freud Documentation

Release 1.0.0

The Regents of the University of Michigan

Feb 09, 2019

Contents

CHAPTER 1

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, and clusters, as well as original analysis methods including potentials of mean force and torque (PMFTs) and local environment matching. The freud library uses NumPy arrays for input and output, enabling integration with the scientific Python ecosystem for many typical materials science workflows.

1.1 Installation

1.1.1 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:

- Python (2.7+ required, 3.5+ recommended)
- NumPy
- Intel Threading Building Blocks (TBB)

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

- Cython: The freud repository contains Cython-generated `*.cpp` files in the `freud/` directory that can be used directly. However, Cython is necessary if you wish to recompile these files.

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. In general, the following options are available for setup.py in addition to the standard setuptools options (notes are included to indicate which options are only available for specific subcommands such as build_ext):

–PRINT-WARNINGS Specify whether or not to print compilation warnings resulting from the build even if the build succeeds with no errors.

–ENABLE-CYTHON Rebuild the Cython-generated C++ files. If there are any unexpected issues with compiling the C++ shipped with the build, using this flag may help. It is also necessary any time modifications are made to the Cython files.

-j Compile in parallel. This affects both the generation of C++ files from Cython files and the subsequent compilation of the source files. In the latter case, this option controls the number of Python modules that will be compiled in parallel.

–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. Note that this information can also be provided using the environment variable TBB_ROOT. The options –TBB-INCLUDE and –TBB-LINK will take precedence over –TBB-ROOT if both are specified.

–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. Note that this information can also be provided using the environment variable TBB_ROOT. The options –TBB-INCLUDE and –TBB-LINK will take precedence over –TBB-ROOT if both are specified.

–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 that this information can also be provided using the environment variable TBB_ROOT. The options –TBB-INCLUDE and –TBB-LINK will take precedence over –TBB-ROOT if both are specified.

The following additional arguments are primarily useful for developers:

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

–NTHREAD Specify the number of threads to allocate to compiling each module. This option is primarily useful for rapid development, particularly when all changes are in one module. While the -j option will not help parallelize this case, this option allows compilation of multiple source files belonging to the same module in parallel.

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

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

1.1.2 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.

1.1.3 Documentation

The documentation for freud is hosted online at ReadTheDocs, but you may also build the documentation yourself:

Building the documentation

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

- Sphinx

You can install sphinx using conda

```
conda install sphinx
```

or from PyPi

```
pip install sphinx
```

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
```

1.2 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 [Matplotlib] and Bokeh [Bokeh].

1.2.1 Key concepts

There are a few critical concepts, algorithms, and data structures that are central to all of *freud*. The *box* module defines the concept of a periodic simulation box, and the *locality* module defines methods for finding nearest neighbors for particles. Since both of these are used throughout *freud*, we recommend familiarizing yourself with these before delving too deep into the workings of specific *freud* modules.

Box

The goal of `freud` is to perform generic analyses of particle simulations. Such simulations are always conducted within some region representing physical space; in `freud`, these regions are known as *simulation boxes*, or simply *boxes*. An important characteristic of many simulations is that the simulation box is periodic, *i.e.* particles can travel and interact across system boundaries (for more information, see the Wikipedia page). Simulations frequently use periodic boundary conditions to effectively simulate infinite systems without actually having to include an infinite number of particles. In such systems, a box in N dimensions can be represented by N linearly independent vectors.

The `Box` class provides the standard API for such simulation boxes throughout `freud`. The class represents some 2- or 3-dimensional region of space, and it provides utility functions for interacting with this space, including the ability to wrap vectors outside this box into the box according to periodic boundary conditions. Boxes are represented according to the HOOMD-blue convention for boxes. According to this convention, a 3D (2D) simulation box is fully defined by 3 (2) linearly independent vectors, which are represented by 3 (2) characteristic lengths and 3 (1) tilt factors indicating how these vectors are angled with respect to one another. With this convention, a generic box is represented by the following 3×3 matrix:

$$\begin{pmatrix} L_x & xy \times L_x & xz \times L_z \\ 0 & L_y & yz \times L_z \\ 0 & 0 & L_z \end{pmatrix}$$

where xy, xz, and yz are the tilt factors. Note that this convention imposes the requirement that the box vectors form a right-handed coordinate system, which manifests itself in the form of an upper (rather than lower) triangular box matrix.

In this notebook, we demonstrate the basic features of the `Box` class, particularly the facility for wrapping particles back into the box under periodic boundary conditions. For more information, see the `freud.box` documentation.

Box Creation

There are many ways to construct a box. We demonstrate all of these below, with some discussion of when they might be useful.

Default (full) API

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 a 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:
        Box(Lx=1.0, Ly=2.0, Lz=3.0, xy=0.0, xz=0.0, yz=0.0, dimensions=3)
The copied box:
        Box(Lx=1.0, Ly=2.0, Lz=3.0, xy=0.0, xz=0.0, yz=0.0, dimensions=3)

The original box is modified:
        Box(Lx=5.0, Ly=2.0, Lz=3.0, xy=0.0, xz=0.0, yz=0.0, dimensions=3)
The copied box is not:
        Box(Lx=1.0, Ly=2.0, Lz=3.0, xy=0.0, xz=0.0, yz=0.0, dimensions=3)

The new copy:
        Box(Lx=1.0, Ly=2.0, Lz=3.0, xy=0.0, xz=0.0, yz=0.0, dimensions=3)
The new copy after the original is modified:
        Box(Lx=2.0, Ly=2.0, Lz=3.0, xy=0.0, xz=0.0, yz=0.0, dimensions=3)
The modified original box:
        Box(Lx=2.0, Ly=2.0, Lz=3.0, xy=0.0, xz=0.0, yz=0.0, dimensions=3)
```

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:
        Box(Lx=1.0, Ly=1.0, Lz=0.5, xy=1.0, xz=0.0, yz=1.0, dimensions=3)

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

The box matrix was automatically detected:
        Box(Lx=1.0, Ly=1.0, Lz=0.5, xy=1.0, xz=0.0, yz=1.0, dimensions=3)

Using a 3x3 numpy array:
        Box(Lx=1.0, Ly=1.0, Lz=0.5, xy=1.0, xz=0.0, yz=1.0, dimensions=3)
```

From a namedtuple or dict

A box can be also be constructed from a namedtuple with the appropriate entries. Any other object that provides a similar API for attribute-based access of L_x, L_y, L_z, xy, xz, and yz (or some subset) will work equally well. 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', 'dimensions'])

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

```
Box from named tuple:
        Box(Lx=5.0, Ly=3.0, Lz=2.0, xy=0.0, xz=0.0, yz=0.0, dimensions=3)

2D Box from named tuple:
        Box(Lx=5.0, Ly=3.0, Lz=0.0, xy=0.0, xz=0.0, yz=0.0, dimensions=2)
```

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:
        Box(Lx=5.0, Ly=3.0, Lz=2.0, xy=0.0, xz=0.0, yz=0.0, dimensions=3)
```

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:
        Box(Lx=5.0, Ly=6.0, Lz=7.0, xy=0.5, xz=0.0, yz=0.5, dimensions=3)

2D Box from list:
        Box(Lx=5.0, Ly=6.0, Lz=0.0, xy=0.0, xz=0.0, yz=0.0, dimensions=2)
```

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:
        Box(Lx=5.0, Ly=5.0, Lz=5.0, xy=0.0, xz=0.0, yz=0.0, dimensions=3)

Square Box:
        Box(Lx=5.0, Ly=5.0, Lz=0.0, xy=0.0, xz=0.0, yz=0.0, dimensions=2)
```

Export

If you want to export or display the box, you can export box objects into their matrix or namedtuple representations, which provide completely specified descriptions of the box. Note that the namedtuple type used by freud boxes, the BoxTuple, is simply an internal representation.

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

[8]:
```
[[5.0, 0.0, 0.0], [0, 5.0, 0.0], [0, 0, 5.0]]
```

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

[9]:
```
BoxTuple(Lx=5.0, Ly=5.0, Lz=5.0, xy=0.0, xz=0.0, yz=0.0)
```

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.Linv]))
```

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

Boxes also support converting to and from fractional 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]:
```
# Conversion to coordinate representation from fractions.
print(box.makeCoordinates([0, 0, 0]))
print(box.makeCoordinates([0.5, 0.5, 0.5]))
print(box.makeCoordinates([0.8, 0.3, 1]))
print()

# Conversion to and from coordinate representation, resulting
# in the input fractions.
print(box.makeFraction(box.makeCoordinates([0, 0, 0])))
print(box.makeFraction(box.makeCoordinates([0.5, 0.5, 0.5])))
print("[{:1.1f}, {:1.1f}, {:1.1f}]".format(*box.makeFraction(box.makeCoordinates([0.8,
↪ 0.3, 1]))))
```

```
[-5.0, -5.0, -5.0]
[0.0, 0.0, 0.0]
[3.0, -2.0, 5.0]

[0.0, 0.0, 0.0]
[0.5, 0.5, 0.5]
[0.8, 0.3, 1.0]
```

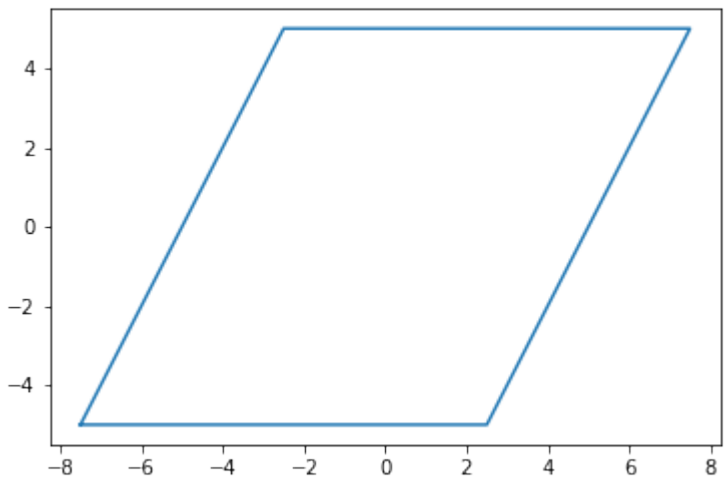
Finally (and most critically for enforcing periodicity), boxes support wrapping vectors from outside the box into the box. The concept of periodicity and box wrapping is most easily demonstrated visually.

[12]:
```
# We define box plot generation separately
from util import box_2d_to_points

# 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]])
points = box_2d_to_points(box)
```

[13]:
```
from matplotlib import pyplot as plt
fig, ax = plt.subplots(figsize=(9, 6))
ax.plot(points[:, 0], points[:, 1], color='k')
plt.show()
```

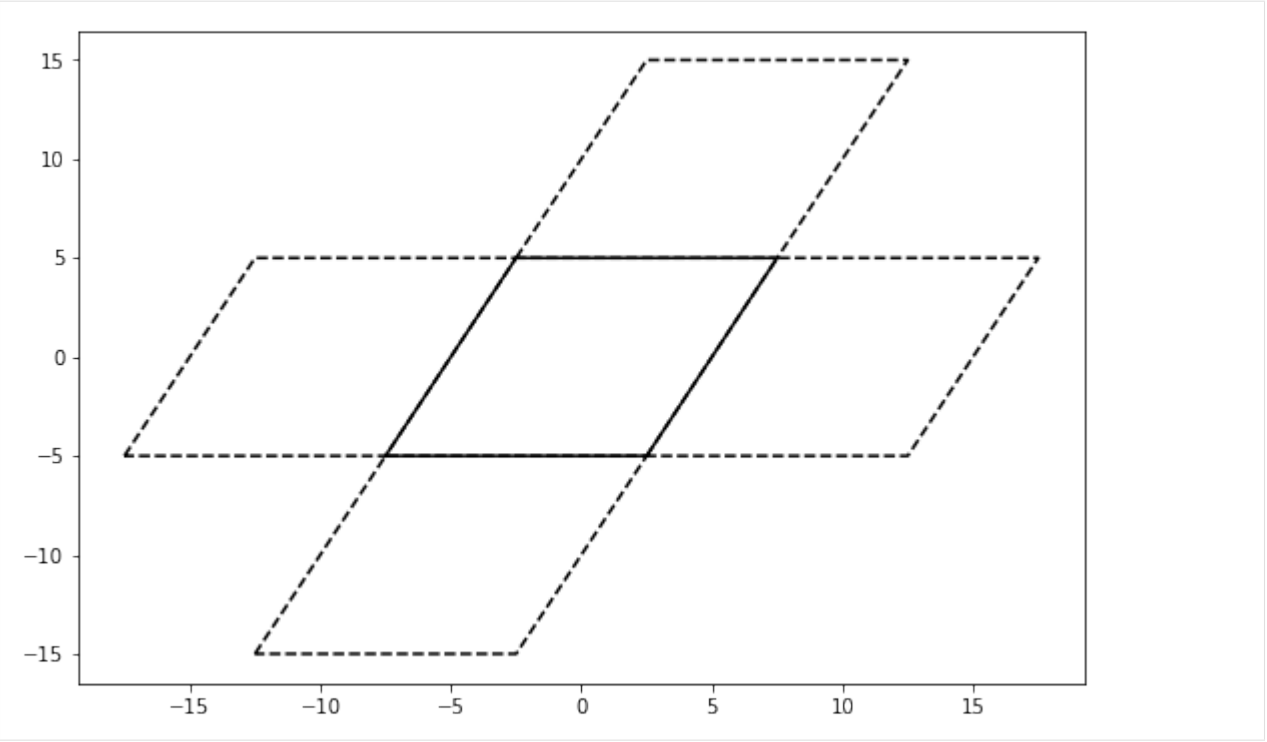
```
<Figure size 900x600 with 1 Axes>
```

[14]:
```
plt.figure()
plt.plot(points[:, 0], points[:, 1])
plt.show()
```



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.

[15]:
```
fig, ax = plt.subplots(figsize=(9, 6))
ax.plot(points[:, 0], points[:, 1], color='k')
ax.plot(points[:, 0] + Lx, points[:, 1], linestyle='dashed', color='k')
ax.plot(points[:, 0] - Lx, points[:, 1], linestyle='dashed', color='k')
ax.plot(points[:, 0] + xy*Ly, points[:, 1] + Ly, linestyle='dashed', color='k')
ax.plot(points[:, 0] - xy*Ly, points[:, 1] - Ly, linestyle='dashed', color='k')
plt.show()
```



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

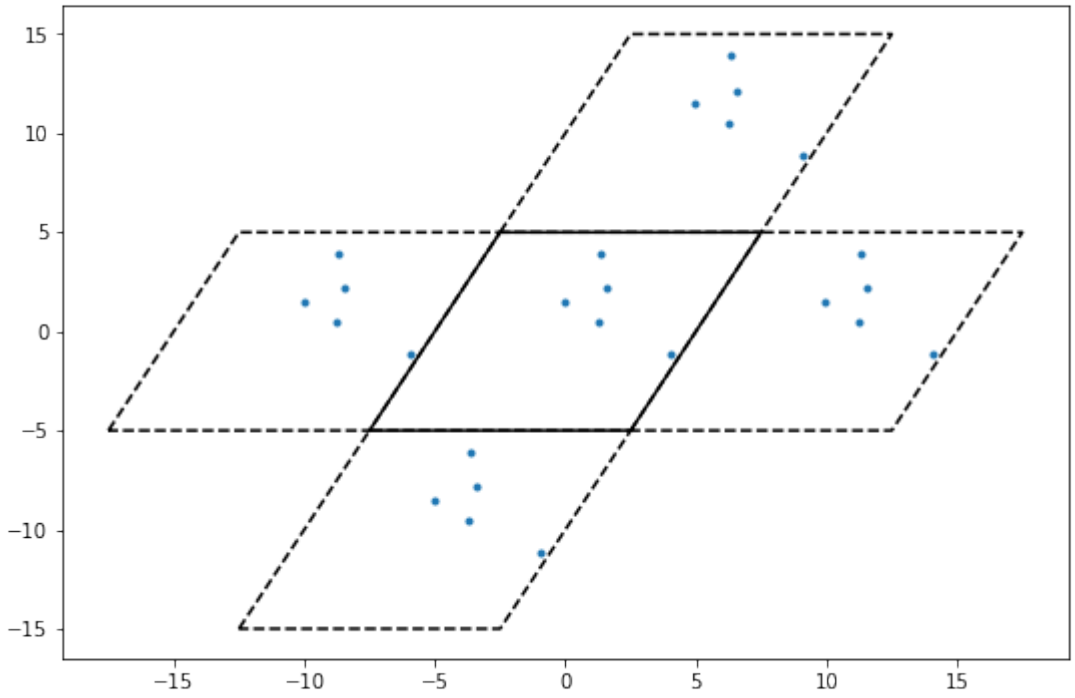
[16]:
```
np.random.seed(0)
tmp = np.random.rand(5, 2)
origin = np.array(box.makeCoordinates([0, 0, 0]))
u = np.array(box.makeCoordinates([1, 0, 0])) - origin
v = np.array(box.makeCoordinates([0, 1, 0])) - origin
particles = u*tmp[:, [0]] + v*tmp[:, [1]]
```

[17]:
```
fig, ax = plt.subplots(figsize=(9, 6))

# Plot the boxes.
ax.plot(points[:, 0], points[:, 1], color='k')
ax.plot(points[:, 0] + Lx, points[:, 1], linestyle='dashed', color='k')
ax.plot(points[:, 0] - Lx, points[:, 1], linestyle='dashed', color='k')
ax.plot(points[:, 0] + xy*Ly, points[:, 1] + Ly, linestyle='dashed', color='k')
ax.plot(points[:, 0] - xy*Ly, points[:, 1] - Ly, linestyle='dashed', color='k')

# Plot the points in the original box.
ax.plot(particles[:, 0] + origin[0], particles[:, 1] + origin[1],
        linestyle='None', marker='.', color='#1f77b4')

# Define the different origins.
origins = []
origins.append(np.array(box.makeCoordinates([-1, 0, 0])))
origins.append(np.array(box.makeCoordinates([1, 0, 0])))
origins.append(np.array(box.makeCoordinates([0, -1, 0])))
origins.append(np.array(box.makeCoordinates([0, 1, 0])))

# Plot particles in each of the periodic boxes.
for o in origins:
```

(continues on next page)

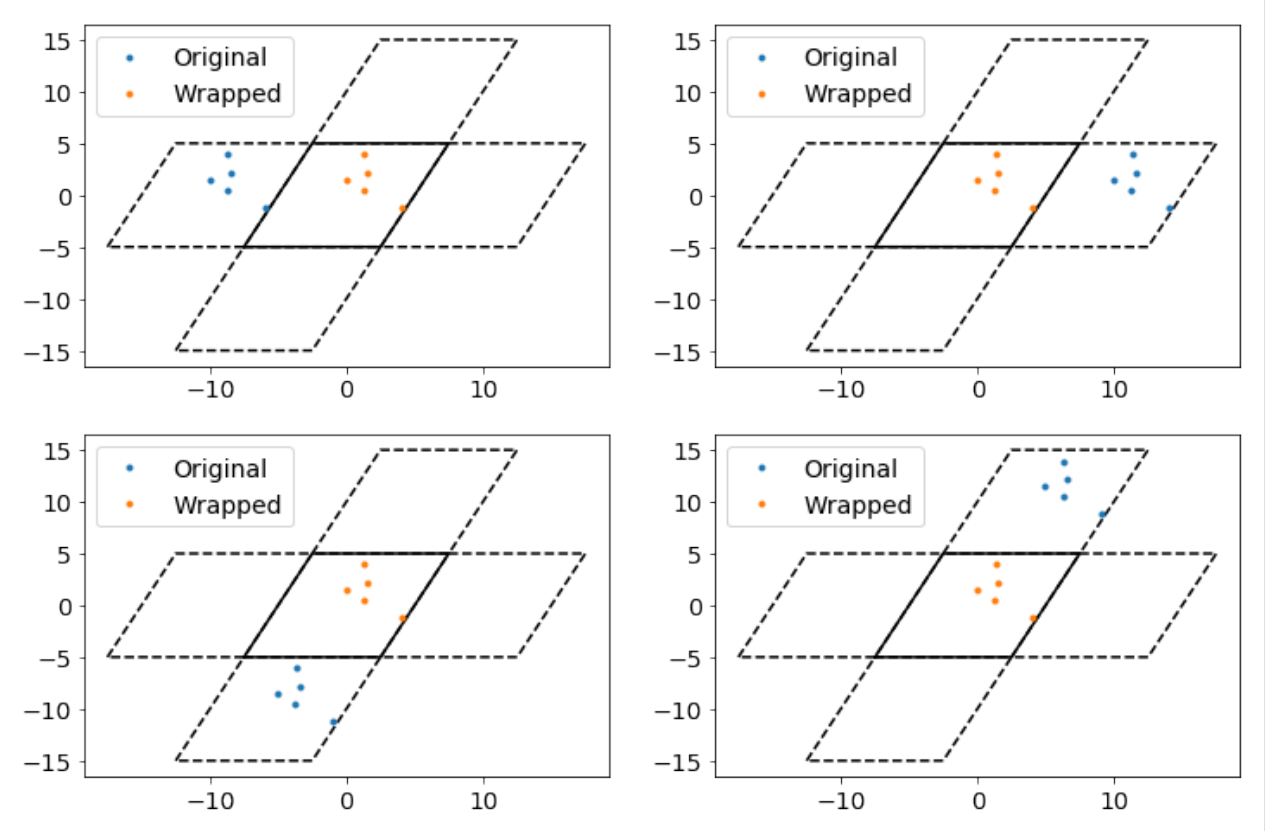
(continued from previous page)

```
    ax.plot(particles[:, 0] + o[0], particles[:, 1] + o[1],
            linestyle='None', marker='.', color='#1f77b4')
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.

[18]:
```
fig, axes = plt.subplots(2, 2, figsize=(12, 8))

# Plot the boxes.
for i, ax in enumerate(axes.flatten()):
    ax.plot(points[:, 0], points[:, 1], color='k')
    ax.plot(points[:, 0] + Lx, points[:, 1], linestyle='dashed', color='k')
    ax.plot(points[:, 0] - Lx, points[:, 1], linestyle='dashed', color='k')
    ax.plot(points[:, 0] + xy*Ly, points[:, 1] + Ly, linestyle='dashed', color='k')
    ax.plot(points[:, 0] - xy*Ly, points[:, 1] - Ly, linestyle='dashed', color='k')

    # Plot the points relative to origin i.
    o = origins[i]
    ax.plot(particles[:, 0] + o[0], particles[:, 1] + o[1],
            linestyle='None', marker='.', label='Original')

    # Now wrap these points and plot them.
    wrapped_particles = box.wrap(particles + o)
    ax.plot(wrapped_particles[:, 0], wrapped_particles[:, 1],
            linestyle='None', marker='.', label='Wrapped')
    ax.tick_params(axis="both", which="both", labelsize=14)

    ax.legend(fontsize=14)
plt.show()
```



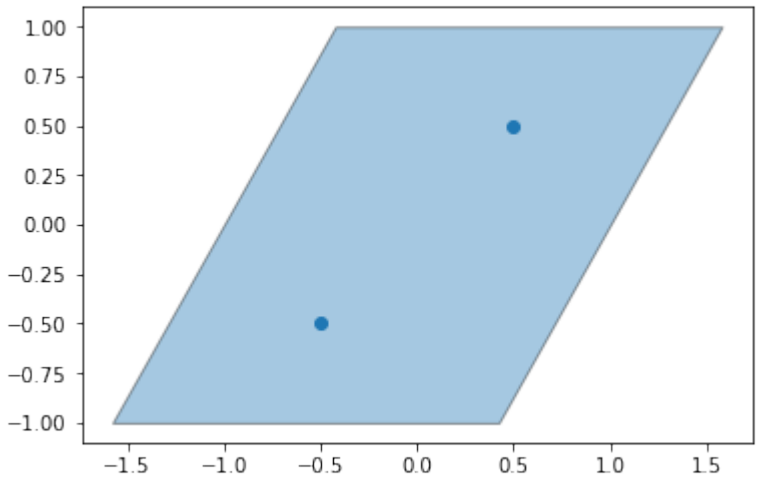
ParticleBuffer - Unit Cell RDF

The `ParticleBuffer` class is meant to replicate particles 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]:
```
import freud
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from util import box_2d_to_points
```

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.asarray([[-0.5, -0.5, -0.5], [0.5, 0.5, 0.5]])
corners = box_2d_to_points(box)
ax = plt.gca()
box_patch = plt.Polygon(corners[:, :2])
patch_collection = matplotlib.collections.PatchCollection([box_patch], edgecolors=
↪'black', alpha=0.4)
ax.add_collection(patch_collection)
plt.scatter(points[:, 1], points[:, 2])
plt.show()
```



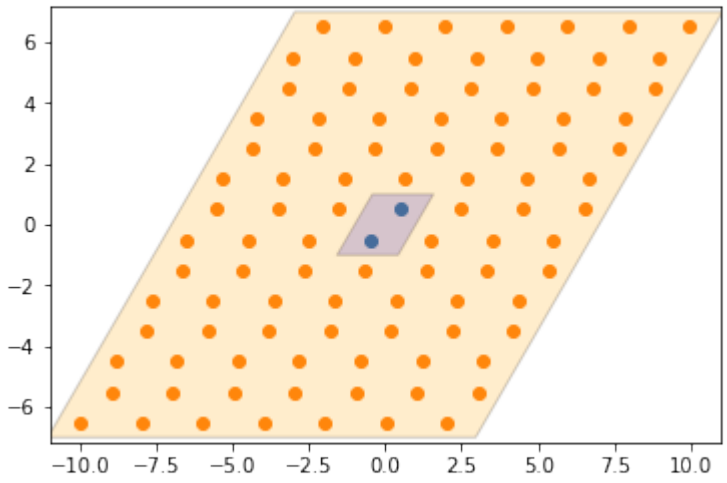
Next, we create a `ParticleBuffer` instance and have it compute the "buffer" particles that lie outside the first periodicity. These positions are stored in the `buffer_positions` attribute. The corresponding `buffer_ids` array gives a mapping from the index of the buffer particle to the index of the 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.box.ParticleBuffer(box)
pbuff.compute(points, 6, images=True)
print(pbuff.buffer_particles[:10], '...')
```

```
[[-9.964102  -6.5        0.       ]
 [-8.809401  -4.5        0.       ]
 [-7.6547003 -2.5        0.       ]
 [-6.5       -0.5        0.       ]
 [-5.3452992  1.5        0.       ]
 [-4.1905985  3.5        0.       ]
 [-3.0358982  5.5        0.       ]
 [-7.9641013 -6.5        0.       ]
 [-6.8094006 -4.5        0.       ]
 [-5.6547003 -2.5        0.       ]] ...
```

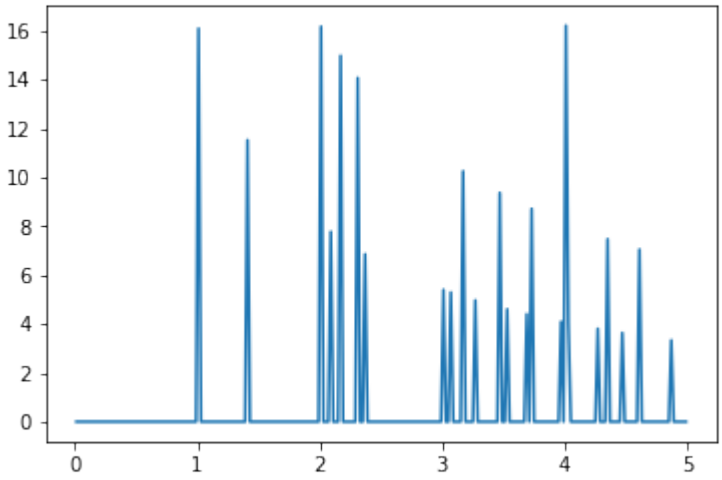
Below, we plot the original unit cell and the replicated buffer points and buffer box.

[4]:
```
plt.scatter(points[:, 0], points[:, 1])
plt.scatter(pbuff.buffer_particles[:, 0], pbuff.buffer_particles[:, 1])
box_patch = plt.Polygon(corners[:, :2])
buff_corners = box_2d_to_points(pbuff.buffer_box)
buff_box_patch = plt.Polygon(buff_corners[:, :2])
patch_collection = matplotlib.collections.PatchCollection(
    [box_patch, buff_box_patch], facecolors=['blue', 'orange'],
    edgecolors='black', alpha=0.2)
plt.gca().add_collection(patch_collection)
plt.show()
```



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

[5]:
```
rdf = freud.density.RDF(rmax=5, dr=0.02)
rdf.compute(pbuff.buffer_box, ref_points=points, points=pbuff.buffer_particles)
plt.plot(rdf.R, rdf.RDF)
plt.show()
```



LinkCell

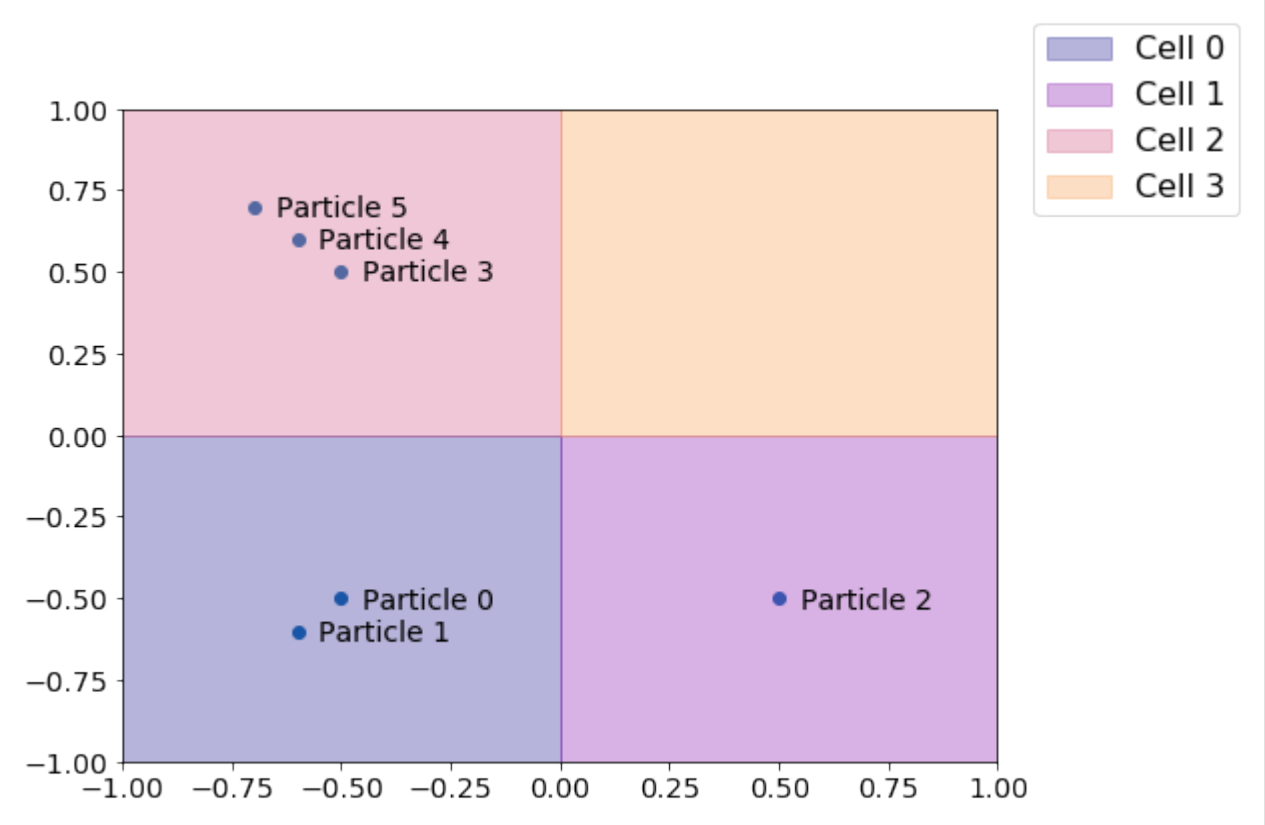
Many of the most powerful analyses of particle simulations involve some characterization of the local environments of particles. Whether the analyses involve finding clusters, identifying interfaces, computing order parameters, or something else entirely, they always require finding particles in proximity to others so that properties of the local environment can be computed. The `freud.locality.NeighborList` and `freud.locality.LinkCell` classes are the fundamental building blocks for this type of calculation. The `NeighborList` class is essentially a container for particle pairs that are determined to be adjacent to one another. The `LinkCell` class implements the standard linked-list cell algorithm, in which a cell list is computed using linked lists to store the particles in each cell. In this notebook, we provide a brief demonstration of how this data structure works and how it is used throughout freud.

We begin by demonstrating how a cell list works, which is essentially by dividing space into fixed width cells.

[1]:
```
from __future__ import division
import freud
import numpy as np
from matplotlib import pyplot as plt
import timeit

# place particles 0 and 1 in cell 0
# place particle 2 in cell 1
# place particles 3,4,5 in cell 3
# and no particles in cells 4,5,6,7
particles = np.array([[-0.5, -0.5, 0],
                      [-0.6, -0.6, 0],
                      [0.5, -0.5, 0],
                      [-0.5, 0.5, 0],
                      [-0.6, 0.6, 0],
                      [-0.7, 0.7, 0]], dtype='float32')


L = 2  # The box size
r_max = 1  # The cell width, and the nearest neighbor distance
box = freud.box.Box.square(L)
lc = freud.locality.LinkCell(box, r_max)
lc.compute(box, particles)

for c in range(0, lc.num_cells):
    print("The following particles are in cell {}: {}".format(c, ', '.join([str(x)␣
↪for x in lc.itercell(c)])))
```

```
The following particles are in cell 0: 0, 1
The following particles are in cell 1: 2
The following particles are in cell 2: 3, 4, 5
The following particles are in cell 3:
```

[2]:
```
from matplotlib import patches
from matplotlib import cm
cmap = cm.get_cmap('plasma')
colors = [cmap(i/lc.num_cells) for i in range(lc.num_cells)]

fig, ax = plt.subplots(figsize=(9, 6))
ax.scatter(particles[:, 0], particles[:, 1])
ax.set_xlim([-1, 1])
ax.set_ylim([-1, 1])
corners = [(-1, -1), (0, -1), (-1, 0), (0, 0)]
handles = []
labels = []
for i, corner in enumerate(corners):
    p = patches.Rectangle(corner, 1, 1, color=colors[i], alpha=0.3)
    ax.add_patch(p)
    handles.append(p)
    labels.append("Cell {}".format(i))
ax.tick_params(axis='both', which='both', labelsize=14)
fig.legend(handles, labels, fontsize=16)
fig.subplots_adjust(right=0.8)
for i, p in enumerate(particles):
    ax.text(p[0]+0.05, p[1]-0.03, "Particle {}".format(i), fontsize=14)
```



The principle behind a cell list is that depending on how close particles have to be to be considered neighbors, we can construct a cell list of an appropriate width such that a given particle's neighbors can always be found by only looking in the neighboring cells, saving us the work of checking all the other particles in the system. We can now extract the NeighborList object computed using this cell list for finding particle neighbors.

[3]:
```
nlist = lc.nlist
for i in set(nlist.index_i):
    js = nlist.index_j[nlist.index_i == i]
    print("The particles within a distance 1 of particle {} are: {}".format(
        i, ', '.join([str(j) for j in js])))
```

```
The particles within a distance 1 of particle 0 are: 1, 4, 5
The particles within a distance 1 of particle 1 are: 0, 2, 3, 4, 5
The particles within a distance 1 of particle 2 are: 1
The particles within a distance 1 of particle 3 are: 1, 4, 5
The particles within a distance 1 of particle 4 are: 0, 1, 3, 5
The particles within a distance 1 of particle 5 are: 0, 1, 3, 4
```

Finally, we can easily check this computation manually by just computing particle distances. Note that we need to be careful to make sure that we properly respect the box periodicity, which means that interparticle distances should be calculated according to the minimum image convention. In essence, this means that since the box is treated as being infinitely replicated in all directions, we have to ensure that each particle is only interacting with the closest copy of another particle. We can easily enforce this here by making sure that particle distances are never large than half the box length in any given dimension.

[4]:
```
def compute_distances(box, positions):
    """Compute pairwise particle distances, taking into account PBCs.
```

(continues on next page)

(continued from previous page)

```
    Args:
        box (:class:`freud.box.Box`): The simulation box the particles live in.
        positions (:class:`np.ndarray`): The particle positions.
    """
    # First we shift all the particles so that the coordinates lie from
    # [0, L] rather than [-L/2, L/2].
    positions[:, 0] = np.mod(positions[:, 0]+box.Lx/2, box.Lx)
    positions[:, 1] = np.mod(positions[:, 1]+box.Ly/2, box.Ly)
    positions[:, 0] = np.mod(positions[:, 0]+box.Lx/2, box.Lx)
    positions[:, 1] = np.mod(positions[:, 1]+box.Ly/2, box.Ly)

    # To apply minimum image convention, we check if the distance is
    # greater than half the box length in either direction, and if it
    # is, we replace it with L-distance instead. We use broadcasting
    # to get all pairwise positions, then modify the pos2 array where
    # the distance is found to be too large for a specific pair.
    pos1, pos2 = np.broadcast_arrays(positions[np.newaxis, :, :], positions[:, np.
↪newaxis, :])
    vectors = pos1 - pos2
    pos2[:, :, 0] = np.where(np.abs(vectors[:, :, 0]) > box.Lx/2,
                             box.Lx - np.abs(pos2[:, :, 0]),
                             pos2[:, :, 0])
    pos2[:, :, 1] = np.where(np.abs(vectors[:, :, 1]) > box.Ly/2,
                             box.Ly - np.abs(pos2[:, :, 1]),
                             pos2[:, :, 1])

    distances = np.linalg.norm(pos1 - pos2, axis=-1)
    return distances
```

[5]:
```
pairwise_distances = compute_distances(box, particles)
for i in range(pairwise_distances.shape[0]):
    js = np.where(pairwise_distances[i, :] < r_max)
    print("The particles within a distance 1 of particle {} are: {}".format(
        i, ', '.join([str(j) for j in js[0] if not j==i])))
```

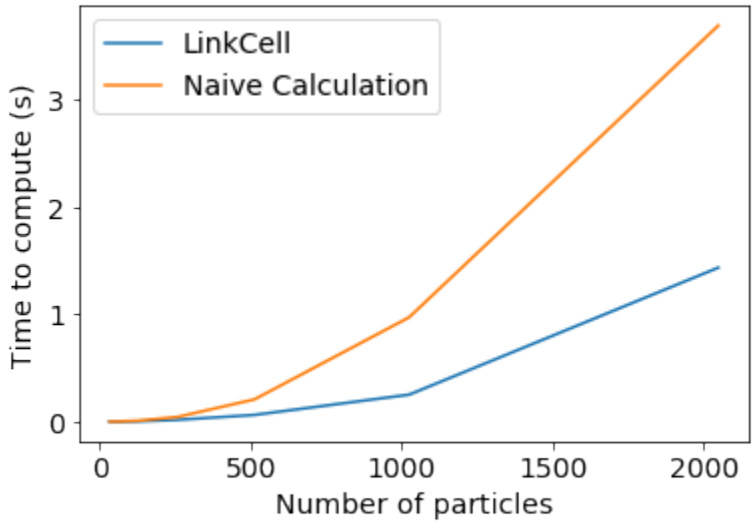
```
The particles within a distance 1 of particle 0 are: 1, 4, 5
The particles within a distance 1 of particle 1 are: 0, 2, 3, 4, 5
The particles within a distance 1 of particle 2 are: 1
The particles within a distance 1 of particle 3 are: 1, 4, 5
The particles within a distance 1 of particle 4 are: 0, 1, 3, 5
The particles within a distance 1 of particle 5 are: 0, 1, 3, 4
```

For larger systems, however, such pairwise calculations would quickly become prohibitively expensive. The primary benefit of the LinkCell object is that it can dramatically improve this cost.

[6]:
```
log_Ns = np.arange(5, 12)
lc_times = []
naive_times = []
for log_N in log_Ns:
    print("Running for log_N = {}".format(log_N))
    particles = np.random.rand(int(2**log_N), 3)*L-L/2
    particles[:, 0] = 0
    lc_times.append(timeit.timeit("lc.compute(box, particles)", number=10,␣
↪globals=globals()))
    naive_times.append(timeit.timeit("compute_distances(box, particles)", number=10,␣
↪globals=globals()))
```

```
Running for log_N = 5
Running for log_N = 6
Running for log_N = 7
Running for log_N = 8
Running for log_N = 9
Running for log_N = 10
Running for log_N = 11
```

[7]:
```
fig, ax = plt.subplots()
ax.plot(2**log_Ns, lc_times, label="LinkCell")
ax.plot(2**log_Ns, naive_times, label="Naive Calculation")
ax.legend(fontsize=14)
ax.tick_params(axis='both', which='both', labelsize=14)
ax.set_xlabel("Number of particles", fontsize=14)
ax.set_ylabel("Time to compute (s)", fontsize=14);
```



Nearest Neighbors

One of the basic computations required for higher-level computations (such as the *hexatic order parameter*) is finding the nearest neighbors of a particle. This tutorial will show you how to compute the nearest neighbors and visualize that data.

The algorithm is straightforward:

```
for each particle i:
    for each particle j in neighbor_cells(i):
        r_ij = position[j] - position[i]
        r = sqrt(dot(r_ij, r_ij))
        l_r_array.append(r)
        l_n_array.append(j)
        # sort by distance
        sort(n_array, r_array)
        neighbor_array[i] = n_array[:k]
```

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 a solid at packing fractions $\phi = 0.75$.

