lambda function to `Histogram::reduceOverThreadsPerBin`.

Computing with Histograms

The `Histogram` class is designed as a data structure for the histogram. Most histogram computations in **freud** involve standard neighbor finding to get bonds, followed by binning some function of these bonds into a histogram. Examples include RDFs (binning bond distances), PMFTs (binning bonds by the different vector components of the bond), and bond order diagrams (binning bond angles). An important distinction between histogram computations and most others is that histograms naturally support an accumulation of information over multiple frames of data, an operation that is ill-defined for many other computations. As a result, histogram computations also need to implement some boilerplate for handling accumulating and averaging data over multiple frames.

The details of these computations are encapsulated by the `BondComputeHistogram` class, which contains a histogram, provides accessors to standard histogram properties like bin counts and axis sizes, and has a generic accumulation method that accepts a lambda compute function. This signature is very similar to the utility functions for looping over neighbors, and in fact the function is transparently forwarded to `locality::loopOverNeighbors`. Any compute that matches this pattern should inherit from the `BondComputeHistogram` class and must implement an `accumulate` method to perform the computation and a `reduce` to reduce thread local histograms into a single histogram..

7.21.4 Making freud Releases

Release Process

Documented below are the steps needed to make a new release of **freud**.

1. Create a release branch, numbered according to Sematic Versioning:

```
git checkout -b release/vX.Y.Z
```

Changelog

2. Review headings (Added, Changed, Fixed, Deprecated, Removed) and ensure consistent formatting.
3. Update the release version and release date from `next` to `vX.Y.Z - YYYY-MM-DD`.

Submodules

4. Update git submodules (optional, but should be done regularly).

Code Formatting

5. Reformat C++ code with `clang-format` 6.0:

```
clang-format -style=file cpp/**/*
```

Contributors

6. Update the contributor list:

```
git shortlog -sne > contributors.txt
```

Bump version

7. Commit previous changes before running `bumpversion`.
8. Use the bumpversion package to increase the version number and automatically generate a git tag:

```
bumpversion patch  # for X.Y.Z
bumpversion minor  # for X.Y
bumpversion major  # for X
```

9. Push the release branch to the remote:

```
git push -u origin release/vX.Y.Z
```

10. Ensure that ReadTheDocs and continuous integration pass (you will need to manually enable the branch on ReadTheDocs' web interface to test it). Then push the tag:

```
git push --tags
```

Automatic Builds

11. Pushing the tag will cause CircleCI to create a release for PyPI automatically (see automation in `.circleci/config.yml`). Make sure this succeeds – it takes a while to run.
12. Create a pull request and merge the release branch into the `master` branch. Delete the release branch on ReadTheDocs' web interface, since there is now a tagged version.
13. The conda-forge autotick bot should discover that the PyPI source distribution has changed, and will create a pull request to the conda-forge feedstock. This pull request may take a few hours to appear. If other changes are needed in the conda-forge recipe (e.g. new dependencies), follow the conda-forge documentation to create a pull request from *your own fork* of the feedstock. Merge the pull request after all continuous integration passes to trigger release builds for conda-forge.

Release Announcement

14. Verify that ReadTheDocs, PyPI, and conda-forge have been updated to the newest version.
15. Send a release notification via the freud-users group. Follow the template of previous release notifications.

7.22 How to cite freud

Please acknowledge the use of this software within the body of your publication for example by copying or adapting the following formulation:

Data analysis for this publication utilized the freud library[1].

> [1] V. Ramasubramani, B. D. Dice, E. S. Harper, M. P. Spellings, J. A. Anderson, and S. C. Glotzer. freud: A Software Suite for High Throughput Analysis of Particle Simulation Data. Computer Physics Communications Volume 254, September 2020, 107275. doi:10.1016/j.cpc.2020.107275.

The paper is available online from Computer Physics Communications and a pre-print is freely available on arXiv.

To cite this reference, you can use the following BibTeX entry:

```
@article{freud2020,
    title = {freud: A Software Suite for High Throughput
             Analysis of Particle Simulation Data},
    author = {Vyas Ramasubramani and
              Bradley D. Dice and
              Eric S. Harper and
              Matthew P. Spellings and
              Joshua A. Anderson and
              Sharon C. Glotzer},
    journal = {Computer Physics Communications},
    volume = {254},
    pages = {107275},
    year = {2020},
    issn = {0010-4655},
    doi = {https://doi.org/10.1016/j.cpc.2020.107275},
    url = {http://www.sciencedirect.com/science/article/pii/S0010465520300916},
    keywords = {Simulation analysis, Molecular dynamics,
                Monte Carlo, Computational materials science},
}
```

Optionally, publications using **freud** in the context of machine learning or data visualization may also wish to cite this reference.

> [2] B. D. Dice, V. Ramasubramani, E. S. Harper, M. P. Spellings, J. A. Anderson, and S. C. Glotzer. Analyzing Particle Systems for Machine Learning and Data Visualization with freud. Proceedings of the 18th Python in Science Conference, 2019, 27-33. doi:10.25080/Majora-7ddc1dd1-004.

The paper is freely available from the SciPy Conference website.

To cite this reference, you can use the following BibTeX entry:

```
@InProceedings{freud2019,
    title = {Analyzing Particle Systems for Machine Learning
             and Data Visualization with freud},
    author = {Bradley D. Dice and
              Vyas Ramasubramani and
              Eric S. Harper and
              Matthew P. Spellings and
              Joshua A. Anderson and
              Sharon C. Glotzer },
    booktitle = {Proceedings of the 18th Python in Science Conference},
    pages = {27-33},
    year = {2019},
    editor = {Chris Calloway and David Lippa and Dillon Niederhut and David Shupe},
    doi = {https://doi.org/10.25080/Majora-7ddc1dd1-004},
    url = {http://conference.scipy.org/proceedings/scipy2019/bradley_dice.html}
}
```

7.23 References

7.24 License

```
BSD 3-Clause License for freud

Copyright (c) 2010-2020 The Regents of the University of Michigan
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors
   may be used to endorse or promote products derived from this software without
   specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
```

(continues on next page)