[1]:
```
from bokeh.io import output_notebook
output_notebook()
from bokeh.models import Legend
from bokeh.plotting import figure, output_file, show
from bokeh.layouts import gridplot
import numpy as np
import time
from freud import parallel, box, locality
parallel.setNumThreads(4)
import util

# Create hexagon vertices
verts = util.make_polygon(sides=6, radius=0.6204)

# Define colors for our system
c_list = ["#30A2DA", "#FC4F30", "#E5AE38", "#6D904F", "#9757DB",
          "#188487", "#FF7F00", "#9A2C66", "#626DDA", "#8B8B8B"]
c_dict = dict()
c_dict[6] = c_list[0]
c_dict[5] = c_list[1]
c_dict[4] = c_list[2]
c_dict[3] = c_list[7]
c_dict[7] = c_list[4]

def render_plot(p, fbox):
    # Display box
    corners = util.box_2d_to_points(fbox)
    p.patches(xs=[corners[:-1, 0]], ys=[corners[:-1, 1]],
              fill_color=(0, 0, 0, 0), line_color="black", line_width=2)
    p.legend.location = 'bottom_center'
    p.legend.orientation = 'horizontal'
    util.default_bokeh(p)
    show(p)
```

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

[2]:
```
# Load the data
data_path = "ex_data/phi065"
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]
```

Viewing our system

Before proceeding, we should probably view our system first. `freud` does not make any assumptions about your data and is not specifically designed for any one visualization package. Here we use bokeh to render our system. Bokeh is not appropriate for real-time interaction with your simulation data, nor is it appropriate for 3D data, but is perfectly fine for rendering individual simulation frames, so we will use it here

[3]:
```
# Grab data from last frame
l_box = box_data[-1].tolist()
l_pos = pos_data[-1]
```

(continues on next page)

(continued from previous page)

```
l_quat = quat_data[-1]
l_ang = 2*np.arctan2(np.copy(l_quat[:, 3]), np.copy(l_quat[:, 0]))

# Create box
fbox = box.Box.from_box(l_box)
side_length = max(fbox.Lx, fbox.Ly)
l_max = side_length / 2.0
l_max *= 1.1

# Take local vertices and rotate, translate into system coordinates
patches = util.local_to_global(verts, l_pos[:, :2], l_ang)

# Plot
p = figure(title="System Visualization",
           x_range=(-l_max, l_max),
           y_range=(-l_max, l_max))
p.patches(xs=patches[:, :, 0].tolist(), ys=patches[:, :, 1].tolist(),
          fill_color=(42, 126, 187), line_color="black", line_width=1.5, legend=
↪"Hexagons")
render_plot(p, fbox)
```

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

By eye, we can see regions where the hexagons appear close-packed, as well as regions where there are vacancies. We will be using the Nearest Neighbor object to investigate this in our system.

The Nearest Neighbor object

This module will give the indices of the \(k) particles which are nearest to another particle. `freud` provides two different modes by which to compute the nearest neighbors, selected by the `strict_cut` variable:

- `strict_cut=False` (*default*): The value for `rmax` is expanded until every particle has \(k) nearest neighbors
- `strict_cut=True`: the `rmax` value is not expanded, so that any "vacancies" in the number of neighbors found are filled with `UINTMAX`

strict_cut=False

First we show how to use the `strict_cut=False` mode to find the neighbors of a specific particle

[4]:
```
# Create freud nearest neighbor object
n_neigh = 6
nn = locality.NearestNeighbors(rmax=1.5, n_neigh=n_neigh, strict_cut=False)

# Compute nearest neighbors
nn.compute(fbox, l_pos, l_pos)
# Get the NeighborList
n_list = nn.nlist
# Get the neighbors for particle 0
pidx = 0
n_idxs = n_list.index_j[np.where(n_list.index_i == pidx)[0]]
```

(continues on next page)

(continued from previous page)

```
# Get position, orientation for the central particle
center_pos = l_pos[np.newaxis, pidx]
center_ang = l_ang[np.newaxis, pidx]

# Get the positions, orientations for the neighbor particles
neigh_pos = np.zeros(shape=(n_neigh, 3), dtype=np.float32)
neigh_ang = np.zeros(shape=(n_neigh), dtype=np.float32)
neigh_pos[:] = l_pos[n_idxs]
neigh_ang[:] = l_ang[n_idxs]

# Create array of transformed positions
c_patches = util.local_to_global(verts, center_pos[:, 0:2], center_ang)
n_patches = util.local_to_global(verts, neigh_pos[:, 0:2], neigh_ang)

# Create array of colors
center_color = np.array([c_list[0] for _ in range(center_pos.shape[0])])
neigh_color = np.array([c_list[-1] for _ in range(neigh_pos.shape[0])])

# Plot
p = figure(title="Nearest Neighbors Visualization",
           x_range=(-l_max, l_max), y_range=(-l_max, l_max))
p.patches(xs=n_patches[:, :, 0].tolist(), ys=n_patches[:, :, 1].tolist(),
          fill_color=neigh_color.tolist(), line_color="black", legend="Neighbors")
p.patches(xs=c_patches[:, :, 0].tolist(), ys=c_patches[:, :, 1].tolist(),
          fill_color=center_color.tolist(), line_color="black", legend="Center")
render_plot(p, fbox)
```

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

Notice that nearest neighbors properly handles periodic boundary conditions.

We do the same thing below, but for a particle/neighbors not spanning the box.

[5]:
```
# Get the neighbors for particle 1000
pidx = 1000
segment_start = n_list.segments[pidx]
segment_end = segment_start + n_list.neighbor_counts[pidx]
n_idxs = n_list.index_j[segment_start:segment_end]

# Get position, orientation for the central particle
center_pos = l_pos[np.newaxis, pidx]
center_ang = l_ang[np.newaxis, pidx]

# Get positions, orientations for the neighbors and one non-neighbor
neigh_pos = l_pos[n_idxs]
neigh_ang = l_ang[n_idxs]
non_neigh_pos = neigh_pos[np.newaxis, -1]
non_neigh_ang = neigh_ang[np.newaxis, -1]

# Create array of transformed positions
c_patches = util.local_to_global(verts, center_pos[:, :2], center_ang)
n_patches = util.local_to_global(verts, neigh_pos[:, :2], neigh_ang)
non_n_patches = util.local_to_global(verts, non_neigh_pos[:, :2], non_neigh_ang)

# Create array of colors
```

(continues on next page)

(continued from previous page)

```
center_color = np.array([c_list[0] for _ in range(center_pos.shape[0])])
neigh_color = np.array([c_list[-1] for _ in range(neigh_pos.shape[0])])

# Color the last particle differently
non_neigh_color = np.array([c_list[-2] for _ in range(non_neigh_pos.shape[0])])

# Plot
p = figure(title="Nearest Neighbors Visualization",
           x_range=(-l_max, l_max), y_range=(-l_max, l_max))
p.patches(xs=n_patches[:, :, 0].tolist(), ys=n_patches[:, :, 1].tolist(),
          fill_color=neigh_color.tolist(), line_color="black", legend="Neighbors")
p.patches(xs=non_n_patches[:, :, 0].tolist(), ys=non_n_patches[:, :, 1].tolist(),
          fill_color=non_neigh_color.tolist(), line_color="black", legend="Non-
↪neighbor")
p.patches(xs=c_patches[:, :, 0].tolist(), ys=c_patches[:, :, 1].tolist(),
          fill_color=center_color.tolist(), line_color="black", legend="Center")
render_plot(p, fbox)
```

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

Notice that while `freud` found the 6 nearest neighbors, one of the particles isn't really in a neighbor position (which we have colored purple). How do we go about finding particles with a deficit or surplus of neighbors?

strict_cut=True

Now for `strict_cut=True`. This mode allow you to find particles which have fewer than the specified number of particles. For this system, we'll search for 8 neighbors, so that we can display particles with both a deficit and a surplus of neighbors.

[6]:
```
# Create freud nearest neighbors object
n_neigh = 8
rmax = 1.7
nn = locality.NearestNeighbors(rmax=rmax, n_neigh=n_neigh, strict_cut=True)

# Compute nearest neighbors
nn.compute(fbox, l_pos, l_pos)
# Get the neighborlist
n_list = nn.nlist
# Get the number of particles
num_particles = nn.n_ref

p = figure(title="Nearest Neighbors visualization",
           x_range=(-l_max, l_max), y_range=(-l_max, l_max))
for k in np.unique(n_list.neighbor_counts):
    # Find particles with k neighbors
    c_idxs = np.where(n_list.neighbor_counts == k)[0]
    center_pos = l_pos[c_idxs]
    center_ang = l_ang[c_idxs]
    c_patches = util.local_to_global(verts, center_pos[:, 0:2], center_ang)
    center_color = np.array([c_dict[k] for _ in range(center_pos.shape[0])])
    p.patches(xs=c_patches[:, :, 0].tolist(), ys=c_patches[:, :, 1].tolist(),
              fill_color=center_color.tolist(), line_color="black", legend="k={}".
↪format(k))
```

(continues on next page)

(continued from previous page)

```
render_plot(p, fbox)
```

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

Visualize each set of k values independently

[7]:
```
for k in np.unique(n_list.neighbor_counts):
    p = figure(title="Nearest Neighbors: k={}".format(k),
               x_range=(-l_max, l_max), y_range=(-l_max, l_max))
    # Find particles with k neighbors
    c_idxs = np.where(n_list.neighbor_counts == k)[0]
    center_pos = l_pos[c_idxs]
    center_ang = l_ang[c_idxs]
    c_patches = util.local_to_global(verts, center_pos[:, 0:2], center_ang)
    patches = util.local_to_global(verts, l_pos[:, 0:2], l_ang)
    center_color = np.array([c_dict[k] for _ in range(center_pos.shape[0])])
    p.patches(xs=patches[:, :, 0].tolist(), ys=patches[:, :, 1].tolist(),
              fill_color=(128, 128, 128, 0.1), line_color=(0, 0, 0, 0.1), legend=
↪"other")
    p.patches(xs=c_patches[:, :, 0].tolist(), ys=c_patches[:, :, 1].tolist(),
              fill_color=center_color.tolist(), line_color="black", legend="k={}".
↪format(k))
    render_plot(p, fbox)
```

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

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

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

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

[8]:
```
for k in np.unique(n_list.neighbor_counts):
    p = figure(title="Nearest Neighbors: k={}".format(k),
               x_range=(-l_max, l_max), y_range=(-l_max, l_max))
    # Find particles with k neighbors
    c_idxs = np.copy(np.where(n_list.neighbor_counts == k)[0])
    center_pos = l_pos[c_idxs]
```

(continues on next page)

(continued from previous page)

```
    center_ang = l_ang[c_idxs]

    neigh_pos = np.zeros(shape=(k*len(c_idxs), 3), dtype=np.float32)
    neigh_ang = np.zeros(shape=(k*len(c_idxs)), dtype=np.float32)
    for i, pidx in enumerate(c_idxs):
        # Create a list of positions, angles to draw
        segment_start = n_list.segments[pidx]
        segment_end = segment_start + n_list.neighbor_counts[pidx]
        n_idxs = n_list.index_j[segment_start:segment_end]
        for j, nidx in enumerate(n_idxs):
            neigh_pos[k*i+j] = l_pos[nidx]
            neigh_ang[k*i+j] = l_ang[nidx]
    c_patches = util.local_to_global(verts, center_pos[:, 0:2], center_ang)
    n_patches = util.local_to_global(verts, neigh_pos[:, 0:2], neigh_ang)
    patches = util.local_to_global(verts, l_pos[:, 0:2], l_ang)
    center_color = np.array([c_dict[k] for _ in range(center_pos.shape[0])])
    neigh_color = np.array([c_list[-1] for _ in range(neigh_pos.shape[0])])
    p.patches(xs=patches[:, :, 0].tolist(), ys=patches[:, :, 1].tolist(),
              fill_color=(128, 128, 128, 0.1), line_color=(0, 0, 0, 0.1), legend=
↪"other")
    p.patches(xs=n_patches[:, :, 0].tolist(), ys=n_patches[:, :, 1].tolist(),
              fill_color=neigh_color.tolist(), line_color="black", legend="neighbors")
    p.patches(xs=c_patches[:, :, 0].tolist(), ys=c_patches[:, :, 1].tolist(),
              fill_color=center_color.tolist(), line_color="black", legend="k={}".
↪format(k))
    render_plot(p, fbox)
```

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

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

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

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

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

Compare against a solid/crystalline system

Visualize in the same way, but with a solid system. Notice that almost all the hexagons have 6 nearest neighbors.

[9]:
```
data_path = "ex_data/phi075"
box_data = np.load("{}/box_data.npy".format(data_path))
```

(continues on next page)

(continued from previous page)

```
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]

# Grab data from last frame
l_box = box_data[-1].tolist()
l_pos = pos_data[-1]
l_quat = quat_data[-1]
l_ang = 2*np.arctan2(np.copy(l_quat[:, 3]), np.copy(l_quat[:, 0]))

# Create box
fbox = box.Box.from_box(l_box)
side_length = max(fbox.Lx, fbox.Ly)
l_max = side_length / 2.0
l_max *= 1.1

# Create freud nearest neighbor object
n_neigh = 8
rmax = 1.65
nn = locality.NearestNeighbors(rmax=rmax, n_neigh=n_neigh, strict_cut=True)

# Compute nearest neighbors
nn.compute(fbox, l_pos, l_pos)
# Get the neighborlist
n_list = nn.nlist

p = figure(title="Nearest Neighbors Visualization",
           x_range=(-l_max, l_max), y_range=(-l_max, l_max))
for k in np.unique(n_list.neighbor_counts):
    # find particles with k neighbors
    c_idxs = np.where(n_list.neighbor_counts == k)[0]
    center_pos = l_pos[c_idxs]
    center_ang = l_ang[c_idxs]
    c_patches = util.local_to_global(verts, center_pos[:, 0:2], center_ang)
    center_color = np.array([c_dict[k] for _ in range(center_pos.shape[0])])
    p.patches(xs=c_patches[:, :, 0].tolist(), ys=c_patches[:, :, 1].tolist(),
              fill_color=center_color.tolist(), line_color="black", legend="k={}".
↪format(k))
render_plot(p, fbox)
```

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

1.2.2 Module Examples

The remaining examples go into greater detail on the functionality of specific modules, showing how they can be used to perform specific types of analyses of simulations.

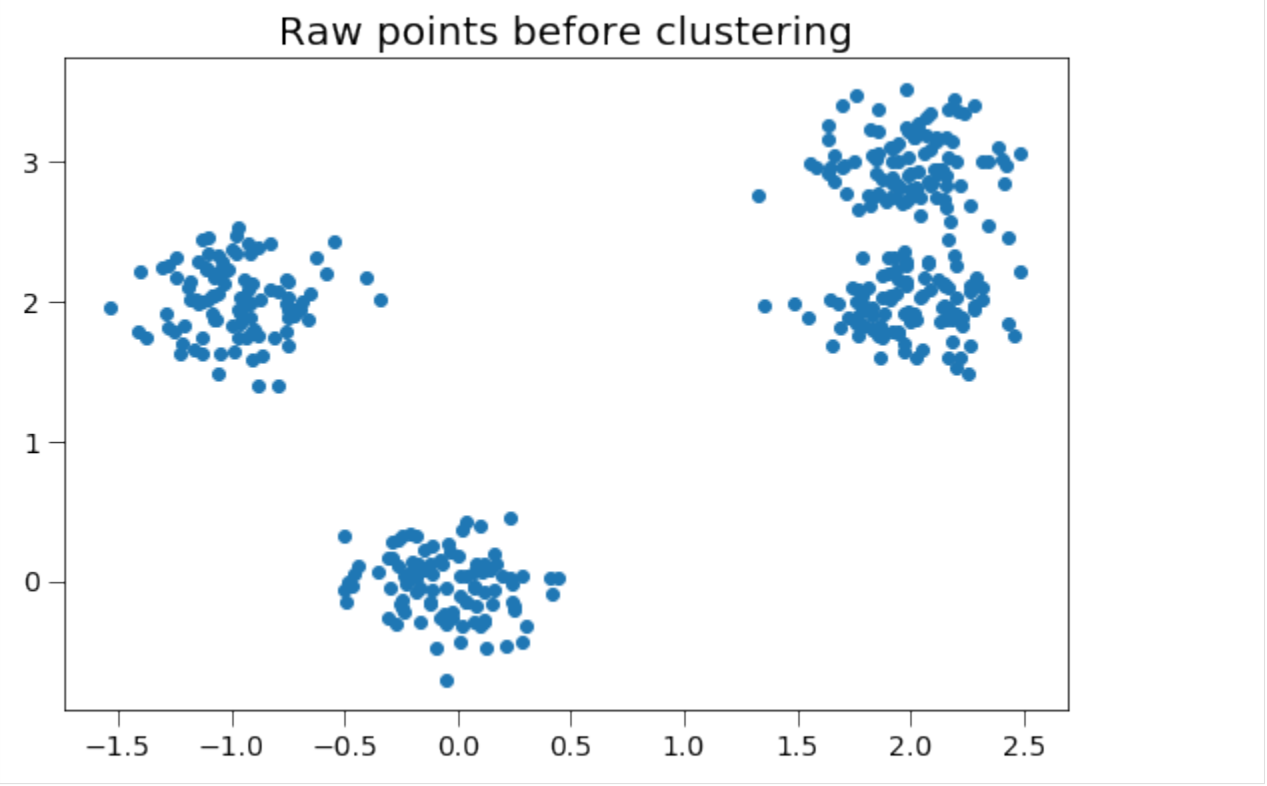
Cluster

The cluster module uses a set of coordinates and a cutoff distance to determine clustered points. 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 random points to cluster.

[2]:
```
points = np.empty(shape=(0, 2))
for center_point in [(0, 0), (2, 2), (-1, 2), (2, 3)]:
    points = np.concatenate(
        (points, np.random.multivariate_normal(mean=center_point, cov=0.05*np.eye(2),␣
↪size=(100,))))
fig, ax = plt.subplots(1, 1, figsize=(9, 6))
ax.scatter(points[:,0], points[:,1])
ax.set_title('Raw points before clustering', fontsize=20)
ax.tick_params(axis='both', which='both', labelsize=14, size=8)
plt.show()

# We must add a z=0 component to this array for freud
points = np.hstack((points, np.zeros((points.shape[0], 1))))
```



Now we create a box and a cluster compute object.

[3]:
```
box = freud.box.Box.square(L=10)
cl = freud.cluster.Cluster(box=box, rcut=1.0)
```

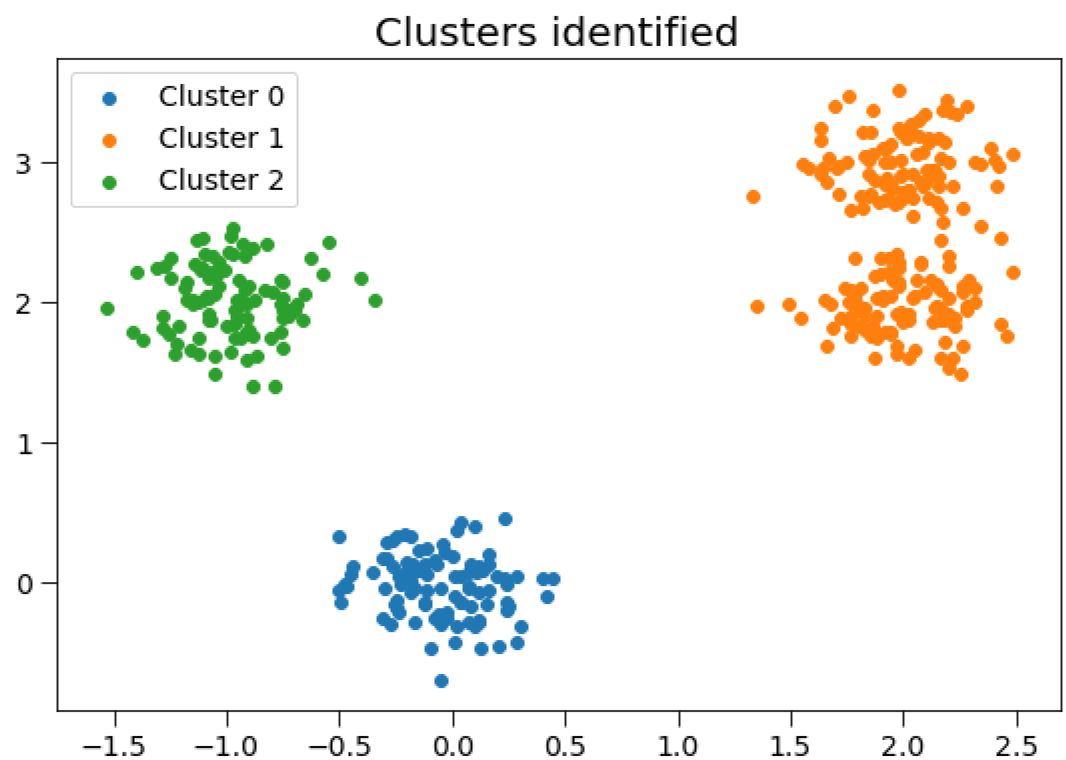
Next, we use the `computeClusters` method to determine clusters and the `clusterIdx` property to return their identities. Note that we use `freud`'s *method chaining* here, where a compute method returns the compute object.

[4]:
```
cluster_idx = cl.computeClusters(points).cluster_idx
print(cluster_idx)
```

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

[5]:
```python
fig, ax = plt.subplots(1, 1, figsize=(9, 6))
for cluster_id in range(cl.num_clusters):
    cluster_point_indices = np.where(cluster_idx == cluster_id)[0]
    ax.scatter(points[cluster_point_indices, 0], points[cluster_point_indices, 1],␣
↪label="Cluster {}".format(cluster_id))
    print("There are {} points in cluster {}.".format(len(cluster_point_indices),␣
↪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 100 points in cluster 0.
There are 200 points in cluster 1.
There are 100 points in cluster 2.
```



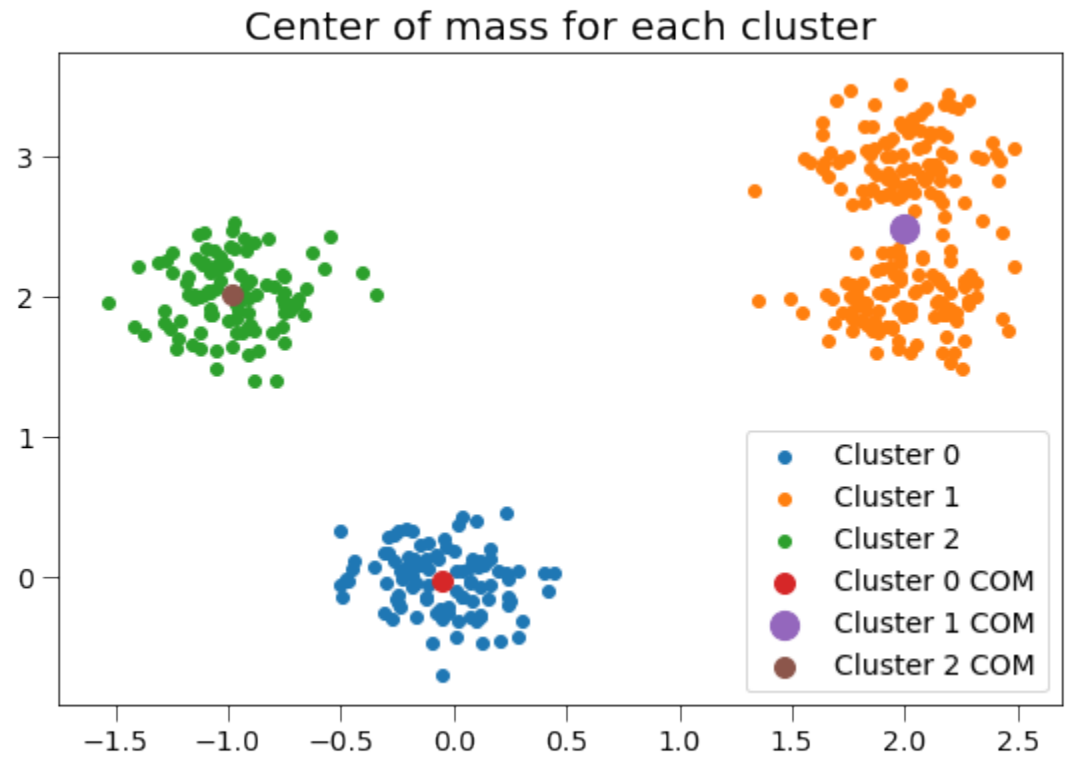
We may also compute the clusters' centers of mass and gyration tensor using the `ClusterProperties` class.

[6]:
```
clp = freud.cluster.ClusterProperties(box=box)
clp.computeProperties(points, cl.cluster_idx)
```

[6]:
```
<freud.cluster.ClusterProperties at 0x7fe984f50198>
```

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

[7]:
```
plt.figure(figsize=(9, 6))
for cluster_id in range(cl.num_clusters):
    cluster_point_indices = np.where(cluster_idx == cluster_id)[0]
    plt.scatter(points[cluster_point_indices, 0], points[cluster_point_indices, 1],
                label="Cluster {}".format(cluster_id))
for cluster_id in range(cl.num_clusters):
    cluster_point_indices = np.where(cluster_idx == cluster_id)[0]
    plt.scatter(clp.cluster_COM[cluster_id, 0], clp.cluster_COM[cluster_id, 1],
                s=len(cluster_point_indices),
                label="Cluster {} COM".format(cluster_id))
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.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.

[8]:
```
for cluster_id in range(cl.num_clusters):
    G = clp.cluster_G[cluster_id]
    print("The gyration tensor of cluster {} is:\n{}".format(cluster_id, G))
```

```
The gyration tensor of cluster 0 is:
[[ 0.04618965 -0.00847244  0.        ]
```

(continues on next page)

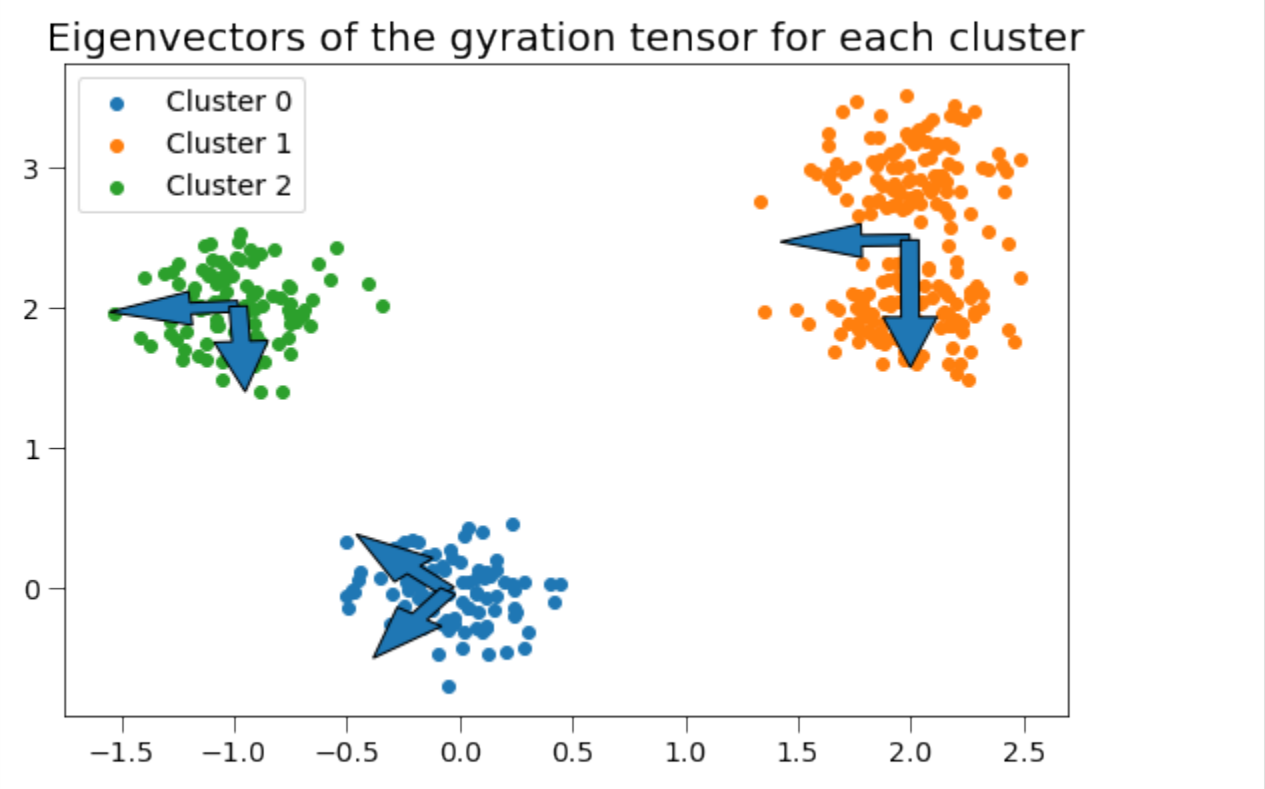
(continued from previous page)

```
 [-0.00847244  0.04932979  0.        ]
 [ 0.          0.          0.        ]]
The gyration tensor of cluster 1 is:
[[0.04560551 0.00232111 0.        ]
 [0.00232111 0.3034907  0.        ]
 [0.         0.         0.        ]]
The gyration tensor of cluster 2 is:
[[0.04483054 0.00113422 0.        ]
 [0.00113422 0.06339595 0.        ]
 [0.         0.         0.        ]]
```

This 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).

[9]:
```
plt.figure(figsize=(9, 6))
for cluster_id in range(cl.num_clusters):
    cluster_point_indices = np.where(cluster_idx == cluster_id)[0]
    plt.scatter(points[cluster_point_indices, 0], points[cluster_point_indices, 1],␣
↪label="Cluster {}".format(cluster_id))
for cluster_id in range(cl.num_clusters):
    com = clp.cluster_COM[cluster_id]
    G = clp.cluster_G[cluster_id]
    evals, evecs = np.linalg.eig(G)
    radii = np.sqrt(evals)
    for evalue, evec in zip(evals[:2], evecs[:2, :2]):
        print("Cluster {} has radius of gyration {:.4f} along the axis of ({:.4f}, {:.
↪4f}).".format(cluster_id, evalue, *evec))
    arrows = (radii * evecs)[:2, :2]
    for arrow in arrows:
        plt.arrow(com[0], com[1], arrow[0], arrow[1], width=0.08)
plt.title('Eigenvectors of the gyration tensor for each cluster', fontsize=20)
plt.legend(loc='best', fontsize=14)
plt.gca().tick_params(axis='both', which='both', labelsize=14, size=8)
```

```
Cluster 0 has radius of gyration 0.0391 along the axis of (-0.7688, 0.6394).
Cluster 0 has radius of gyration 0.0564 along the axis of (-0.6394, -0.7688).
Cluster 1 has radius of gyration 0.0456 along the axis of (-1.0000, -0.0090).
Cluster 1 has radius of gyration 0.3035 along the axis of (0.0090, -1.0000).
Cluster 2 has radius of gyration 0.0448 along the axis of (-0.9982, -0.0608).
Cluster 2 has radius of gyration 0.0635 along the axis of (0.0608, -0.9982).
```



Density - ComplexCF

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
```

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

[2]:
```
def plot_data(title, points, angles, values, box, ccf, s=200):
    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=angle, vmin=-np.pi/4, vmax=np.pi/4, 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')
    plt.colorbar()
    plt.subplot(122)
    plt.title('Orientation Spatial Autocorrelation Function')
    plt.plot(ccf.R, np.real(ccf.RDF))
```

(continues on next page)

(continued from previous page)

```
    plt.xlabel(r'$r$')
    plt.ylabel(r'$C(r)$')
    plt.show()
```

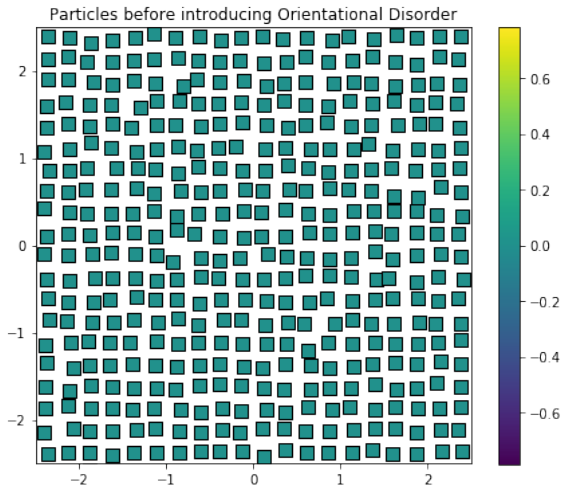
First, let's generate a 2D structure with perfect orientational order and slight positional disorder (the particles are not perfectly on a grid, but they are perfectly aligned). The color of the particles corresponds to their angle of rotation, so all the particles will be the same color to begin with.

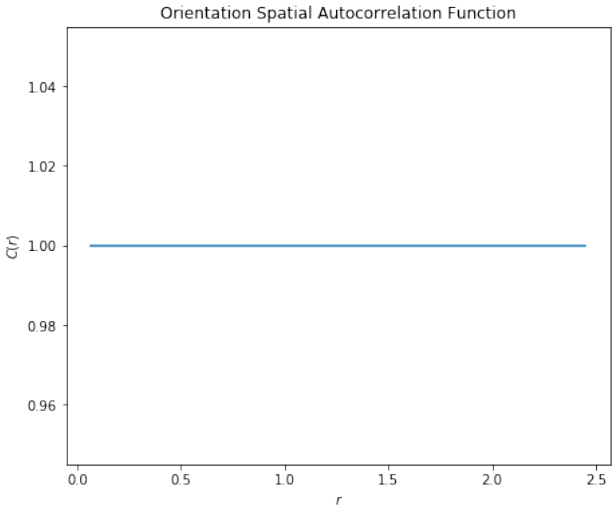
We create a `freud.density.ComplexCF` object to compute the correlation functions. Given a particle orientation θ, we compute its complex orientation value (the quantity we are correlating) as $s = e^{4i\theta}$, to account for the fourfold symmetry of the particles. We will compute the correlation function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

When we compute the correlations between particles, we must use the complex conjugate of the `values` array for one of the arguments. 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):
    # Initialize a box and particle spacing
    box = freud.box.Box.square(L=L)
    dx = box.Lx/repeats

    # Generate points and add random positional noise
    points = np.array([[i, j, 0] for j in np.arange(-(box.Ly+dx)/2, (box.Ly+dx)/2, dx)
                       for i in np.arange(-(box.Lx+dx)/2, (box.Lx+dx)/2, dx)])
    cov = 1e-4*box.Lx*np.eye(3)
    cov[2, 2] = 0
    points += np.random.multivariate_normal(mean=np.zeros(3), cov=cov,␣
↪size=len(points))
    return box, points

# 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 ComplexCF compute object and compute the correlation function
ccf = freud.density.ComplexCF(rmax=box.Lx/2, dr=box.Lx/50)
ccf.compute(box, points, np.conj(values), points, values)

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

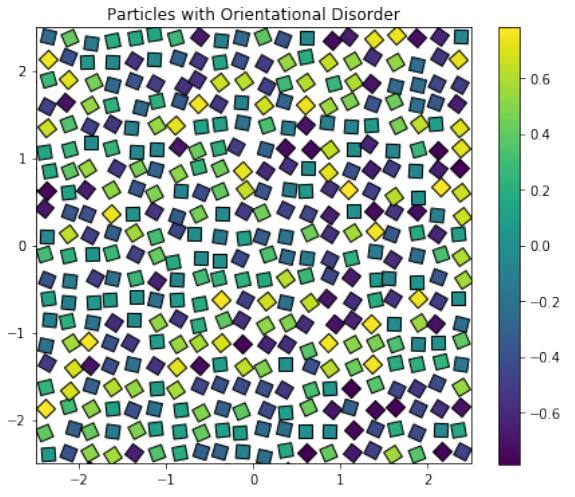


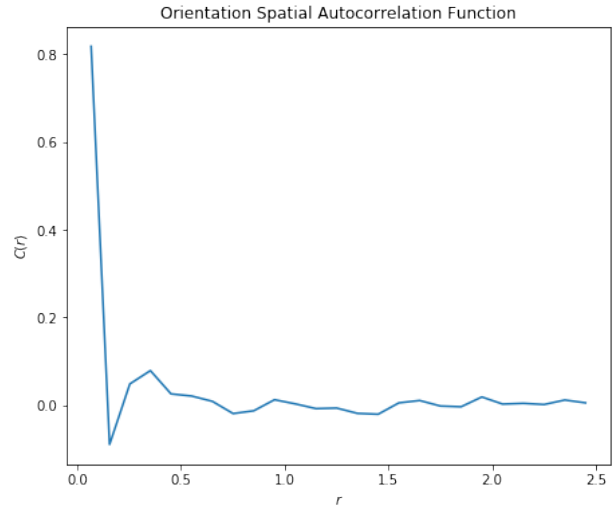


Now we will generate random angles from $-\frac{\pi}{4}$ to $\frac{\pi}{4}$, which orients our squares randomly. The four-fold symmetry of the squares means that the space of unique angles is restricted to a range of $\frac{\pi}{2}$. Again, we compute a complex value for each particle, $s = e^{4i\theta}$.

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.

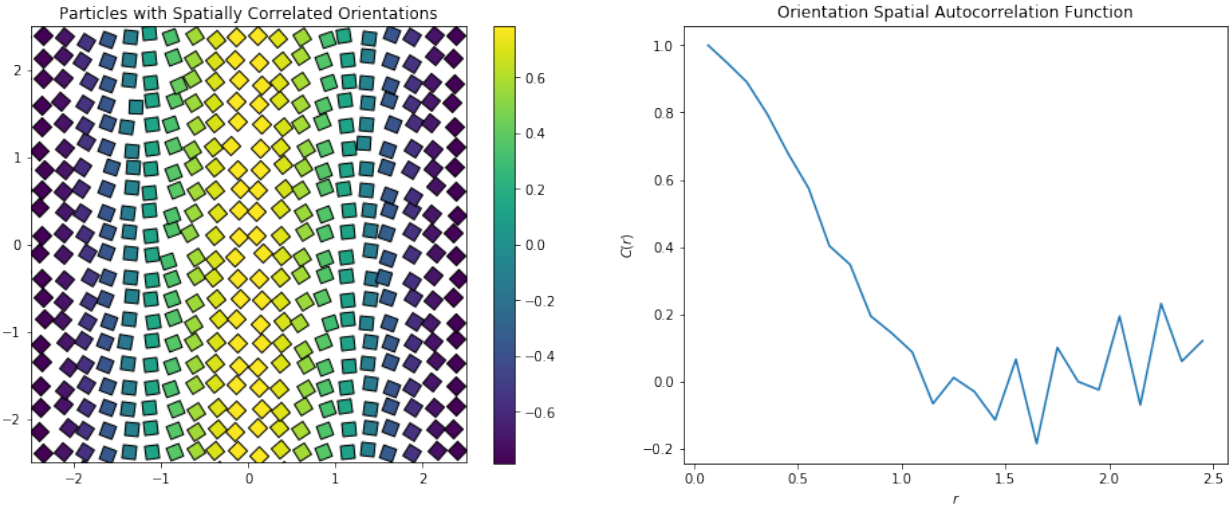
[4]:
```
# Change the angles to values randomly drawn from a uniform distribution
angles = np.random.uniform(-np.pi/4, np.pi/4, size=len(points))
values = np.exp(angles * 4j)

# Recompute the correlation functions
ccf.compute(box, points, np.conj(values), points, values)

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





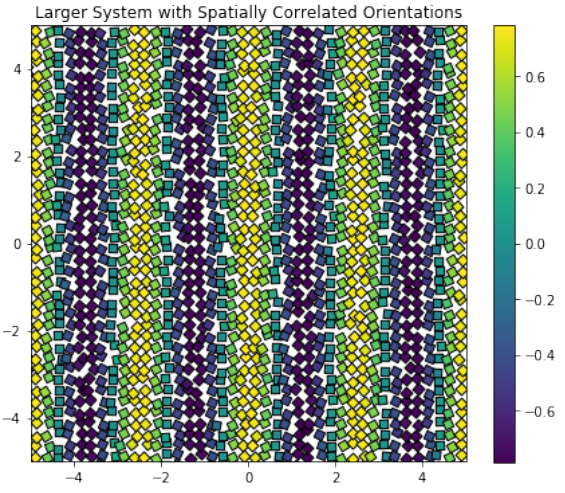
The plot below shows what happens when we intentionally introduce a correlation length by adding a spatial pattern to the particle orientations. At short distances, the correlation is very high. As r increases, the oppositely-aligned part of the pattern some distance away causes the correlation to drop.

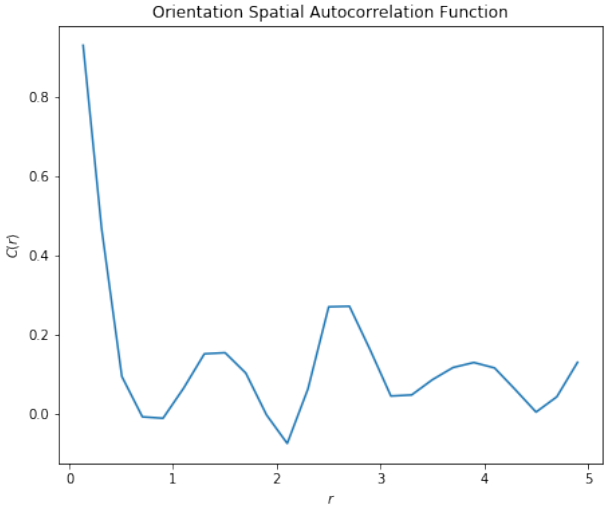
[5]:
```
# Use angles that vary spatially in a pattern
angles = np.pi/4 * np.cos(2*np.pi*points[:, 0]/box.Lx)
values = np.exp(angles * 4j)

# Recompute the correlation functions
ccf.compute(box, points, np.conj(values), points, values)

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



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 ComplexCF compute object
ccf = freud.density.ComplexCF(rmax=box.Lx/2, dr=box.Lx/50)
ccf.compute(box, points, np.conj(values), points, values)

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





FloatCF

Grain Size Determination

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 approximate the grain size within a system.

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

To show the correlation function, we need a pretend data set. We start with a phase separated Ising lattice and assign type values, either **blue (-1)** or **red (+1)**, to generate "grains."

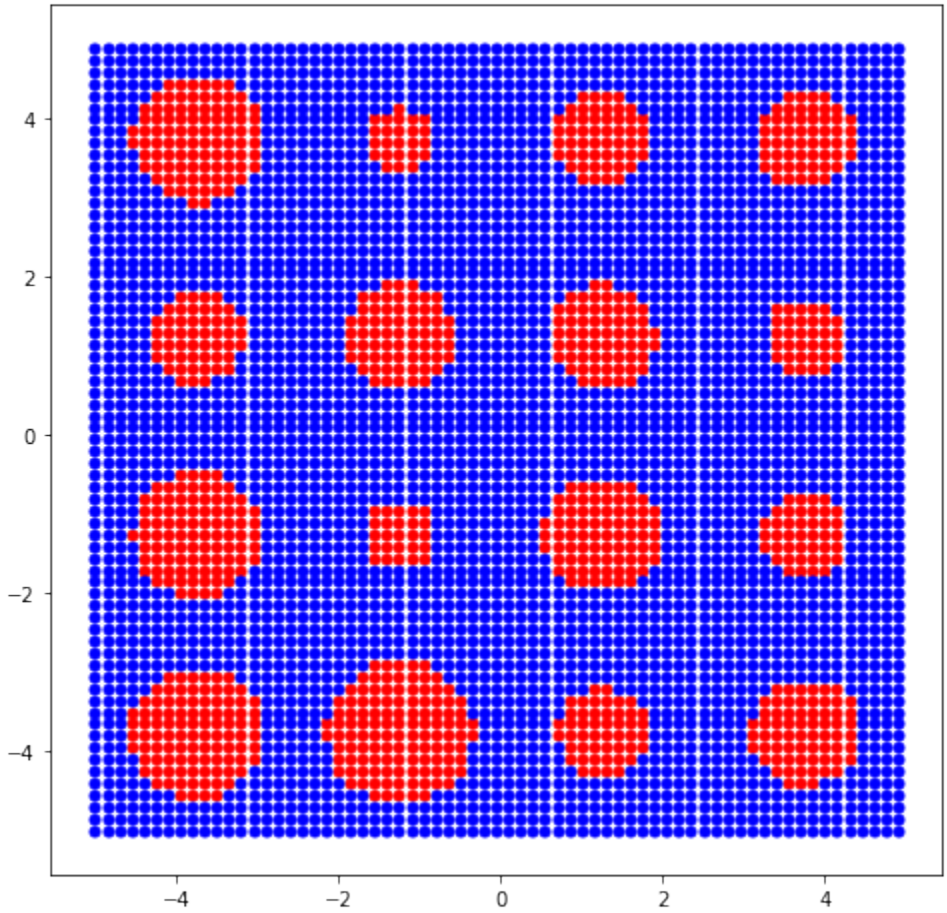
To make the pretend data set, we create a large number of **blue (-1)** particles on a square grid. Then we place grain centers on a larger grid and draw grain radii from a normal distribution. We color the particles **red (+1)** if their distance from a grain center is less than the grain radius.

[2]:
```
# Set up the system
box = freud.box.Box.square(L=10)
dx = 0.15
num_grains = 4
dg = box.Lx/num_grains
points = np.array([[i, j, 0]
                   for j in np.arange(-box.Ly/2, box.Ly/2, dx)
                   for i in np.arange(-box.Lx/2, box.Lx/2, dx)])
values = np.array([-1 for p in points])
centroids = [[i*dg + 0.5*dg, j*dg + 0.5*dg, 0]
             for i in range(num_grains) for j in range(num_grains)]
grain_radii = np.abs(np.random.normal(size=num_grains**2, loc=0.25*dg, scale=0.05*dg))
for center, radius in zip(centroids, grain_radii):
    lc = freud.locality.LinkCell(box, radius).compute(box, points, [center])
    for i in lc.nlist.index_i:
        values[i] = 1
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(8, 8))
plt.scatter(points[values > 0, 0],
            points[values > 0, 1],
            marker='o', color='red', s=25)
plt.scatter(points[values < 0, 0],
            points[values < 0, 1],
            marker='o', color='blue', s=25)
plt.show()
```



This system is **phase-separated** because the red particles are generally near one another, and so are the blue particles. However, there is some **length scale** at which the phase separation is no longer visible. Imagine looking at this image from a far distance away: the red and blue regions would be indistinguishable, and the picture would appear purple.

We can use the correlation lengths between the grain centers and the surrounding red and blue particles to find where the grains end. First, we need to locate the centers of the grains using `freud.cluster`.

[3]:
```
cl = freud.cluster.Cluster(box=box, rcut=2*dx)
cluster_idx = cl.computeClusters(points[values > 0, :]).cluster_idx
```

Now we'll find the cluster centroids (the grain centers).

[4]:
```
clp = freud.cluster.ClusterProperties(box=box)
clp.computeProperties(points[values > 0, :], cl.cluster_idx)
```

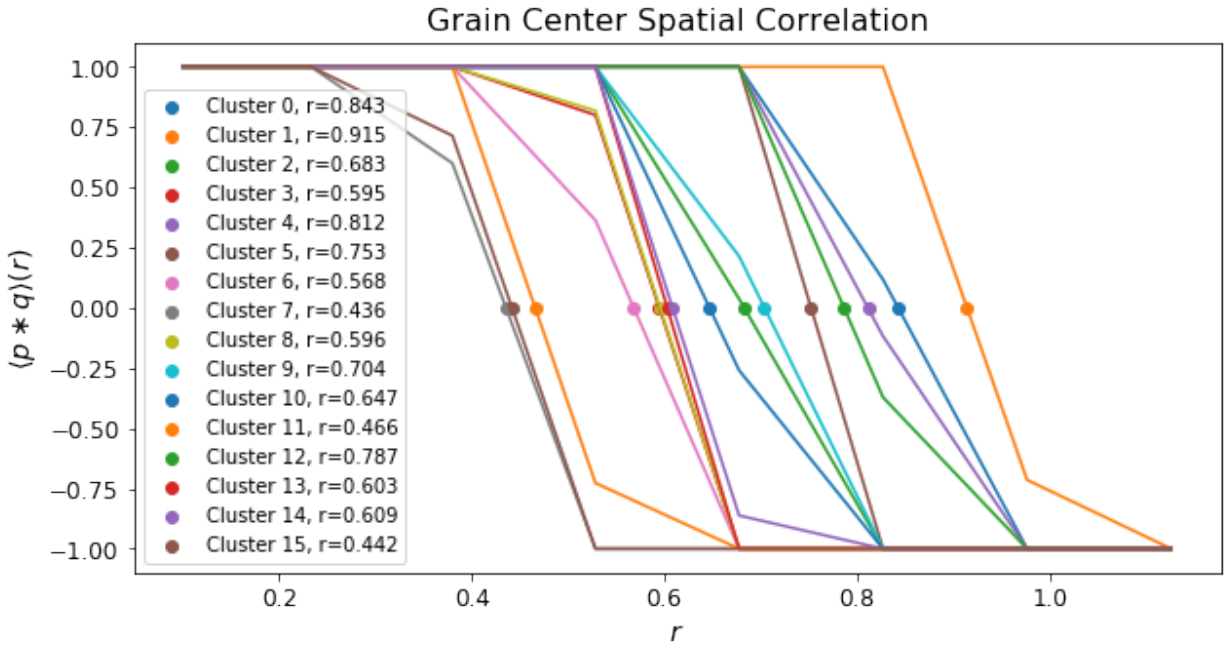
[4]:
```
<freud.cluster.ClusterProperties at 0x1147de550>
```

Now we create a `freud.density.FloatCF` compute object, to compute the correlation function.

[5]:
```
fcf = freud.density.FloatCF(rmax=0.5*box.Lx/num_grains, dr=dx)
fcf.reset()
```

Now we compute the product of type values. Knowing that two sites with the same type will have a product of 1, we can estimate the radius of a grain as the smallest value of r such that $\langle p * q \rangle(r) < 0$, where the set p represents our red and blue particles and q represents the grain centers.

[6]:
```
plt.figure(figsize=(10, 5))
weights = []
sizes = []
measured_grain_radii = []
for cluster_id in range(cl.num_clusters):
    fcf.compute(box=box,
                ref_points=points,  # all points in the system
                ref_values=values,   # all types in the system
                points=[clp.cluster_COM[cluster_id]],
                values=[1])
    # get the center of the histogram bins
    r = fcf.R
    # get the value of the histogram bins
    y = fcf.RDF
    grainsize = y[y > 0][-1] * (r[y < 0][0]-r[y > 0][-1]) / \
        (y[y > 0][-1]-y[y < 0][0]) + r[y > 0][-1]
    measured_grain_radii.append(grainsize)

    plt.plot(r, y)
    plt.scatter(x=grainsize, y=0, label='Cluster {}, r={}'.format(
        cluster_id, round(grainsize, 3)))

plt.title("Grain Center Spatial Correlation", fontsize=16)
plt.xlabel(r"$r$", fontsize=14)
plt.ylabel(r"$\langle p \ast q \rangle (r)$", fontsize=14)
plt.legend()
plt.tick_params(which="both", axis="both", labelsize=12)
plt.show()
```



This table shows the grain radii we expected compared to the ones we computed.

[7]:
```
print('Actual Radius\tMeasured Radius')
for actual, measured in zip(sorted(grain_radii), sorted(measured_grain_radii)):
    print('{:.4f}\t\t{:.4f}'.format(actual, measured))
```

```
Actual Radius   Measured Radius
0.4255          0.4357
0.4384          0.4419
0.5010          0.4660
0.5462          0.5684
0.5692          0.5949
0.5988          0.5957
0.6055          0.6032
0.6366          0.6087
0.6605          0.6471
0.6752          0.6831
0.6959          0.7042
0.7234          0.7525
0.8065          0.7868
0.8141          0.8115
0.8491          0.8430
0.9105          0.9146
```

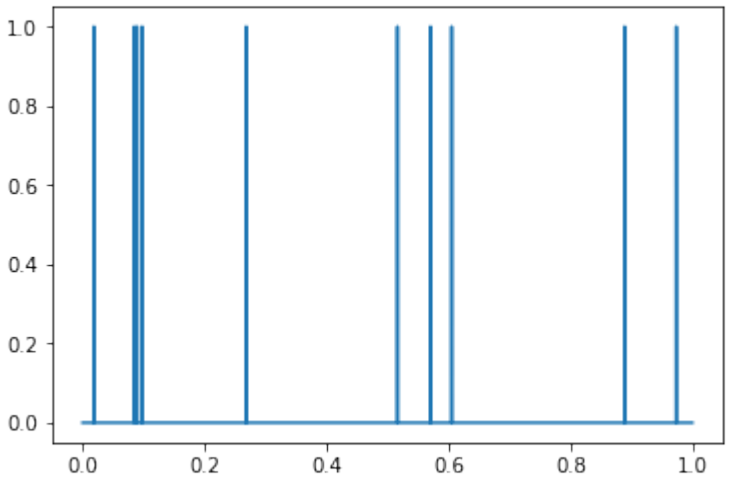
GaussianDensity

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
```

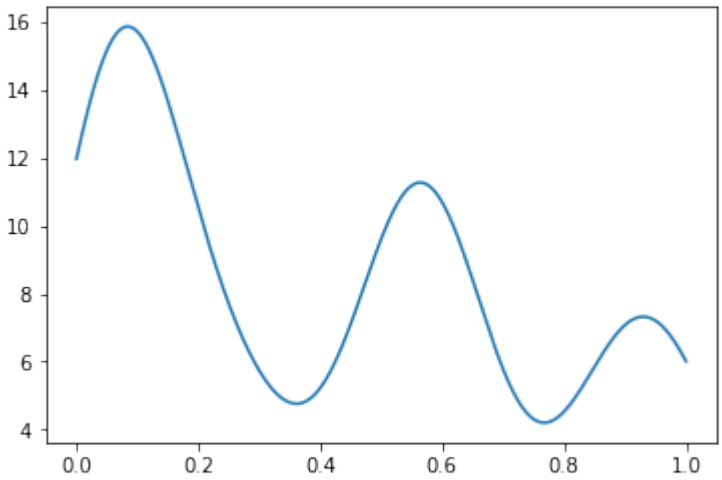
To illustrate the basic concept, consider a toy example: a simple set of point particles with unit mass on a line. For analytical purposes, the standard way to accomplish this would be using Dirac delta functions.

[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()
```



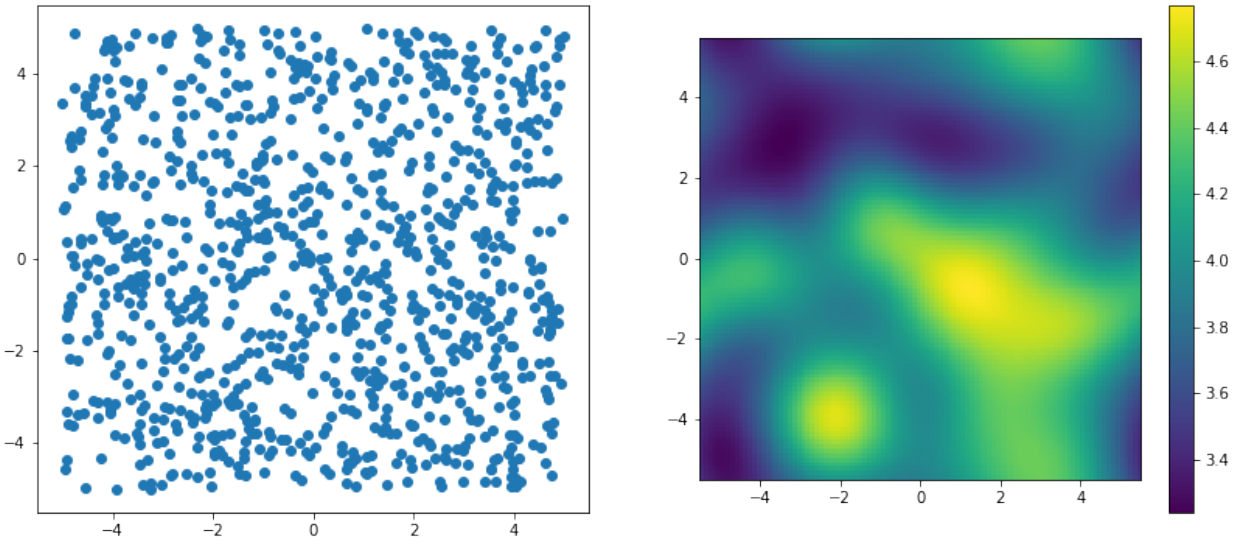
However, delta functions can be cumbersome to work with, so we might instead want to smooth out these particles. One option is to instead represent particles as Gaussians centered at the location of the points. In that case, the total particle density at any point in the interval $[0, 1]$ represented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



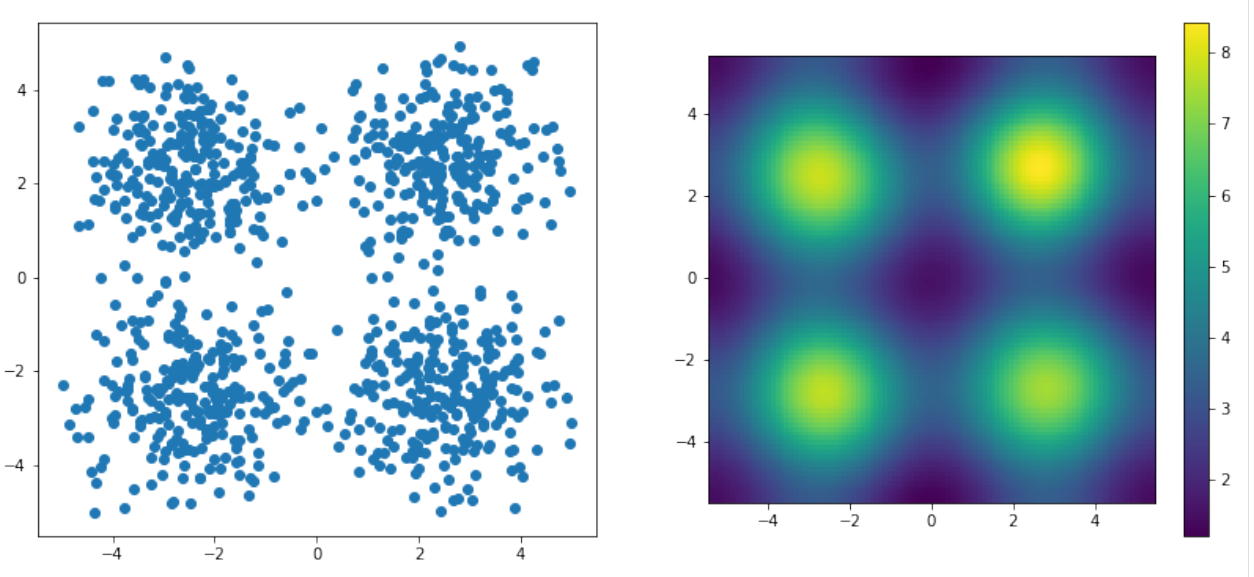
The goal of the GaussianDensity class is to perform the same interpolation for points on a 2D or 3D grid, accounting for Box periodicity.

[4]:
```
N = 1000  # Number of points
L = 10  # Box length
box = freud.box.Box.square(L)
points = box.wrap(np.random.rand(N, 3)*L)
points[:, 2] = 0
gd = freud.density.GaussianDensity(L*L, L/3, 1)
gd.compute(box, points)

fig, axes = plt.subplots(1, 2, figsize=(14, 6))
axes[0].scatter(points[:, 0], points[:, 1])
im = axes[1].imshow(gd.gaussian_density, origin='lower',
                    extent=axes[0].get_xlim() + axes[0].get_ylim())
plt.colorbar(im);
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.eye(3, 3), size=(int(N/
↪4),)))
points = box.wrap(np.concatenate(points))
points[:, 2] = 0

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

fig, axes = plt.subplots(1, 2, figsize=(14, 6))
axes[0].scatter(points[:, 0], points[:, 1])
im = axes[1].imshow(gd.gaussian_density, origin='lower',
                    extent=axes[0].get_xlim() + axes[0].get_ylim())
plt.colorbar(im);
plt.show()
```



LocalDensity

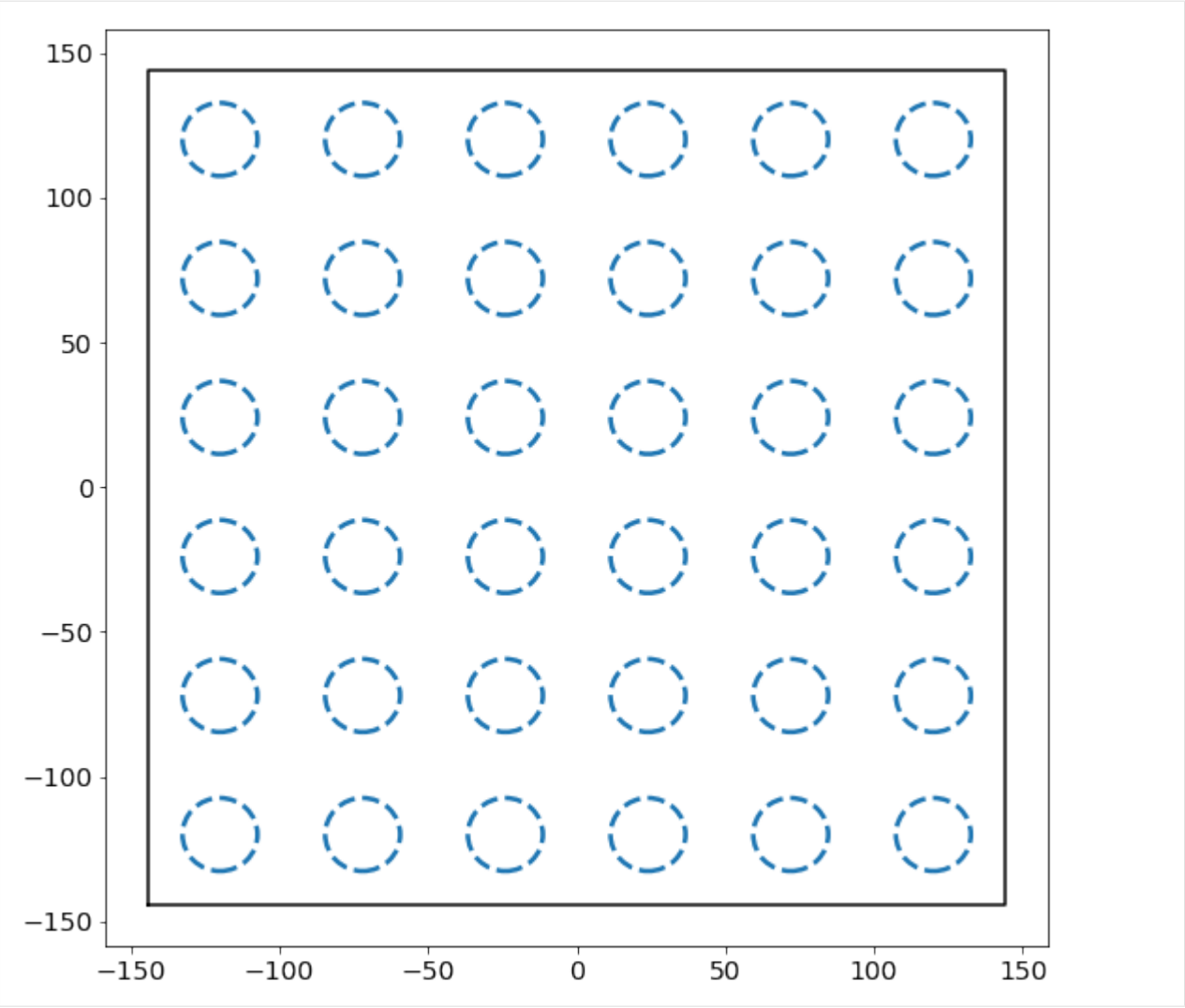
The `freud.density` module is intended to compute a variety of quantities that relate spatial distributions of particles with other particles. In this notebook, we demonstrate `freud`'s local density calculation, which can be used to characterize the particle distributions in some systems. In this example, 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 util
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_points = util.box_2d_to_points(box)
    ax.plot(box_points[:, 0], box_points[:, 1], color='k')

    add_patches(ax, points, radius, ls=ls, lw=lw)
    return fig, ax
```

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

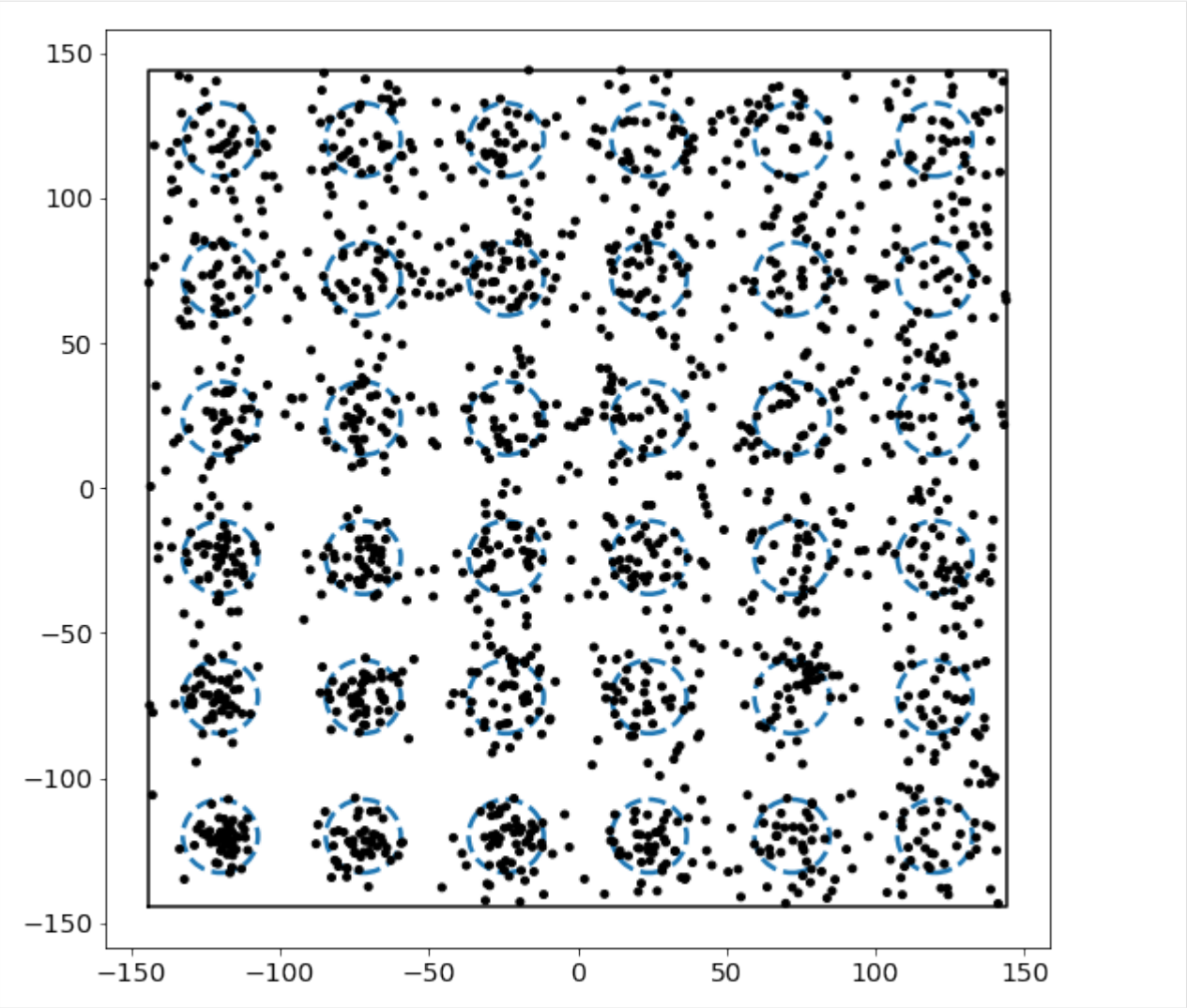
[3]:
```
area = 5
radius = np.sqrt(area/np.pi)
ref_area = area*100
ref_radius = np.sqrt(ref_area/np.pi)
num = 6
scale = num*4
box, ref_points = util.make_square(num, num, scale=scale)
ref_points[..., [0, 1]] += scale
fig, ax = plot_lattice(box, ref_points, ref_radius, ls="dashed", lw=2.5)
plt.tick_params(axis="both", 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 = []
distances = np.linalg.norm(ref_points + np.array(box.L)/2, axis=-1)
cov_basis = 20*distances/np.min(distances)
for i, p in enumerate(ref_points):
    cov = cov_basis[i]*np.eye(3)
    cov[2, 2] = 0  # Nothing in z
    points.append(
        np.random.multivariate_normal(p, cov, size=(50,)))
points = box.wrap(np.concatenate(points))
```

[5]:
```
fig, ax = plot_lattice(box, ref_points, ref_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 increases 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 reference points around which the number of other points is computed. These other points can, but need not be, distinct from the reference points. In our case, we want to use the regions as our reference points with the small circles as our data points.

When we construct the `LocalDensity` class, there are three arguments. The first is the radius from the reference points within which particles should be included in the reference point's counter. The second and third are the volume and the circumsphere diameters of the **data points**, not the reference points. This distinction is critical for getting appropriate density values, since these values are used to actually check cutoffs and calculate the density.

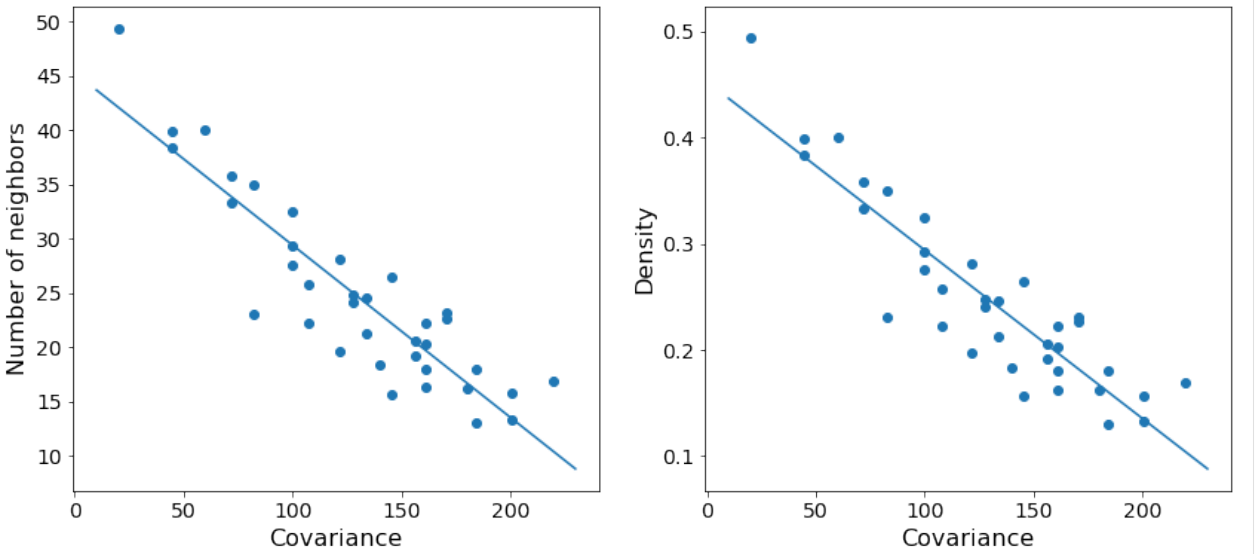
[6]:
```
density = freud.density.LocalDensity(ref_radius, area, radius)
num_neighbors = density.compute(box, ref_points, points).num_neighbors
densities = density.compute(box, ref_points, points).density
```

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

for i, data in enumerate([num_neighbors, densities]):
    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)
```

(continues on next page)

(continued from previous page)

```
    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 reference 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.

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
from util import box_2d_to_points
import matplotlib.pyplot as plt

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

def plot_rdf(box_arr, points_arr, prop, rmax=10, dr=0.1, 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(rmax, dr)
    for box, points in zip(box_arr, points_arr):
        rdf.accumulate(box, points)
```

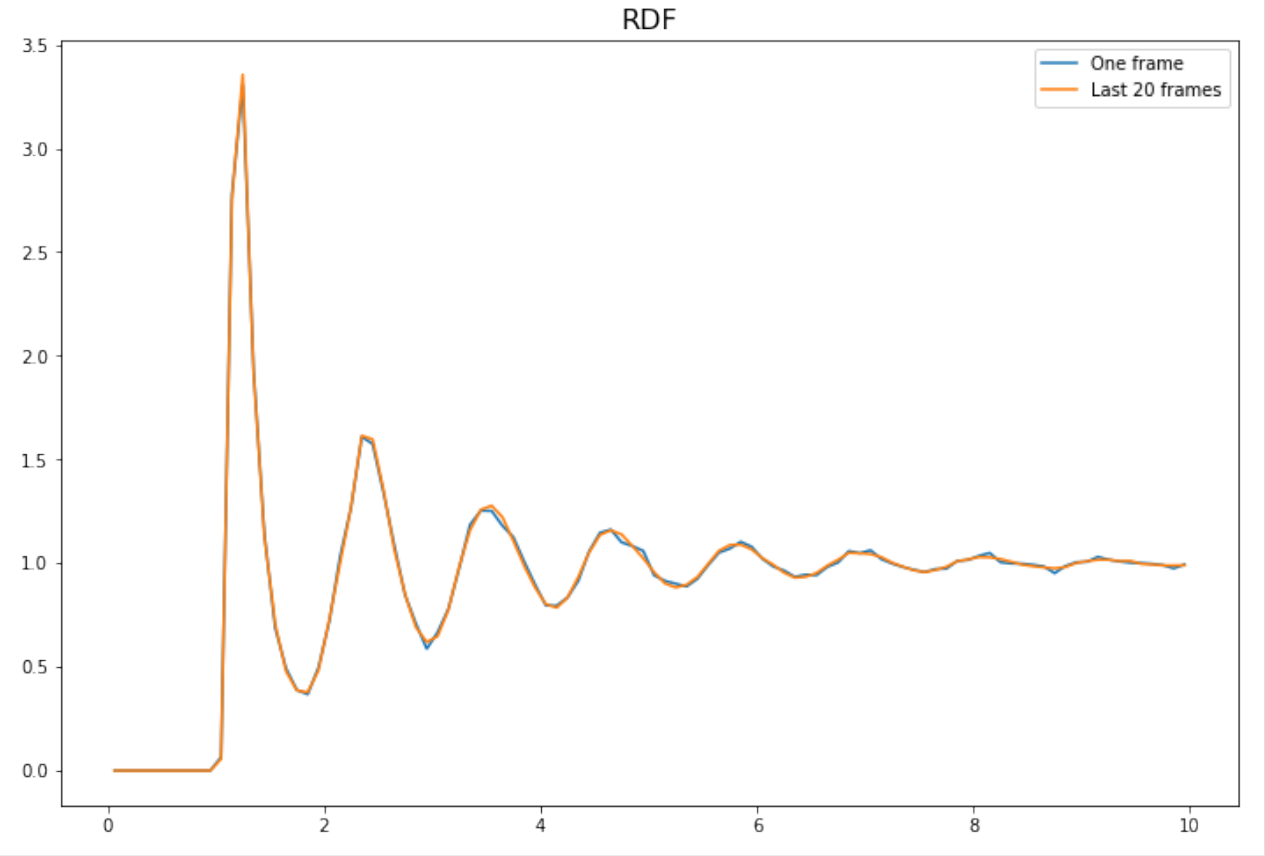
(continues on next page)

(continued from previous page)

```
    if label is not None:
        ax.plot(rdf.R, getattr(rdf, prop), label=label)
        ax.legend()
    else:
        ax.plot(rdf.R, 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', dr=0.1, 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', dr=0.1, label='Last 20 frames', ax=ax)

plt.show()
```



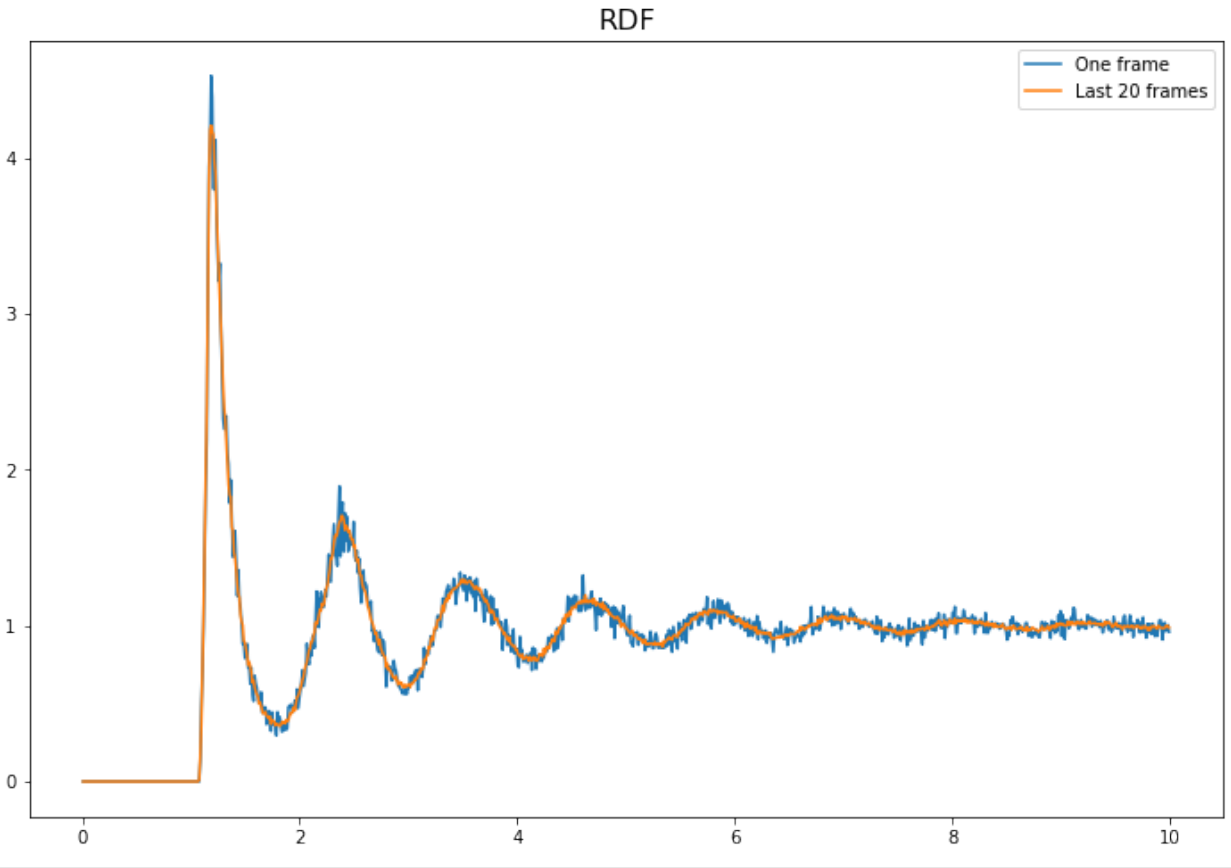
The difference between `accumulate` (which should be called on a series of frames) and `compute` (meant for a single frame) is more striking for smaller bin sizes, which are statistically noisier.

[3]:
```
# Compute the RDF for the last frame
box_arr = [box_data[-1].tolist()]
```

(continues on next page)

(continued from previous page)

```
pos_arr = [pos_data[-1]]
ax = plot_rdf(box_arr, pos_arr, 'RDF', dr=0.01, 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', dr=0.01, label='Last 20 frames', ax=ax)

plt.show()
```



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 util
import matplotlib.pyplot as plt
```

[2]:
```
# Define some helper plotting functions.
def plot_lattice(box, points, colors=None):
    """Helper function for plotting points on a lattice."""
```

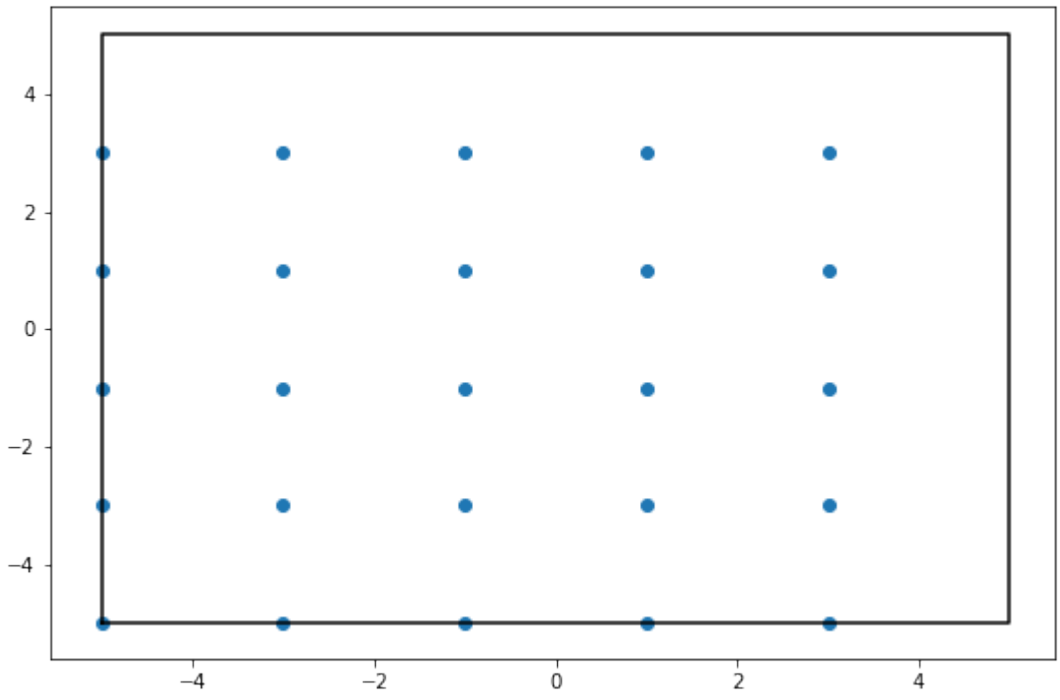
(continues on next page)

(continued from previous page)

```
    fig, ax = plt.subplots(1, 1, figsize=(9, 6))
    box_points = util.box_2d_to_points(box)
    ax.plot(box_points[:, 0], box_points[:, 1], color='k')

    if colors is not None:
        p = ax.scatter(points[:, 0], points[:, 1], c=colors, cmap='plasma')
        plt.colorbar(p)
    else:
        ax.scatter(points[:, 0], points[:, 1])
    return fig, ax

def plot_rdf(box, points, prop, rmax=3, drs=[0.15, 0.04, 0.001]):
    """Helper function for plotting RDFs."""
    fig, axes = plt.subplots(1, len(drs), figsize=(16, 3))
    for i, dr in enumerate(drs):
        rdf = freud.density.RDF(rmax, dr)
        rdf.compute(box, points)
        axes[i].plot(rdf.R, getattr(rdf, prop))
        axes[i].set_title("Bin width: {:.3f}".format(dr), fontsize=16)
    return fig, ax
```

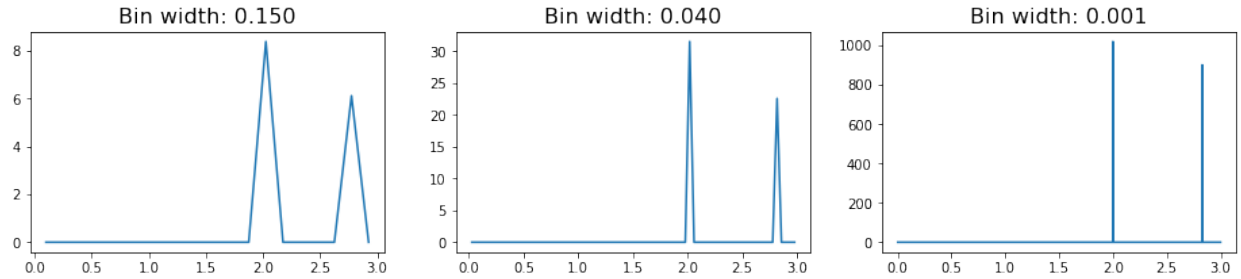
To start, we construct and visualize a set of points sitting on a simple square lattice.