(continued from previous page)

```
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

7.25 Credits

7.25.1 freud Developers

The following people contributed to the development of freud.

Vyas Ramasubramani - **Lead developer**

- Ensured PEP8 compliance.
- Added CircleCI continuous integration support.
- Create environment module and refactored order module.
- Rewrote most of freud docs, including order, density, and environment modules.
- Fixed nematic order parameter.
- Add properties for accessing class members.
- Various minor bug fixes.
- Refactored PMFT code.
- Refactored Steinhardt order parameter code.
- Wrote numerous examples of freud usage.
- Rewrote most of freud tests.
- Replaced CMake-based installation with setup.py using Cython.
- Add code coverage metrics.
- Added support for installing from PyPI, including ensuring that NumPy is installed.
- Converted all docstrings to Google format, fixed various incorrect docs.
- Debugged and added rotational autocorrelation code.
- Added MSD module.
- Wrote NeighborQuery, _QueryArgs, NeighborQueryResult classes.
- Wrote neighbor iterator infrastructure.
- Wrote PairCompute and SpatialHistogram parent classes.
- Wrote ManagedArray class.
- Wrote C++ histogram-related classes.
- Initial design of freud 2.0 API (NeighborQuery objects, neighbor computations, histograms).
- Standardized neighbor API in Python to use dictionaries of arguments or NeighborList objects for all pair computations.

- Standardized all attribute access into C++ with Python properties.
- Standardized variable naming of points/query_points across all of freud.
- Standardized vector directionality in computes.
- Enabled usage of quaternions in place of angles for orientations in 2D PMFT calculations.
- Wrote new freud 2.0 compute APIs based on neighbor_query objects and neighbors as either dictionaries or NeighborLists.
- Rewrote MatchEnv code to fit freud 2.0 API, splitting it into 3 separate calculations and rewriting internals using NeighborQuery objects.
- Wrote tutorial and reference sections of documentation.
- Unified util and common packages.
- Rewrote all docstrings in the package for freud 2.0.
- Changed Cubatic to use Mersenne Twisters for rng.
- Moved all citations into Bibtex format.
- Created data module.
- Standardized PMFT normalization.
- Enabled optional normalization of RDF.
- Changed correlation function to properly take the complex conjugate of inputs.
- Wrote developer documentation for version 2.0.
- Fixed handling of 2D systems from various data sources.
- Fixed usage of query orientations in PMFTXY, PMFTXYT and PMFTXYZ when points and query points are not identical.
- Refactored and standardized PMFT tests.
- Rewrote build system to use scikit-build.
- Added support for pre-commit hooks.

Bradley Dice - **Lead developer**

- Cleaned up various docstrings.
- Fixed bugs in HexOrderParameter.
- Cleaned up testing code.
- Added bumpversion support.
- Reduced all compile warnings.
- Added Python interface for box periodicity.
- Added Voronoi support for neighbor lists across periodic boundaries.
- Added Voronoi weights for 3D.
- Added Voronoi cell volume computation.
- Incorporated internal BiMap class for Boost removal.
- Wrote numerous examples of freud usage.
- Added some freud tests.

- Added ReadTheDocs support.
- Rewrote interface module into pure Cython.
- Added box duck-typing.
- Removed nose from unit testing.
- Use lambda function for parallelizing CorrelationFunction with TBB.
- Finalized boost removal.
- Wrote AABBQuery class.
- Consolidated cluster module functionality.
- Rewrote SolidLiquid order parameter class.
- Updated AngularSeparation class.
- Rewrote Voronoi implementation to leverage voro++.
- Implemented Voronoi bond weighting to enable Minkowski structure metrics.
- Refactored methods in Box and PeriodicBuffer for v2.0.
- Added checks to C++ for 2D boxes where required.
- Refactored cluster module.
- Standardized vector directionality in computes.
- NeighborQuery support to ClusterProperties, GaussianDensity, Voronoi, PeriodicBuffer, Interface.
- Standardized APIs for order parameters.
- Added radius of gyration to ClusterProperties.
- Improved Voronoi plotting code.
- Corrected number of points/query points in LocalDensity.
- Made PeriodicBuffer inherit from _Compute.
- Removed cudacpu and HOOMDMath includes.
- Added plotting functionality for Box and NeighborQuery objects.
- Added support for reading system data directly from MDAnalysis, garnett, gsd, HOOMD-blue, and OVITO.
- Revised tutorials and documentation on data inputs.
- Updated MSD to perform accumulation with `compute(..., reset=False)`.
- Added test PyPI support to continuous integration.
- Added continuous integration to freud-examples.
- Implemented periodic center of mass computations in C++.
- Revised docs about query modes.
- Implemented smarter heuristics in Voronoi for voro++ block sizes, resulting in significant performance gains for large systems.
- Corrected calculation of neighbor distances in the Voronoi NeighborList.
- Added finite tolerance to ensure stability of 2D Voronoi NeighborList computations.
- Improved stability of Histogram bin calculations.

- Improved error handling of Cubatic input parameters.
- Added 2D Minkowski Structure Metrics to Hexatic, enabled by using `weighted=True` along with a Voronoi NeighborList.
- Worked with Tommy Waltmann to add the SphereVoxelization feature.
- Fixed GaussianDensity normalization in 2D systems.
- Prevented GaussianDensity from computing 3D systems after it has computed 2D systems.
- Contributed code, design, and testing for `DiffractionPattern` class.
- Fixed `Hexatic` order parameter (unweighted) to normalize by number of neighbors instead of the symmetry order k.
- Added `num_query_points` and `num_points` attributes to NeighborList class.
- Added scikit-build support for Windows.

Eric Harper, University of Michigan - **Former lead developer**

- Added TBB parallelism.
- Wrote PMFT module.
- Added NearestNeighbors (since removed).
- Wrote RDF.
- Added bonding module (since removed).
- Added cubatic order parameter.
- Added hexatic order parameter.
- Added Pairing2D (since removed).
- Created common array conversion logic.

Joshua A. Anderson, University of Michigan - **Creator and former lead developer**

- Initial design and implementation.
- Wrote LinkCell and IteratorLinkCell.
- Wrote GaussianDensity, LocalDensity.
- Added parallel module.
- Added indexing modules (since removed).
- Wrote Cluster and ClusterProperties modules.

Matthew Spellings - **Former lead developer**

- Added generic neighbor list.
- Enabled neighbor list usage across freud modules.
- Added correlation functions.
- Added LocalDescriptors class.
- Added interface module.

Erin Teich

- Wrote environment matching (MatchEnv) class.
- Wrote BondOrder class (with Julia Dshemuchadse).

- Wrote AngularSeparation class (with Andrew Karas).
- Contributed to LocalQl development.
- Wrote LocalBondProjection class.

M. Eric Irrgang

- Authored kspace module (since removed).
- Fixed numerous bugs.
- Contributed to freud.shape (since removed).

Chrisy Du

- Authored Steinhardt order parameter classes.
- Fixed support for triclinic boxes.

Antonio Osorio

- Developed TrajectoryXML class.
- Various bug fixes.
- OpenMP support.

Richmond Newman

- Developed the freud box.
- Solid liquid order parameter.

Carl Simon Adorf

- Developed the Python box module.

Jens Glaser

- Wrote kspace front-end (since removed).
- Modified kspace module (since removed).
- Wrote Nematic order parameter class.

Benjamin Schultz

- Wrote Voronoi class.
- Fix normalization in GaussianDensity.
- Bug fixes in shape module (since removed).

Bryan VanSaders

- Make Cython catch C++ exceptions.
- Add shiftvec option to PMFT.

Ryan Marson

- Various GaussianDensity bugfixes.

Yina Geng

- Co-wrote Voronoi neighbor list module.
- Add properties for accessing class members.

Carolyn Phillips

- Initial design and implementation.
- Package name.

Ben Swerdlow

- Documentation and installation improvements.

James Antonaglia

- Added number of neighbors as an argument to HexOrderParameter.
- Bugfixes.
- Analysis of deprecated kspace module.

Mayank Agrawal

- Co-wrote Voronoi neighbor list module.

William Zygmunt

- Helped with Boost removal.

Greg van Anders

- Bugfixes for CMake and SSE2 installation instructions.

James Proctor

- Cythonization of the cluster module.

Rose Cersonsky

- Enabled TBB-parallelism in density module.
- Fixed how C++ arrays were pulled into Cython.

Wenbo Shen

- Translational order parameter.

Andrew Karas

- Angular separation.
- Wrote reference implementation for rotational autocorrelation.

Paul Dodd

- Fixed CorrelationFunction namespace, added ComputeOCF class for TBB parallelization.

Tim Moore

- Added optional rmin argument to density.RDF.
- Enabled NeighborList indexing.

Alex Dutton

- BiMap class for MatchEnv.

Matthew Palathingal

- Replaced use of boost shared arrays with shared ptr in Cython.
- Helped incorporate BiMap class into MatchEnv.

Kelly Wang

- Enabled NeighborList indexing.

- Added methods `compute_distances` and `compute_all_distances` to Box.
- Added method `crop` to Box.

Yezhi Jin

- Added support for 2D arrays in the Python interface to Box functions.
- Rewrote Voronoi implementation to leverage voro++.
- Implemented Voronoi bond weighting to enable Minkowski structure metrics.
- Contributed code, design, and testing for `DiffractionPattern` class.

Brandon Butler

- Rewrote Steinhardt order parameter.

Jin Soo Ihm

- Added benchmarks.
- Contributed to NeighborQuery classes.
- Refactored C++ to perform neighbor queries on-the-fly.
- Added plotting functions to analysis classes.
- Wrote RawPoints class.
- Created Compute parent class with decorators to ensure properties have been computed.
- Updated common array conversion logic.
- Added many validation tests.

Mike Henry

- Fixed syntax in freud-examples notebooks for v2.0.
- Updated documentation links

Michael Stryk

- Added short examples into Cluster, Density, Environment, and Order Modules.

Tommy Waltmann

- Worked with Bradley Dice to add the SphereVoxelization feature.
- Contributed code, design, and testing for `DiffractionPattern` class.

Maya Martirossyan

- Added test for Steinhardt for particles without neighbors.

Pavel Buslaev

- Added `values` argument to compute method of `GaussianDensity` class.

7.25.2 Source code

Eigen (http://eigen.tuxfamily.org) is included as a git submodule in freud. Eigen is made available under the Mozilla Public License v2.0 (http://mozilla.org/MPL/2.0/). Its linear algebra routines are used for various tasks including the computation of eigenvalues and eigenvectors.

fsph (https://github.com/glotzerlab/fsph) is included as a git submodule in freud. It is used for the calculation of spherical harmonics. fsph is made available under the MIT license:

```
Copyright (c) 2016 The Regents of the University of Michigan

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

HOOMD-blue (https://github.com/glotzerlab/hoomd-blue) is the original source of some algorithms and tools for vector math implemented in freud. HOOMD-blue is made available under the BSD 3-Clause license:

```
BSD 3-Clause License for HOOMD-blue

Copyright (c) 2009-2019 The Regents of the University of Michigan All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors
   may be used to endorse or promote products derived from this software without
   specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
```

(continues on next page)

(continued from previous page)

```
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

voro++ (https://github.com/chr1shr/voro) is included as a git submodule in freud. It is used for computing Voronoi diagrams. voro++ is made available under the following license:

```
Voro++ Copyright (c) 2008, The Regents of the University of California, through
Lawrence Berkeley National Laboratory (subject to receipt of any required
approvals from the U.S. Dept. of Energy). All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

(1) Redistributions of source code must retain the above copyright notice, this
list of conditions and the following disclaimer.

(2) Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.

(3) Neither the name of the University of California, Lawrence Berkeley
National Laboratory, U.S. Dept. of Energy nor the names of its contributors may
be used to endorse or promote products derived from this software without
specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

You are under no obligation whatsoever to provide any bug fixes, patches, or
upgrades to the features, functionality or performance of the source code
("Enhancements") to anyone; however, if you choose to make your Enhancements
available either publicly, or directly to Lawrence Berkeley National
Laboratory, without imposing a separate written license agreement for such
Enhancements, then you hereby grant the following license: a non-exclusive,
royalty-free perpetual license to install, use, modify, prepare derivative
works, incorporate into other computer software, distribute, and sublicense
such enhancements or derivative works thereof, in binary and source code form.
```

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

T