[3]:
```
box, points = util.make_square(5, 5)
fig, ax = plot_lattice(box, points)
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.

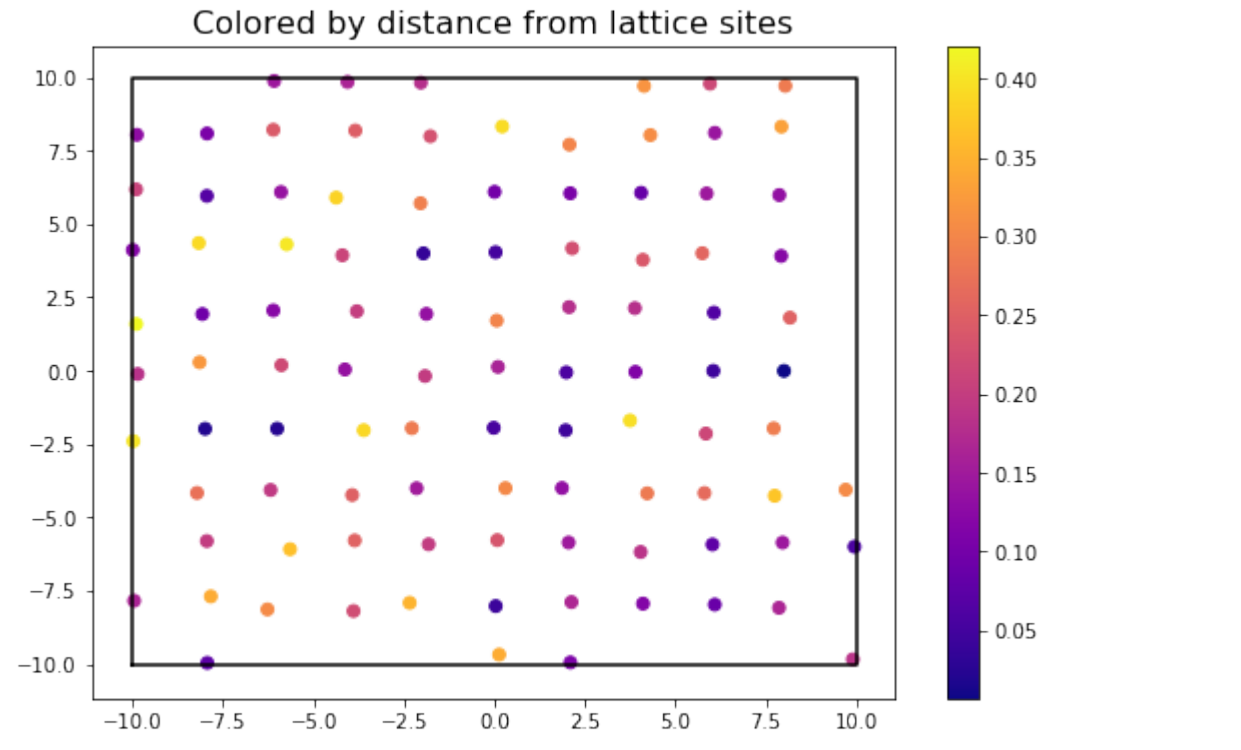
[4]:
```
fig, ax = plot_rdf(box, points, 'RDF')
plt.show()
```



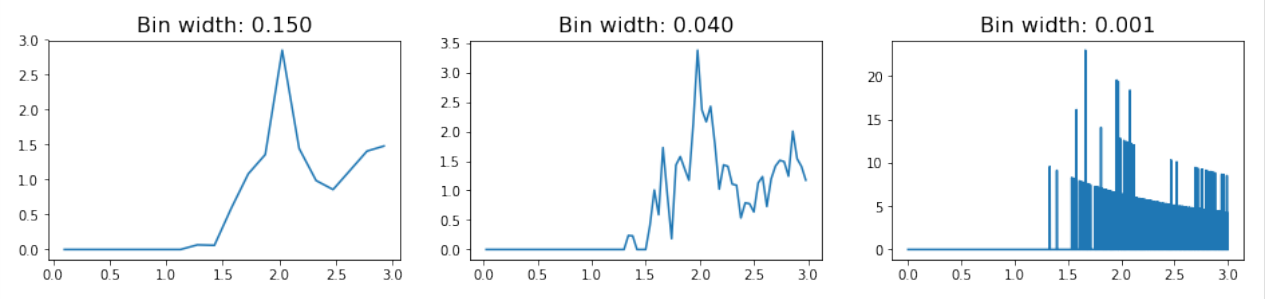
In these RDFs, we see two sharply defined peaks, with the first corresponding to the nearest neighbors on the lattice (which are all at a distance 2 from each other), and the second, smaller peak caused by the particles on the diagonal (which sit at distance $\sqrt{2^2 + 2^2} \approx 2.83$.

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, points = util.make_square(10, 10, noise=0.15)
fig, ax = plot_lattice(box, box.wrap(points), np.linalg.norm(points-np.round(points),␣
↪axis=1))
ax.set_title("Colored by distance from lattice sites", fontsize=16);
plt.show()
```



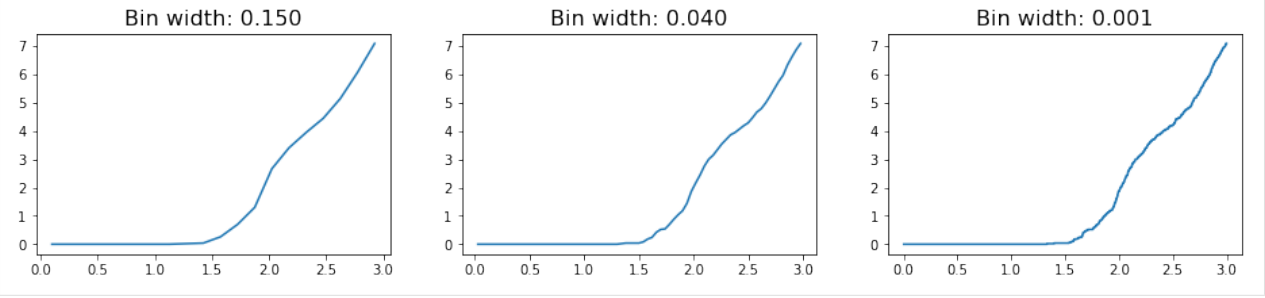
[6]:
```
fig, ax = plot_rdf(box, points, 'RDF')
plt.show()
```



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]:
```
fig, ax = plot_rdf(box, points, 'n_r')
plt.show()
```



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 util
```

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. 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]:
```
# These functions are adapted from the rowan quaternion library.
# See rowan.readthedocs.io for more information.
def quat_multiply(qi, qj):
    """Multiply two sets of quaternions."""
    output = np.empty(np.broadcast(qi, qj).shape)

    output[..., 0] = qi[..., 0] * qj[..., 0] - \
        np.sum(qi[..., 1:] * qj[..., 1:], axis=-1)
    output[..., 1:] = (qi[..., 0, np.newaxis] * qj[..., 1:] +
                       qj[..., 0, np.newaxis] * qi[..., 1:] +
                       np.cross(qi[..., 1:], qj[..., 1:]))
    return output

def quat_rotate(q, v):
    """Rotate a vector by a quaternion."""
    v = np.array([0, *v])

    q_conj = q.copy()
    q_conj[..., 1:] *= -1

    return quat_multiply(q, quat_multiply(v, q_conj))[..., 1:]

def quat_to_angle(q):
    """Get rotation angles of quaternions."""
    norms = np.linalg.norm(q[..., 1:], axis=-1)
    return 2.0 * np.arctan2(norms, q[..., 0])
```

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.

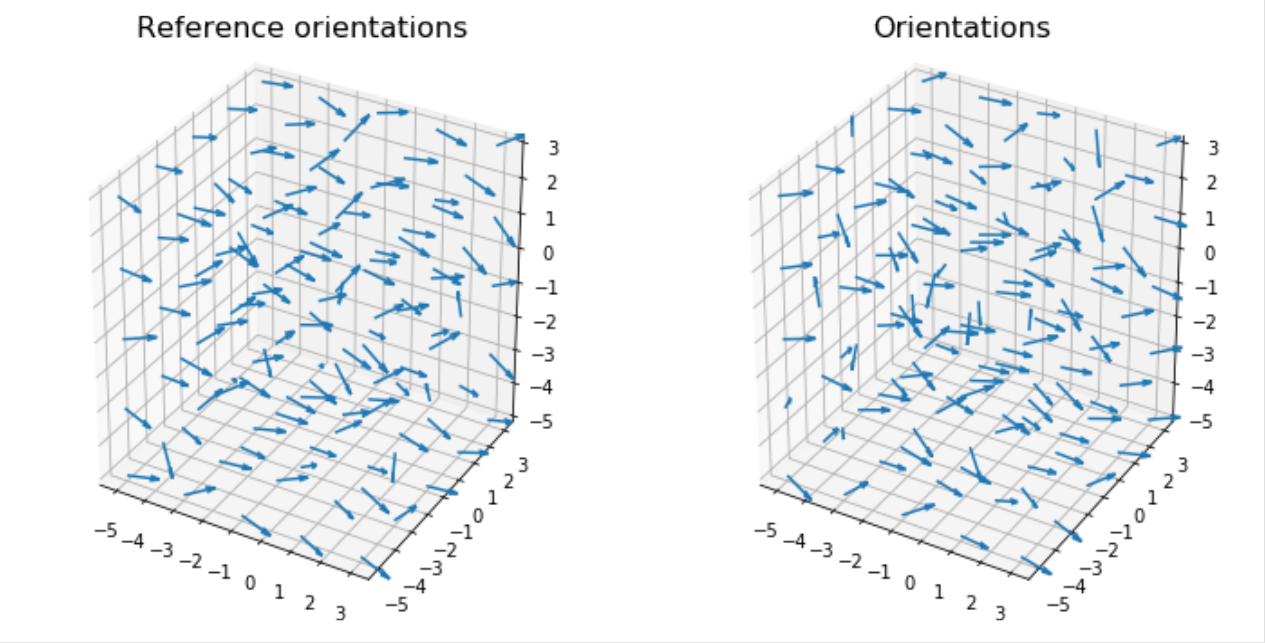
[3]:
```
box, positions = util.make_sc(5, 5, 5)
v = 0.05

# Quaternions can be simply sampled as 4-vectors.
# Note that these samples are not uniformly distributed rotations,
# but that is not important for our current applications.
np.random.seed(0)
ref_orientations = np.random.multivariate_normal(mean=[1, 0, 0, 0], cov=v*np.eye(4),␣
↪size=positions.shape[0])
orientations = np.random.multivariate_normal(mean=[1, 0, 0, 0], cov=v*np.eye(4),␣
↪size=positions.shape[0])

# However, they must be normalized: only unit quaternions represent rotations.
ref_orientations /= np.linalg.norm(ref_orientations, axis=1)[:, np.newaxis]
orientations /= np.linalg.norm(orientations, axis=1)[:, np.newaxis]
```

[4]:
```
# To show orientations, we use arrows rotated by the quaternions.
ref_arrowheads = quat_rotate(ref_orientations, np.array([1, 0, 0]))
arrowheads = quat_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')
```

(continues on next page)

(continued from previous page)

```
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.

[5]:
```
num_neighbors=12
r_max = 1.8

# 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_ors = np.array([[1, 0, 0, 0]])
ang_sep = freud.environment.AngularSeparation(r_max, num_neighbors)
ang_sep.computeNeighbor(box, ref_orientations, orientations,
                        positions, positions, equiv_ors)

#convert angles from radians to degrees
neighbor_angles = np.rad2deg(ang_sep.neighbor_angles)
neighbor_angles
```

[5]:
```
array([100.32801 ,  90.07115 ,  60.706676, ...,  42.66844 ,  61.745235,
       52.767185], dtype=float32)
```

Global Angles

Alternatively, the AngularSeparation 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.

[6]:
```
global_orientations = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0,␣
↪1]])
ang_sep.computeGlobal(ref_orientations, global_orientations, equiv_ors)
global_angles = np.rad2deg(ang_sep.global_angles)
```

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]:
```
np.allclose(global_angles[0, :], np.rad2deg(quat_to_angle(ref_orientations)))
```

[7]:
```
True
```

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
import util
```

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 use `accumulate` to gather data from multiple frames in order to smooth the resulting array's statistics.

[2]:
```
box, points = util.make_fcc(nx=40, ny=40, nz=40, noise=0.1)
orientations = np.array([[1, 0, 0, 0]]*len(points))
```

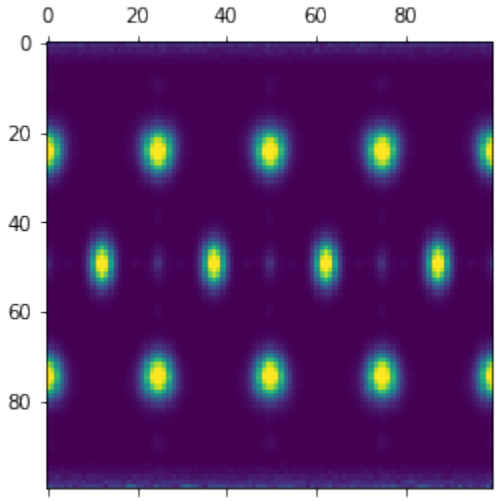
Now we create a `BondOrder` compute object and create some arrays useful for plotting.

[3]:
```
rmax = 3 # This is intentionally large
n_bins_theta = 100
n_bins_phi = 100
k = 0 # This parameter is ignored
n = 12 # Chosen for FCC structure
bod = freud.environment.BondOrder(rmax=rmax, k=k, n=n, n_bins_t=n_bins_theta, n_bins_
↪p=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. Note that we use `freud`'s *method chaining* here, where a compute method returns the compute object.

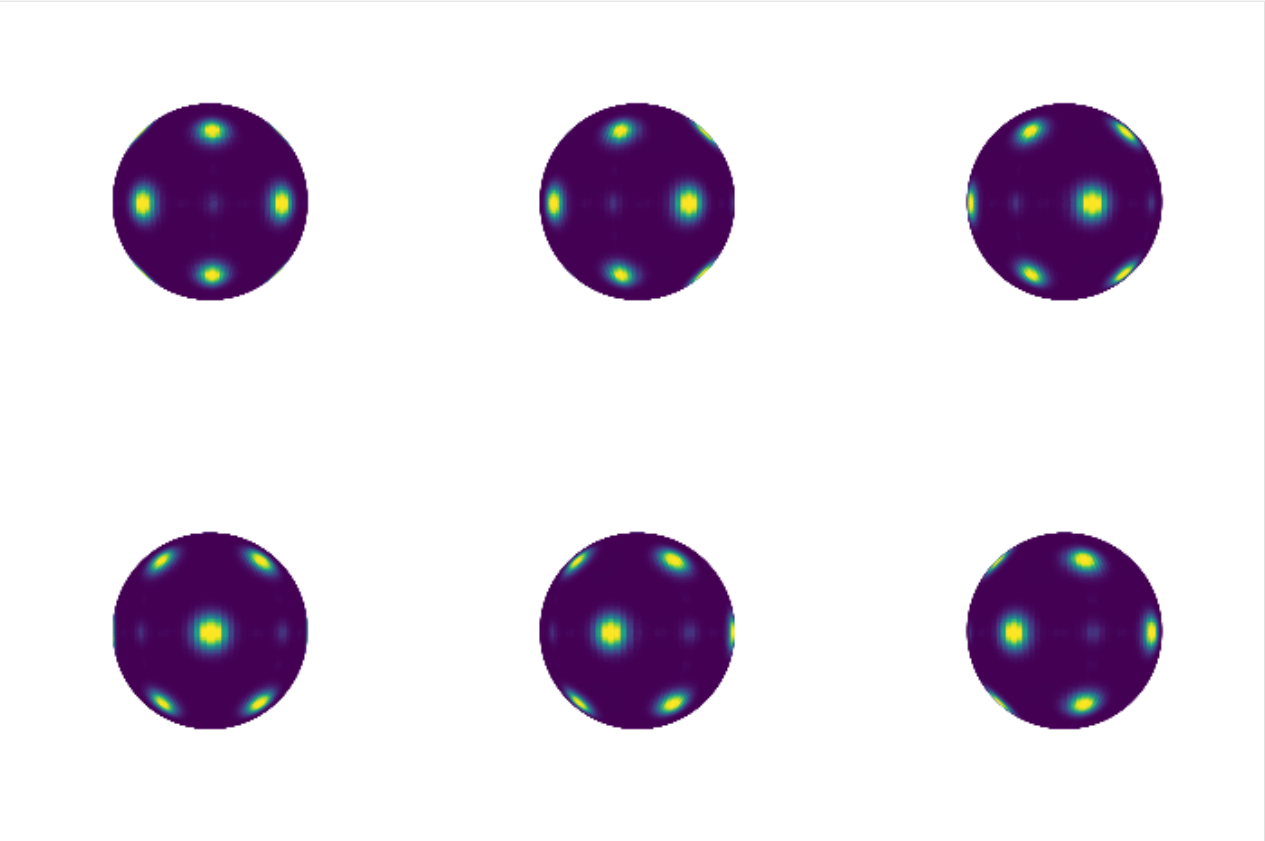
[4]:
```
bod_array = bod.compute(box=box, ref_points=points, ref_orientations=orientations,␣
↪points=points, orientations=orientations).bond_order
bod_array = np.clip(bod_array, 0, np.percentile(bod_array, 99)) # This cleans up bad␣
↪bins for plotting
plt.matshow(bod_array)
plt.show()
```



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.

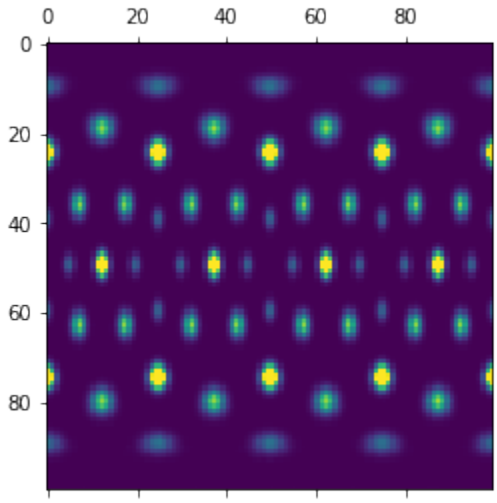
[5]:
```
fig = plt.figure(figsize=(12, 8))
for plot_num in range(6):
    ax = fig.add_subplot(231 + plot_num, projection='3d')
    ax.plot_surface(x, y, z, rstride=1, cstride=1, shade=False,
                    facecolors=matplotlib.cm.viridis(bod_array.T / np.max(bod_array)))
    ax.set_xlim(-1, 1)
    ax.set_ylim(-1, 1)
    ax.set_zlim(-1, 1)
    ax.set_axis_off()
    # View angles in degrees
    view_angle = 0, plot_num*15
    ax.view_init(*view_angle)
plt.show()
```



Using Neighbor Lists

We can also construct neighbor lists and use those to determine bonds instead of the `rmax` and `n` values in the `BondOrder` constructor. 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]:
```
lc = freud.locality.LinkCell(box=box, cell_width=3)
nlist = lc.compute(box, points, points).nlist
nlist.filter_r(box, points, points, rmax=3, rmin=2.5)
bod_array = bod.compute(box=box, ref_points=points, ref_orientations=orientations,
                        points=points, orientations=orientations, nlist=nlist).bond_
↪order
bod_array = np.clip(bod_array, 0, np.percentile(bod_array, 99)) # This cleans up bad␣
↪bins for plotting
plt.matshow(bod_array)
plt.show()
```



LocalDescriptors: Steinhardt Order Parameters

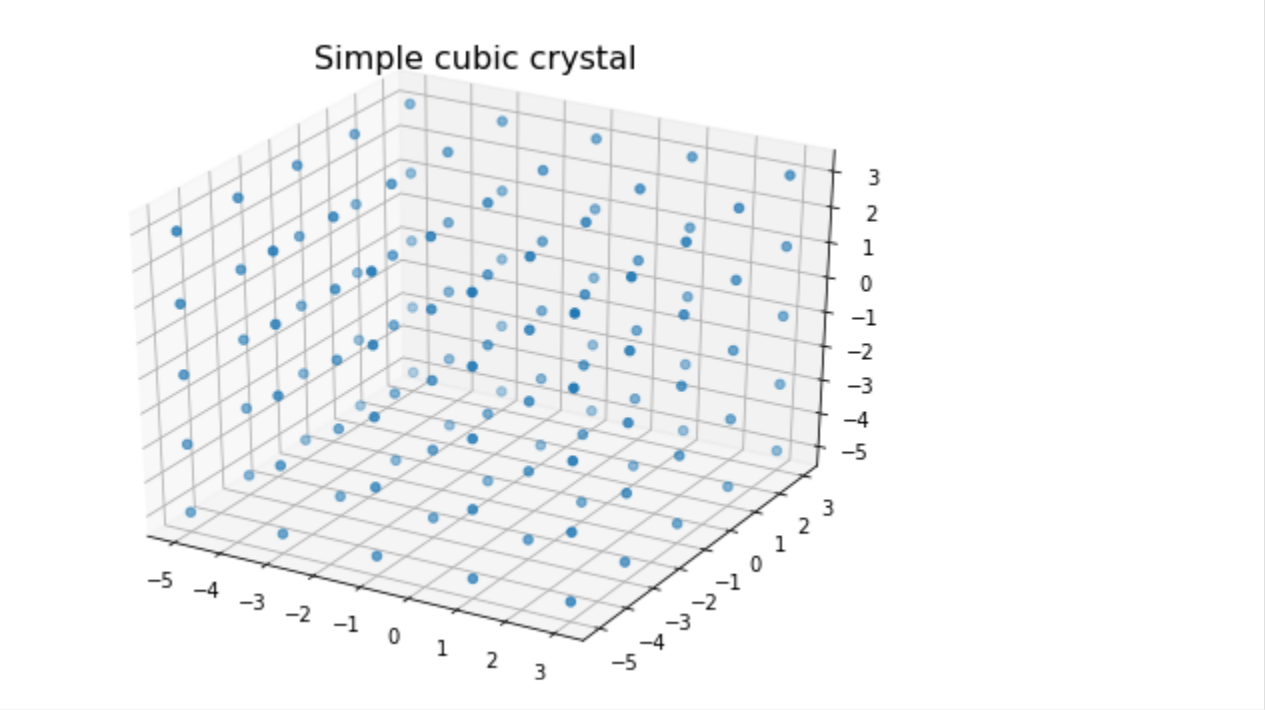
The `freud.environment` module analyzes the local environments of particles. The `freud.environment.LocalDescriptors` class is a useful tool for analyzing identifying crystal structures in a rotationally invariant manner using local particle environments. The primary purpose of this class is to compute 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
import util
```

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]:
```
box, points = util.make_sc(5, 5, 5)
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(points[:, 0], points[:, 1], points[:, 2])
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
r_max = 2

# In order to be able to access information on which particles are bonded
# to which ones, we precompute the neighborlist
nn = freud.locality.NearestNeighbors(r_max, num_neighbors)
nn.compute(box, points)
nl = nn.nlist
ld = freud.environment.LocalDescriptors(num_neighbors, l_max, r_max)
ld.compute(box, num_neighbors, points, mode='global', nlist=nl);
```

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
```

(continues on next page)

(continued from previous page)

```
    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.LocalQl` documentation.

[5]:
```
def get_Ql(p, descriptors, nlist):
    """Given a set of points and a LocalDescriptors object (and the underlying␣
↪neighborlist,
    compute the per-particle Steinhardt order parameter for all :math:`l` values up␣
↪to the
    maximum quantum number used in the computation of the descriptors."""
    Qbar_lm = np.zeros((p.shape[0], descriptors.sph.shape[1]), dtype=np.complex128)
    num_neighbors = descriptors.sph.shape[0]/p.shape[0]
    for i in range(p.shape[0]):
        indices = nlist.index_i == i
        Qbar_lm[i, :] = np.sum(descriptors.sph[indices, :], 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
            Ql[i, l] = np.sqrt(4*np.pi/(2*l + 1) * Ql[i, l])

    return Ql

Ql = get_Ql(points, ld, nl)
```

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 `LocalQl` class can be found in the documentation or in the `LocalQl` example.

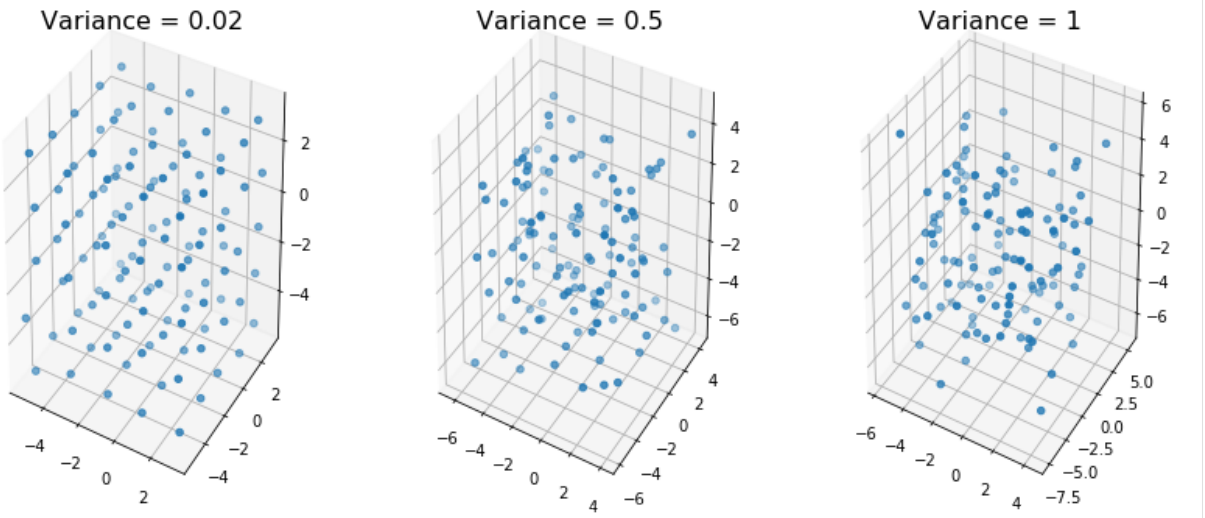
[6]:
```
L = 6
ql = freud.order.LocalQl(box, r_max*2, L, 0)
ql.compute(points, nl)
if np.allclose(ql.Ql, Ql[:, L]):
    print("Our manual Ql calculation matches the Steinhardt OP class!")
```

```
Our manual Ql calculation matches the Steinhardt OP 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]:
```
variances = [0.02, 0.5, 1]
point_arrays = []
nns = []
nls = []
for v in variances:
    point_arrays.append(
        points + np.random.multivariate_normal(
            mean=(0, 0, 0), cov=v*np.eye(3), size=points.shape[0]))
    nns.append(freud.locality.NearestNeighbors(r_max, num_neighbors))
    nns[-1].compute(box, point_arrays[-1])
    nls.append(nns[-1].nlist)
```

[8]:
```
box, points = util.make_sc(5, 5, 5)
fig = plt.figure(figsize=(14, 6))
axes = []
plot_str = "1" + str(len(variances)) + "{}"
for i, v in enumerate(variances):
    axes.append(fig.add_subplot(plot_str.format(i+1), projection='3d'))
    axes[-1].scatter(point_arrays[i][:, 0], point_arrays[i][:, 1], point_arrays[i][:,␣
↪2])
    axes[-1].set_title("Variance = {}".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.

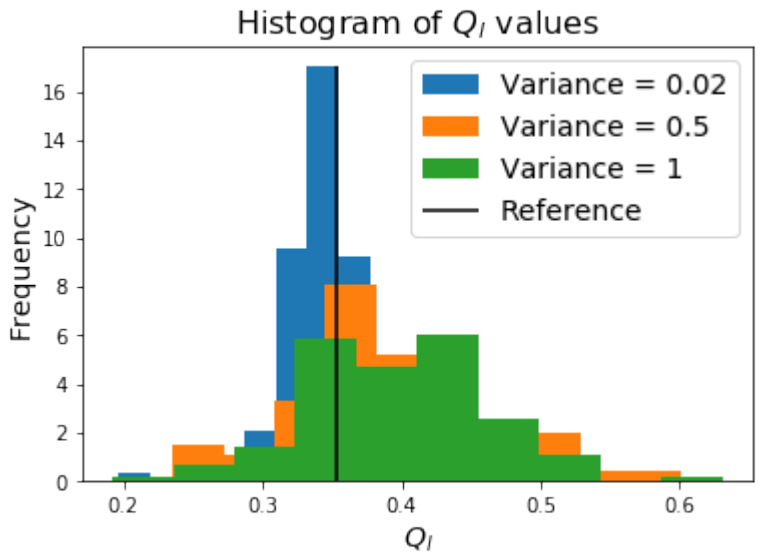
[9]:
```
lds = []
Qls = []
for i, v in enumerate(variances):
    lds.append(freud.environment.LocalDescriptors(num_neighbors, l_max, r_max))
    lds[-1].compute(box, num_neighbors, point_arrays[i], mode='global', nlist=nls[i]);
    Qls.append(get_Ql(point_arrays[i], lds[-1], nls[i]))
```

[10]:
```
fig, ax = plt.subplots()
for i, Q in enumerate(Qls):
    lim_out = ax.hist(Q[:, L], label="Variance = {}".format(variances[i]),␣
↪density=True)
```

(continues on next page)

(continued from previous page)

```
    if i == 0:
        # Can choose any element, all are identical in the reference case
        ax.vlines(Ql[:, L][0], 0, np.max(lim_out[0]), label='Reference')
ax.set_title("Histogram of $Q_l$ values", fontsize=16)
ax.set_ylabel("Frequency", fontsize=14)
ax.set_xlabel("$Q_l$", 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 LocalQl class again

[11]:
```
for i, pa in enumerate(point_arrays):
    ql = freud.order.LocalQl(box, r_max*2, L, 0)
    ql.compute(pa, nls[i])
    if np.allclose(ql.Ql, Qls[i][:, L]):
        print("Our manual Ql calculation matches the Steinhardt OP class!")
```

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

MatchEnv

The `freud.environment.MatchEnv` 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. `MatchEnv` 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.

Simulation data courtesy of Shannon Moran, sample code courtesy of Erin Teich.

[1]:
```
import numpy as np
import freud
from collections import Counter
import matplotlib.pyplot as plt
from util import box_2d_to_points

def get_cluster_arr(box, pos, rcut, num_neigh, threshold, hard_r=False,
                    registration=False, global_search=False):
    """Computes clusters of particles' local environments.

    Args:
        rcut (float):
            Cutoff radius for particles' neighbors.
        num_neigh (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.
        hard_r (bool):
            If True, add all particles that fall within the threshold of
            rcut to the environment.
        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
            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
    match = freud.environment.MatchEnv(box, rcut, num_neigh)
    match.cluster(pos, threshold, hard_r=hard_r,
                  registration=registration, global_search=global_search)
    # Get all clusters. This returns an array in which every
    # particle is indexed by the cluster that it belongs to.
    cluster_envs = {}
    # Get the sets of vectors that correspond to all clusters.
    for cluster_ind in match.clusters:
        if cluster_ind not in cluster_envs:
            cluster_envs[cluster_ind] = np.copy(match.getEnvironment(cluster_ind))

    return np.copy(match.clusters), cluster_envs

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):
```

(continues on next page)

(continued from previous page)

```
            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('ex_data/MatchEnv_Hexagons.npz')
box = ex_data['box']
positions = ex_data['positions']
orientations = ex_data['orientations']

cluster_index_arr, cluster_envs = get_cluster_arr(box, positions, rcut=5.0, num_
↪neigh=6,
                                                  threshold=0.2, hard_r=False,
                                                  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]
```

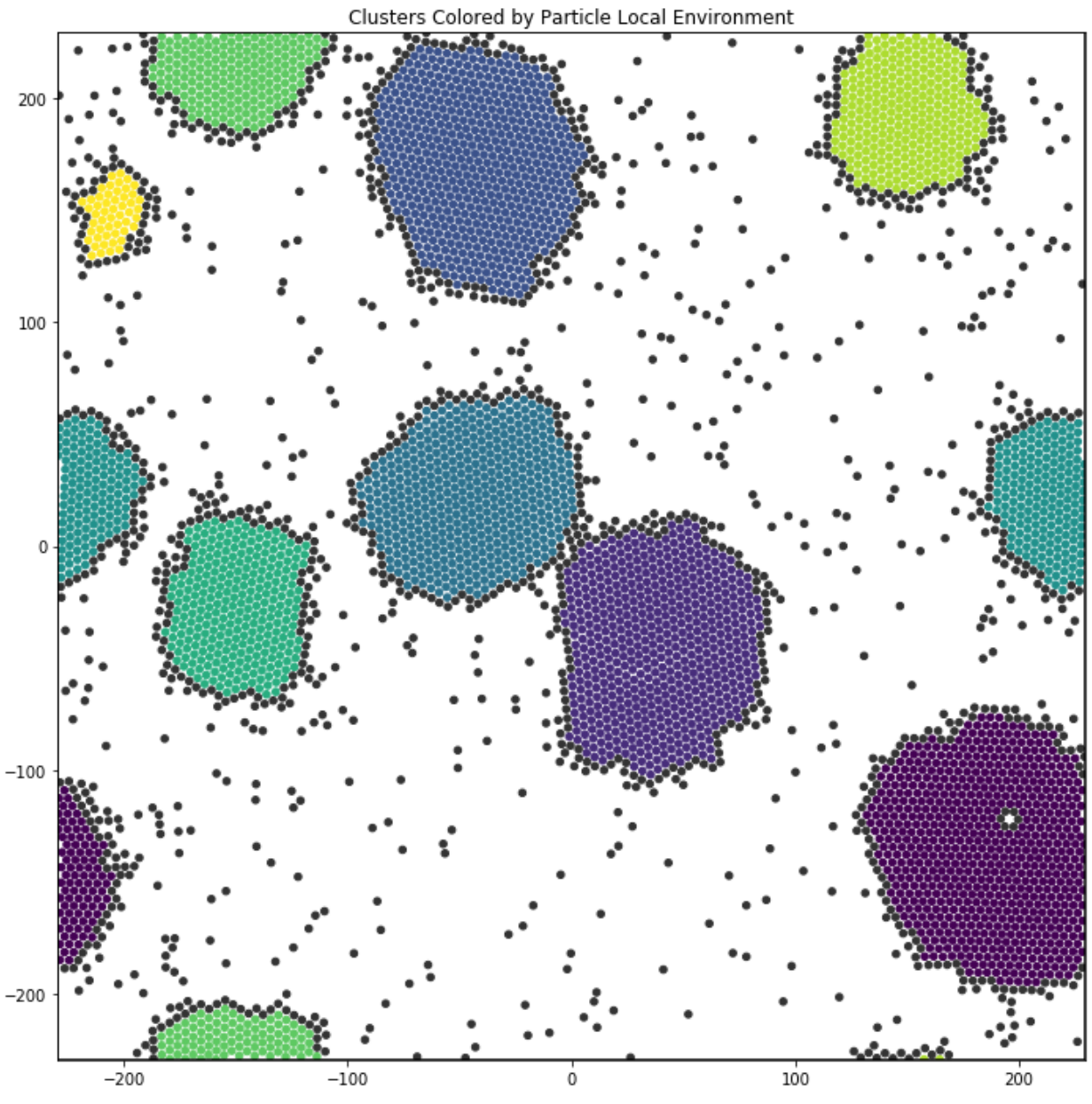
Below, we plot the resulting clusters. The colors correspond to the cluster size.

[3]:
```
plt.figure(figsize=(12, 12), facecolor='white')
box_points = box_2d_to_points(freud.box.Box.from_box(box))
plt.plot(box_points[:, 0], box_points[:, 1], c='black')
plt.scatter(positions[:, 0], positions[:, 1], c=colors, s=20)
ax = plt.gca()
```

(continues on next page)

(continued from previous page)

```
ax.set_xlim((min(box_points[:, 0]), max(box_points[:, 0])))
ax.set_ylim((min(box_points[:, 1]), max(box_points[:, 1])))
ax.set_aspect('equal')
plt.title('Clusters Colored by Particle Local Environment')
plt.show()
```



Interface

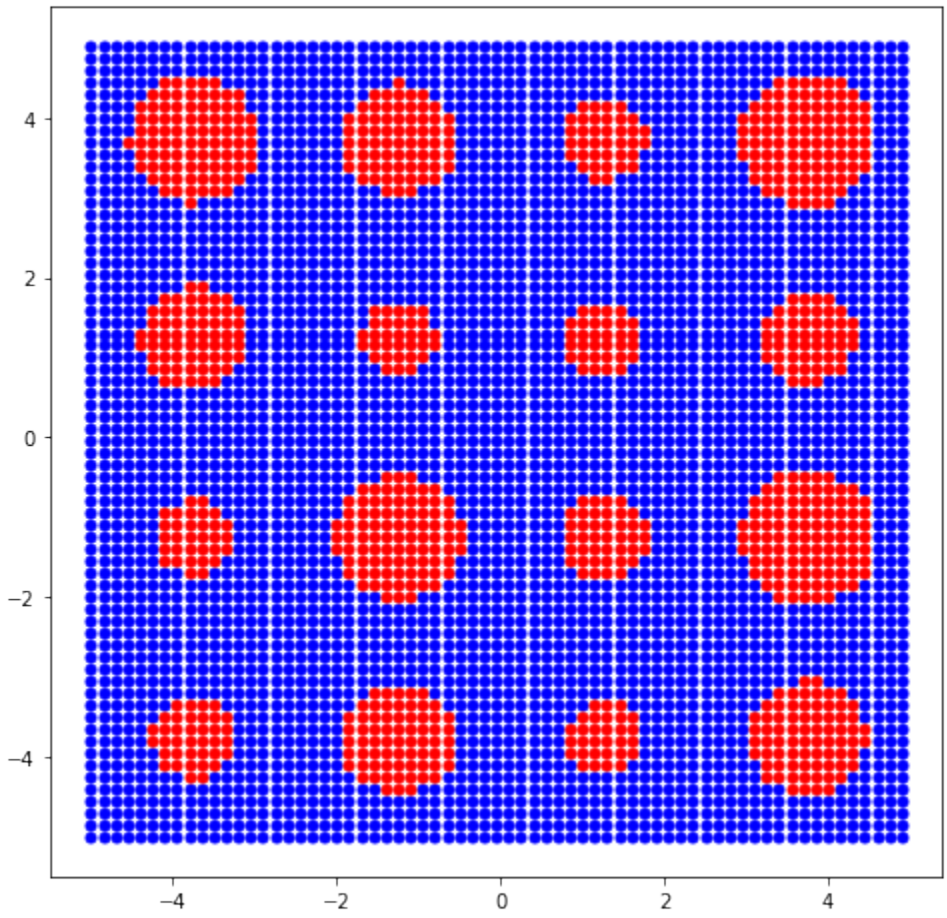
Locating Particles on Interfacial Boundaries

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 **red (+1)** if their distance from a grain center is less than the grain radius.

[2]:
```
# Set up the system
box = freud.box.Box.square(L=10)
dx = 0.15
num_grains = 4
dg = box.Lx/num_grains
points = np.array([[i, j, 0]
                   for j in np.arange(-box.Ly/2, box.Ly/2, dx)
                   for i in np.arange(-box.Lx/2, box.Lx/2, dx)])
values = np.array([-1]*points.shape[0])
centroids = [[i*dg + 0.5*dg, j*dg + 0.5*dg, 0]
             for i in range(num_grains) for j in range(num_grains)]
grain_radii = np.abs(np.random.normal(size=num_grains**2, loc=0.25*dg, scale=0.05*dg))
for center, radius in zip(centroids, grain_radii):
    lc = freud.locality.LinkCell(box, radius).compute(box, points, [center])
    for i in lc.nlist.index_i:
        values[i] = 1

blue_points = points[values < 0]
red_points = points[values > 0]

plt.figure(figsize=(8, 8))
plt.scatter(blue_points[:, 0],
            blue_points[:, 1],
            marker='o', color='blue', s=25)
plt.scatter(red_points[:, 0],
            red_points[:, 1],
            marker='o', color='red', s=25)
plt.show()
```



This system is **phase-separated** because the red 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 red-blue boundary. The class can tell us how many points are on either side of the interface:

[3]:
```
iface = freud.interface.InterfaceMeasure(r_cut=0.2)
iface.compute(box=box, ref_points=blue_points, points=red_points)

print('There are', iface.ref_point_count, 'reference (blue) points on the interface.')
print('There are', iface.point_count, '(red) points on the interface.')
```

```
There are 410 reference (blue) points on the interface.
There are 346 (red) points on the interface.
```

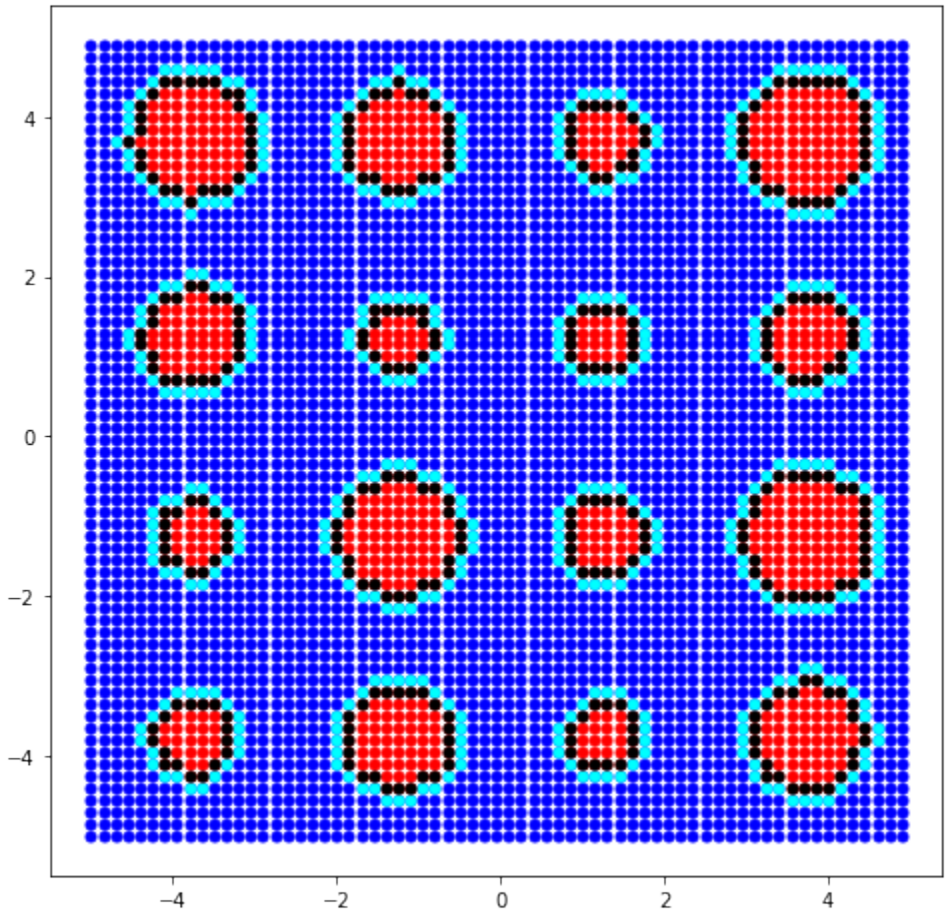
Now we can plot the particles on the interface. We color the outside of the interface cyan and the inside of the interface black.

[4]:
```
plt.figure(figsize=(8, 8))

plt.scatter(blue_points[:, 0],
            blue_points[:, 1],
            marker='o', color='blue', s=25)
plt.scatter(red_points[:, 0],
            red_points[:, 1],
            marker='o', color='red', s=25)
```

(continues on next page)

(continued from previous page)

```
plt.scatter(blue_points[iface.ref_point_ids, 0],
            blue_points[iface.ref_point_ids, 1],
            marker='o', color='cyan', s=25)
plt.scatter(red_points[iface.point_ids, 0],
            red_points[iface.point_ids, 1],
            marker='o', color='black', s=25)

plt.show()
```



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:

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

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):
    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
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.HexOrderParameter(rmax=1.2, k=6, n=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
    l_box = box_data[-1].tolist()
    l_pos = pos_data[-1]
    l_quat = quat_data[-1]
    l_ang = 2*np.arctan2(l_quat[:, 3], l_quat[:, 0])

    # Compute hexatic order for 6 nearest neighbors
    hex_order.compute(l_box, l_pos)
    psi_k = hex_order.psi
    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, l_pos[:, :2], l_ang)
    # 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('ex_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('ex_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('ex_data/phi085', 'Hexatic Order Parameter, 0.85 density')
```

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

NematicOrderParameter

The `freud.order` module provides 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
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import util
```

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) $D 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]:
```
# These functions are adapted from the rowan quaternion library.
# See rowan.readthedocs.io for more information.
def quat_multiply(qi, qj):
    """Multiply two sets of quaternions."""
    output = np.empty(np.broadcast(qi, qj).shape)

    output[..., 0] = qi[..., 0] * qj[..., 0] - \
        np.sum(qi[..., 1:] * qj[..., 1:], axis=-1)
    output[..., 1:] = (qi[..., 0, np.newaxis] * qj[..., 1:] +
```

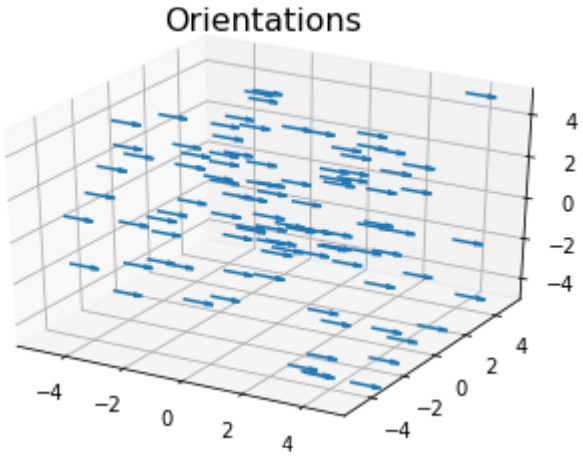
(continues on next page)

(continued from previous page)

```
                        qj[..., 0, np.newaxis] * qi[..., 1:] +
                        np.cross(qi[..., 1:], qj[..., 1:]))
    return output

def quat_rotate(q, v):
    """Rotate a vector by a quaternion."""
    v = np.array([0, *v])

    q_conj = q.copy()
    q_conj[..., 1:] *= -1

    return quat_multiply(q, quat_multiply(v, q_conj))[..., 1:]
```

[3]:
```
# Random positions are fine for this. Order is measured
# in terms of similarity of orientations, not positions.
L = 10
positions = np.random.rand(100, 3)*L - L/2
box = freud.box.Box.cube(L=L)
orientations = np.zeros((100, 4))
orientations[:, 0] = 1  # Quaternion (1, 0, 0, 0) is default orientation
```

[4]:
```
# To show orientations, we use arrows rotated by the quaternions.
arrowheads = quat_rotate(orientations, np.array([1, 0, 0]))

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.quiver3D(positions[:, 0], positions[:, 1], positions[:, 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.

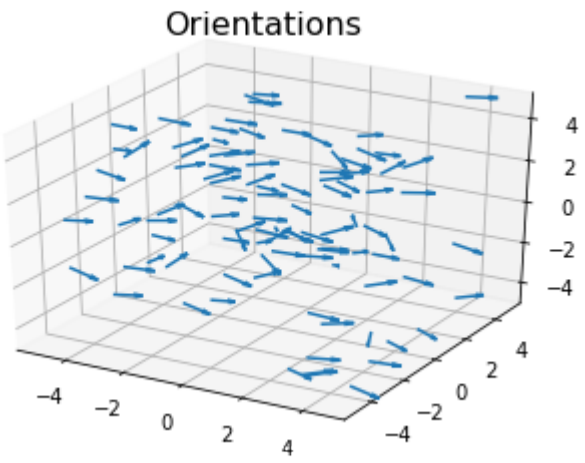
[5]:
```
nop = freud.order.NematicOrderParameter([1, 0, 0])
nop.compute(orientations)
print("The value of the order parameter is {}.".format(nop.nematic_order_parameter))
```

```
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.

[6]:
```
# Quaternions can be simply sampled as 4-vectors.
# Note that these samples are not uniformly distributed rotations,
# but that is not important for our current applications.
# However, we must ensure that the quaternions are normalized:
# only unit quaternions represent rotations.
np.random.seed(0)
v = 0.05
orientations = np.random.multivariate_normal(mean=[1, 0, 0, 0], cov=v*np.eye(4),␣
↪size=positions.shape[0])
orientations /= np.linalg.norm(orientations, axis=1)[:, np.newaxis]
```

[7]:
```
# To show orientations, we use arrows rotated by the quaternions.
arrowheads = quat_rotate(orientations, np.array([1, 0, 0]))

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.quiver3D(positions[:, 0], positions[:, 1], positions[:, 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.

[8]:
```
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.NematicOrderParameter(ax)
    nop.compute(orientations)
    print("For axis {}, the value of the order parameter is {:0.3f}.".format(ax, nop.
↪nematic_order_parameter))
```

```
For axis [1, 0, 0], the value of the order parameter is 0.608.
For axis [0, 1, 0], the value of the order parameter is 0.564.
For axis [0, 0, 1], the value of the order parameter is 0.606.
For axis [1, 1, 0], the value of the order parameter is 0.611.
```

(continues on next page)

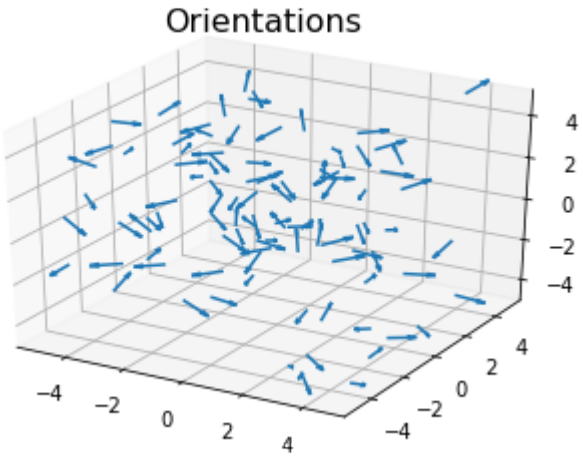
(continued from previous page)

```
For axis [1, 0, 1], the value of the order parameter is 0.616.
For axis [0, 1, 1], the value of the order parameter is 0.577.
For axis [1, 1, 1], the value of the order parameter is 0.608.
```

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

[9]:
```
v = 0.5
orientations = np.random.multivariate_normal(mean=[1, 0, 0, 0], cov=v*np.eye(4),␣
↪size=positions.shape[0])
orientations /= np.linalg.norm(orientations, axis=1)[:, np.newaxis]

arrowheads = quat_rotate(orientations, np.array([1, 0, 0]))
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.quiver3D(positions[:, 0], positions[:, 1], positions[:, 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.NematicOrderParameter(ax)
    nop.compute(orientations)
    print("For axis {}, the value of the order parameter is {:0.3f}.".format(ax, nop.
↪nematic_order_parameter))
```

```
For axis [1, 0, 0], the value of the order parameter is 0.047.
For axis [0, 1, 0], the value of the order parameter is 0.075.
For axis [0, 0, 1], the value of the order parameter is 0.062.
For axis [1, 1, 0], the value of the order parameter is 0.090.
For axis [1, 0, 1], the value of the order parameter is 0.071.
For axis [0, 1, 1], the value of the order parameter is 0.069.
For axis [1, 1, 1], the value of the order parameter is 0.122.
```



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

[10]:
```
# 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:")
```

(continues on next page)

(continued from previous page)

```
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.nematic_order_parameter))
```

```
True
The nematic tensor:
[[-0.08164977 -0.00958257  0.05754685]
 [-0.00958257  0.02508486  0.07119219]
 [ 0.05754685  0.07119219  0.05656486]]
The eigenvalues of the nematic tensor:
[-0.11201978 -0.00962728  0.121647  ]
The eigenvectors of the nematic tensor:
[[-0.86846197 -0.45401222  0.19911498]
 [-0.27491584  0.7752717   0.5686608 ]
 [ 0.41254714 -0.43912038  0.7981091 ]]
The largest eigenvalue, 0.122, is equal to the order parameter 0.122.
```

LocalQl, LocalWl

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

(continues on next page)

(continued from previous page)

```
np.random.seed(1)
```

[2]:
```
%matplotlib inline
```

We first construct ideal crystals and then extract the characteristic value of Q_l for each of these structures. Note that we are using the `LocalQlNear` class, which takes as a parameter the number of nearest neighbors to use in additional to a distance cutoff. The base `LocalQl` class can also be used, but it can be much more sensitive to the choice of distance cutoff; conversely, the corresponding `LocalQlNear` class is guaranteed to find the number of neighbors required. Such a guarantee is especially useful when trying to identify known structures that have specific coordination numbers. 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.

[3]:
```
r_max = 2

L = 6
box, sc = util.make_sc(5, 5, 5)
# The last two arguments are the quantum number l and the number of nearest neighbors.
ql = freud.order.LocalQlNear(box, r_max*2, L, L)
Ql_sc = ql.compute(sc).Ql
mean_sc = np.mean(Ql_sc)
print("The standard deviation in the values computed for simple cubic is {}".
↪format(np.std(Ql_sc)))

L = 8
box, bcc = util.make_bcc(5, 5, 5)
ql = freud.order.LocalQlNear(box, r_max*2, L, L)
Ql_bcc = ql.compute(bcc).Ql
mean_bcc = np.mean(Ql_bcc)
print("The standard deviation in the values computed for BCC is {}".format(np.std(Ql_
↪bcc)))

L = 12
box, fcc = util.make_fcc(5, 5, 5)
ql = freud.order.LocalQlNear(box, r_max*2, L, L)
Ql_fcc = ql.compute(fcc).Ql
mean_fcc = np.mean(Ql_fcc)
print("The standard deviation in the values computed for FCC is {}".format(np.std(Ql_
↪fcc)))
```

```
The standard deviation in the values computed for simple cubic is 0.014370555989444256
The standard deviation in the values computed for BCC is 0.010574632324278355
The standard deviation in the values computed for FCC is 0.002546764211729169
```

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.

[4]:
```
def make_noisy_replicas(points, variances):
    """Given a set of points, return an array of those points with noise."""
    point_arrays = []
    for v in variances:
        point_arrays.append(
            points + np.random.multivariate_normal(
```

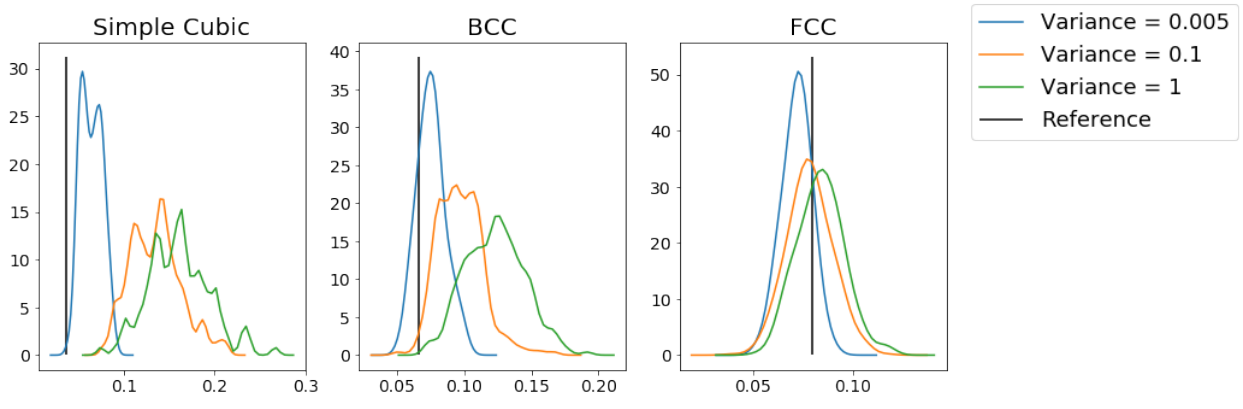
(continues on next page)

(continued from previous page)

```
                mean=(0, 0, 0), cov=v*np.eye(3), size=points.shape[0]))
    return point_arrays
```

[5]:
```
variances = [0.005, 0.1, 1]
sc_arrays = make_noisy_replicas(sc, variances)
bcc_arrays = make_noisy_replicas(bcc, variances)
fcc_arrays = make_noisy_replicas(fcc, variances)
```

[6]:
```
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_arrays, bcc_arrays, fcc_arrays], [mean_sc, mean_bcc, mean_fcc],
              [6, 8, 12], ["Simple Cubic", "BCC", "FCC"])

for i, (arrays, ref_val, L, title) in enumerate(zip_obj):
    ax = axes[i]
    for j, (array, var) in enumerate(zip(arrays, variances)):
        ql = freud.order.LocalQlNear(box, r_max*2, L, L)
        ql.compute(array)
        if not kde:
            ax.hist(ql.Ql, label="Variance = {}".format(var), density=True)
        else:
            padding = 0.02
            N = 50
            bins = np.linspace(np.min(ql.Ql)-padding, np.max(ql.Ql)+padding, N)

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

            ax.plot(bins, Ql, label="Variance = {}".format(var))
        ax.set_title(title, fontsize=20)
        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. However, you can see even from these plots that the measures are not always good; for

example, the BCC example shows minimal distinction between variances of 0.1 and 1, which you might hope to be easily distinguishable, while adding minimal noise to the FCC crystals makes the system deviate substantially from the optimal value of the order parameter. As a result, choosing the appropriate parameterization for the order parameter (which quantum number l to use, how many nearest neighbors, the r_{cut}, etc) can be very important.

In addition to the simple `LocalQlNear` class demonstrated here and the r_{cut} based `LocalQl` variant, there are also the `LocalWl` and `LocalWlNear` classes. The latter two classes use the same spherical harmonics to compute a slightly different quantity: Q_l involves one way of averaging the spherical harmonics between particles and their neighbors, and W_l uses a different type of average. The W_l averages 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's original paper).

In addition to the difference between the classes, the classes also contain additional compute methods that perform an additional type of averaging. Calling `computeAve` instead of compute will populate the `ave_Ql` (or `ave_Wl`) arrays, which perform an additional level of implicit averaging over the second neighbor shells of particles to accumulate more information on particle environments (see the original reference). 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.

PMFTXY2D

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 cell lists 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.parallel.setNumThreads(4)
import numpy as np
import matplotlib
from matplotlib import pyplot as plt
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
```

(continues on next page)

(continued from previous page)

```
    pmft = freud.pmft.PMFTXY2D(x_max=3.0, y_max=3.0, n_x=300, n_y=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
        l_box = box_data[i].tolist()
        l_pos = pos_data[i]
        l_quat = quat_data[i]
        l_ang = 2*np.arctan2(l_quat[:, 3], l_quat[:, 0])
        l_ang = l_ang % (2 * np.pi)

        pmft.accumulate(l_box, l_pos, l_ang, l_pos, l_ang)

    # Get the value of the PMFT histogram bins
    pmft_arr = np.copy(pmft.PMFT)

    # Do some simple post-processing for plotting purposes
    pmft_arr[np.isinf(pmft_arr)] = np.nan
    dx = (2.0 * 3.0) / pmft.n_bins_X
    dy = (2.0 * 3.0) / pmft.n_bins_Y
    nan_arr = np.where(np.isnan(pmft_arr))
    for i in range(pmft.n_bins_X):
        x = -3.0 + dx * i
        for j in range(pmft.n_bins_Y):
            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.n_bins_X / (2.0 * 3.0))
    center = int(pmft.n_bins_X / 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
    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 = pmft.X
    y = pmft.Y
    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)
```

(continues on next page)

(continued from previous page)

```
    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()
```

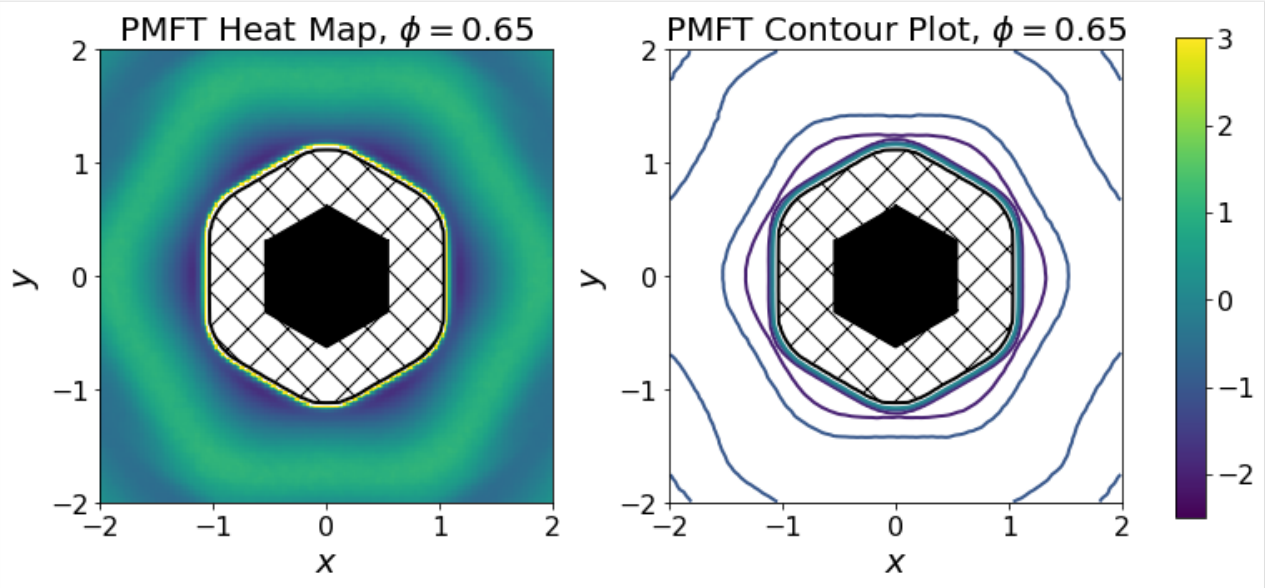
65% density

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).

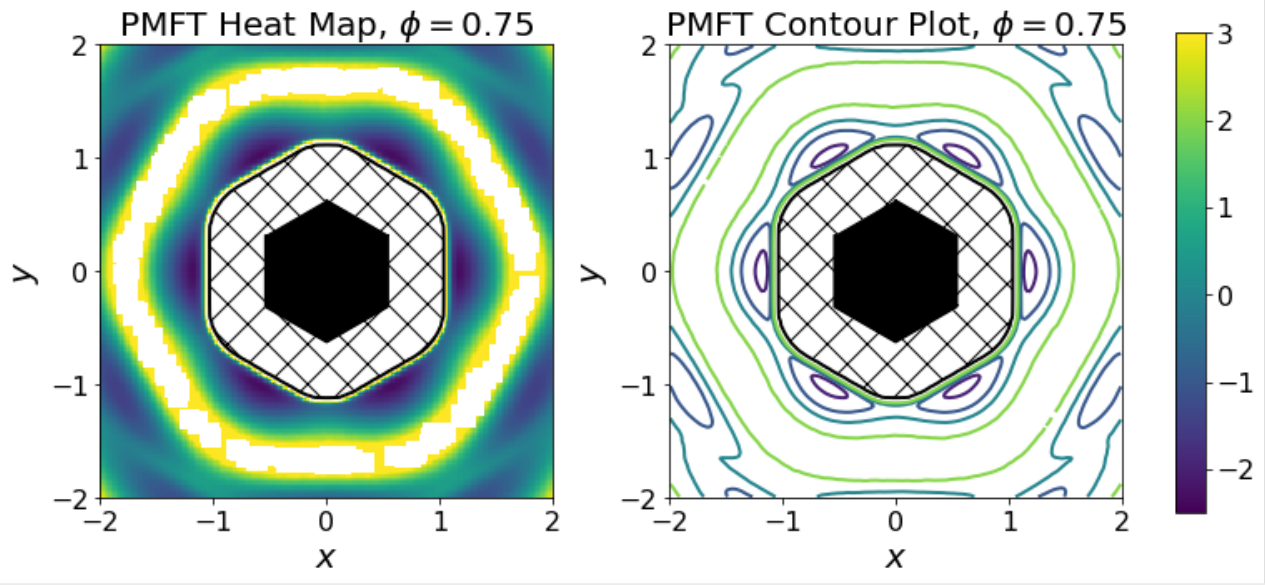
[3]:
```
plot_pmft('ex_data/phi065', 0.65)
```



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).

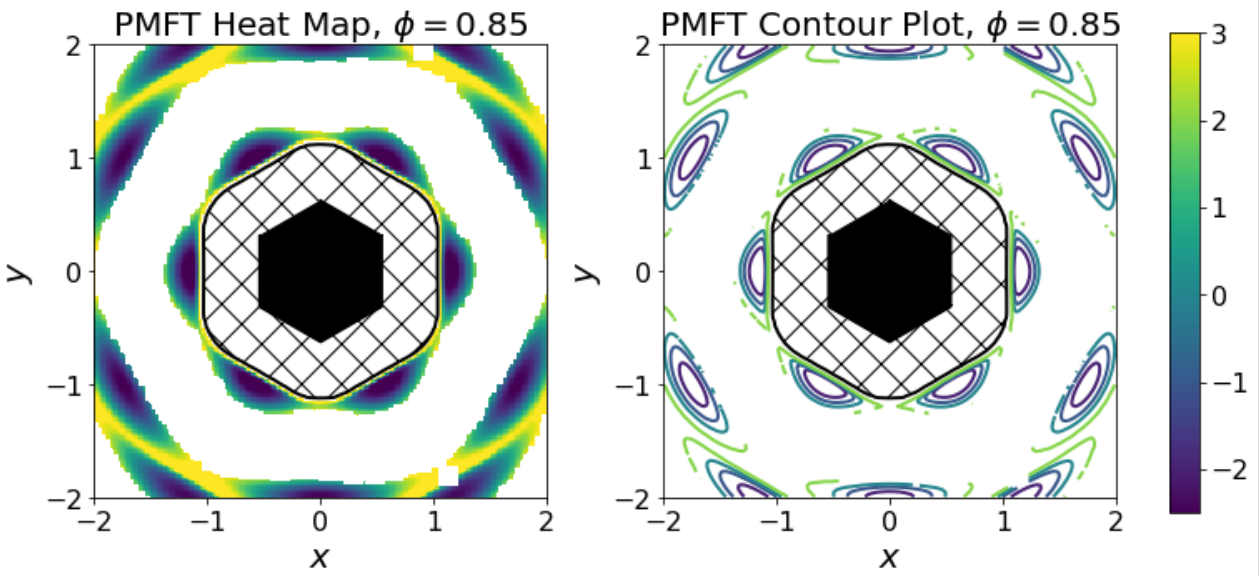
[4]:
```
plot_pmft('ex_data/phi075', 0.75)
```



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('ex_data/phi085', 0.85)
```



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
from freud import box, pmft

from scipy.interpolate import griddata
from scipy.interpolate import RegularGridInterpolator

import warnings
warnings.simplefilter('ignore')

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('ex_data/XYZ/positions.npy').astype(np.float32)
box_data = np.load('ex_data/XYZ/boxes.npy').astype(np.float32)
```

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

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

res = (100,100,100)
pmft_arr = np.zeros(res)

mypmft = pmft.PMFTXYZ(x_max=window, y_max=window, z_max=window,
                      n_x=res[0], n_y=res[1], n_z=res[2])

# This data is for isotropic particles, so we will just make some unit quaternions
```

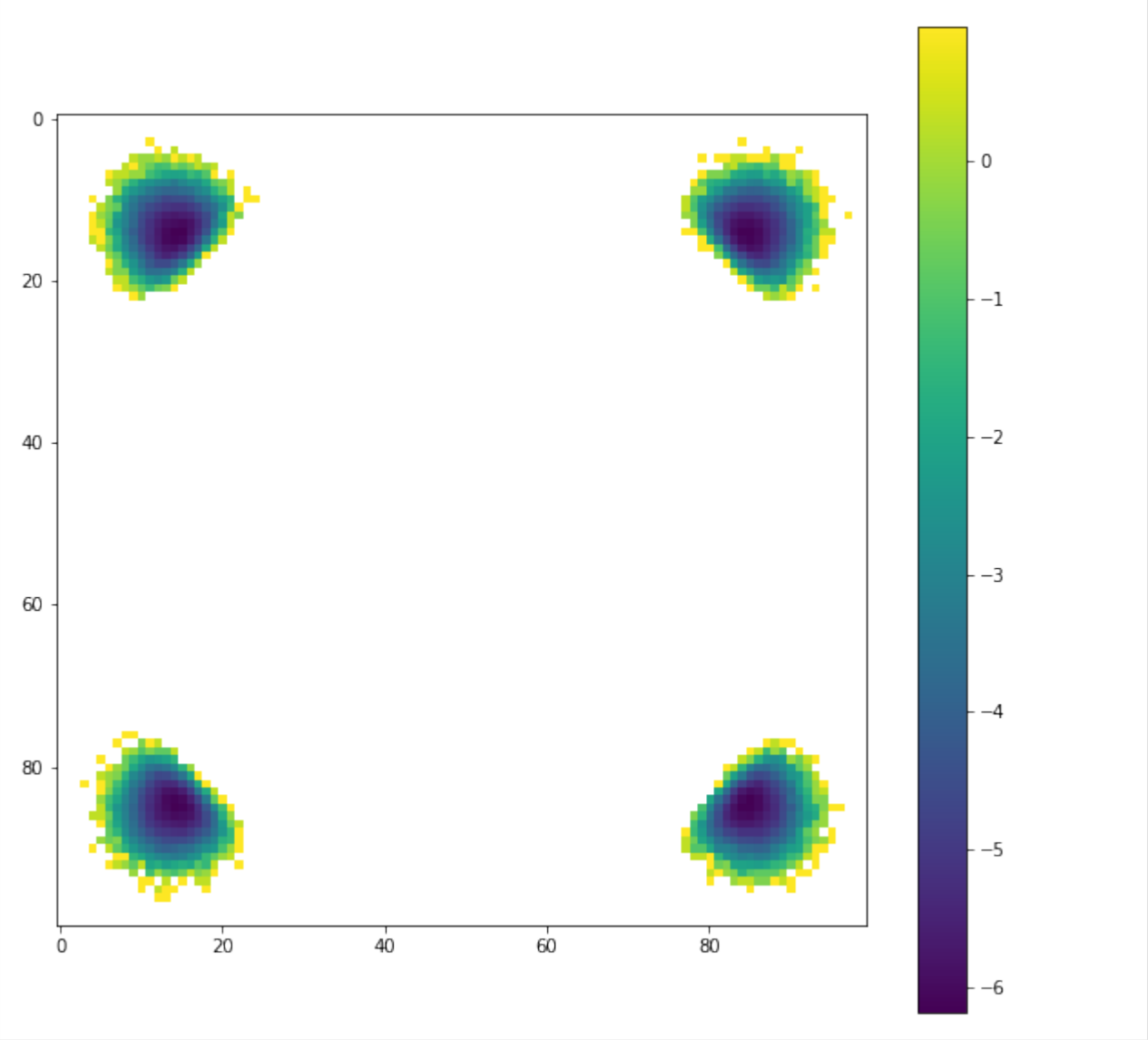
(continues on next page)

(continued from previous page)

```
# 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]):
    l_box = box_data[i]
    l_pos = pos_data[i]
    mypmft.accumulate(l_box, l_pos, quats, l_pos, quats)

unshifted = np.copy(mypmft.PMFT)

x = mypmft.X
y = mypmft.Y
z = mypmft.Z
```

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(res[0]/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

res = (50,50,50)
pmft_arr = np.zeros(res)

mypmft = pmft.PMFTXYZ(x_max=window, y_max=window, z_max=window,
                      n_x=res[0], n_y=res[1], n_z=res[2], shiftvec=shiftvec)
```

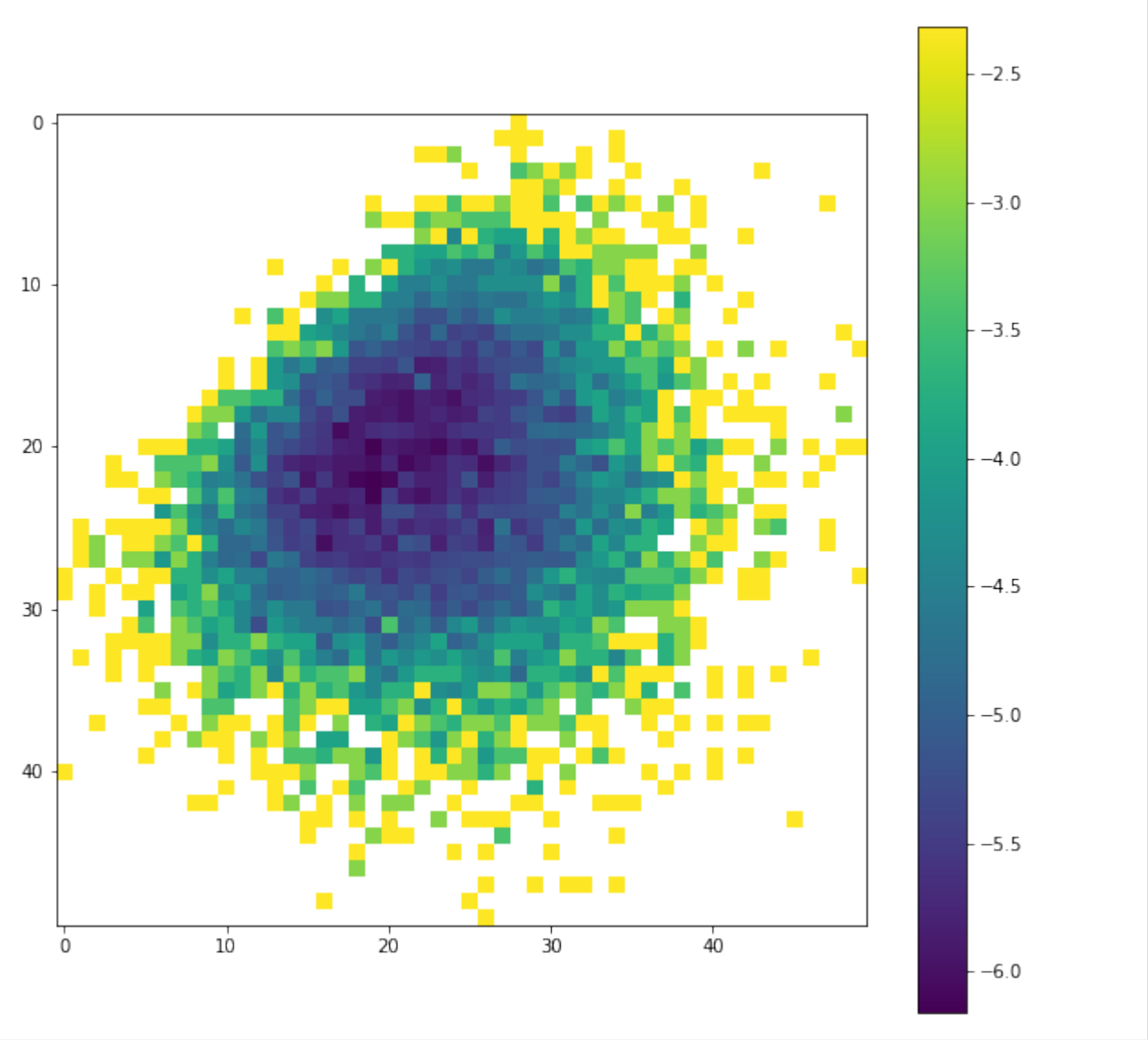
(continues on next page)

(continued from previous page)

```
# 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]):
    l_box = box_data[i]
    l_pos = pos_data[i]
    mypmft.accumulate(l_box, l_pos, quats, l_pos, quats)

shifted = np.copy(mypmft.PMFT)

x = mypmft.X
y = mypmft.Y
z = mypmft.Z
```

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(res[2]/2),:,:])
plt.colorbar()
plt.show()
```



Voronoi

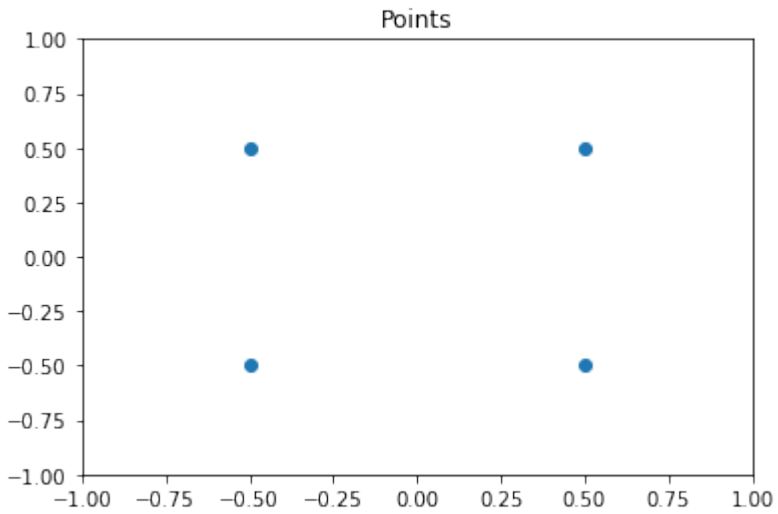
The `voronoi` module finds the Voronoi diagram of a set of points, **while respecting periodic boundary conditions** (which are not handled by `scipy.spatial.Voronoi`, documentation). This is handled by replicating the points using periodic images that lie outside the box, up to a specified buffer distance.

This case is two-dimensional (with z=0 for all particles) for simplicity, but the `voronoi` module works for both 2D and 3D simulations.

[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],
    [0.5, -0.5],
    [-0.5, 0.5],
    [0.5, 0.5]])
plt.scatter(points[:,0], points[:,1])
plt.title('Points')
plt.xlim((-1, 1))
plt.ylim((-1, 1))
plt.show()

# We must add a z=0 component to this array for freud
points = np.hstack((points, np.zeros((points.shape[0], 1))))
```



Now we create a box and a `voronoi` compute object. Note that the buffer distance must be large enough to ensure that the points are duplicated sufficiently far outside the box for the `qhull` algorithm. **Results are only guaranteed to be correct when the buffer is large enough:** `buffer` $>= L/2$, for the longest box side length L.

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

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=box, positions=points).polytopes
print(cells)
```

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

(continues on next page)

(continued from previous page)

```
       [0., 1., 0.],
       [1., 1., 0.],
       [1., 0., 0.]])]
```

We create a helper function to draw the Voronoi polygons using `matplotlib`.

[5]:
```
def draw_voronoi(box, points, cells, nlist=None, color_by_sides=False):
    ax = plt.gca()
    # Draw Voronoi cells
    patches = [plt.Polygon(cell[:, :2]) for cell in cells]
    patch_collection = matplotlib.collections.PatchCollection(patches, edgecolors=
↪'black', alpha=0.4)
    cmap = plt.cm.Set1

    if color_by_sides:
        colors = [len(cell) for cell in voro.polytopes]
    else:
        colors = np.random.permutation(np.arange(len(patches)))

    cmap = plt.cm.get_cmap('Set1', np.unique(colors).size)
    bounds = np.array(range(min(colors), max(colors)+2))

    patch_collection.set_array(np.array(colors))
    patch_collection.set_cmap(cmap)
    patch_collection.set_clim(bounds[0], bounds[-1])
    ax.add_collection(patch_collection)

    # Draw points
    plt.scatter(points[:,0], points[:,1], c=colors)
    plt.title('Voronoi Diagram')
    plt.xlim((-box.Lx/2, box.Lx/2))
    plt.ylim((-box.Ly/2, box.Ly/2))

    # Set equal aspect and draw box
    ax.set_aspect('equal', 'datalim')
    box_patch = plt.Rectangle([-box.Lx/2, -box.Ly/2], box.Lx, box.Ly, alpha=1,␣
↪fill=None)
    ax.add_patch(box_patch)

    # Draw neighbor lines
    if nlist is not None:
        bonds = np.asarray([points[j] - points[i] for i, j in zip(nlist.index_i,␣
↪nlist.index_j)])
        box.wrap(bonds)
        line_data = np.asarray([[points[nlist.index_i[i]],
                                 points[nlist.index_i[i]]+bonds[i]] for i in␣
↪range(len(nlist.index_i))])
        line_data = line_data[:, :, :2]
        line_collection = matplotlib.collections.LineCollection(line_data, alpha=0.3)
        ax.add_collection(line_collection)

    # Show colorbar for number of sides
    if color_by_sides:
        cb = plt.colorbar(patch_collection, ax=ax, ticks=bounds, boundaries=bounds)
        cb.set_ticks(cb.formatter.locs + 0.5)
        cb.set_ticklabels((cb.formatter.locs - 0.5).astype('int'))
        cb.set_label("Number of sides", fontsize=12)
```

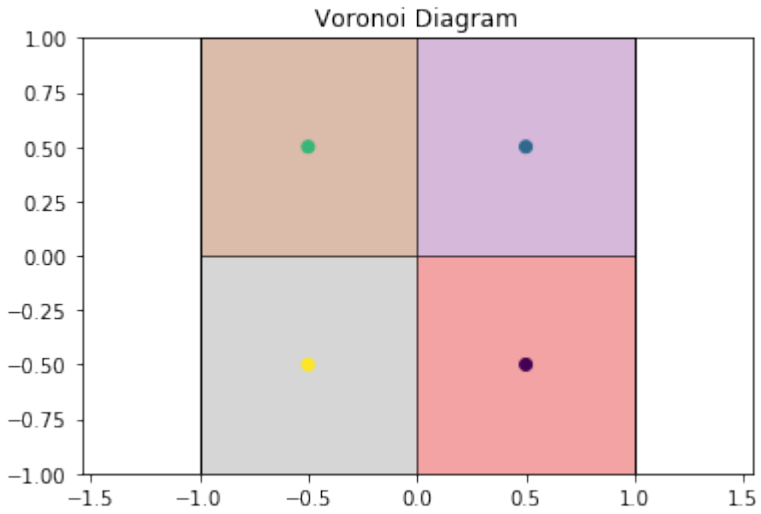
(continues on next page)

(continued from previous page)

```
    plt.show()
```

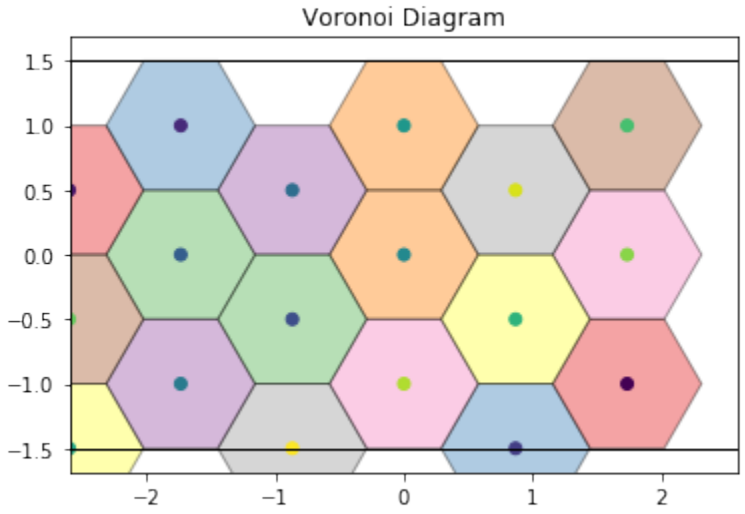
Now we can draw the Voronoi diagram from the example above.

[6]:
```
draw_voronoi(box, points, voro.polytopes)
```



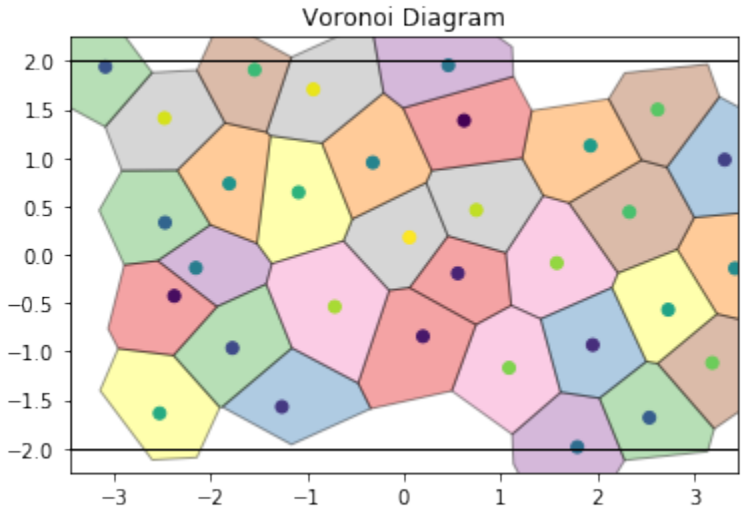
This also works for more complex cases, such as this hexagonal lattice.

[7]:
```
def hexagonal_lattice(rows=3, cols=3, noise=0):
    # 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

    # 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

# Compute the Voronoi diagram and plot
box, points = hexagonal_lattice()
voro = freud.voronoi.Voronoi(box, np.max(box.L)/2)
voro.compute(box=box, positions=points)
draw_voronoi(box, points, voro.polytopes)
```



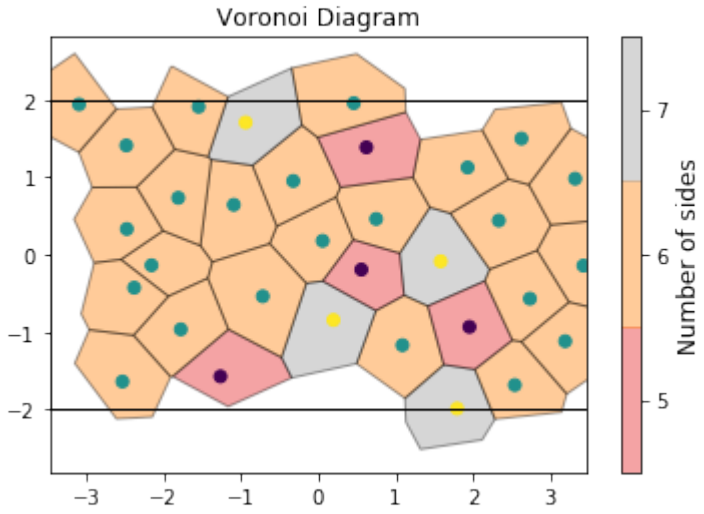
For noisy data, we see that the Voronoi diagram can change substantially. We perturb the positions with 2D Gaussian noise:

[8]:
```
box, points = hexagonal_lattice(rows=4, cols=4, noise=0.04)
voro = freud.voronoi.Voronoi(box, np.max(box.L)/2)
voro.compute(box=box, positions=points)
draw_voronoi(box, points, voro.polytopes)
```



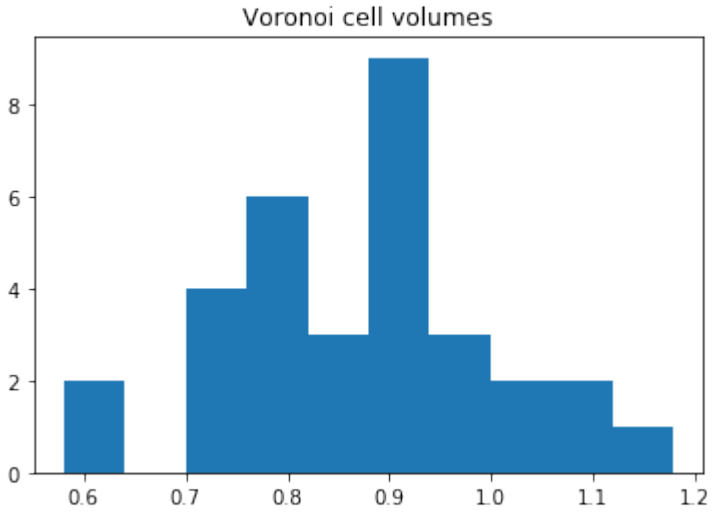
If we color by the number of sides of each Voronoi cell, we can see patterns in the defects: 5-gons and 7-gons tend to pair up.

[9]:
```
draw_voronoi(box, points, voro.polytopes, color_by_sides=True)
```



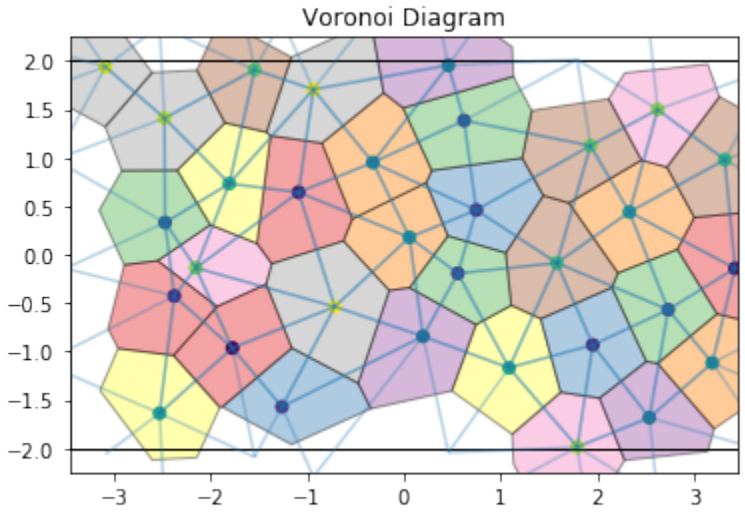
We can also compute the volumes of the Voronoi cells. Here, we plot them as a histogram:

[10]:
```
voro.computeVolumes()
plt.hist(voro.volumes)
plt.title('Voronoi cell volumes')
plt.show()
```



The `voronoi` module also provides `freud.locality.NeighborList` objects, where particles are neighbors if they share an edge in the Voronoi diagram. The `NeighborList` effectively represents the bonds in the Delaunay triangulation.

[11]:
```
voro.computeNeighbors(box=box, positions=points)
nlist = voro.nlist
draw_voronoi(box, points, voro.polytopes, nlist=nlist)
```

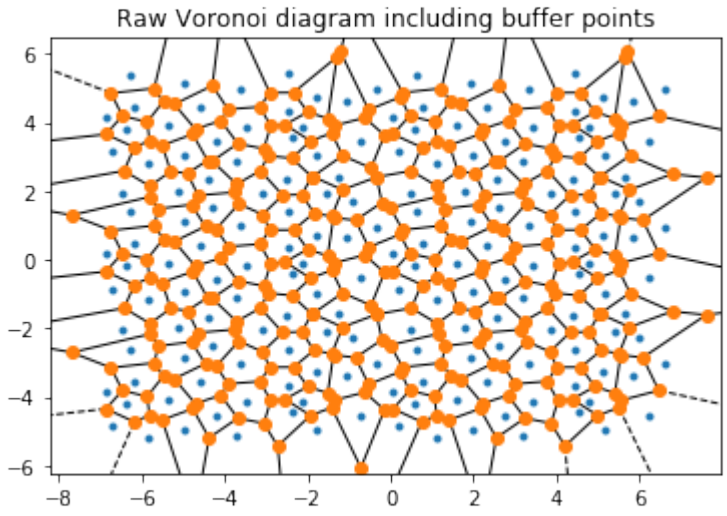


The `voronoi` property stores the raw `scipy.spatial.qhull.Voronoi` object from `scipy`.

Important: The plots below also show the *replicated buffer points*, not just the ones inside the periodic box. `freud.voronoi` is intended for use with periodic systems while `scipy.spatial.Voronoi` does not recognize periodic boundaries.

[12]:
```
print(type(voro.voronoi))
from scipy.spatial import voronoi_plot_2d
voronoi_plot_2d(voro.voronoi)
plt.title('Raw Voronoi diagram including buffer points')
plt.show()
```

```
<class 'scipy.spatial.qhull.Voronoi'>
```



1.3 Modules

Below is a list of modules in freud. To add your own module, read the *development guide*.

1.3.1 Box Module

Overview

freud.box.Box	The freud Box class for simulation boxes.
freud.box.ParticleBuffer	Replicates particles outside the box via periodic images.

Details

The *Box* class defines the geometry of a simulation box. The module natively supports periodicity by providing the fundamental features for wrapping vectors outside the box back into it. The *ParticleBuffer* class is used to replicate particles across the periodic boundary to assist analysis methods that do not recognize periodic boundary conditions or extend beyond the limits of one periodicity of the box.

Box

class freud.box.**Box**(*Lx*, *Ly*, *Lz*, *xy*, *xz*, *yz*, *is2D=None*)

The freud Box class for simulation boxes.

Module author: Richmond Newman <newmanrs@umich.edu>

Module author: Carl Simon Adorf <csadorf@umich.edu>

Module author: Bradley Dice <bdice@bradleydice.com>

Changed in version 0.7.0: Added box periodicity interface

The Box class is defined according to the conventions of the HOOMD-blue simulation software. For more information, please see:

http://hoomd-blue.readthedocs.io/en/stable/box.html

Parameters

- **Lx** (*float*) – Length of side x.
- **Ly** (*float*) – Length of side y.
- **Lz** (*float*) – Length of side z.
- **xy** (*float*) – Tilt of xy plane.
- **xz** (*float*) – Tilt of xz plane.
- **yz** (*float*) – Tilt of yz plane.
- **is2D** (*bool*) – Specify that this box is 2-dimensional, default is 3-dimensional.

Variables

- **xy** (*float*) – The xy tilt factor.
- **xz** (*float*) – The xz tilt factor.
- **yz** (*float*) – The yz tilt factor.
- **L** (*tuple, settable*) – The box lengths
- **Lx** (*tuple, settable*) – The x-dimension length.
- **Ly** (*tuple, settable*) – The y-dimension length.

- **Lz** (*tuple, settable*) – The z-dimension length.
- **Linv** (*tuple*) – The inverse box lengths.
- **volume** (*float*) – The box volume (area in 2D).
- **dimensions** (*int, settable*) – The number of dimensions (2 or 3).
- **periodic** (*list, settable*) – Whether or not the box is periodic.
- **periodic_x** (*bool, settable*) – Whether or not the box is periodic in x.
- **periodic_y** (*bool, settable*) – Whether or not the box is periodic in y.
- **periodic_z** (*bool, settable*) – Whether or not the box is periodic in z.

classmethod cube(*type cls*, *L=None*)
Construct a cubic box with equal lengths.

Parameters **L** (*float*) – The edge length

classmethod from_box(*type cls*, *box*, *dimensions=None*)
Initialize a box instance from a box-like object.

Parameters

- **box** – A box-like object
- **dimensions** (*int*) – Dimensionality of the box (Default value = None)

Note: Objects that can be converted to freud boxes include lists like `[Lx, Ly, Lz, xy, xz, yz]`, dictionaries with keys `'Lx', 'Ly', 'Lz', 'xy', 'xz', 'yz', 'dimensions'`, namedtuples with properties `Lx, Ly, Lz, xy, xz, yz, dimensions`, 3x3 matrices (see *from_matrix()*), or existing *freud.box.Box* objects.

If any of `Lz, xy, xz, yz` are not provided, they will be set to 0.

If all values are provided, a triclinic box will be constructed. If only `Lx, Ly, Lz` are provided, an orthorhombic box will be constructed. If only `Lx, Ly` are provided, a rectangular (2D) box will be constructed.

If the optional `dimensions` argument is given, this will be used as the box dimensionality. Otherwise, the box dimensionality will be detected from the `dimensions` of the provided box. If no dimensions can be detected, the box will be 2D if `Lz == 0`, and 3D otherwise.

Returns The resulting box object.

Return type *freud.box.Box*

classmethod from_matrix(*type cls*, *boxMatrix*, *dimensions=None*)
Initialize a box instance from a box matrix.

For more information and the source for this code, see: http://hoomd-blue.readthedocs.io/en/stable/box.html

Parameters

- **boxMatrix** (*array-like*) – A 3x3 matrix or list of lists
- **dimensions** (*int*) – Number of dimensions (Default value = None)

getImage
Returns the image corresponding to a wrapped vector.

New in version 0.8.

Parameters **vec** ((3) numpy.ndarray) – Coordinates of unwrapped vector.

Returns Image index vector.

Return type (3) numpy.ndarray

getLatticeVector
Get the lattice vector with index i.

Parameters **i** (*unsigned int*) – Index ($0 \leq i < d$) of the lattice vector, where d is the box dimension (2 or 3).

Returns Lattice vector with index i.

Return type list[float, float, float]

is2D
Return if box is 2D (True) or 3D (False).

Returns True if 2D, False if 3D.

Return type bool

makeCoordinates
Convert fractional coordinates into real coordinates.

Parameters **f** ((3) numpy.ndarray) – Fractional coordinates (x, y, z) between 0 and 1 within parallelepipedal box.

Returns Vector of real coordinates (x, y, z).

Return type list[float, float, float]

makeFraction
Convert real coordinates into fractional coordinates.

Parameters **vec** ((3) numpy.ndarray) – Real coordinates within parallelepipedal box.

Returns A fractional coordinate vector.

Return type list[float, float, float]

classmethod square(*type cls*, *L=None*)
Construct a 2-dimensional (square) box with equal lengths.

Parameters **L** (*float*) – The edge length

to_dict
Return box as dictionary.

Returns Box parameters

Return type dict

to_matrix
Returns the box matrix (3x3).

Returns box matrix

Return type list of lists, shape 3x3

to_tuple
Returns the box as named tuple.

Returns Box parameters

Return type namedtuple

unwrap
Unwrap a given array of vectors inside the box back into real space, using an array of image indices that determine how many times to unwrap in each dimension.

Parameters

- **vecs** ((3) or $(N, 3)$ numpy.ndarray) – Single vector or array of N vectors. The vectors are modified in place.
- **imgs** ((3) or $(N, 3)$ numpy.ndarray) – Single image index or array of N image indices.

Returns Vectors unwrapped by the image indices provided.

Return type (3) or $(N, 3)$ numpy.ndarray

wrap
Wrap a given array of vectors from real space into the box, using the periodic boundaries.

Note: Since the origin of the box is in the center, wrapping is equivalent to applying the minimum image convention to the input vectors.

Parameters **vecs** ((3) or $(N, 3)$ numpy.ndarray) – Single vector or array of N vectors. The vectors are altered in place and returned.

Returns Vectors wrapped into the box.

Return type (3) or $(N, 3)$ numpy.ndarray

Particle Buffer

class freud.box.**ParticleBuffer**(*box*)
Replicates particles outside the box via periodic images.

Module author: Ben Schultz <baschult@umich.edu>

Module author: Bradley Dice <bdice@bradleydice.com>

New in version 0.11.

Parameters **box** (*freud.box.Box*) – Simulation box.

Variables

- **buffer_particles** (numpy.ndarray) – The buffer particles.
- **buffer_ids** (numpy.ndarray) – The buffer ids.
- **buffer_box** (*freud.box.Box*) – The buffer box, expanded to hold the replicated particles.

compute
Compute the particle buffer.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points used to calculate particle buffer.
- **buffer** (*float*) – Buffer distance for replication outside the box.

- **images** (*bool*) – If `False` (default), `buffer` is a distance. If `True`, `buffer` is a number of images to replicate in each dimension. Note that one image adds half of a box length to each side, meaning that one image doubles the box side lengths, two images triples the box side lengths, and so on.

1.3.2 Cluster Module

Overview

freud.cluster.Cluster	Finds clusters in a set of points.
freud.cluster.ClusterProperties	Routines for computing properties of point clusters.

Details

The *freud.cluster* module aids in finding and computing the properties of clusters of points in a system.

Cluster

class freud.cluster.**Cluster**(*box*, *rcut*)

Finds clusters in a set of points.

Given a set of coordinates and a cutoff, *freud.cluster.Cluster* will determine all of the clusters of points that are made up of points that are closer than the cutoff. Clusters are 0-indexed. The class contains an index array, the `cluster_idx` attribute, which can be used to identify which cluster a particle is associated with: `cluster_obj.cluster_idx[i]` is the cluster index in which particle `i` is found. By the definition of a cluster, points that are not within the cutoff of another point end up in their own 1-particle cluster.

Identifying micelles is one primary use-case for finding clusters. This operation is somewhat different, though. In a cluster of points, each and every point belongs to one and only one cluster. However, because a string of points belongs to a polymer, that single polymer may be present in more than one cluster. To handle this situation, an optional layer is presented on top of the `cluster_idx` array. Given a key value per particle (i.e. the polymer id), the computeClusterMembership function will process `cluster_idx` with the key values in mind and provide a list of keys that are present in each cluster.

Module author: Joshua Anderson <joaander@umich.edu>

Parameters

- **box** (*freud.box.Box*) – The simulation box.
- **rcut** (*float*) – Particle distance cutoff.

Note: 2D: *freud.cluster.Cluster* properly handles 2D boxes. The points must be passed in as `[x, y, 0]`. Failing to set z=0 will lead to undefined behavior.

Variables

- ***box*** (*freud.box.Box*) – Box used in the calculation.
- **num_clusters** (*int*) – The number of clusters.
- **num_particles** (*int*) – The number of particles.
- **cluster_idx** ((*$N_{particles}$*) numpy.ndarray) – The cluster index for each particle.

- **cluster_keys** (*list (list)*) – A list of lists of the keys contained in each cluster.

computeClusterMembership
Compute the clusters with key membership. Loops over all particles and adds them to a list of sets. Each set contains all the keys that are part of that cluster. Get the computed list with `cluster_keys`.

Parameters **keys** (($N_{particles}$) numpy.ndarray) – Membership keys, one for each particle.

computeClusters
Compute the clusters for the given set of points.

Parameters

- **points** (($N_{particles}$, 3) np.ndarray) – Particle coordinates.
- **nlist** (*freud.locality.NeighborList*, optional) – Object to use to find bonds (Default value = None).
- **box** (*freud.box.Box*, optional) – Simulation box (Default value = None).

Cluster Properties

class freud.cluster.**ClusterProperties**(*box*)
Routines for computing properties of point clusters.

Given a set of points and cluster ids (from *Cluster*, or another source), ClusterProperties determines the following properties for each cluster:

- Center of mass
- Gyration tensor

The computed center of mass for each cluster (properly handling periodic boundary conditions) can be accessed with `getClusterCOM()`. This returns a ($N_{clusters}$, 3) numpy.ndarray.

The 3×3 gyration tensor G can be accessed with `getClusterG()`. This returns a numpy.ndarray, shape= ($N_{clusters} \times 3 \times 3$). The tensor is symmetric for each cluster.

Module author: Joshua Anderson <joaander@umich.edu>

Parameters **box** (*freud.box.Box*) – Simulation box.

Variables

- ***box*** (*freud.box.Box*) – Box used in the calculation.
- **num_clusters** (*int*) – The number of clusters.
- **cluster_COM** (($N_{clusters}$, 3) numpy.ndarray) – The center of mass of the last computed cluster.
- **cluster_G** (($N_{clusters}$, 3, 3) numpy.ndarray) – The cluster G tensors computed by the last call to `computeProperties()`.
- **cluster_sizes** (($N_{clusters}$) numpy.ndarray) – The cluster sizes computed by the last call to `computeProperties()`.

computeProperties
Compute properties of the point clusters. Loops over all points in the given array and determines the center of mass of the cluster as well as the G tensor. These can be accessed after the call to `computeProperties()` with `getClusterCOM()` and `getClusterG()`.

Parameters

- **points** ((*$N_{particles}$*, 3) np.ndarray) – Positions of the particles making up the clusters.
- **cluster_idx** (np.ndarray) – List of cluster indexes for each particle.
- **box** (*freud.box.Box*, optional) – Simulation box (Default value = None).

1.3.3 Density Module

Overview

freud.density.FloatCF	Computes the real pairwise correlation function.
freud.density.ComplexCF	Computes the complex pairwise correlation function.
freud.density.GaussianDensity	Computes the density of a system on a grid.
freud.density.LocalDensity	Computes the local density around a particle.
freud.density.RDF	Computes RDF for supplied data.

Details

The *freud.density* module contains various classes relating to the density of the system. These functions allow evaluation of particle distributions with respect to other particles.

Correlation Functions

class freud.density.**FloatCF**(*rmax*, *dr*)

Computes the real pairwise correlation function.

The correlation function is given by $C(r) = \langle s_1(0) \cdot s_2(r) \rangle$ between two sets of points p_1 (ref_points) and p_2 (points) with associated values s_1 (ref_values) and s_2 (values). Computing the correlation function results in an array of the expected (average) product of all values at a given radial distance r.

The values of r where the correlation function is computed are controlled by the rmax and dr parameters to the constructor. rmax determines the maximum distance at which to compute the correlation function and dr is the step size for each bin.

Note: 2D: *freud.density.FloatCF* properly handles 2D boxes. The points must be passed in as [x, y, 0]. Failing to set z=0 will lead to undefined behavior.

Note: Self-correlation: It is often the case that we wish to compute the correlation function of a set of points with itself. If points is the same as ref_points, not provided, or None, we omit accumulating the self-correlation value in the first bin.

Module author: Matthew Spellings <mspells@umich.edu>

Parameters

- **rmax** (*float*) – Maximum pointwise distance to include in the calculation.
- **dr** (*float*) – Bin size.

Variables

- ***RDF*** ((N_{bins}) numpy.ndarray) – Expected (average) product of all values whose radial distance falls within a given distance bin.
- ***box*** (*freud.box.Box*) – The box used in the calculation.
- **counts** ((N_{bins}) numpy.ndarray) – The number of points in each histogram bin.
- **R** ((N_{bins}) numpy.ndarray) – The centers of each bin.

accumulate

Calculates the correlation function and adds to the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** ((N_{ref_points}, 3) numpy.ndarray) – Reference points used to calculate the correlation function.
- **ref_values** ((N_{ref_points}) numpy.ndarray) – Real values used to calculate the correlation function.
- **points** ((N_{points}, 3) numpy.ndarray, optional) – Points used to calculate the correlation function. Uses ref_points if not provided or None.
- **values** ((N_{points}) numpy.ndarray, optional) – Real values used to calculate the correlation function. Uses ref_values if not provided or None.
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList to use to find bonds (Default value = None).

compute

Calculates the correlation function for the given points. Will overwrite the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** ((N_{ref_points}, 3) numpy.ndarray) – Reference points used to calculate the correlation function.
- **ref_values** ((N_{ref_points}) numpy.ndarray) – Real values used to calculate the correlation function.
- **points** ((N_{points}, 3) numpy.ndarray, optional) – Points used to calculate the correlation function. Uses ref_points if not provided or None.
- **values** ((N_{points}) numpy.ndarray, optional) – Real values used to calculate the correlation function. Uses ref_values if not provided or None.
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList to use to find bonds (Default value = None).

reset

Resets the values of the correlation function histogram in memory.

class freud.density.**ComplexCF**(*rmax*, *dr*)

Computes the complex pairwise correlation function.

The correlation function is given by $C(r) = \langle s_1(0) \cdot s_2(r) \rangle$ between two sets of points p_1 (ref_points) and p_2 (points) with associated values s_1 (ref_values) and s_2 (values). Computing the correlation function results in an array of the expected (average) product of all values at a given radial distance r.

The values of r where the correlation function is computed are controlled by the `rmax` and `dr` parameters to the constructor. `rmax` determines the maximum distance at which to compute the correlation function and `dr` is the step size for each bin.

Note: 2D: *freud.density.ComplexCF* properly handles 2D boxes. The points must be passed in as `[x, y, 0]`. Failing to set z=0 will lead to undefined behavior.

Note: Self-correlation: It is often the case that we wish to compute the correlation function of a set of points with itself. If `points` is the same as `ref_points`, not provided, or `None`, we omit accumulating the self-correlation value in the first bin.

Module author: Matthew Spellings <mspells@umich.edu>

Parameters

- **rmax** (*float*) – Maximum pointwise distance to include in the calculation.
- **dr** (*float*) – Bin size.

Variables

- ***RDF*** ((N_{bins}) `numpy.ndarray`) – Expected (average) product of all values at a given radial distance.
- ***box*** (*freud.box.Box*) – Box used in the calculation.
- **counts** ((N_{bins}) `numpy.ndarray`) – The number of points in each histogram bin.
- **R** ((N_{bins}) `numpy.ndarray`) – The centers of each bin.

accumulate

Calculates the correlation function and adds to the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** ((N_{ref_points}, 3) `numpy.ndarray`) – Reference points used to calculate the correlation function.
- **ref_values** ((N_{ref_points}) `numpy.ndarray`) – Complex values used to calculate the correlation function.
- **points** ((N_{points}, 3) `numpy.ndarray`, optional) – Points used to calculate the correlation function. Uses `ref_points` if not provided or `None`.
- **values** ((N_{points}) `numpy.ndarray`, optional) – Complex values used to calculate the correlation function. Uses `ref_values` if not provided or `None`.
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList to use to find bonds (Default value = `None`).

compute

Calculates the correlation function for the given points. Will overwrite the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** ((N_{ref_points}, 3) `numpy.ndarray`) – Reference points used to calculate the correlation function.

- **ref_values** ((N_{ref_points}) numpy.ndarray) – Complex values used to calculate the correlation function.
- **points** ((N_{points}, 3) numpy.ndarray, optional) – Points used to calculate the correlation function. Uses ref_points if not provided or None.
- **values** ((N_{points}) numpy.ndarray, optional) – Complex values used to calculate the correlation function. Uses ref_values if not provided or None.
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList to use to find bonds (Default value = None).

reset
Resets the values of the correlation function histogram in memory.

Gaussian Density

class freud.density.**GaussianDensity**(**args*)
Computes the density of a system on a grid.

Replaces particle positions with a Gaussian blur and calculates the contribution from each to the proscribed grid based upon the distance of the grid cell from the center of the Gaussian. The resulting data is a regular grid of particle densities that can be used in standard algorithms requiring evenly spaced point, such as Fast Fourier Transforms. The dimensions of the image (grid) are set in the constructor, and can either be set equally for all dimensions or for each dimension independently.

- Constructor Calls:

  Initialize with all dimensions identical:

```
freud.density.GaussianDensity(width, r_cut, dr)
```

  Initialize with each dimension specified:

```
freud.density.GaussianDensity(width_x, width_y, width_z, r_cut, dr)
```

Module author: Joshua Anderson <joaander@umich.edu>

Parameters

- **width** (*unsigned int*) – Number of pixels to make the image.
- **width_x** (*unsigned int*) – Number of pixels to make the image in x.
- **width_y** (*unsigned int*) – Number of pixels to make the image in y.
- **width_z** (*unsigned int*) – Number of pixels to make the image in z.
- **r_cut** (*float*) – Distance over which to blur.
- **sigma** (*float*) – Sigma parameter for Gaussian.

Variables

- ***box*** (*freud.box.Box*) – Box used in the calculation.
- **gaussian_density** ((w_x, w_y, w_z) numpy.ndarray) – The image grid with the Gaussian density.
- **counts** ((N_{bins}) numpy.ndarray) – The number of points in each histogram bin.
- **R** ((N_{bins}) numpy.ndarray) – The centers of each bin.

compute
Calculates the Gaussian blur for the specified points. Does not accumulate (will overwrite current image).

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **points** ((N_{points}, 3) numpy.ndarray) – Points to calculate the local density.

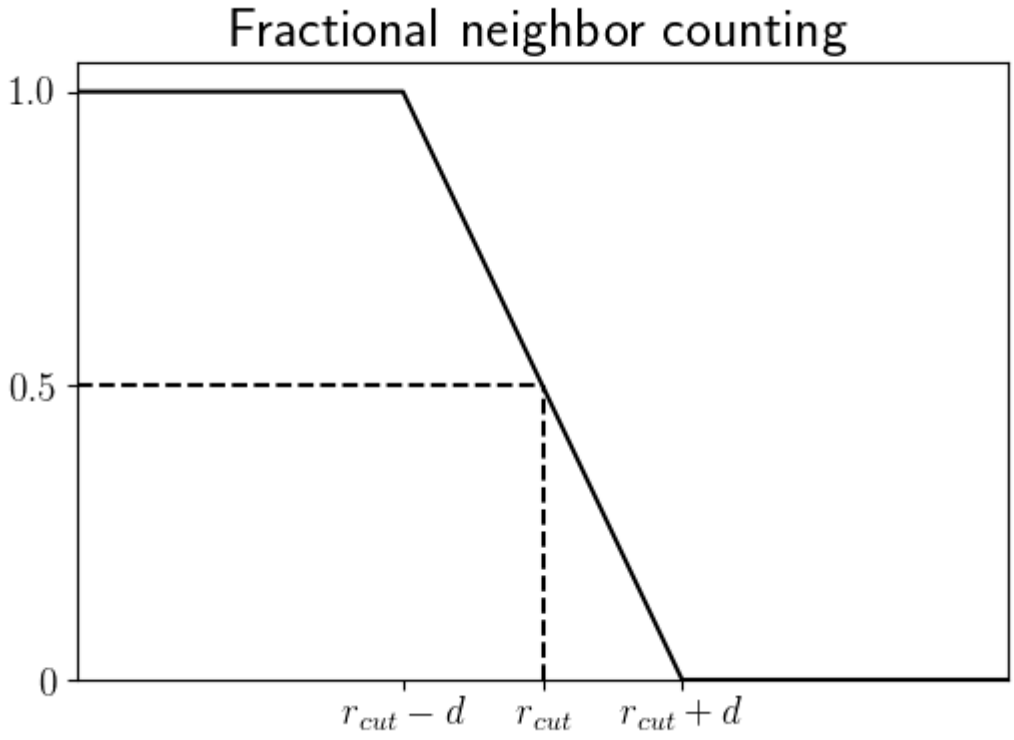
Local Density

class freud.density.**LocalDensity**(*r_cut*, *volume*, *diameter*)
Computes the local density around a particle.

The density of the local environment is computed and averaged for a given set of reference points in a sea of data points. Providing the same points calculates them against themselves. Computing the local density results in an array listing the value of the local density around each reference point. Also available is the number of neighbors for each reference point, giving the user the ability to count the number of particles in that region.

The values to compute the local density are set in the constructor. `r_cut` sets the maximum distance at which data points are included relative to a given reference point. `volume` is the volume of a single data points, and `diameter` is the diameter of the circumsphere of an individual data point. Note that the volume and diameter do not affect the reference point; whether or not data points are counted as neighbors of a given reference point is entirely determined by the distance between reference point and data point center relative to `r_cut` and the `diameter` of the data point.

In order to provide sufficiently smooth data, data points can be fractionally counted towards the density. Rather than perform compute-intensive area (volume) overlap calculations to determine the exact amount of overlap area (volume), the LocalDensity class performs a simple linear interpolation relative to the centers of the data points. Specifically, a point is counted as one neighbor of a given reference point if it is entirely contained within the `r_cut`, half of a neighbor if the distance to its center is exactly `r_cut`, and zero if its center is a distance greater than or equal to `r_cut + diameter` from the reference point's center. Graphically, this looks like:



Note: 2D: `freud.density.LocalDensity` properly handles 2D boxes. The points must be passed in as `[x, y, 0]`. Failing to set z=0 will lead to undefined behavior.

Module author: Joshua Anderson <joaander@umich.edu>

Parameters

- **r_cut** (*float*) – Maximum distance over which to calculate the density.
- **volume** (*float*) – Volume of a single particle.
- **diameter** (*float*) – Diameter of particle circumsphere.

Variables

- ***box*** (*freud.box.Box*) – Box used in the calculation.
- ***density*** ((N_{ref_points}) numpy.ndarray) – Density of points per ref_point.
- **num_neighbors** ((N_{ref_points}) numpy.ndarray) – Number of neighbor points for each ref_point.

compute

Calculates the local density for the specified points. Does not accumulate (will overwrite current data).

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** ((N_{ref_points}, 3) numpy.ndarray) – Reference points to calculate the local density.

- **points** ((N_{points}, 3) numpy.ndarray, optional) – Points to calculate the local density. Uses ref_points if not provided or None.
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList to use to find bonds (Default value = None).

Radial Distribution Function

class freud.density.**RDF**(*rmax*, *dr*, *rmin=0*)

Computes RDF for supplied data.

The RDF ($g(r)$) is computed and averaged for a given set of reference points in a sea of data points. Providing the same points calculates them against themselves. Computing the RDF results in an RDF array listing the value of the RDF at each given r, listed in the R array.

The values of r to compute the RDF are set by the values of rmin, rmax, dr in the constructor. rmax sets the maximum distance at which to calculate the $g(r)$, rmin sets the minimum distance at which to calculate the $g(r)$, and dr determines the step size for each bin.

Module author: Eric Harper <harperic@umich.edu>

Note: **2D:** *freud.density.RDF* properly handles 2D boxes. The points must be passed in as [x, y, 0]. Failing to set z=0 will lead to undefined behavior.

Parameters

- **rmax** (*float*) – Maximum interparticle distance to include in the calculation.
- **dr** (*float*) – Distance between histogram bins.
- **rmin** (*float, optional*) – Minimum interparticle distance to include in the calculation. Defaults to 0.

Variables

- ***box*** (*freud.box.Box*) – Box used in the calculation.
- ***RDF*** ((N_{bins},) numpy.ndarray) – Histogram of RDF values.
- **R** ((N_{bins}) numpy.ndarray) – The centers of each bin.
- **n_r** ((N_{bins},) numpy.ndarray) – Histogram of cumulative RDF values (*i.e.* the integrated RDF).

Changed in version 0.7.0: Added optional *rmin* argument.

accumulate

Calculates the RDF and adds to the current RDF histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** ((N_{ref_points}, 3) numpy.ndarray) – Reference points used to calculate the RDF.
- **points** ((N_{points}, 3) numpy.ndarray, optional) – Points used to calculate the RDF. Uses ref_points if not provided or None.
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList to use to find bonds (Default value = None).

compute
Calculates the RDF for the specified points. Will overwrite the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** ((N_{ref_points}, 3) numpy.ndarray) – Reference points used to calculate the RDF.
- **points** ((N_{points}, 3) numpy.ndarray, optional) – Points used to calculate the RDF. Uses ref_points if not provided or None.
- **nlist** (*freud.locality.NeighborList*) – NeighborList to use to find bonds (Default value = None).

reset
Resets the values of RDF in memory.

1.3.4 Environment Module

Overview

freud.environment.BondOrder	Compute the bond orientational order diagram for the system of particles.
freud.environment.LocalDescriptors	Compute a set of descriptors (a numerical "fingerprint") of a particle's local environment.
freud.environment.MatchEnv	Clusters particles according to whether their local environments match or not, according to various shape matching metrics.
freud.environment.AngularSeparation	Calculates the minimum angles of separation between particles and references.

Details

The *freud.environment* module contains functions which characterize the local environments of particles in the system. These methods use the positions and orientations of particles in the local neighborhood of a given particle to characterize the particle environment.

Bond Order

class freud.environment.**BondOrder**(*rmax*, *k*, *n*, *nBinsT*, *nBinsP*)
Compute the bond orientational order diagram for the system of particles.

The bond orientational order diagram (BOOD) is a way of studying the average local environments experienced by particles. In a BOOD, a particle and its nearest neighbors (determined by either a prespecified number of neighbors or simply a cutoff distance) are treated as connected by a bond joining their centers. All of the bonds in the system are then binned by their azimuthal (θ) and polar (ϕ) angles to indicate the location of a particle's neighbors relative to itself. The distance between the particle and its neighbor is only important when determining whether it is counted as a neighbor, but is not part of the BOOD; as such, the BOOD can be viewed as a projection of all bonds onto the unit sphere. The resulting 2D histogram provides insight into how particles are situated relative to one-another in a system.

This class provides access to the classical BOOD as well as a few useful variants. These variants can be accessed *via* the mode arguments to the *compute()* or *accumulate()* methods. Available modes of calculation are:

- `'bod'` (Bond Order Diagram, *default*): This mode constructs the default BOOD, which is the 2D histogram containing the number of bonds formed through each azimuthal (θ) and polar (ϕ) angle.
- `'lbod'` (Local Bond Order Diagram): In this mode, a particle's neighbors are rotated into the local frame of the particle before the BOOD is calculated, *i.e.* the directions of bonds are determined relative to the orientation of the particle rather than relative to the global reference frame. An example of when this mode would be useful is when a system is composed of multiple grains of the same crystal; the normal BOOD would show twice as many peaks as expected, but using this mode, the bonds would be superimposed.
- `'obcd'` (Orientation Bond Correlation Diagram): This mode aims to quantify the degree of orientational as well as translational ordering. As a first step, the rotation that would align a particle's neighbor with the particle is calculated. Then, the neighbor is rotated **around the central particle** by that amount, which actually changes the direction of the bond. One example of how this mode could be useful is in identifying plastic crystals, which exhibit translational but not orientational ordering. Normally, the BOOD for a plastic crystal would exhibit clear structure since there is translational order, but with this mode, the neighbor positions would actually be modified, resulting in an isotropic (disordered) BOOD.
- `'oocd'` (Orientation Orientation Correlation Diagram): This mode is substantially different from the other modes. Rather than compute the histogram of neighbor bonds, this mode instead computes a histogram of the directors of neighboring particles, where the director is defined as the basis vector $\hat{z}$ rotated by the neighbor's quaternion. The directors are then rotated into the central particle's reference frame. This mode provides insight into the local orientational environment of particles, indicating, on average, how a particle's neighbors are oriented.

Module author: Erin Teich <erteich@umich.edu>

Parameters

- **r_max** (*float*) – Distance over which to calculate.
- **k** (*unsigned int*) – Order parameter i. To be removed.
- **n** (*unsigned int*) – Number of neighbors to find.
- **n_bins_t** (*unsigned int*) – Number of θ bins.
- **n_bins_p** (*unsigned int*) – Number of ϕ bins.

Variables

- **bond_order** ((N_ϕ, N_θ) numpy.ndarray) – Bond order.
- ***box*** (*freud.box.Box*) – Box used in the calculation.
- **theta** ((N_θ) numpy.ndarray) – The values of bin centers for θ.
- **phi** ((N_ϕ) numpy.ndarray) – The values of bin centers for ϕ.
- **n_bins_theta** (*unsigned int*) – The number of bins in the θ dimension.
- **n_bins_phi** (*unsigned int*) – The number of bins in the ϕ dimension.

accumulate

Calculates the correlation function and adds to the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** (($N_{particles}$, 3) numpy.ndarray) – Reference points used to calculate bonds.
- **ref_orientations** (($N_{particles}$, 4) numpy.ndarray) – Reference orientations used to calculate bonds.

- **points** (($N_{particles}$, 3) numpy.ndarray, optional) – Points used to calculate bonds. Uses ref_points if not provided or None.
- **orientations** (($N_{particles}$, 3) numpy.ndarray) – Orientations used to calculate bonds. Uses ref_orientations if not provided or None.
- **mode** (*str, optional*) – Mode to calculate bond order. Options are 'bod', 'lbod', 'obcd', or 'oocd' (Default value = 'bod').
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList to use to find bonds (Default value = None).

compute

Calculates the bond order histogram. Will overwrite the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** (($N_{particles}$, 3) numpy.ndarray) – Reference points used to calculate bonds.
- **ref_orientations** (($N_{particles}$, 4) numpy.ndarray) – Reference orientations used to calculate bonds.
- **points** (($N_{particles}$, 3) numpy.ndarray, optional) – Points used to calculate bonds. Uses ref_points if not provided or None.
- **orientations** (($N_{particles}$, 3) numpy.ndarray, optional) – Orientations used to calculate bonds. Uses ref_orientations if not provided or None.
- **mode** (*str, optional*) – Mode to calculate bond order. Options are 'bod', 'lbod', 'obcd', or 'oocd' (Default value = 'bod').
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList to use to find bonds (Default value = None).

reset

Resets the values of the bond order in memory.

Local Descriptors

class freud.environment.**LocalDescriptors**(*num_neighbors*, *lmax*, *rmax*, *negative_m=True*)

Compute a set of descriptors (a numerical "fingerprint") of a particle's local environment.

The resulting spherical harmonic array will be a complex-valued array of shape *(num_bonds, num_sphs)*. Spherical harmonic calculation can be restricted to some number of nearest neighbors through the *num_neighbors* argument; if a particle has more bonds than this number, the last one or more rows of bond spherical harmonics for each particle will not be set.

Module author: Matthew Spellings <mspells@umich.edu>

Parameters

- **num_neighbors** (*unsigned int*) – Maximum number of neighbors to compute descriptors for.
- **lmax** (*unsigned int*) – Maximum spherical harmonic l to consider.
- **rmax** (*float*) – Initial guess of the maximum radius to looks for neighbors.
- **negative_m** (*bool*) – True if we should also calculate Y_{lm} for negative m.

Variables

- **sph** ((*N_{bonds}*, SphWidth) numpy.ndarray) – A reference to the last computed spherical harmonic array.
- **num_particles** (*unsigned int*) – The number of points passed to the last call to compute().
- **num_neighbors** (*unsigned int*) – The number of neighbors used by the last call to compute. Bounded from above by the number of reference points multiplied by the lower of the num_neighbors arguments passed to the last compute call or the constructor.
- **l_max** (*unsigned int*) – The maximum spherical harmonic *l* to calculate for.
- **r_max** (*float*) – The cutoff radius.

compute

Calculates the local descriptors of bonds from a set of source points to a set of destination points.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **num_neighbors** (*unsigned int*) – Number of neighbors to compute with or to limit to, if the neighbor list is precomputed.
- **points_ref** ((*$N_{particles}$*, 3) numpy.ndarray) – Source points to calculate the order parameter.
- **points** ((*$N_{particles}$*, 3) numpy.ndarray, optional) – Destination points to calculate the order parameter (Default value = None).
- **orientations** ((*$N_{particles}$*, 4) numpy.ndarray, optional) – Orientation of each reference point (Default value = None).
- **mode** (*str, optional*) – Orientation mode to use for environments, either 'neighborhood' to use the orientation of the local neighborhood, 'particle_local' to use the given particle orientations, or 'global' to not rotate environments (Default value = 'neighborhood').
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList to use to find bonds or 'precomputed' if using computeNList() (Default value = None).

computeNList

Compute the neighbor list for bonds from a set of source points to a set of destination points.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **points_ref** ((*$N_{particles}$*, 3) numpy.ndarray) – Source points to calculate the order parameter.
- **points** ((*$N_{particles}$*, 3) numpy.ndarray, optional) – Destination points to calculate the order parameter (Default value = None).

Match Environments

class freud.environment.**MatchEnv**(*box*, *rmax*, *k*)

Clusters particles according to whether their local environments match or not, according to various shape matching metrics.

Module author: Erin Teich <erteich@umich.edu>

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **rmax** (*float*) – Cutoff radius for cell list and clustering algorithm. Values near the first minimum of the RDF are recommended.
- **k** (*unsigned int*) – Number of nearest neighbors taken to define the local environment of any given particle.

Variables

- **tot_environment** (($N_{particles}, N_{neighbors}, 3$) numpy.ndarray) – All environments for all particles.
- **num_particles** (*unsigned int*) – The number of particles.
- **num_clusters** (*unsigned int*) – The number of clusters.
- **clusters** (($N_{particles}$) numpy.ndarray) – The per-particle index indicating cluster membership.

cluster

Determine clusters of particles with matching environments.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Destination points to calculate the order parameter.
- **threshold** (*float*) – Maximum magnitude of the vector difference between two vectors, below which they are "matching."
- **hard_r** (*bool*) – If True, add all particles that fall within the threshold of m_rmaxsq to the environment.
- **registration** (*bool*) – If True, first use brute force registration to orient one set of environment vectors with respect to the other set such that it minimizes the RMSD between the two sets.
- **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.
- **env_nlist** (*freud.locality.NeighborList*, optional) – NeighborList to use to find the environment of every particle (Default value = None).
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList to use to find neighbors of every particle, to compare environments (Default value = None).

getEnvironment

Returns the set of vectors defining the environment indexed by i.

Parameters **i** (*unsigned int*) – Environment index.

Returns The array of vectors.

Return type ($N_{neighbors}, 3$) numpy.ndarray

isSimilar

Test if the motif provided by refPoints1 is similar to the motif provided by refPoints2.

Parameters

- **refPoints1** (($N_{particles}$, 3) numpy.ndarray) – Vectors that make up motif 1.
- **refPoints2** (($N_{particles}$, 3) numpy.ndarray) – Vectors that make up motif 2.

- **threshold** (*float*) – Maximum magnitude of the vector difference between two vectors, below which they are considered "matching."
- **registration** (*bool, optional*) – If True, first use brute force registration to orient one set of environment vectors with respect to the other set such that it minimizes the RMSD between the two sets (Default value = False).

Returns A doublet that gives the rotated (or not) set of `refPoints2`, and the mapping between the vectors of `refPoints1` and `refPoints2` that will make them correspond to each other. Empty if they do not correspond to each other.

Return type tuple (((*N*$_{particles}$, 3) `numpy.ndarray`), map[int, int])

matchMotif

Determine clusters of particles that match the motif provided by refPoints.

Parameters

- **points** ((*N*$_{particles}$, 3) `numpy.ndarray`) – Particle positions.
- **refPoints** ((*N*$_{particles}$, 3) `numpy.ndarray`) – Vectors that make up the motif against which we are matching.
- **threshold** (*float*) – Maximum magnitude of the vector difference between two vectors, below which they are considered "matching."
- **registration** (*bool, optional*) – If True, first use brute force registration to orient one set of environment vectors with respect to the other set such that it minimizes the RMSD between the two sets (Default value = False).
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList to use to find bonds (Default value = `None`).

minRMSDMotif

Rotate (if registration=True) and permute the environments of all particles to minimize their RMSD with respect to the motif provided by refPoints.

Parameters

- **points** ((*N*$_{particles}$, 3) `numpy.ndarray`) – Particle positions.
- **refPoints** ((*N*$_{particles}$, 3) `numpy.ndarray`) – Vectors that make up the motif against which we are matching.
- **registration** (*bool, optional*) – If True, first use brute force registration to orient one set of environment vectors with respect to the other set such that it minimizes the RMSD between the two sets (Default value = False).
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList to use to find bonds (Default value = `None`).

Returns Vector of minimal RMSD values, one value per particle.

Return type (*N*$_{particles}$) `numpy.ndarray`

minimizeRMSD

Get the somewhat-optimal RMSD between the set of vectors refPoints1 and the set of vectors refPoints2.

Parameters

- **refPoints1** ((*N*$_{particles}$, 3) `numpy.ndarray`) – Vectors that make up motif 1.
- **refPoints2** ((*N*$_{particles}$, 3) `numpy.ndarray`) – Vectors that make up motif 2.

- **registration** (*bool, optional*) – If true, first use brute force registration to orient one set of environment vectors with respect to the other set such that it minimizes the RMSD between the two sets (Default value = False).

Returns A triplet that gives the associated min_rmsd, rotated (or not) set of refPoints2, and the mapping between the vectors of refPoints1 and refPoints2 that somewhat minimizes the RMSD.

Return type tuple (float, ($(N_{particles}, 3)$ numpy.ndarray), map[int, int])

setBox

Reset the simulation box.

Parameters **box** (*freud.box.Box*) – Simulation box.

Angular Separation

class freud.environment.**AngularSeparation**(*box*, *rmax*, *n*)

Calculates the minimum angles of separation between particles and references.

Module author: Erin Teich <erteich@umich.edu>

Module author: Andrew Karas <askaras@umich.edu>

Parameters

- **rmax** (*float*) – Cutoff radius for cell list and clustering algorithm. Values near the first minimum of the RDF are recommended.
- **n** (*int*) – The number of neighbors.

Variables

- **nlist** (*freud.locality.NeighborList*) – The neighbor list.
- **n_p** (*unsigned int*) – The number of particles used in computing the last set.
- **n_ref** (*unsigned int*) – The number of reference particles used in computing the neighbor angles.
- **n_global** (*unsigned int*) – The number of global orientations to check against.
- **neighbor_angles** ((N_{bonds}) numpy.ndarray) – The neighbor angles in radians. **This field is only populated after** computeNeighbor() **is called.** The angles are stored in the order of the neighborlist object.
- **global_angles** ($(N_{global}, N_{particles})$ numpy.ndarray) – The global angles in radians. **This field is only populated after** computeGlobal() **is called.** The angles are stored in the order of the neighborlist object.

computeGlobal

Calculates the minimum angles of separation between global_ors and ors, checking for underlying symmetry as encoded in equiv_quats. The result is stored in the global_angles class attribute.

Parameters

- **ors** ($(N_{particles}$, 3) numpy.ndarray) – Orientations to calculate the order parameter.
- **global_ors** ($(N_{particles}$, 4) numpy.ndarray) – Reference orientations to calculate the order parameter.
- **equiv_quats** ($(N_{particles}$, 4) numpy.ndarray) – The set of all equivalent quaternions that takes the particle as it is defined to some global reference orientation. Important: equiv_quats must include both q and $-q$, for all included quaternions.

computeNeighbor
Calculates the minimum angles of separation between ref_ors and ors, checking for underlying symmetry as encoded in equiv_quats. The result is stored in the `neighbor_angles` class attribute.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_ors** (($N_{particles}$, 4) numpy.ndarray) – Reference orientations used to calculate the order parameter.
- **ors** (($N_{particles}$, 3) numpy.ndarray) – Orientations used to calculate the order parameter.
- **ref_points** (($N_{particles}$, 3) numpy.ndarray) – Reference points used to calculate the order parameter.
- **points** (($N_{particles}$, 3) numpy.ndarray) – Points used to calculate the order parameter.
- **equiv_quats** (($N_{particles}$, 4) numpy.ndarray, optional) – The set of all equivalent quaternions that takes the particle as it is defined to some global reference orientation. Important: `equiv_quats` must include both q and $-q$, for all included quaternions.
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList to use to find bonds (Default value = None).

1.3.5 Index Module

Overview

freud.index.Index2D	freud-style indexer for flat arrays.
freud.index.Index3D	freud-style indexer for flat arrays.

Details

The *freud.index* module exposes the 1-dimensional indexer utilized in freud at the C++ level. At the C++ level, freud utilizes flat arrays to represent multidimensional arrays. N-dimensional arrays with n_i elements in each dimension i are represented as 1-dimensional arrays with $\prod_{i=1}^{N} n_i$ elements.

Index2D

class freud.index.**Index2D**(**args*)
freud-style indexer for flat arrays.

Once constructed, the object provides direct access to the flat index equivalent:

- Constructor Calls:

  Initialize with all dimensions identical:

```
freud.index.Index2D(w)
```

  Initialize with each dimension specified:

```
freud.index.Index2D(w, h)
```

Note: freud indexes column-first i.e. `Index2D(i, j)` will return the 1-dimensional index of the i^{th} column and the j^{th} row. This is the opposite of what occurs in a numpy array, in which `array[i, j]` returns the element in the i^{th} row and the j^{th} column.

Module author: Joshua Anderson <joaander@umich.edu>

Parameters

- **w** (*unsigned int*) – Width of 2D array (number of columns).
- **h** (*unsigned int*) – Height of 2D array (number of rows).

Variables **num_elements** (*unsigned int*) – Number of elements in the array.

Example:

```
index = Index2D(10)
i = index(3, 5)
```

__call__(*self*, *i*, *j*)

Parameters

- **i** (*unsigned int*) – Column index.
- **j** (*unsigned int*) – Row index.

Returns Index in flat (*e.g.* 1-dimensional) array.

Return type unsigned int

Index3D

class freud.index.**Index3D**(**args*)

freud-style indexer for flat arrays.

Once constructed, the object provides direct access to the flat index equivalent:

- Constructor Calls:

  Initialize with all dimensions identical:

  ```
  freud.index.Index3D(w)
  ```

  Initialize with each dimension specified:

  ```
  freud.index.Index3D(w, h, d)
  ```

Note: freud indexes column-first i.e. Index3D(i, j, k) will return the 1-dimensional index of the i^{th} column, j^{th} row, and the k^{th} frame. This is the opposite of what occurs in a numpy array, in which `array[i, j, k]` returns the element in the i^{th} frame, j^{th} row, and the k^{th} column.

Module author: Joshua Anderson <joaander@umich.edu>

Parameters

- **w** (*unsigned int*) – Width of 2D array (number of columns).
- **h** (*unsigned int*) – Height of 2D array (number of rows).

- **d** (*unsigned int*) – Depth of 2D array (number of frames).

Variables **num_elements** (*unsigned int*) – Number of elements in the array.

Example:

```
index = Index3D(10)
i = index(3, 5, 4)
```

__call__(*self*, *i*, *j*, *k*)

Parameters

- **i** (*unsigned int*) – Column index.
- **j** (*unsigned int*) – Row index.
- **k** (*unsigned int*) – Frame index.

Returns Index in flat (*e.g.* 1-dimensional) array.

Return type unsigned int

1.3.6 Interface Module

Overview

freud.interface.InterfaceMeasure	Measures the interface between two sets of points.

Details

The *freud.interface* module contains functions to measure the interface between sets of points.

Interface Measure

class freud.interface.**InterfaceMeasure**(*box*, *r_cut*)

Measures the interface between two sets of points.

Module author: Matthew Spellings <*mspells@umich.edu*>

Module author: Bradley Dice <*bdice@bradleydice.com*>

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **r_cut** (*float*) – Distance to search for particle neighbors.

Variables

- **ref_point_count** (*int*) – Number of particles from `ref_points` on the interface.
- **ref_point_ids** (`np.ndarray`) – The particle IDs from `ref_points`.
- **point_count** (*int*) – Number of particles from `points` on the interface.
- **point_ids** (`np.ndarray`) – The particle IDs from `points`.

compute

Compute the particles at the interface between the two given sets of points.

Parameters

- **ref_points** (($N_{particles}$, 3) numpy.ndarray) – One set of particle positions.
- **points** (($N_{particles}$, 3) numpy.ndarray) – Other set of particle positions.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

1.3.7 Locality Module

Overview

freud.locality.NeighborList	Class representing a certain number of "bonds" between particles.
freud.locality.IteratorLinkCell	Iterates over the particles in a cell.
freud.locality.LinkCell	Supports efficiently finding all points in a set within a certain distance from a given point.
freud.locality.NearestNeighbors	Supports efficiently finding the N nearest neighbors of each point in a set for some fixed integer N.

Details

The *freud.locality* module contains data structures to efficiently locate points based on their proximity to other points.

Neighbor List

class freud.locality.**NeighborList**

Class representing a certain number of "bonds" between particles. Computation methods will iterate over these bonds when searching for neighboring particles.

NeighborList objects are constructed for two sets of position arrays A (alternatively *reference points*; of length n_A) and B (alternatively *target points*; of length n_B) and hold a set of $(i, j) : i < n_A, j < n_B$ index pairs corresponding to near-neighbor points in A and B, respectively.

For efficiency, all bonds for a particular reference particle i are contiguous and bonds are stored in order based on reference particle index i. The first bond index corresponding to a given particle can be found in $\log(n_{bonds})$ time using *find_first_index()*.

Module author: Matthew Spellings <mspells@umich.edu>

New in version 0.6.4.

Note: Typically, there is no need to instantiate this class directly. In most cases, users should manipulate *freud.locality.NeighborList* objects received from a neighbor search algorithm, such as *freud.locality.LinkCell*, *freud.locality.NearestNeighbors*, or *freud.voronoi.Voronoi*.

Variables

- **index_i** (np.ndarray) – The reference point indices from the last set of points this object was evaluated with. This array is read-only to prevent breakage of find_first_index().

- **index_j** (np.ndarray) – The reference point indices from the last set of points this object was evaluated with. This array is read-only to prevent breakage of find_first_index().
- **weights** ((N_{bonds}) np.ndarray) – The per-bond weights from the last set of points this object was evaluated with.
- **segments** ((N_{ref_points}) np.ndarray) – A segment array, which is an array of length N_{ref} indicating the first bond index for each reference particle from the last set of points this object was evaluated with.
- **neighbor_counts** ((N_{ref_points}) np.ndarray) – A neighbor count array, which is an array of length N_{ref} indicating the number of neighbors for each reference particle from the last set of points this object was evaluated with.

Example:

```
# Assume we have position as Nx3 array
lc = LinkCell(box, 1.5).compute(box, positions)
nlist = lc.nlist

# Get all vectors from central particles to their neighbors
rijs = positions[nlist.index_j] - positions[nlist.index_i]
box.wrap(rijs)
```

copy

Create a copy. If other is given, copy its contents into this object. Otherwise, return a copy of this object.

Parameters **other** (*freud.locality.NeighborList*, optional) – A NeighborList to copy into this object (Default value = None).

filter

Removes bonds that satisfy a boolean criterion.

Parameters **filt** (np.ndarray) – Boolean-like array of bonds to keep (True means the bond will not be removed).

Note: This method modifies this object in-place.

Example:

```
# Keep only the bonds between particles of type A and type B
nlist.filter(types[nlist.index_i] != types[nlist.index_j])
```

filter_r

Removes bonds that are outside of a given radius range.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** (($N_{particles}$, 3) numpy.ndarray) – Reference points to use for filtering.
- **points** (($N_{particles}$, 3) numpy.ndarray) – Target points to use for filtering.
- **rmax** (*float*) – Maximum bond distance in the resulting neighbor list.
- **rmin** (*float, optional*) – Minimum bond distance in the resulting neighbor list (Default value = 0).

find_first_index
Returns the lowest bond index corresponding to a reference particle with an index $\geq i$.

Parameters **i** (*unsigned int*) – The particle index.

classmethod from_arrays(*type cls*, *Nref*, *Ntarget*, *index_i*, *index_j*, *weights=None*)
Create a NeighborList from a set of bond information arrays.

Parameters

- **Nref** (*int*) – Number of reference points (corresponding to `index_i`).
- **Ntarget** (*int*) – Number of target points (corresponding to `index_j`).
- **index_i** (`np.ndarray`) – Array of integers corresponding to indices in the set of reference points.
- **index_j** (`np.ndarray`) – Array of integers corresponding to indices in the set of target points.
- **weights** (`np.ndarray`, optional) – Array of per-bond weights (if `None` is given, use a value of 1 for each weight) (Default value = `None`).

Cell Lists

class freud.locality.**IteratorLinkCell**
Iterates over the particles in a cell.

Module author: Joshua Anderson <*joaander@umich.edu*>

Example:

```
# Grab particles in cell 0
for j in linkcell.itercell(0):
    print(positions[j])
```

next
Implements iterator interface

class freud.locality.**LinkCell**(*box*, *cell_width*)
Supports efficiently finding all points in a set within a certain distance from a given point.

Module author: Joshua Anderson <*joaander@umich.edu*>

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **cell_width** (*float*) – Maximum distance to find particles within.

Variables

- ***box*** (*freud.box.Box*) – Simulation box.
- **num_cells** (*unsigned int*) – The number of cells in the box.
- **nlist** (*freud.locality.NeighborList*) – The neighbor list stored by this object, generated by `compute()`.

Note: 2D: *freud.locality.LinkCell* properly handles 2D boxes. The points must be passed in as `[x, y, 0]`. Failing to set z=0 will lead to undefined behavior.

Example:

```
# Assume positions are an Nx3 array
lc = LinkCell(box, 1.5)
lc.compute(box, positions)
for i in range(positions.shape[0]):
    # Cell containing particle i
    cell = lc.getCell(positions[0])
    # List of cell's neighboring cells
    cellNeighbors = lc.getCellNeighbors(cell)
    # Iterate over neighboring cells (including our own)
    for neighborCell in cellNeighbors:
        # Iterate over particles in each neighboring cell
        for neighbor in lc.itercell(neighborCell):
            pass # Do something with neighbor index

# Using NeighborList API
dens = density.LocalDensity(1.5, 1, 1)
dens.compute(box, positions, nlist=lc.nlist)
```

compute

Update the data structure for the given set of points and compute a NeighborList.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** (($N_{particles}$, 3) numpy.ndarray) – Reference point coordinates.
- **points** (($N_{particles}$, 3) numpy.ndarray, optional) – Point coordinates (Default value = None).
- **exclude_ii** (*bool, optional*) – Set this to True if pairs of points with identical indices should be excluded. If this is None, it will be treated as True if points is None or the same object as ref_points (Defaults to None).

getCell

Returns the index of the cell containing the given point.

Parameters **point** ((3) numpy.ndarray) – Point coordinates (x, y, z).

Returns Cell index.

Return type unsigned int

getCellNeighbors

Returns the neighboring cell indices of the given cell.

Parameters **cell** (*unsigned int*) – Cell index.

Returns Array of cell neighbors.

Return type ($N_{neighbors}$) numpy.ndarray

itercell

Return an iterator over all particles in the given cell.

Parameters **cell** (*unsigned int*) – Cell index.

Returns Iterator to particle indices in specified cell.

Return type iter

Nearest Neighbors

class freud.locality.**NearestNeighbors**(*rmax*, *n_neigh*, *scale=1.1*, *strict_cut=False*)

Supports efficiently finding the N nearest neighbors of each point in a set for some fixed integer N.

- `strict_cut == True`: `rmax` will be strictly obeyed, and any particle which has fewer than N neighbors will have values of `UINT_MAX` assigned.
- `strict_cut == False` (default): `rmax` will be expanded to find the requested number of neighbors. If `rmax` increases to the point that a cell list cannot be constructed, a warning will be raised and the neighbors already found will be returned.

Module author: Eric Harper <harperic@umich.edu>

Parameters

- **rmax** (*float*) – Initial guess of a distance to search within to find N neighbors.
- **n_neigh** (*unsigned int*) – Number of neighbors to find for each point.
- **scale** (*float*) – Multiplier by which to automatically increase `rmax` value if the requested number of neighbors is not found. Only utilized if `strict_cut` is False. Scale must be greater than 1.
- **strict_cut** (*bool*) – Whether to use a strict `rmax` or allow for automatic expansion, default is False.

Variables

- **UINTMAX** (*unsigned int*) – Value of C++ UINTMAX used to pad the arrays.
- ***box*** (*freud.box.Box*) – Simulation box.
- **num_neighbors** (*unsigned int*) – The number of neighbors this object will find.
- **n_ref** (*unsigned int*) – The number of particles this object found neighbors of.
- **r_max** (*float*) – Current nearest neighbors search radius guess.
- **wrapped_vectors** ((*$N_{particles}$*) numpy.ndarray) – The wrapped vectors padded with -1 for empty neighbors.
- **r_sq_list** ((*$N_{particles}$*, *$N_{neighbors}$*) numpy.ndarray) – The Rsq values list.
- **nlist** (*freud.locality.NeighborList*) – The neighbor list stored by this object, generated by `compute()`.

Example:

```
nn = NearestNeighbors(2, 6)
nn.compute(box, positions, positions)
hexatic = order.HexOrderParameter(2)
hexatic.compute(box, positions, nlist=nn.nlist)
```

compute

Update the data structure for the given set of points.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** ((*$N_{particles}$*, 3) numpy.ndarray) – Reference point coordinates.
- **points** ((*$N_{particles}$*, 3) numpy.ndarray, optional) – Point coordinates. Defaults to `ref_points` if not provided or `None`.

- **exclude_ii** (*bool, optional*) – Set this to `True` if pairs of points with identical indices should be excluded. If this is `None`, it will be treated as `True` if `points` is `None` or the same object as `ref_points` (Defaults to `None`).

getNeighborList
Return the entire neighbor list.

Returns Indices of up to $N_{neighbors}$ points that are neighbors of the $N_{particles}$ reference points, padded with UINTMAX if fewer neighbors than requested were found.

Return type $(N_{particles}, N_{neighbors})$ `numpy.ndarray`

getNeighbors
Return the N nearest neighbors of the reference point with index i.

Parameters **i** (*unsigned int*) – Index of the reference point whose neighbors will be returned.

Returns Indices of points that are neighbors of reference point i, padded with UINTMAX if fewer neighbors than requested were found.

Return type $(N_{neighbors})$ `numpy.ndarray`

getRsq
Return the squared distances to the N nearest neighbors of the reference point with index i.

Parameters **i** (*unsigned int*) – Index of the reference point of which to fetch the neighboring point distances.

Returns Squared distances to the N nearest neighbors.

Return type $(N_{particles})$ `numpy.ndarray`

1.3.8 MSD Module

Overview

freud.msd.MSD	Compute the mean squared displacement.

Details

The *freud.msd* module provides functions for computing the mean-squared-displacement (MSD) of particles in periodic systems.

MSD

class freud.msd.**MSD**(*box=None*, *mode=None*)
Compute the mean squared displacement.

The mean squared displacement (MSD) measures how much particles move over time. The MSD plays an important role in characterizing Brownian motion, since it provides a measure of whether particles are moving according to diffusion alone or if there are other forces contributing. There are a number of definitions for the mean squared displacement. This function provides access to the two most common definitions through the mode argument.

- `'window'` (*default*): This mode calculates the most common form of the MSD, which is defined as

$$MSD(m) = \left\langle \frac{1}{N-m} \sum_{k=0}^{N-m-1} (\vec{r}(k+m) - \vec{r}(k))^2 \right\rangle_{particles}$$

  According to this definition, the mean squared displacement is the average displacement over all windows of length m over the course of the simulation. Therefore, for any m, $MSD(m)$ is averaged over all windows of length m and over all particles. This calculation can be accessed using the 'window' mode of this function.

  The windowed calculation can be quite computationally intensive. To perform this calculation efficiently, we use the algorithm described in [Calandrini2011] as described in this StackOverflow thread.

- `'direct'`: Under some circumstances, however, we may be more interested in calculating a different quantity described by

$$\begin{aligned} MSD(t) = & \quad \langle (\vec{r} - \vec{r}_0)^2 \rangle_{particles} \\ = & \quad \frac{1}{N} \sum_{n=1}^{N} (x_n(t) - x_n(0))^2 \end{aligned}$$

  In this case, we simply compute how much particles have moved from their initial position, averaged over all particles. For more information on this calculation, see the Wikipedia page.

Note: The MSD is only well-defined when the box is constant over the course of the simulation. Additionally, the number of particles must be constant over the course of the simulation.

Module author: Vyas Ramasubramani <vramasub@umich.edu>

New in version 1.0.

Parameters

- **box** (*freud.box.Box*, optional) – If not provided, the class will assume that all positions provided in calls to *compute()* or *accumulate()* are already unwrapped.
- **mode** (*str, optional*) – Mode of calculation. Options are `'window'` and `'direct'`. (Default value = `'window'`).

Variables

- ***box*** (*freud.box.Box*) – Box used in the calculation.
- ***msd*** ($(N_{frames},)$ numpy.ndarray) – The mean squared displacement.

accumulate

Calculate the MSD for the positions provided and add to the existing per-particle data.

Note: Unlike most methods in freud, accumulation for the MSD is split over particles rather than frames of a simulation. The reason for this choice is that efficient computation of the MSD requires using the entire trajectory for a given particle. As a result, this accumulation is primarily useful when the trajectory is so large that computing an MSD on all particles at once is prohibitive.

Parameters

- **positions** ($(N_{frames}, N_{particles}, 3)$ numpy.ndarray) – The particle positions over a trajectory. If neither box nor images are provided, the positions are assumed to be unwrapped already.

- **images** ((*N*$_{frames}$, *N*$_{particles}$, 3) numpy.ndarray, optional) – The particle images to unwrap with if provided. Must be provided along with a simulation box (in the constructor) if particle positions need to be unwrapped. If neither are provided, positions are assumed to be unwrapped already.

compute
Calculate the MSD for the positions provided.

Parameters

- **positions** ((*N*$_{frames}$, *N*$_{particles}$, 3) numpy.ndarray) – The particle positions over a trajectory. If neither box nor images are provided, the positions are assumed to be unwrapped already.
- **images** ((*N*$_{frames}$, *N*$_{particles}$, 3) numpy.ndarray, optional) – The particle images to unwrap with if provided. Must be provided along with a simulation box (in the constructor) if particle positions need to be unwrapped. If neither are provided, positions are assumed to be unwrapped already.

reset
Clears the stored MSD values from previous calls to accumulate (or the last call to compute).

1.3.9 Order Module

Overview

freud.order.CubaticOrderParameter	Compute the cubatic order parameter [HajiAkbari2015] for a system of particles using simulated annealing instead of Newton-Raphson root finding.
freud.order.NematicOrderParameter	Compute the nematic order parameter for a system of particles.
freud.order.HexOrderParameter	Calculates the k-atic order parameter for each particle in the system.
freud.order.TransOrderParameter	Compute the translational order parameter for each particle.
freud.order.LocalQl	Compute the local Steinhardt [Steinhardt1983] rotationally invariant Q_l order parameter for a set of points.
freud.order.LocalQlNear	A variant of the LocalQl class that performs its average over nearest neighbor particles as determined by an instance of *freud.locality.NeighborList*.
freud.order.LocalWl	Compute the local Steinhardt [Steinhardt1983] rotationally invariant W_l order parameter for a set of points.
freud.order.LocalWlNear	A variant of the LocalWl class that performs its average over nearest neighbor particles as determined by an instance of *freud.locality.NeighborList*.
freud.order.SolLiq	Uses dot products of Q_{lm} between particles for clustering.
freud.order.SolLiqNear	A variant of the SolLiq class that performs its average over nearest neighbor particles as determined by an instance of *freud.locality.NeighborList*.

Continued on next page

Table 9 – continued from previous page

freud.order.RotationalAutocorrelation	Calculates a measure of total rotational autocorrelation based on hyperspherical harmonics as laid out in "Design rules for engineering colloidal plastic crystals of hard polyhedra - phase behavior and directional entropic forces" by Karas et al.

Details

The *freud.order* module contains functions which compute order parameters for the whole system or individual particles. Order parameters take bond order data and interpret it in some way to quantify the degree of order in a system using a scalar value. This is often done through computing spherical harmonics of the bond order diagram, which are the spherical analogue of Fourier Transforms.

Cubatic Order Parameter

class freud.order.**CubaticOrderParameter**(*t_initial*, *t_final*, *scale*, *n_replicates*, *seed*)

Compute the cubatic order parameter [HajiAkbari2015] for a system of particles using simulated annealing instead of Newton-Raphson root finding.

Module author: Eric Harper <harperic@umich.edu>

Parameters

- **t_initial** (*float*) – Starting temperature.
- **t_final** (*float*) – Final temperature.
- **scale** (*float*) – Scaling factor to reduce temperature.
- **n_replicates** (*unsigned int*) – Number of replicate simulated annealing runs.
- **seed** (*unsigned int*) – Random seed to use in calculations. If None, system time is used.

Variables

- **t_initial** (*float*) – The value of the initial temperature.
- **t_final** (*float*) – The value of the final temperature.
- **scale** (*float*) – The scale
- **cubatic_order_parameter** (*float*) – The cubatic order parameter.
- **orientation** ((4) numpy.ndarray) – The quaternion of global orientation.
- **particle_order_parameter** (numpy.ndarray) – Cubatic order parameter.
- **particle_tensor** ($(N_{particles}, 3, 3, 3, 3)$ numpy.ndarray) – Rank 5 tensor corresponding to each individual particle orientation.
- **global_tensor** $((3, 3, 3, 3)$ numpy.ndarray) – Rank 4 tensor corresponding to global orientation.
- **cubatic_tensor** $((3, 3, 3, 3)$ numpy.ndarray) – Rank 4 cubatic tensor.
- **gen_r4_tensor** $((3, 3, 3, 3)$ numpy.ndarray) – Rank 4 tensor corresponding to each individual particle orientation.

compute

Calculates the per-particle and global order parameter.

Parameters **orientations** (($N_{particles}$, 4) numpy.ndarray) – Orientations as angles to use in computation.

Nematic Order Parameter

class freud.order.**NematicOrderParameter**(*u*)

Compute the nematic order parameter for a system of particles.

Module author: Jens Glaser <jsglaser@umich.edu>

New in version 0.7.0.

Parameters **u** ((3) numpy.ndarray) – The nematic director of a single particle in the reference state (without any rotation applied).

Variables

- **nematic_order_parameter** (*float*) – Nematic order parameter.
- **director** ((3) numpy.ndarray) – The average nematic director.
- **particle_tensor** (($N_{particles}$, 3, 3) numpy.ndarray) – One 3x3 matrix per-particle corresponding to each individual particle orientation.
- **nematic_tensor** ((3, 3) numpy.ndarray) – 3x3 matrix corresponding to the average particle orientation.

compute

Calculates the per-particle and global order parameter.

Parameters **orientations** (($N_{particles}$, 4) numpy.ndarray) – Orientations to calculate the order parameter.

Hexatic Order Parameter

class freud.order.**HexOrderParameter**(*rmax*, *k*, *n*)

Calculates the k-atic order parameter for each particle in the system.

The k-atic order parameter for a particle i and its n neighbors j is given by:

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

The parameter k governs the symmetry of the order parameter while the parameter n governs the number of neighbors of particle i to average over. ϕ_{ij} is the angle between the vector r_{ij} and $(1, 0)$.

Note: **2D:** *freud.order.HexOrderParameter* properly handles 2D boxes. The points must be passed in as `[x, y, 0]`. Failing to set z=0 will lead to undefined behavior.

Module author: Eric Harper <harperic@umich.edu>

Parameters

- **rmax** (*float*) – +/- r distance to search for neighbors.
- **k** (*unsigned int*) – Symmetry of order parameter ($k = 6$ is hexatic).
- **n** (*unsigned int*) – Number of neighbors ($n = k$ if n not specified).

Variables

- **psi** (($N_{particles}$) numpy.ndarray) – Order parameter.

- **box** (*freud.box.Box*) – Box used in the calculation.
- **num_particles** (*unsigned int*) – Number of particles.
- **K** (*unsigned int*) – Symmetry of the order parameter.

compute
Calculates the correlation function and adds to the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*) – Neighborlist to use to find bonds.

Translational Order Parameter

class freud.order.**TransOrderParameter**(*rmax*, *k*, *n*)
Compute the translational order parameter for each particle.

Module author: Wenbo Shen <shenwb@umich.edu>

Parameters

- **rmax** (*float*) – +/- r distance to search for neighbors.
- **k** (*float*) – Symmetry of order parameter ($k = 6$ is hexatic).
- **n** (*unsigned int*) – Number of neighbors ($n = k$ if n not specified).

Variables

- **d_r** (($N_{particles}$) numpy.ndarray) – Reference to the last computed translational order array.
- **box** (*freud.box.Box*) – Box used in the calculation.
- **num_particles** (*unsigned int*) – Number of particles.

compute
Calculates the local descriptors.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*) – Neighborlist to use to find bonds.

Steinhardt Q_l Order Parameter

class freud.order.**LocalQl**(*box*, *rmax*, *l*, *rmin*)
Compute the local Steinhardt [Steinhardt1983] rotationally invariant Q_l order parameter for a set of points.

Implements the local rotationally invariant Q_l order parameter described by Steinhardt. For a particle i, we calculate the average Q_l by summing the spherical harmonics between particle i and its neighbors j in a local region: $\overline{Q}_{lm}(i) = \frac{1}{N_b}\sum_{j=1}^{N_b} Y_{lm}(\theta(\vec{r}_{ij}), \phi(\vec{r}_{ij}))$. The particles included in the sum are determined by the rmax argument to the constructor.

This is then combined in a rotationally invariant fashion to remove local orientational order as follows: $Q_l(i) = \sqrt{\frac{4\pi}{2l+1}\sum_{m=-l}^{l}|\overline{Q}_{lm}|^2}$.

The `computeAve()` method provides access to a variant of this parameter that performs a average over the first and second shell combined [Lechner2008]. To compute this parameter, we perform a second averaging over the first neighbor shell of the particle to implicitly include information about the second neighbor shell. This averaging is performed by replacing the value $\overline{Q}_{lm}(i)$ in the original definition by the average value of $\overline{Q}_{lm}(k)$ over all the k neighbors of particle i as well as itself.

The `computeNorm()` and `computeAveNorm()` methods provide normalized versions of `compute()` and `computeAve()`, where the normalization is performed by dividing by the average Q_{lm} values over all particles.

Module author: Xiyu Du <xiyudu@umich.edu>

Module author: Vyas Ramasubramani <vramasub@umich.edu>

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **rmax** (*float*) – Cutoff radius for the local order parameter. Values near the first minimum of the RDF are recommended.
- **l** (*unsigned int*) – Spherical harmonic quantum number l. Must be a positive number.
- **rmin** (*float*) – Can look at only the second shell or some arbitrary RDF region.

Variables

- ***box*** (*freud.box.Box*) – Box used in the calculation.
- **num_particles** (*unsigned int*) – Number of particles.
- **Ql** ($(N_{particles})$ numpy.ndarray) – The last computed Q_l for each particle (filled with NaN for particles with no neighbors).
- **ave_Ql** ($(N_{particles})$ numpy.ndarray) – The last computed $\bar{Q}_l$ for each particle (filled with NaN for particles with no neighbors).
- **norm_Ql** ($(N_{particles})$ numpy.ndarray) – The last computed Q_l for each particle normalized by the value over all particles (filled with NaN for particles with no neighbors).
- **ave_norm_Ql** ($(N_{particles})$ numpy.ndarray) – The last computed $\bar{Q}_l$ for each particle normalized by the value over all particles (filled with NaN for particles with no neighbors).

compute

Compute the order parameter.

Parameters

- **points** ($(N_{particles}, 3)$ numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

computeAve

Compute the order parameter over two nearest neighbor shells.

Parameters

- **points** ($(N_{particles}, 3)$ numpy.ndarray) – Points to calculate the order parameter.

- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

computeAveNorm
Compute the order parameter over two nearest neighbor shells normalized by the average spherical harmonic value over all the particles.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

computeNorm
Compute the order parameter normalized by the average spherical harmonic value over all the particles.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

setBox
Reset the simulation box.

Parameters **box** (*freud.box.Box*) – Simulation box.

class freud.order.**LocalQlNear**(*box*, *rmax*, *l*, *kn*)
A variant of the *LocalQl* class that performs its average over nearest neighbor particles as determined by an instance of *freud.locality.NeighborList*. The number of included neighbors is determined by the kn parameter to the constructor.

Module author: Xiyu Du <xiyudu@umich.edu>

Module author: Vyas Ramasubramani <vramasub@umich.edu>

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **rmax** (*float*) – Cutoff radius for the local order parameter. Values near the first minimum of the RDF are recommended.
- **l** (*unsigned int*) – Spherical harmonic quantum number l. Must be a positive number.
- **kn** (*unsigned int*) – Number of nearest neighbors. must be a positive integer.

Variables

- ***box*** (*freud.box.Box*) – Box used in the calculation.
- **num_particles** (*unsigned int*) – Number of particles.
- **Ql** (($N_{particles}$) numpy.ndarray) – The last computed Q_l for each particle (filled with NaN for particles with no neighbors).
- **ave_Ql** (($N_{particles}$) numpy.ndarray) – The last computed $\bar{Q}_l$ for each particle (filled with NaN for particles with no neighbors).
- **norm_Ql** (($N_{particles}$) numpy.ndarray) – The last computed Q_l for each particle normalized by the value over all particles (filled with NaN for particles with no neighbors).
- **ave_norm_Ql** (($N_{particles}$) numpy.ndarray) – The last computed $\bar{Q}_l$ for each particle normalized by the value over all particles (filled with NaN for particles with no neighbors).

compute
Compute the order parameter.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

computeAve
Compute the order parameter over two nearest neighbor shells.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

computeAveNorm
Compute the order parameter over two nearest neighbor shells normalized by the average spherical harmonic value over all the particles.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

computeNorm
Compute the order parameter normalized by the average spherical harmonic value over all the particles.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

setBox
Reset the simulation box.

Parameters **box** (*freud.box.Box*) – Simulation box.

Steinhardt W_l Order Parameter

class freud.order.**LocalWl**(*box*, *rmax*, *l*)
Compute the local Steinhardt [Steinhardt1983] rotationally invariant W_l order parameter for a set of points.

Implements the local rotationally invariant W_l order parameter described by Steinhardt. For a particle i, we calculate the average W_l by summing the spherical harmonics between particle i and its neighbors j in a local region: $\overline{Q}_{lm}(i) = \frac{1}{N_b}\sum_{j=1}^{N_b} Y_{lm}(\theta(\vec{r}_{ij}), \phi(\vec{r}_{ij}))$. The particles included in the sum are determined by the rmax argument to the constructor.

The W_l is then defined as a weighted average over the $\overline{Q}_{lm}(i)$ values using Wigner 3j symbols (Clebsch-Gordan coefficients). The resulting combination is rotationally (i.e. frame) invariant.

The *computeAve()* method provides access to a variant of this parameter that performs a average over the first and second shell combined [Lechner2008]. To compute this parameter, we perform a second averaging over the first neighbor shell of the particle to implicitly include information about the second neighbor shell. This

averaging is performed by replacing the value $\overline{Q}_{lm}(i)$ in the original definition by the average value of $\overline{Q}_{lm}(k)$ over all the k neighbors of particle i as well as itself.

The *computeNorm()* and *computeAveNorm()* methods provide normalized versions of *compute()* and *computeAve()*, where the normalization is performed by dividing by the average Q_{lm} values over all particles.

Module author: Xiyu Du <xiyudu@umich.edu>

Module author: Vyas Ramasubramani <vramasub@umich.edu>

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **rmax** (*float*) – Cutoff radius for the local order parameter. Values near the first minimum of the RDF are recommended.
- **l** (*unsigned int*) – Spherical harmonic quantum number l. Must be a positive number
- **rmin** (*float*) – Lower bound for computing the local order parameter. Allows looking at, for instance, only the second shell, or some other arbitrary RDF region.

Variables

- ***box*** (*freud.box.Box*) – Box used in the calculation.
- **num_particles** (*unsigned int*) – Number of particles.
- **Wl** ($(N_{particles})$ numpy.ndarray) – The last computed W_l for each particle (filled with NaN for particles with no neighbors).
- **ave_Wl** ($(N_{particles})$ numpy.ndarray) – The last computed $\bar{W}_l$ for each particle (filled with NaN for particles with no neighbors).
- **norm_Wl** ($(N_{particles})$ numpy.ndarray) – The last computed W_l for each particle normalized by the value over all particles (filled with NaN for particles with no neighbors).
- **ave_norm_Wl** ($(N_{particles})$ numpy.ndarray) – The last computed $\bar{W}_l$ for each particle normalized by the value over all particles (filled with NaN for particles with no neighbors).

compute

Compute the order parameter.

Parameters

- **points** ($(N_{particles}, 3)$ numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

computeAve

Compute the order parameter over two nearest neighbor shells.

Parameters

- **points** ($(N_{particles}, 3)$ numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

computeAveNorm

Compute the order parameter over two nearest neighbor shells normalized by the average spherical harmonic value over all the particles.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

computeNorm
Compute the order parameter normalized by the average spherical harmonic value over all the particles.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

setBox
Reset the simulation box.

Parameters **box** (*freud.box.Box*) – Simulation box.

class freud.order.**LocalWlNear**(*box*, *rmax*, *l*, *kn*)
A variant of the *LocalWl* class that performs its average over nearest neighbor particles as determined by an instance of *freud.locality.NeighborList*. The number of included neighbors is determined by the kn parameter to the constructor.

Module author: Xiyu Du <xiyudu@umich.edu>

Module author: Vyas Ramasubramani <vramasub@umich.edu>

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **rmax** (*float*) – Cutoff radius for the local order parameter. Values near the first minimum of the RDF are recommended.
- **l** (*unsigned int*) – Spherical harmonic quantum number l. Must be a positive number
- **kn** (*unsigned int*) – Number of nearest neighbors. Must be a positive number.

Variables

- ***box*** (*freud.box.Box*) – Box used in the calculation.
- **num_particles** (*unsigned int*) – Number of particles.
- **Wl** (($N_{particles}$) numpy.ndarray) – The last computed W_l for each particle (filled with NaN for particles with no neighbors).
- **ave_Wl** (($N_{particles}$) numpy.ndarray) – The last computed $\bar{W}_l$ for each particle (filled with NaN for particles with no neighbors).
- **norm_Wl** (($N_{particles}$) numpy.ndarray) – The last computed W_l for each particle normalized by the value over all particles (filled with NaN for particles with no neighbors).
- **ave_norm_Wl** (($N_{particles}$) numpy.ndarray) – The last computed $\bar{W}_l$ for each particle normalized by the value over all particles (filled with NaN for particles with no neighbors).

compute
Compute the order parameter.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

computeAve
Compute the order parameter over two nearest neighbor shells.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

computeAveNorm
Compute the order parameter over two nearest neighbor shells normalized by the average spherical harmonic value over all the particles.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

computeNorm
Compute the order parameter normalized by the average spherical harmonic value over all the particles.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

setBox
Reset the simulation box.

Parameters **box** (*freud.box.Box*) – Simulation box.

Solid-Liquid Order Parameter

class freud.order.**SolLiq**(*box*, *rmax*, *Qthreshold*, *Sthreshold*, *l*)
Uses dot products of Q_{lm} between particles for clustering.

Module author: Richmond Newman <newmanrs@umich.edu>

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **rmax** (*float*) – Cutoff radius for the local order parameter. Values near first minimum of the RDF are recommended.
- **Qthreshold** (*float*) – Value of dot product threshold when evaluating $Q^*_{lm}(i)Q_{lm}(j)$ to determine if a neighbor pair is a solid-like bond. (For $l = 6$, 0.7 generally good for FCC or BCC structures).
- **Sthreshold** (*unsigned int*) – Minimum required number of adjacent solid-link bonds for a particle to be considered solid-like for clustering. (For $l = 6$, 6-8 is generally good for FCC or BCC structures).
- **l** (*unsigned int*) – Choose spherical harmonic Q_l. Must be positive and even.

Variables

- ***box*** (*freud.box.Box*) – Box used in the calculation.

- **largest_cluster_size** (*unsigned int*) – The largest cluster size. Must call a compute method first.
- **cluster_sizes** (*unsigned int*) – The sizes of all clusters.
- **largest_cluster_size** – The largest cluster size. Must call a compute method first.
- **Ql_mi** (($N_{particles}$) numpy.ndarray) – The last computed Q_{lmi} for each particle.
- **clusters** (($N_{particles}$) numpy.ndarray) – The last computed set of solid-like cluster indices for each particle.
- **num_connections** (($N_{particles}$) numpy.ndarray) – The number of connections per particle.
- **Ql_dot_ij** (($N_{particles}$) numpy.ndarray) – Reference to the qldot_ij values.
- **num_particles** (*unsigned int*) – Number of particles.

compute
Compute the solid-liquid order parameter.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

computeSolLiqNoNorm
Compute the solid-liquid order parameter without normalizing the dot product.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

computeSolLiqVariant
Compute a variant of the solid-liquid order parameter.

This variant method places a minimum threshold on the number of solid-like bonds a particle must have to be considered solid-like for clustering purposes.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*, optional) – Neighborlist to use to find bonds (Default value = None).

class freud.order.**SolLiqNear**(*box*, *rmax*, *Qthreshold*, *Sthreshold*, *l*, *kn*)
A variant of the *SolLiq* class that performs its average over nearest neighbor particles as determined by an instance of *freud.locality.NeighborList*. The number of included neighbors is determined by the kn parameter to the constructor.

Module author: Richmond Newman <newmanrs@umich.edu>

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **rmax** (*float*) – Cutoff radius for the local order parameter. Values near the first minimum of the RDF are recommended.

- **Qthreshold** (*float*) – Value of dot product threshold when evaluating $Q^*_{lm}(i)Q_{lm}(j)$ to determine if a neighbor pair is a solid-like bond. (For $l = 6$, 0.7 generally good for FCC or BCC structures).
- **Sthreshold** (*unsigned int*) – Minimum required number of adjacent solid-link bonds for a particle to be considered solid-like for clustering. (For $l = 6$, 6-8 is generally good for FCC or BCC structures).
- **l** (*unsigned int*) – Choose spherical harmonic Q_l. Must be positive and even.
- **kn** (*unsigned int*) – Number of nearest neighbors. Must be a positive number.

Variables

- ***box*** (*freud.box.Box*) – Box used in the calculation.
- **largest_cluster_size** (*unsigned int*) – The largest cluster size. Must call a compute method first.
- **cluster_sizes** (*unsigned int*) – The sizes of all clusters.
- **largest_cluster_size** – The largest cluster size. Must call a compute method first.
- **Ql_mi** (($N_{particles}$) numpy.ndarray) – The last computed Q_{lmi} for each particle.
- **clusters** (($N_{particles}$) numpy.ndarray) – The last computed set of solid-like cluster indices for each particle.
- **num_connections** (($N_{particles}$) numpy.ndarray) – The number of connections per particle.
- **Ql_dot_ij** (($N_{particles}$) numpy.ndarray) – Reference to the qldot_ij values.
- **num_particles** (*unsigned int*) – Number of particles.

compute

Compute the local rotationally invariant Q_l order parameter.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*) – Neighborlist to use to find bonds.

computeSolLiqNoNorm

Compute the local rotationally invariant Q_l order parameter.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*) – Neighborlist to use to find bonds.

computeSolLiqVariant

Compute the local rotationally invariant Q_l order parameter.

Parameters

- **points** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate the order parameter.
- **nlist** (*freud.locality.NeighborList*) – Neighborlist to use to find bonds.

Rotational Autocorrelation

class freud.order.**RotationalAutocorrelation**(*l*)

Calculates a measure of total rotational autocorrelation based on hyperspherical harmonics as laid out in "Design rules for engineering colloidal plastic crystals of hard polyhedra - phase behavior and directional entropic forces" by Karas et al. (currently in preparation). The output is not a correlation function, but rather a scalar value that measures total system orientational correlation with an initial state. As such, the output can be treated as an order parameter measuring degrees of rotational (de)correlation. For analysis of a trajectory, the compute call needs to be done at each trajectory frame.

Module author: Andrew Karas <askaras@umich.edu>

Module author: Vyas Ramasubramani <vramasub@umich.edu>

New in version 1.0.

Parameters **l** (*int*) – Order of the hyperspherical harmonic. Must be a positive, even integer.

Variables

- **num_orientations** (*unsigned int*) – The number of orientations used in computing the last set.
- **azimuthal** (*int*) – The azimuthal quantum number, which defines the order of the hyperspherical harmonic. Must be a positive, even integer.
- **ra_array** (($N_{orientations}$) numpy.ndarray) – The per-orientation array of rotational autocorrelation values calculated by the last call to compute.
- **autocorrelation** (*float*) – The autocorrelation computed in the last call to compute.

compute

Calculates the rotational autocorrelation function for a single frame.

Parameters

- **ref_ors** (($N_{orientations}$, 4) numpy.ndarray) – Reference orientations for the initial frame.
- **ors** (($N_{orientations}$, 4) numpy.ndarray) – Orientations for the frame of interest.

1.3.10 Parallel Module

Overview

freud.parallel.NumThreads	Context manager for managing the number of threads to use.
freud.parallel.setNumThreads	Set the number of threads for parallel computation.

Details

The *freud.parallel* module controls the parallelization behavior of freud, determining how many threads the TBB-enabled parts of freud will use. By default, freud tries to use all available threads for parallelization unless directed otherwise, with one exception.

parallel.**setNumThreads**(*nthreads=None*)

Set the number of threads for parallel computation.

Module author: Joshua Anderson <joaander@umich.edu>

Parameters **nthreads** (*int, optional*) – Number of threads to use. If None (default), use all threads available.

class freud.parallel.**NumThreads**
Context manager for managing the number of threads to use.

Module author: Joshua Anderson <joaander@umich.edu>

Parameters **N** (*int*) – Number of threads to use in this context. Defaults to None, which will use all available threads.

freud.parallel.**setNumThreads**
Set the number of threads for parallel computation.

Module author: Joshua Anderson <joaander@umich.edu>

Parameters **nthreads** (*int, optional*) – Number of threads to use. If None (default), use all threads available.

1.3.11 PMFT Module

Overview

freud.pmft.PMFTR12	Computes the PMFT [vanAndersKlotsa2014] [vanAndersAhmed2014] in a 2D system described by r, θ_1, θ_2.
freud.pmft.PMFTXYT	Computes the PMFT [vanAndersKlotsa2014] [vanAndersAhmed2014] for systems described by coordinates x, y, θ listed in the `X`, `Y`, and `T` arrays.
freud.pmft.PMFTXY2D	Computes the PMFT [vanAndersKlotsa2014] [vanAndersAhmed2014] in coordinates x, y listed in the `X` and `Y` arrays.
freud.pmft.PMFTXYZ	Computes the PMFT [vanAndersKlotsa2014] [vanAndersAhmed2014] in coordinates x, y, z, listed in the `X`, `Y`, and `Z` arrays.

Details

The *freud.pmft* module allows for the calculation of the Potential of Mean Force and Torque (PMFT) [vanAndersKlotsa2014] [vanAndersAhmed2014] in a number of different coordinate systems. The shape of the arrays computed by this module depend on the coordinate system used, with space discretized into a set of bins created by the PMFT object's constructor. Each reference point's neighboring points are assigned to bins, determined by the relative positions and/or orientations of the particles. Next, the positional correlation function (PCF) is computed by normalizing the binned histogram, by dividing out the number of accumulated frames, bin sizes (the Jacobian), and reference point number density. The PMFT is then defined as the negative logarithm of the PCF. For further descriptions of the numerical methods used to compute the PMFT, refer to the supplementary information of [vanAndersKlotsa2014].

Note: The coordinate system in which the calculation is performed is not the same as the coordinate system in which particle positions and orientations should be supplied. Only certain coordinate systems are available for certain particle positions and orientations:

- 2D particle coordinates (position: $[x, y, 0]$, orientation: θ):
  - r, θ_1, θ_2.
  - x, y.

  - x, y, θ.
- 3D particle coordinates:
  - x, y, z.

Note: For any bins where the histogram is zero (i.e. no observations were made with that relative position/orientation of particles), the PCF will be zero and the PMFT will return `nan`.

PMFT (r, θ_1, θ_2)

class freud.pmft.**PMFTR12**(*r_max*, *n_r*, *n_t1*, *n_t2*)

Computes the PMFT [vanAndersKlotsa2014] [vanAndersAhmed2014] in a 2D system described by r, θ_1, θ_2.

Note: **2D:** *freud.pmft.PMFTR12* is only defined for 2D systems. The points must be passed in as `[x, y, 0]`. Failing to set z=0 will lead to undefined behavior.

Module author: Eric Harper <harperic@umich.edu>

Module author: Vyas Ramasubramani <vramasub@umich.edu>

Parameters

- **r_max** (*float*) – Maximum distance at which to compute the PMFT.
- **n_r** (*unsigned int*) – Number of bins in r.
- **n_t1** (*unsigned int*) – Number of bins in θ_1.
- **n_t2** (*unsigned int*) – Number of bins in θ_2.

Variables

- ***box*** (*freud.box.Box*) – Box used in the calculation.
- **bin_counts** ($(N_r, N_{\theta 2}, N_{\theta 1})$) – Bin counts.
- **PCF** ($(N_r, N_{\theta 2}, N_{\theta 1})$) – The positional correlation function.
- **PMFT** ($(N_r, N_{\theta 2}, N_{\theta 1})$) – The potential of mean force and torque.
- **r_cut** (*float*) – The cutoff used in the cell list.
- **R** ((N_r) numpy.ndarray) – The array of r-values for the PCF histogram.
- **T1** ($(N_{\theta 1})$ numpy.ndarray) – The array of θ_1-values for the PCF histogram.
- **T2** ($(N_{\theta 2})$ numpy.ndarray) – The array of θ_2-values for the PCF histogram.
- **inverse_jacobian** ($(N_r, N_{\theta 2}, N_{\theta 1})$) – The inverse Jacobian used in the PMFT.
- **n_bins_R** (*unsigned int*) – The number of bins in the r-dimension of the histogram.
- **n_bins_T1** (*unsigned int*) – The number of bins in the θ_1-dimension of the histogram.
- **n_bins_T2** (*unsigned int*) – The number of bins in the θ_2-dimension of the histogram.

accumulate

Calculates the positional correlation function and adds to the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** (($N_{particles}$, 3) numpy.ndarray) – Reference points used in computation.
- **ref_orientations** (($N_{particles}$, 1) or ($N_{particles}$,) numpy.ndarray) – Reference orientations as angles used in computation.
- **points** (($N_{particles}$, 3) numpy.ndarray, optional) – Points used in computation. Uses ref_points if not provided or None.
- **orientations** (($N_{particles}$, 1) or ($N_{particles}$,) numpy.ndarray, optional) – Orientations as angles used in computation. Uses ref_orientations if not provided or None.
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList used to find bonds (Default value = None).

compute

Calculates the positional correlation function for the given points. Will overwrite the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** (($N_{particles}$, 3) numpy.ndarray) – Reference points used in computation.
- **ref_orientations** (($N_{particles}$, 4) numpy.ndarray) – Reference orientations as angles used in computation.
- **points** (($N_{particles}$, 3) numpy.ndarray, optional) – Points used in computation. Uses ref_points if not provided or None.
- **orientations** (($N_{particles}$, 4) numpy.ndarray, optional) – Orientations as angles used in computation. Uses ref_orientations if not provided or None.
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList used to find bonds (Default value = None).

reset

Resets the values of the PCF histograms in memory.

PMFT (x, y)

class freud.pmft.**PMFTXY2D**(*x_max*, *y_max*, *n_x*, *n_y*)

Computes the PMFT [vanAndersKlotsa2014] [vanAndersAhmed2014] in coordinates x, y listed in the X and Y arrays.

The values of x and y at which to compute the PCF are controlled by x_max, y_max, n_x, and n_y parameters to the constructor. The x_max and y_max parameters determine the minimum/maximum distance at which to compute the PCF and n_x and n_y are the number of bins in x and y.

Note: 2D: *freud.pmft.PMFTXY2D* is only defined for 2D systems. The points must be passed in as [x, y, 0]. Failing to set z=0 will lead to undefined behavior.

Module author: Eric Harper <harperic@umich.edu>

Module author: Vyas Ramasubramani <vramasub@umich.edu>

Parameters

- **x_max** (*float*) – Maximum x distance at which to compute the PMFT.
- **y_max** (*float*) – Maximum y distance at which to compute the PMFT.
- **n_x** (*unsigned int*) – Number of bins in x.
- **n_y** (*unsigned int*) – Number of bins in y.

Variables

- ***box*** (*freud.box.Box*) – Box used in the calculation.
- **bin_counts** ((N_y, N_x) numpy.ndarray) – Bin counts.
- **PCF** ((N_y, N_x) numpy.ndarray) – The positional correlation function.
- **PMFT** ((N_y, N_x) numpy.ndarray) – The potential of mean force and torque.
- **r_cut** (*float*) – The cutoff used in the cell list.
- **X** ((N_x) numpy.ndarray) – The array of x-values for the PCF histogram.
- **Y** ((N_y) numpy.ndarray) – The array of y-values for the PCF histogram.
- **jacobian** (*float*) – The Jacobian used in the PMFT.
- **n_bins_X** (*unsigned int*) – The number of bins in the x-dimension of the histogram.
- **n_bins_Y** (*unsigned int*) – The number of bins in the y-dimension of the histogram.

accumulate

Calculates the positional correlation function and adds to the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** ($(N_{particles}, 3)$ numpy.ndarray) – Reference points used in computation.
- **ref_orientations** ($(N_{particles}, 1)$ or $(N_{particles},)$ numpy.ndarray) – Reference orientations as angles used in computation.
- **points** ($(N_{particles}, 3)$ numpy.ndarray, optional) – Points used in computation. Uses ref_points if not provided or None.
- **orientations** ($(N_{particles}, 1)$ or $(N_{particles},)$ numpy.ndarray, optional) – Orientations as angles used in computation. Uses ref_orientations if not provided or None.
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList used to find bonds (Default value = None).

compute

Calculates the positional correlation function for the given points. Will overwrite the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** ($(N_{particles}, 3)$ numpy.ndarray) – Reference points used in computation.
- **ref_orientations** ($(N_{particles}, 4)$ numpy.ndarray) – Reference orientations as angles used in computation.

- **points** (($N_{particles}$, 3) numpy.ndarray, optional) – Points used in computation. Uses ref_points if not provided or None.
- **orientations** (($N_{particles}$, 4) numpy.ndarray, optional) – Orientations as angles used in computation. Uses ref_orientations if not provided or None.
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList used to find bonds (Default value = None).

reset
Resets the values of the PCF histograms in memory.

PMFT (x, y, θ)

class freud.pmft.**PMFTXYT**(*x_max*, *y_max*, *n_x*, *n_y*, *n_t*)
Computes the PMFT [vanAndersKlotsa2014] [vanAndersAhmed2014] for systems described by coordinates x, y, θ listed in the X, Y, and T arrays.

The values of x, y, θ at which to compute the PCF are controlled by x_max, y_max, and n_x, n_y, n_t parameters to the constructor. The x_max and y_max parameters determine the minimum/maximum x, y values ($\min(\theta) = 0$, ($\max(\theta) = 2\pi$) at which to compute the PCF and n_x, n_y, n_t are the number of bins in x, y, θ.

Note: 2D: *freud.pmft.PMFTXYT* is only defined for 2D systems. The points must be passed in as [x, y, 0]. Failing to set z=0 will lead to undefined behavior.

Module author: Eric Harper <harperic@umich.edu>

Module author: Vyas Ramasubramani <vramasub@umich.edu>

Parameters

- **x_max** (*float*) – Maximum x distance at which to compute the PMFT.
- **y_max** (*float*) – Maximum y distance at which to compute the PMFT.
- **n_x** (*unsigned int*) – Number of bins in x.
- **n_y** (*unsigned int*) – Number of bins in y.
- **n_t** (*unsigned int*) – Number of bins in θ.

Variables

- ***box*** (*freud.box.Box*) – Box used in the calculation.
- **bin_counts** ((N_θ, N_y, N_x) numpy.ndarray) – Bin counts.
- **PCF** ((N_θ, N_y, N_x) numpy.ndarray) – The positional correlation function.
- **PMFT** ((N_θ, N_y, N_x) numpy.ndarray) – The potential of mean force and torque.
- **r_cut** (*float*) – The cutoff used in the cell list.
- **X** ((N_x) numpy.ndarray) – The array of x-values for the PCF histogram.
- **Y** ((N_y) numpy.ndarray) – The array of y-values for the PCF histogram.
- **T** ((N_θ) numpy.ndarray) – The array of θ-values for the PCF histogram.
- **jacobian** (*float*) – The Jacobian used in the PMFT.
- **n_bins_X** (*unsigned int*) – The number of bins in the x-dimension of the histogram.

- **n_bins_Y** (*unsigned int*) – The number of bins in the y-dimension of the histogram.
- **n_bins_T** (*unsigned int*) – The number of bins in the θ-dimension of the histogram.

accumulate
Calculates the positional correlation function and adds to the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** (($N_{particles}$, 3) numpy.ndarray) – Reference points used in computation.
- **ref_orientations** (($N_{particles}$, 1) or ($N_{particles}$,) numpy.ndarray) – Reference orientations as angles used in computation.
- **points** (($N_{particles}$, 3) numpy.ndarray, optional) – Points used in computation. Uses ref_points if not provided or None.
- **orientations** (($N_{particles}$, 1) or ($N_{particles}$,) numpy.ndarray, optional) – Orientations as angles used in computation. Uses ref_orientations if not provided or None.
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList used to find bonds (Default value = None).

compute
Calculates the positional correlation function for the given points. Will overwrite the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** (($N_{particles}$, 3) numpy.ndarray) – Reference points used in computation.
- **ref_orientations** (($N_{particles}$, 4) numpy.ndarray) – Reference orientations as angles used in computation.
- **points** (($N_{particles}$, 3) numpy.ndarray, optional) – Points used in computation. Uses ref_points if not provided or None.
- **orientations** (($N_{particles}$, 4) numpy.ndarray, optional) – Orientations as angles used in computation. Uses ref_orientations if not provided or None.
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList used to find bonds (Default value = None).

reset
Resets the values of the PCF histograms in memory.

PMFT (x, y, z)

class freud.pmft.**PMFTXYZ**(*x_max*, *y_max*, *z_max*, *n_x*, *n_y*, *n_z*)
Computes the PMFT [vanAndersKlotsa2014] [vanAndersAhmed2014] in coordinates x, y, z, listed in the X, Y, and Z arrays.

The values of x, y, z at which to compute the PCF are controlled by x_max, y_max, z_max, n_x, n_y, and n_z parameters to the constructor. The x_max, y_max, and z_max parameters] determine the minimum/maximum distance at which to compute the PCF and n_x, n_y, and n_z are the number of bins in x, y, z.

Note: 3D: *freud.pmft.PMFTXYZ* is only defined for 3D systems. The points must be passed in as `[x, y, z]`.

Module author: Eric Harper <harperic@umich.edu>

Module author: Vyas Ramasubramani <vramasub@umich.edu>

Parameters

- **x_max** (*float*) – Maximum x distance at which to compute the PMFT.
- **y_max** (*float*) – Maximum y distance at which to compute the PMFT.
- **z_max** (*float*) – Maximum z distance at which to compute the PMFT.
- **n_x** (*unsigned int*) – Number of bins in x.
- **n_y** (*unsigned int*) – Number of bins in y.
- **n_z** (*unsigned int*) – Number of bins in z.
- **shiftvec** (*list*) – Vector pointing from `[0, 0, 0]` to the center of the PMFT.

Variables

- ***box*** (*freud.box.Box*) – Box used in the calculation.
- **bin_counts** ((N_z, N_y, N_x) numpy.ndarray) – Bin counts.
- **PCF** ((N_z, N_y, N_x) numpy.ndarray) – The positional correlation function.
- **PMFT** ((N_z, N_y, N_x) numpy.ndarray) – The potential of mean force and torque.
- **r_cut** (*float*) – The cutoff used in the cell list.
- **X** ((N_x) numpy.ndarray) – The array of x-values for the PCF histogram.
- **Y** ((N_y) numpy.ndarray) – The array of y-values for the PCF histogram.
- **Z** ((N_z) numpy.ndarray) – The array of z-values for the PCF histogram.
- **jacobian** (*float*) – The Jacobian used in the PMFT.
- **n_bins_X** (*unsigned int*) – The number of bins in the x-dimension of the histogram.
- **n_bins_Y** (*unsigned int*) – The number of bins in the y-dimension of the histogram.
- **n_bins_Z** (*unsigned int*) – The number of bins in the z-dimension of the histogram.

accumulate

Calculates the positional correlation function and adds to the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** ($(N_{particles}, 3)$ numpy.ndarray) – Reference points used in computation.
- **ref_orientations** ($(N_{particles}, 4)$ numpy.ndarray) – Reference orientations as angles used in computation.
- **points** ($(N_{particles}, 3)$ numpy.ndarray, optional) – Points used in computation. Uses `ref_points` if not provided or `None`.
- **orientations** ($(N_{particles}, 4)$ numpy.ndarray, optional) – Orientations as angles used in computation. Uses `ref_orientations` if not provided or `None`.

- **face_orientations** (($N_{particles}$, 4) numpy.ndarray, optional) – Orientations of particle faces to account for particle symmetry. If not supplied by user, unit quaternions will be supplied. If a 2D array of shape (N_f, 4) or a 3D array of shape (1, N_f, 4) is supplied, the supplied quaternions will be broadcast for all particles. (Default value = None).
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList used to find bonds (Default value = None).

compute
Calculates the positional correlation function for the given points. Will overwrite the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** (($N_{particles}$, 3) numpy.ndarray) – Reference points used in computation.
- **ref_orientations** (($N_{particles}$, 4) numpy.ndarray) – Reference orientations as angles used in computation.
- **points** (($N_{particles}$, 3) numpy.ndarray, optional) – Points used in computation. Uses ref_points if not provided or None.
- **orientations** (($N_{particles}$, 4) numpy.ndarray, optional) – Orientations as angles used in computation. Uses ref_orientations if not provided or None.
- **face_orientations** (($N_{particles}$, 4) numpy.ndarray, optional) – Orientations of particle faces to account for particle symmetry. If not supplied by user, unit quaternions will be supplied. If a 2D array of shape (N_f, 4) or a 3D array of shape (1, N_f, 4) is supplied, the supplied quaternions will be broadcast for all particles. (Default value = None).
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList used to find bonds (Default value = None).

reset
Resets the values of the PCF histograms in memory.

1.3.12 Voronoi Module

Overview

freud.voronoi.Voronoi	Compute the Voronoi tessellation of a 2D or 3D system using qhull.

Details

The *freud.voronoi* module contains tools to characterize Voronoi cells of a system.

class freud.voronoi.**Voronoi**(*box*, *buff*)
Compute the Voronoi tessellation of a 2D or 3D system using qhull. This uses scipy.spatial.Voronoi, accounting for periodic boundary conditions.

Module author: Benjamin Schultz <baschult@umich.edu>

Module author: Yina Geng <yinageng@umich.edu>

Module author: Mayank Agrawal <amayank@umich.edu>

Module author: Bradley Dice <bdice@bradleydice.com>

Since qhull does not support periodic boundary conditions natively, we expand the box to include a portion of the particles' periodic images. The buffer width is given by the parameter `buff`. The computation of Voronoi tessellations and neighbors is only guaranteed to be correct if `buff >= L/2` where `L` is the longest side of the simulation box. For dense systems with particles filling the entire simulation volume, a smaller value for `buff` is acceptable. If the buffer width is too small, then some polytopes may not be closed (they may have a boundary at infinity), and these polytopes' vertices are excluded from the list. If either the polytopes or volumes lists that are computed is different from the size of the array of positions used in the *freud.voronoi.Voronoi.compute()* method, try recomputing using a larger buffer width.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **buff** (*float*) – Buffer width.

Variables

- **buffer** (*float*) – Buffer width.
- **nlist** (*NeighborList*) – Returns a weighted neighbor list. In 2D systems, the bond weight is the "ridge length" of the Voronoi boundary line between the neighboring particles. In 3D systems, the bond weight is the "ridge area" of the Voronoi boundary polygon between the neighboring particles.
- **polytopes** (list[numpy.ndarray]) – List of arrays, each containing Voronoi polytope vertices.
- **volumes** (((N_{cells})) numpy.ndarray) – Returns an array of volumes (areas in 2D) corresponding to Voronoi cells.

compute

Compute Voronoi diagram.

Parameters

- **positions** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate Voronoi diagram for.
- **box** (*freud.box.Box*) – Simulation box (Default value = None).
- **buff** (*float*) – Buffer distance within which to look for images (Default value = None).

computeNeighbors

Compute the neighbors of each particle based on the Voronoi tessellation. One can include neighbors from multiple Voronoi shells by specifying `numShells` in `getNeighbors()`. An example of computing neighbors from the first two Voronoi shells for a 2D mesh is shown below.

Retrieve the results with `getNeighbors()`.

Example:

```
from freud import box, voronoi
import numpy as np
vor = voronoi.Voronoi(box.Box(5, 5, is2D=True))
pos = np.array([[0, 0, 0], [0, 1, 0], [0, 2, 0],
                [1, 0, 0], [1, 1, 0], [1, 2, 0],
                [2, 0, 0], [2, 1, 0], [2, 2, 0]], dtype=np.float32)
first_shell = vor.computeNeighbors(pos).getNeighbors(1)
second_shell = vor.computeNeighbors(pos).getNeighbors(2)
```

(continues on next page)

(continued from previous page)

```
print('First shell:', first_shell)
print('Second shell:', second_shell)
```

Note: Input positions must be a 3D array. For 2D, set the z value to 0.

Parameters

- **positions** (($N_{particles}$, 3) numpy.ndarray) – Points to calculate Voronoi diagram for.
- **box** (*freud.box.Box*) – Simulation box (Default value = None).
- **buff** (*float*) – Buffer distance within which to look for images (Default value = None).
- **exclude_ii** (*bool, optional*) – True if pairs of points with identical indices should be excluded (Default value = True).

computeVolumes

Computes volumes (areas in 2D) of Voronoi cells.

New in version 0.8.

Must call *freud.voronoi.Voronoi.compute()* before this method. Retrieve the results with the *volumes* attribute.

getNeighbors

Get numShells of neighbors for each particle

Must call computeNeighbors() before this method.

Parameters **numShells** (*int*) – Number of neighbor shells.

1.4 Development Guide

Contributions to freud are highly encouraged. The pages below offer information about freud's design goals and how to contribute new modules.

1.4.1 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:

- In order to remain as agnostic to inputs as possible, freud makes no attempt to interface directly with simulation software. Instead, freud works directly with *NumPy <http://www.numpy.org/>_* arrays to retain maximum flexibility.

- 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).

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
        }
```

(continues on next page)

(continued from previous page)

```
    }

// 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)
```

(continues on next page)

(continued from previous page)

```
                {
                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.

1.4.2 Source Code Conventions

The guidelines below should be followed for any new code added to freud. This guide is separated into three sections, one for guidelines common to Python and C++, one for Python alone, and one for C++.

Both

Naming Conventions

The following conventions should apply to Python, Cython, and C++ code.

- Variable names use `lower_case_with_underscores`
- Function and method names use `lowerCaseWithNoUnderscores`
- Class names use `CapWords`

Python example:

```
class FreudClass(object):
    def __init__(self):
        pass
    def calcSomething(self, position_i, orientation_i, position_j, orientation_j):
        r_ij = position_j - position_i
        theta_ij = calcOrientationThing(orientation_i, orientation_j)
    def calcOrientationThing(self, orientation_i, orientation_j):
        ...
```

C++ example:

```
class FreudCPPClass
    {
    FreudCPPClass()
        {
        }
    computeSomeValue(int variable_a, float variable_b)
        {
        // do some things in here
        }
    };
```

Indentation

- Spaces, not tabs, must be used for indentation
- *4 spaces* are required per level of indentation and continuation lines
- There should be no whitespace at the end of lines in the file.
- Documentation comments and items broken over multiple lines should be *aligned* with spaces

```
class SomeClass
    {
    private:
        int m_some_member;         //!< Documentation for some_member
        int m_some_other_member;   //!< Documentation for some_other_member
    };

template<class BlahBlah> void some_long_func(BlahBlah with_a_really_long_argument_
↪list,
                                             int b,
                                             int c);
```

Formatting Long Lines

All code lines should be hand-wrapped so that they are no more than *79 characters* long. Simply break any excessively long line of code at any natural breaking point to continue on the next line.

```
cout << "This is a really long message, with "
     << message.length()
     << "Characters in it:"
     << message << endl;
```

Try to maintain some element of symmetry in the way the line is broken. For example, the *above* long message is preferred over the below:

```
cout << "This is a really long message, with " << message.length() << "Characters in␣
↪it:"
   << message << endl;
```

There are *special rules* for function definitions and/or calls:

- If the function definition (or call) cleanly fits within the character limit, leave it all on one line

```
int some_function(int arg1, int arg2)
```

- (Option 1) If the function definition (or call) goes over the limit, you may be able to fix it by simply putting the template definition on the previous line:

```
// go from
template<class Foo, class Bar> int some_really_long_function_name(int with_really_
↪long, Foo argument, Bar lists)
// to
template<class Foo, class Bar>
int some_really_long_function_name(int with_really_long, Foo argument, Bar lists)
```

- (Option 2) If the function doesn't have a template specifier, or splitting at that point isn't enough, split out each argument onto a separate line and align them.

```
// Instead of this...
int someReallyLongFunctionName(int with_really_long_arguments, int or, int maybe,␣
↪float there, char are, int just, float a, int lot, char of, int them)

// ...use this.
int someReallyLongFunctionName(int with_really_long_arguments,
                               int or,
                               int maybe,
                               float there,
                               char are,
                               int just,
                               float a,
                               int lot,
                               char of,
                               int them)
```

Python

Code in freud should follow PEP 8, as well as the following guidelines. Anything listed here takes precedence over PEP 8, but try to deviate as little as possible from PEP 8. When in doubt, follow these guidelines over PEP 8.

During continuous integration (CI), all Python and Cython code in freud is tested with flake8 to ensure PEP 8 compliance. It is strongly recommended to set up a pre-commit hook to ensure code is compliant before pushing to the repository:

```
flake8 --install-hook git
git config --bool flake8.strict true
```

Source

- All code should be contained in Cython files
- Python .py files are reserved for module level docstrings and minor miscellaneous tasks for, *e.g*, backwards compatibility.
- Semicolons should not be used to mark the end of lines in Python.

Documentation Comments

- Documentation is generated using sphinx.
- The documentation 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 ((N, 4) (:class:np.ndarray)): The points....`
  - Constructors should be documented at the class level.
  - Class attributes (*including properties*) should be documented as class attributes within the class-level docstring.
  - Optional arguments should be documented as such within the type after the actual type, and the default value should be included within the description *e.g.*, `r_max (float, optional): ... If None (the default), number is inferred....`

  - Properties that are settable should be documented the same way as optional arguments: `Lx (float, settable): Length in x.`

- All docstrings should be contained within the Cython files except module docstrings, which belong in the Python code.
- If you copy an existing file as a template, **make sure to modify the comments to reflect the new file**.
- Good documentation comments are best demonstrated with an in-code example. Liberal addition of examples is encouraged.

CPP

Indentation

- C++ code should follow Whitesmith's style. An extended set of examples follows:

```
class SomeClass
    {
    public:
        SomeClass();
        int SomeMethod(int a);
    private:
        int m_some_member;
    };

// indent function bodies
int SomeClass::SomeMethod(int a)
    {
    // indent loop bodies
    while (condition)
        {
        b = a + 1;
        c = b - 2;
        }

    // indent switch bodies and the statements inside each case
    switch (b)
        {
        case 0:
            c = 1;
            break;
        case 1:
            c = 2;
            break;
        default:
            c = 3;
            break;
        }

    // indent the bodies of if statements
    if (something)
        {
        c = 5;
        b = 10;
        }
     else if (something_else)
```

(continues on next page)

(continued from previous page)

```
        {
        c = 10;
        b = 5;
        }
     else
        {
        c = 20;
        b = 6;
        }

    // omitting the braces is fine if there is only one statement in a body (for␣
↪loops, if, etc.)
    for (int i = 0; i < 10; i++)
        c = c + 1;

    return c;
    // the nice thing about this style is that every brace lines up perfectly with␣
↪its mate
    }
```

- TBB sections should use lambdas, not templates

```
void someC++Function(float some_var,
                     float other_var)
    {
    // code before parallel section
    parallel_for(blocked_range<size_t>(0,n),
        [=] (const blocked_range<size_t>& r)
            {
            // do stuff
            });
```

Documentation Comments

- Documentation should be written in doxygen.

1.4.3 How to Add New Code

This document details the process of adding new code into freud.

Does my code belong in freud?

The freud library is not meant to simply wrap or augment external Python libraries. A good rule of thumb is *if the code I plan to write does not require C++, it does not belong in freud*. There are, of course, exceptions.

Create a new branch

You should branch your code from `master` into a new branch. Do not add new code directly into the `master` branch.

Add a New Module

If the code you are adding is in a *new* module, not an existing module, you must do the following:

- Create `cpp/moduleName` folder
- Edit `freud/__init__.py`
  - Add `from . import moduleName` so that your module is imported by default.
- Edit `freud/_freud.pyx`
  - Add `include "moduleName.pxi"`. This must be done to have freud include your Python-level code.
- Create `freud/moduleName.pxi` file
  - This will house the python-level code.
  - If you have a .pxd file exposing C++ classes, make sure to import that:

```
cimport freud._moduleName as moduleName
```

- Create `freud/moduleName.py` file
  - Make sure there is an import for each C++ class in your module:

```
from ._freud import MyC++Class
```

- Create `freud/_moduleName.pxd`
  - This file will expose the C++ classes in your module to python.
- Edit `setup.py`
  - Add `cpp/moduleName` to the `includes` list.
  - If there are any helper cc files that will not have a corresponding Cython class, add those files to the `sources` list inside the `extensions` list.
- Add line to `doc/source/modules.rst`
  - Make sure your new module is referenced in the documentation.
- Create `doc/source/moduleName.rst`

Add to an Existing Module

To add a new class to an existing module, do the following:

- Create `cpp/moduleName/SubModule.h` and `cpp/moduleName/SubModule.cc`
  - New classes should be grouped into paired `.h`, `.cc` files. There may be a few instances where new classes could be added to an existing `.h`, `.cc` pairing.
- Edit `freud/moduleName.py` file
  - Add a line for each C++ class in your module:

```
from ._freud import MyC++Class
```

- Expose C++ class in `freud/_moduleName.pxd`
- Create Python interface in `freud/moduleName.pxi`

You must include sphinx-style documentation and unit tests.

- Add extra documentation to `doc/source/moduleName.rst`
- Add unit tests to `freud/tests`

1.5 References and Citations

1.6 License

```
freud Open Source Software License Copyright 2010-2019 The Regents of
the University of Michigan All rights reserved.

freud may contain modifications ("Contributions") provided, and to which
copyright is held, by various Contributors who have granted The Regents of the
University of Michigan the right to modify and/or distribute such Contributions.

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
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

1.7 Credits

1.7.1 freud Developers

The following people contributed to the development of freud.

Eric Harper, University of Michigan - **Former lead developer**

- TBB parallelism.
- PMFT module.
- NearestNeighbors.

- RDF.
- Bonding module (since removed).
- Cubatic order parameter.
- Hexatic order parameter.
- Pairing2D (since removed).

Joshua A. Anderson, University of Michigan - **Creator**

- Initial design and implementation.
- IteratorLinkCell.
- LinkCell.
- Various density modules.
- freud.parallel.
- Indexing modules.
- cluster.pxi.

Matthew Spellings - **Former lead developer**

- Added generic neighbor list.
- Enabled neighbor list usage across freud modules.
- Correlation functions.
- LocalDescriptors class.
- interface.pxi.

Erin Teich

- Wrote environment matching module.
- BondOrder (with Julia Dshemuchadse).
- Angular separation (with Andrew Karas).
- Contributed to LocalQl development.
- Wrote LocalBondProjection module.

M. Eric Irrgang

- Authored (now removed) kspace code.
- Numerous bug fixes.
- Various contributions to freud.shape.

Chrisy Du

- Authored all Steinhardt order parameters.
- Fixed support for triclinic boxes.

Antonio Osorio

- Developed TrajectoryXML class.
- Various bug fixes.
- OpenMP support.

Vyas Ramasubramani - **Lead developer**

- Ensured pep8 compliance.
- Added CircleCI continuous integration support.
- Rewrote docs.
- 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.
- Split non-order parameters out of order module into separate environment module..
- Rewrote documentation for order, density, and environment modules.
- Add code coverage metrics.
- Added support for PyPI, including ensuring that NumPy is installed.
- Converted all docstrings to Google format, fixed various incorrect docs.

Bradley Dice - **Lead developer**

- Cleaned up various docstrings.
- HexOrderParameter bug fixes.
- Cleaned up testing code.
- 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.
- Proper box duck-typing.
- Removed nose from unit testing.
- Use lambda function for parallelizing CorrelationFunction with TBB.
- Finalized boost removal.

Richmond Newman

- Developed the freud box.
- Solid liquid order parameter.

Carl Simon Adorf

- Developed the python box module.

Jens Glaser

- Wrote kspace.pxi front-end.
- Modifications to kspace module.
- Nematic order parameter.

Benjamin Schultz

- Wrote Voronoi module.
- Fix normalization in GaussianDensity.
- Bugfixes in freud.shape.

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.

Paul Dodd

- Fixed CorrelationFunction namespace, added ComputeOCF class for TBB parallelization.

Tim Moore

- Added optional rmin argument to density.RDF.

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.

1.7.2 Source code

Eigen (http://eigen.tuxfamily.org/) is included as a git submodule in freud. Eigen is made available under the Mozilla Public License v.2.0 (http://mozilla.org/MPL/2.0/). Its linear algebra routines are used for various tasks including the computation of eigenvalues and eigenvectors.

fsph (https://bitbucket.org/glotzer/fsph) is included as a git submodule in freud. fsph is made available under the MIT license. It is used for the calculation of spherical harmonics, which are then used in the calculation of various order parameters, under the following 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,
```

(continues on next page)

(continued from previous page)

```
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

CHAPTER 2

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 forum. All contributions to freud are welcomed via pull requests! Please see the *development guide* for more information on requirements for new code.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Bibliography

[Matplotlib] Hunter, J. D. (2007). Matplotlib: A 2D Graphics Environment. Computing in Science & Engineering, 9 (3), 90-95. https://doi.org/10.1109/MCSE.2007.55

[Bokeh] Bokeh Development Team (2018). Bokeh: Python library for interactive visualization. https://bokeh.pydata.org

[HajiAkbari2015] Haji-Akbari, A., & Glotzer, S. C. (2015). Strong orientational coordinates and orientational order parameters for symmetric objects. Journal of Physics A: Mathematical and Theoretical, 48. https://doi.org/10.1088/1751-8113/48/48/485201

[vanAndersKlotsa2014] van Anders, G., Klotsa, D., Ahmed, N. K., Engel, M., & Glotzer, S. C. (2014). Understanding shape entropy through local dense packing. Proceedings of the National Academy of Sciences, 111 (45), E4812–E4821. https://doi.org/10.1073/pnas.1418159111

[vanAndersAhmed2014] van Anders, G., Ahmed, N. K., Smith, R., Engel, M., & Glotzer, S. C. (2014). Entropically patchy particles: Engineering valence through shape entropy. ACS Nano, 8 (1), 931–940. https://doi.org/10.1021/nn4057353

[Lechner2008] Lechner, W., & Dellago, C. (2008). Accurate determination of crystal structures based on averaged local bond order parameters. Journal of Chemical Physics, 129 (11). https://doi.org/10.1063/1.2977970

[Steinhardt1983] Steinhardt, P.J., Nelson, D.R., & Ronchetti, M. (1983). Bond-orientational order in liquids and glasses. Phys. Rev. B 28 (784). https://doi.org/10.1103/PhysRevB.28.784

[Calandrini2011] Calandrini, V., Pellegrini, E., Calligari, P., Hinsen, K., & Kneller, G. R. (2011). nMoldyn-Interfacing spectroscopic experiments, molecular dynamics simulations and models for time correlation functions. École thématique de la Société Française de la Neutronique, 12, 201-232. https://doi.org/10.1051/sfn/201112010

Python Module Index

f

Index

Symbols

A

B

C

F

G

H

I

L

M